

Optimizing Cache Usage

Computer Organization

Monday, 26 September 2022

Many slides adapted from:
Computer Organization and Design,
Patterson & Hennessy
5th Edition, © 2014, MK
and from Prof. Mary Jane Irwin, PSU



TÉCNICO LISBOA

Summary

- Previous Class
 - Improving Cache Performance
- Today:
 - Multilevel Caches
 - Code optimization
 - Data access
 - Program access

Reducing Cache Miss Rates: Multilevel Caches

- Use multiple levels of caches
 - With advancing technology have more than enough room on the die for bigger L1 caches *or* for a second level of caches normally a **unified** L2 cache
 - i.e., holds both instructions and data
 - Many high-end systems already include unified L3 cache
- Design considerations for L1 and L2 caches are very different
 - Primary cache attached to CPU
 - focus on **minimizing hit time** (i.e. small, but fast)
 - » Smaller with smaller block sizes
 - Level-2 cache services misses from primary cache
 - focus on **reducing miss rate** (i.e. large, slower than L1)
 - to reduce the penalty of long main memory access times
 - » Larger with larger block sizes
 - » Higher levels of associativity

Multilevel Cache Considerations

- Primary cache
 - Focus on minimal hit time
- L-2 cache
 - Focus on low miss rate to avoid main memory access
 - Hit time has less overall impact
- Results
 - L-1 cache usually smaller than a single cache
 - L-1 block size smaller than L-2 block size

$$t_{\text{access}} = t_{\text{hitL1}} + p_{\text{missL1}} \times t_{\text{penaltyL1}}$$

$$t_{\text{penaltyL1}} = t_{\text{hitL2}} + p_{\text{missL2}} \times t_{\text{penaltyL2}}$$

$$t_{\text{access}} = t_{\text{hitL1}} + p_{\text{missL1}} \times (t_{\text{hitL2}} + p_{\text{missL2}} \times t_{\text{penaltyL2}})$$

Multilevel Cache Example

- Given
 - CPU base CPI = 1, clock rate = 4GHz
 - Miss rate/instruction = 2%
 - Main memory access time = 100ns
- With just primary cache
 - Miss penalty = $100\text{ns}/0.25\text{ns} = 400$ cycles
 - Effective CPI = $1 + 0.02 \times 400 = 9$

Example (cont.)

- Now add L-2 cache
 - Access time = 5ns
 - Global miss rate to main memory = 0.5%
- Primary miss with L-2 hit
 - Penalty = $5\text{ns}/0.25\text{ns} = 20$ cycles
- Primary miss with L-2 miss
 - Extra penalty = 400 cycles

$$\text{CPI} = 1 + 0.02 \times 20 + 0.005 \times 400 = 3.4$$

$$\text{Performance ratio} = 9/3.4 = 2.6$$

Check@home: Two Machines' Cache Parameters

Characteristic	ARM Cortex-A8	Intel Nehalem
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	32 KiB each for instructions/data	32 KiB each for instructions/data per core
L1 cache associativity	4-way (I), 4-way (D) set associative	4-way (I), 8-way (D) set associative
L1 replacement	Random	Approximated LRU
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, Write-allocate(?)	Write-back, No-write-allocate
L1 hit time (load-use)	1 clock cycle	4 clock cycles, pipelined
L2 cache organization	Unified (instruction and data)	Unified (instruction and data) per core
L2 cache size	128 KiB to 1 MiB	256 KiB (0.25 MiB)
L2 cache associativity	8-way set associative	8-way set associative
L2 replacement	Random(?)	Approximated LRU
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate (?)	Write-back, Write-allocate
L2 hit time	11 clock cycles	10 clock cycles
L3 cache organization	–	Unified (instruction and data)
L3 cache size	–	8 MiB, shared
L3 cache associativity	–	16-way set associative
L3 replacement	–	Approximated LRU
L3 block size	–	64 bytes
L3 write policy	–	Write-back, Write-allocate
L3 hit time	–	35 clock cycles

Summary: Improving Cache Performance

1. Reduce the time to hit in the cache

- smaller cache
- direct mapped cache
- smaller blocks
- for writes
 - no write allocate – no “hit” on cache, just write to write buffer
 - write allocate – to avoid two cycles (first check for hit, then write)
pipeline writes via a delayed write buffer to cache

2. Reduce the miss rate

- bigger cache
- more flexible placement (increase associativity)
- larger blocks (16 to 64 bytes typical)
- victim cache – small buffer holding most recently discarded blocks

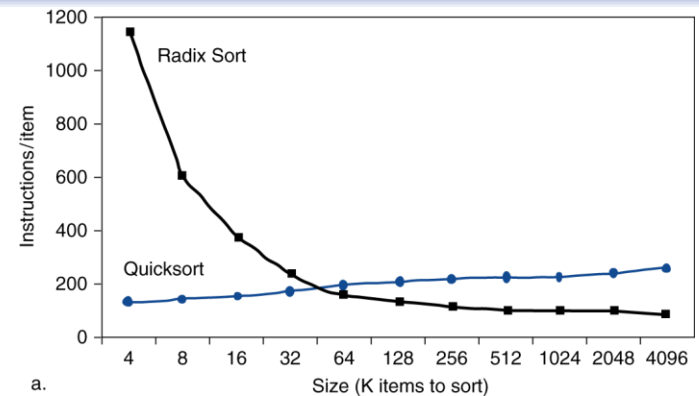
Summary: Improving Cache Performance

3. Reduce the miss penalty

- smaller blocks
- use a write buffer to hold dirty blocks being replaced so don't have to wait for the write to complete before reading
- check write buffer (and/or victim cache) on read miss
 - may get lucky
- for large blocks fetch critical word first
- use multiple cache levels – L2 cache not tied to CPU clock rate
- faster backing store/improved memory bandwidth
 - wider buses
 - memory interleaving, DDR SDRAMs

Interactions with Software

- Misses depend on memory access patterns
 - Algorithm behavior
 - Compiler optimization for memory access
- Inefficient cache use = lower performance
 - How increase cache utilization?
Cache-awareness!



Code Optimization

- The main objective is to reduce the miss-rate by changing the memory access pattern with code optimization techniques
- Which misses should be considered?
 - Mainly, conflict misses
- Which accesses should be considered?
 - Data accesses
 - Program accesses
- Usually... greater flexibility to re-organize the data in memory and their corresponding access patterns.

Data Access Optimization

- Many techniques exist for the optimization of data access:
 - Prefetching and preloading data into cache
 - Cache-conscious structure layout
 - Tree data structures
 - Linearization caching
 - Memory allocation
 - Blocking and strip mining
 - Padding data to align to cache lines
 - Aliasing and “anti-aliasing”
 - “Compressing” data
 - ...

Prefetching and Preloading

- Software prefetching
 - Not too early: data may be evicted before use
 - Not too late: data not fetched in time for use
 - Greedy
- Preloading (pseudo-prefetching)
 - Hit-under-miss processing

Software Prefetching

```
// Loop through and process all 4n elements  
for (int i = 0; i < 4 * n; i++)  
    Process(elem[i]);
```

```
const int kLookAhead = 4; // Some elements ahead  
for (int i = 0; i < 4 * n; i += 4) {  
    Prefetch(elem[i + kLookAhead]);  
    Process(elem[i + 0]);  
    Process(elem[i + 1]);  
    Process(elem[i + 2]);  
    Process(elem[i + 3]);  
}
```

Preloading (pseudo-prefetch)

```
Elem a = elem[0];  
for (int i = 0; i < 4 * n; i += 4) {  
    Elem e = elem[i + 4]; // Cache miss, non-blocking  
    Elem b = elem[i + 1]; // Cache hit  
    Elem c = elem[i + 2]; // Cache hit  
    Elem d = elem[i + 3]; // Cache hit  
    Process(a);  
    Process(b);  
    Process(c);  
    Process(d);  
    a = e;  
}
```

Note: This code reads one element beyond the end of the `elem` array.

Greedy Prefetching

```
void PreorderTraversal(Node *pNode) {  
    // Greedily prefetch left traversal path  
    Prefetch(pNode->left);  
    // Process the current node  
    Process(pNode);  
    // Greedily prefetch right traversal path  
    Prefetch(pNode->right);  
    // Recursively visit left then right subtree  
    PreorderTraversal(pNode->left);  
    PreorderTraversal(pNode->right);  
}
```



Structures

- Cache-conscious layout
 - Field reordering (usually grouped conceptually)
 - Hot/cold splitting
- Let use decide format
 - Array of structures
 - Structures of arrays
- Little compiler support
 - Easier for non-pointer languages (Java)
 - C/C++: do it yourself

Field Reordering

```
struct S {  
    void *key;  
    int count[20];  
    S *pNext;  
};
```

```
struct S {  
    void *key;  
    S *pNext;  
    int count[20];  
};
```



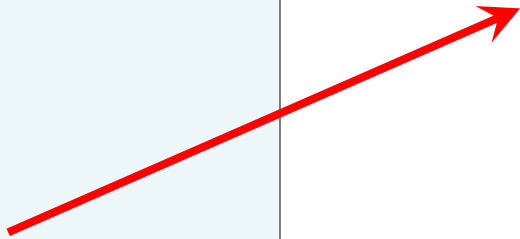
```
void Foo(S *p, void *key, int k){  
    while (p) {  
        if (p->key == key) {  
            p->count[k]++;  
            break;  
        }  
        p = p->pNext;  
    }  
}
```

**Likely accessed
together so store
them together!**

Hot/cold Splitting

Hot fields:

```
struct S {  
    void *key;  
    S *pNext;  
    S2 *pCold;  
};
```

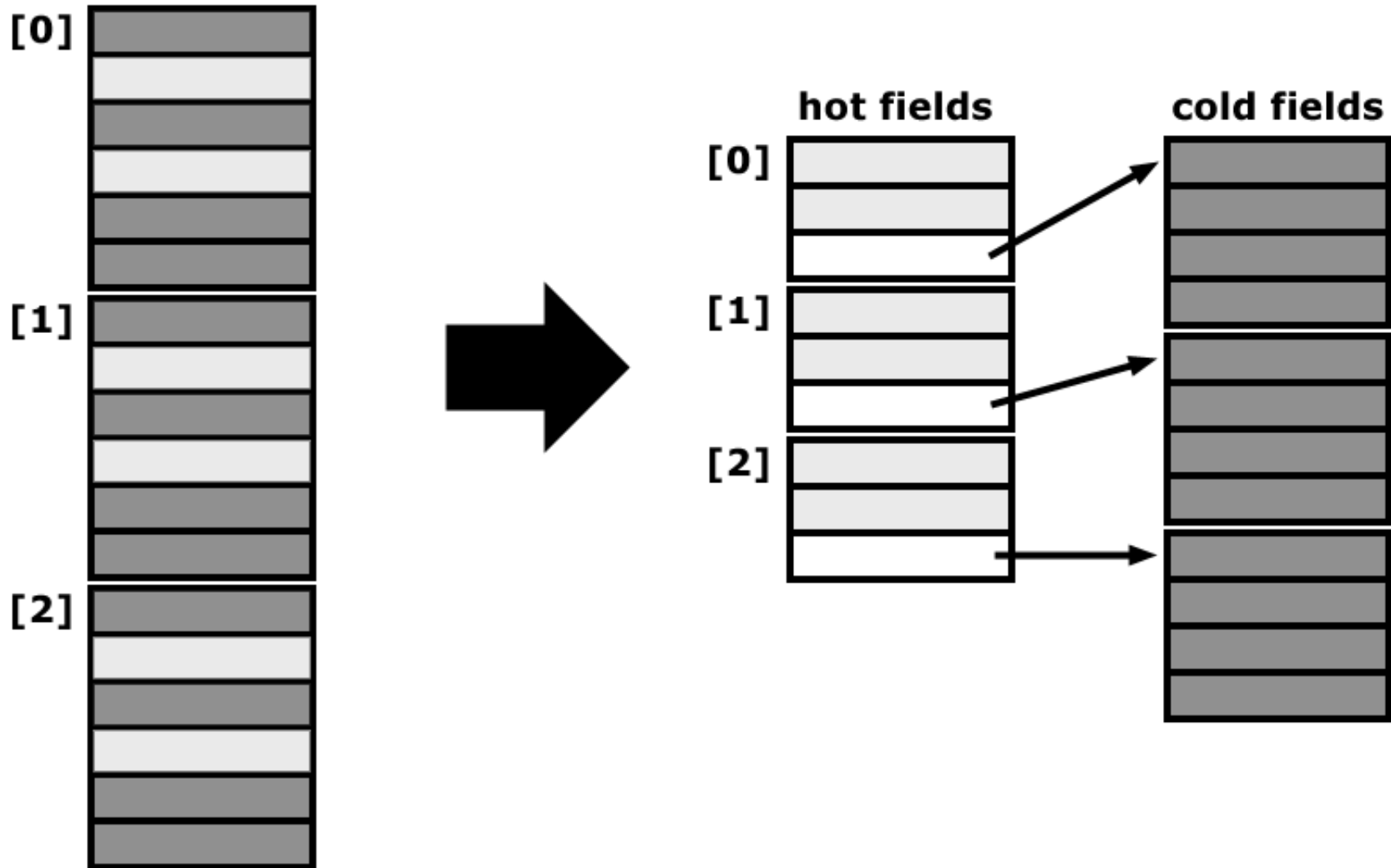


Cold fields:

```
struct S2 {  
    int count[20];  
};
```

- Allocate all 'struct S' from a memory pool
 - Increases coherence
- Prefer array-style allocation
 - No need for actual pointer to cold fields

Hot/cold Splitting



Merging Arrays

- McFarling [1989] reduced caches misses by 75% on 8KB direct mapped cache, 4 byte blocks in software
- Instructions
 - Reorder procedures in memory so as to reduce conflict misses
 - Profiling to look at conflicts (using tools they developed)
- Data
 - **Merging Arrays**: improve spatial locality by single array of compound elements vs. 2 arrays
 - **Loop Fusion**: Combine 2 independent loops that have same looping and some variables overlap
 - **Loop Interchange**: change nesting of loops to access data in order stored in memory
 - **Blocking**: Improve temporal locality by accessing “blocks” of data repeatedly vs. going down whole columns or rows

Variables Organization Example

```
/* Before: 2 sequential arrays */  
int val[SIZE];  
int key[SIZE];
```

```
/* After: 1 array of structures */  
struct merge {  
    int val;  
    int key;  
};  
struct merge merged_array[SIZE];
```

- Reducing conflicts between val & key;
 - improve spatial locality

Loop Fusion Example

```
/* Before */  
for (i = 0; i < N; i = i+1)  
    for (j = 0; j < N; j = j+1)  
        a[i][j] = 1/b[i][j] * c[i][j];  
for (i = 0; i < N; i = i+1)  
    for (j = 0; j < N; j = j+1)  
        d[i][j] = a[i][j] + c[i][j];
```

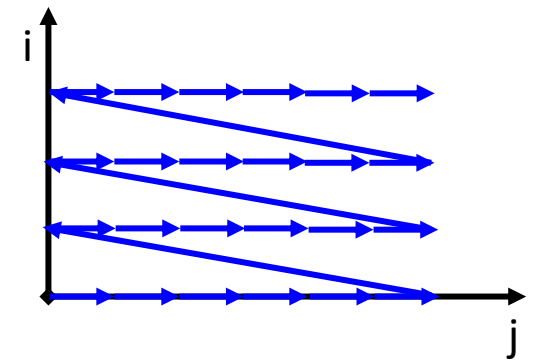
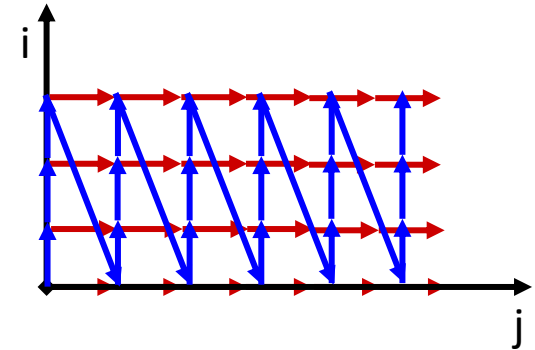
```
/* After */  
for (i = 0; i < N; i = i+1)  
    for (j = 0; j < N; j = j+1) {  
        a[i][j] = 1/b[i][j] * c[i][j];  
        d[i][j] = a[i][j] + c[i][j];  
    }
```

- 2 misses per access to a & c vs. one miss per access;
 - improve spatial locality

Loop Interchange Example

```
/* Before */  
for (k = 0; k < 100; k = k+1)  
    for (j = 0; j < 100; j = j+1)  
        for (i = 0; i < 5000; i = i+1)  
            x[i][j] = 2 * x[i][j];
```

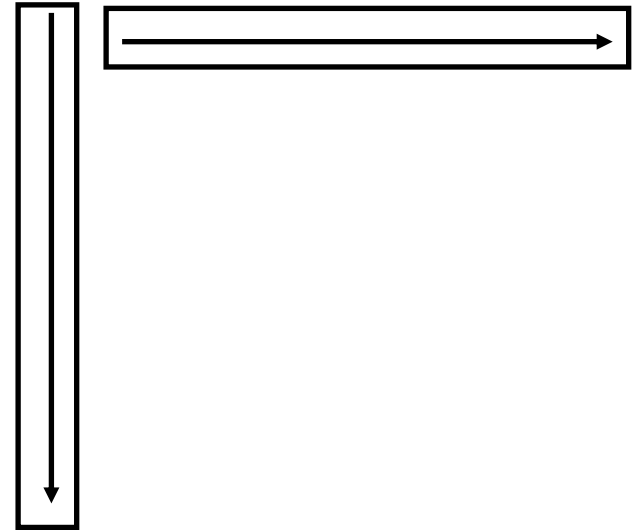
```
/* After */  
for (k = 0; k < 100; k = k+1)  
    for (i = 0; i < 5000; i = i+1)  
        for (j = 0; j < 100; j = j+1)  
            x[i][j] = 2 * x[i][j];
```



- Sequential accesses instead of striding through memory every 100 words
 - improved spatial locality

Blocking (aka Tiling)

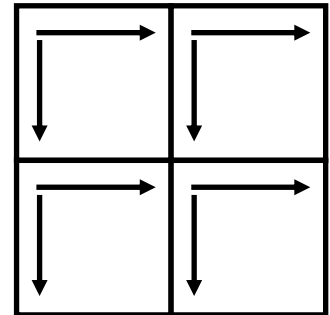
```
/* Before */  
for (i = 0; i < N; i = i+1)  
  for (j = 0; j < N; j = j+1) {  
    r = 0;  
    for (k = 0; k < N; k = k+1)  
      r = r + y[i][k] * z[k][j];  
    x[i][j] = r;  
  }
```



- Two Inner Loops:
 - Read all NxN elements of z[]
 - Read N elements of 1 row of y[] repeatedly
 - Write N elements of 1 row of x[]
- Capacity Misses a function of N & Cache Size:
 $2N^3 + N^2$
- Idea: compute on BxB submatrix that fits the cache

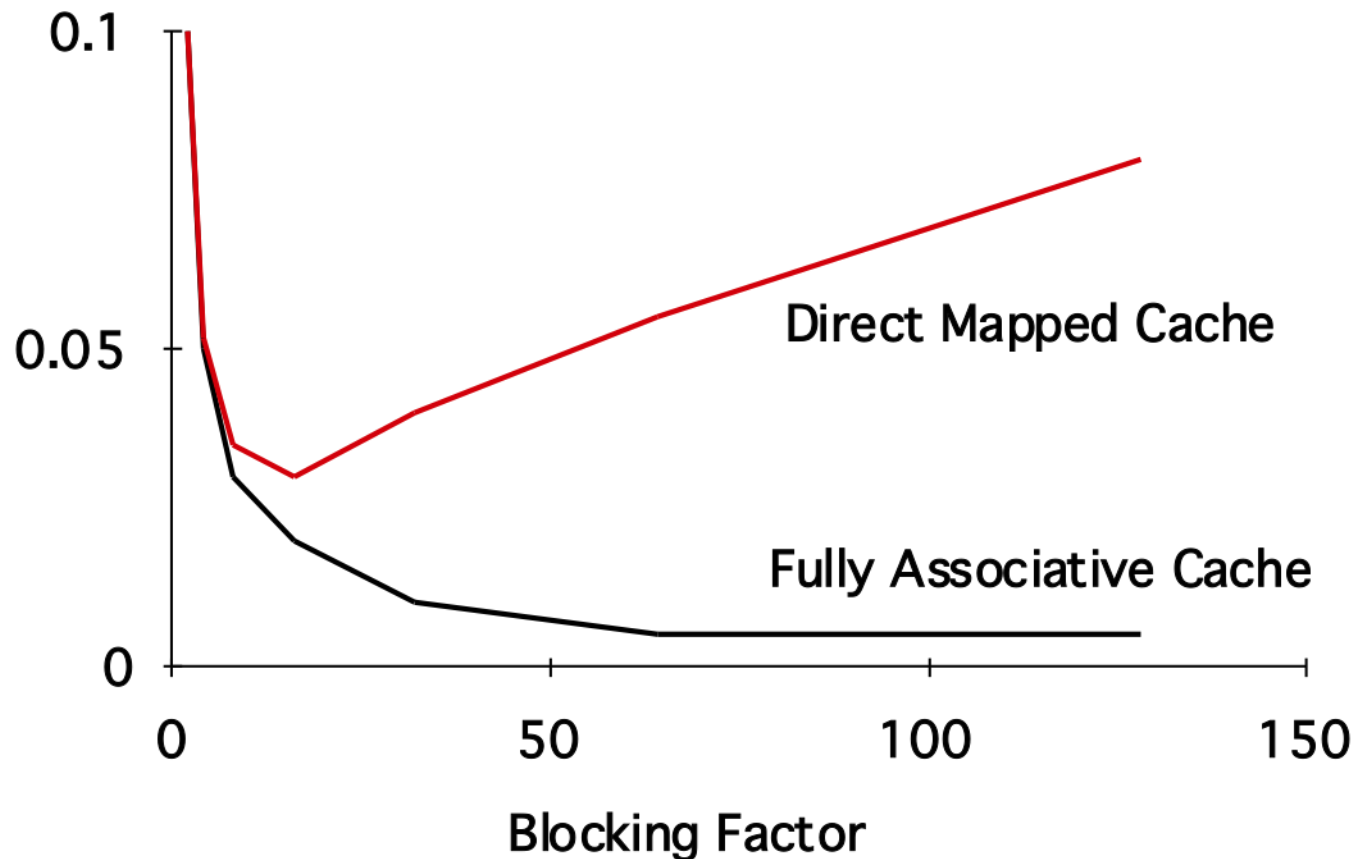
Blocking Example

```
/* After */
for(jj = 0; jj < N; jj = jj + B)
  for(kk = 0; kk < N; kk = kk + B)
    for(i = 0; i < N; i = i+1)
      for(j = jj; j < min(jj+B-1,N); j = j+1) {
        r = 0;
        for (k = kk; k < min(kk+B-1,N); k = k+1)
          r = r + y[i][k]*z[k][j];
        x[i][j] = x[i][j] + r;
      };
```



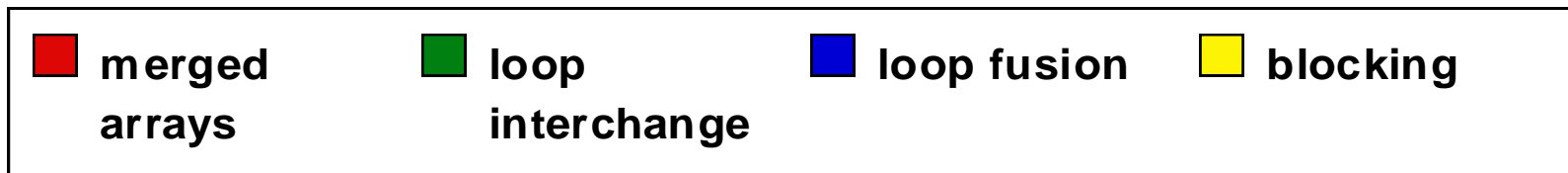
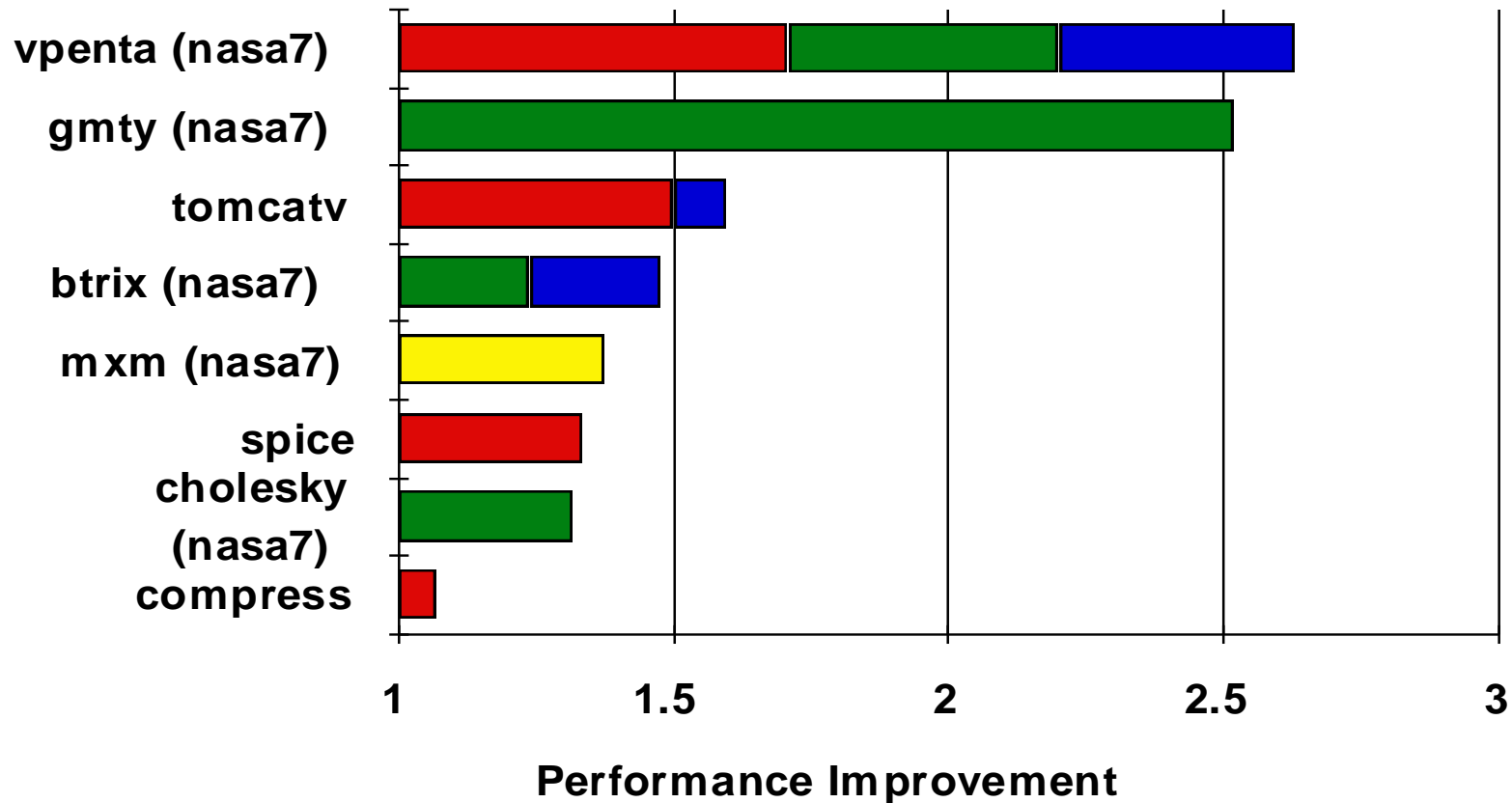
- B called **Blocking Factor**
- Capacity Misses from $2N^3 + N^2$ to $N^3/B + 2N^2$
- Conflict Misses Too?

Reducing Conflict Misses by Blocking



- Conflict misses in caches not FA vs. Blocking size
 - Lam et al [1991] a blocking factor of 24 had a fifth the misses vs. 48 despite both fit in cache

Summary of Optimizations to Reduce Cache Misses



Next Class

- Virtual memory hardware support

Optimizing Cache Usage

Computer Organization

Monday, 26 September 2022

Many slides adapted from:
Computer Organization and Design,
Patterson & Hennessy
5th Edition, © 2014, MK
and from Prof. Mary Jane Irwin, PSU



TÉCNICO LISBOA