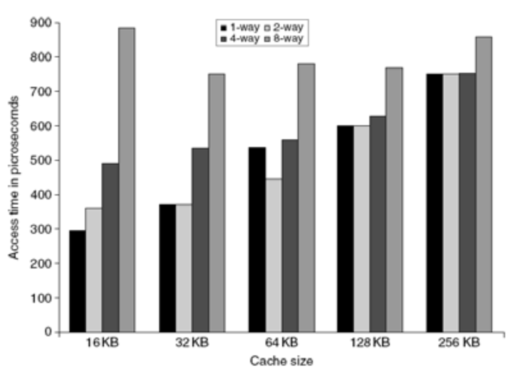
Advanced optimizations of cache performance (§ 2.2)

30

1. Small and Simple Caches to reduce hit time

- Critical timing path:
  - address tag memory, then compare tags, then select set
- Lower associativity
- Direct-mapped caches can overlap tag comparison and transmission of data



31

2. Way prediction to reduce hit time

- Combine fast hit time of Direct Mapped and the lower conflict misses of 2-way SA caches?
  - check one set first (speed of direct mapped cache)
  - on a miss, check the other set, if it hits, call it a *pseudo-hit* (slow hit)
  - way prediction is a bit to indicate which half to check first (changes dynamically)

- May extend prediction to more than 2-way SA caches
- Saves power
- Drawback: CPU pipeline is hard if hit takes sometimes 1 and sometimes 2 cycles

32

3. Pipeline cache access to increase bandwidth

- Examples:
  - » Pentium: 1 cycle
  - » Pentium Pro – Pentium III: 2 cycles
  - » Pentium 4 – Core i7: 4 cycles

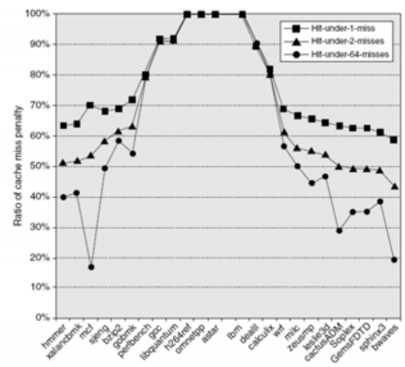
- Increases branch mis-prediction penalty
- Makes it easier to increase associativity

33

4. Non-blocking caches to increase bandwidth

- *Hit under miss* allows data cache to continue to supply cache hits during a miss -- useful only with out-of-order execution.
- *Hit under multiple miss* or *miss under miss* may further lower the effective miss penalty by overlapping multiple misses

– Significantly increases the complexity of the cache controller (multiple outstanding memory accesses)
– Requires multiple memory banks (otherwise cannot support)
– Pentium Pro allows 4 outstanding memory misses



34

5. Multi-bank caches to increase bandwidth

- Individual memory controller for each bank.
- Each bank may have its own address and data lines.
- Banks used for independent accesses vs. faster sequential accesses.
  - ARM Cortex-A8 supports 1-4 banks for L2
  - Intel i7 supports 4 banks for L1 and 8 banks for L2
- How blocks are interleaved affects performance.

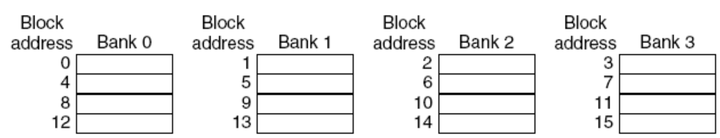


Figure 2.6 Four-way interleaved cache banks using block addressing. Assuming 64 bytes per blocks, each of these addresses would be multiplied by 64 to get byte addressing.

35

6. Critical word first + early restart to reduce miss penalty

- Don't wait for full block to be loaded before restarting CPU
  - *Early restart* - As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution
  - *Critical Word First* - Request the missed word first from memory and send it to CPU as soon as it arrives; Generally useful only in large blocks,
- Beneficial when we have long cache lines (blocks)
- If want next sequential word, early restart may not be useful,

36

7. Merging write buffer to reduce miss penalty

No merging

Write address	V		V		V		V	
100	1	Mem[100]	0		0		0	
108	1	Mem[108]	0		0		0	
116	1	Mem[116]	0		0		0	
124	1	Mem[124]	0		0		0	

Merging

Write address	V		V		V		V	
100	1	Mem[100]	1	Mem[108]	1	Mem[116]	1	Mem[124]
	0		0		0		0	
	0		0		0		0	
	0		0		0		0	

- Most useful in write through caches
- Combine writing individual words into a block
- Writing block is faster than writing individual words

37

8. Compiler optimizations to reduce miss rate

- **Instructions**
  - Reorder procedures in memory so as to reduce conflict misses
  - Aligning basic blocks with cache blocks (lines)
  - Profiling to look at conflicts.

- **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

38

Merging Arrays Example:

```
int val[SIZE];
int key[SIZE];
```

⇨

```
struct merge {
        int val;
        int key;
};
struct merge merged_array[SIZE];
```

Reducing conflicts between val & key improves spatial locality

39

Loop Fusion Example

```
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];
```

⇓

```
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];}
```

One miss per access to `a` and `c` vs. two misses per access.
Improves temporal locality

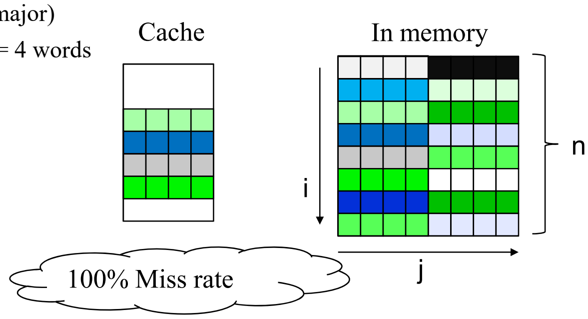
40

loop interchange example

Matrix A is stored Row-wise (row major)

Fully associative cache, block size = 4 words

Cache size < 4n words




```
for (j = 0; j< n; j = j+1)
  for (i = 0; i < n; i = i+1)
        C =+  A[i][j];
```

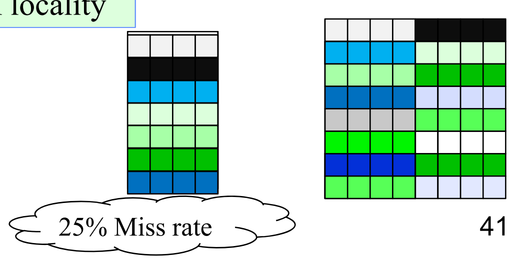
Column-wise memory access

⇓

Take advantage of spatial locality

Row-wise memory access

```
for (i = 0; i < n; i = i+1)
  for (j = 0; j < n; j = j+1)
        C = + A[i][j];
```



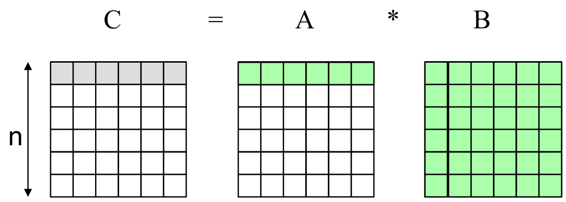
41

Optimization through blocking (partitioning) example

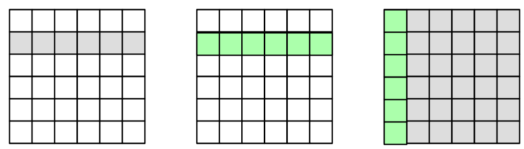
Matrix multiplication

```
for (i = 0 ; i < n ; i++)
  for (j = 0 ; j < n ; j ++)
   {r = 0;
    for (k = 0; k < n; k++)
      r = r + A[i][k]*B[k][j];
    C[i][j] =  r;  };
```

Assume:
A fully associative cache
Block size = 1 word
Cache size < n^2



Data used when i = 0, j = 0, … , n-1



Data used when i = 1, j = 0, … , n-1

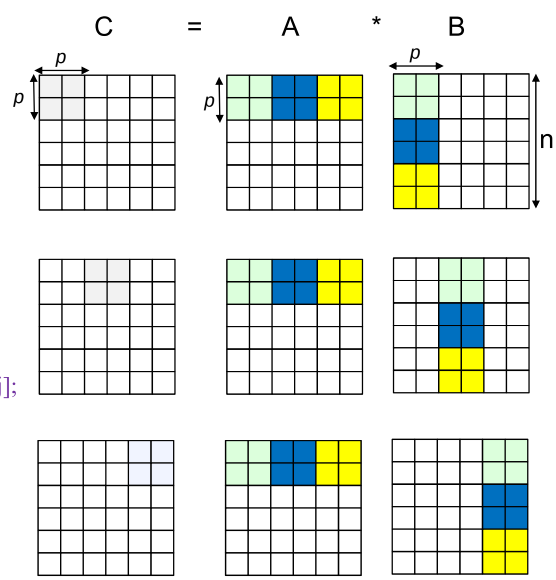
- One row of A will fit in the cache and be repeatedly used (perfect reuse)
- B will not fit in cache and hence a column of B will be evicted before reuse
- Every element of B will be used only once when brought to the cache

Optimization through blocking (partitioning) example

Partition the matrices into submatrices of size *p* x *p*

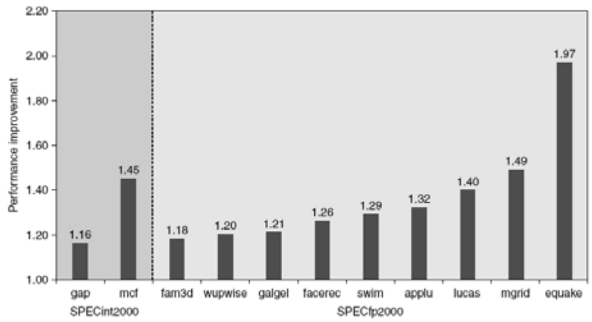
```
for (si = 0; si < n; si =+ p)
 for (sj = 0; sj < n; sj =+ p)
  for (sk = 0; sk < n; sk =+ p)
     for (i=si ; i < si+p ; i++)
       for (j=sj ; j < sj+p ; j++)
        {r = 0;
          for (k=sk; k < sk+p; k++)
                r = r + A[i][k]*B[k][j];
          C[i][j] =+  r;
        };
```

If cache size > $p * n + p^2$, then
- A will be perfectly reused
- Each element of B will be reused "*p*" times (reduce miss rate)



9. Hardware prefetching to reduce miss rate and penalty

- Instruction Prefetching
  - Can fetch 2 (or more) blocks on a miss
  - Extra block placed in "*stream buffer*"
  - On miss, check stream buffer - if found move to cache and prefetch next
- Data Prefetching:
  - May have multiple stream buffers beyond the cache, each prefetching at a different address
- Relies on extra memory bandwidth that can be used without penalty.



44

10.Compiler prefetching to reduce miss rate and penalty

- Data Prefetch
  - Load data into register
  - Cache Prefetch: load into cache
  - Special prefetching instructions should not cause premature page faults.
  - Issuing Prefetch Instructions takes time
  - Is cost of prefetch issues < savings in reduced misses?
- Works only if can overlap prefetching with execution.

- **Example:** Assume that arrays a[] and b[] are aligned at block boundaries and that the cache block size is 4 words.

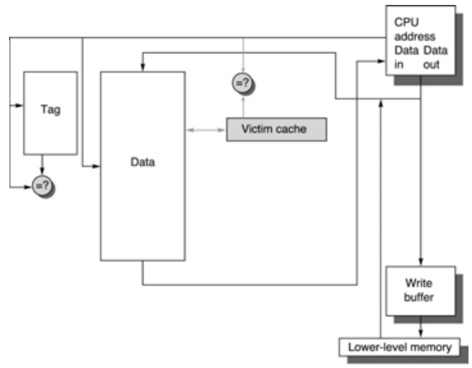
```
for (i=0 ; i < 100 ; i++)
    if (i mod 4 = 0) prefetch (b[i+4], a[i+4])
    b[i] = c * a[i] + d * a[i+1] ;
```

if body of loop takes 20 cycles to execute and cache miss penalty is 80 cycles, then, after the first few iterations, data will be in cache when needed.

45

Using victim caches

- A buffer to place data just evicted from cache
- A small number of fully associative entries
- Accessed in parallel with cache (no increase in hit time)
- On a hit in the VC, swap blocks in VC and cache



- When used with direct mapped caches, it has the effect of adding associativity to the most recently used cache blocks

46