Memory Hierarchy and Cache Design (2)

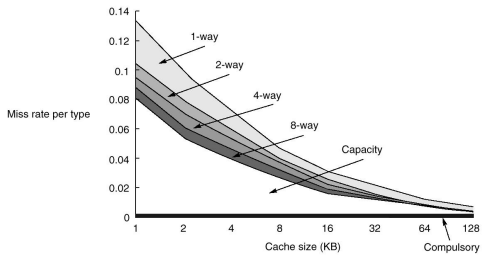
Improving Cache Performance

- **Average memory-access time**
  = Hit time + Miss rate x Miss penalty
- **Improve performance by:**
  1. Reduce the miss rate,
  2. Reduce the miss penalty, or
  3. Reduce the time to hit in the cache.

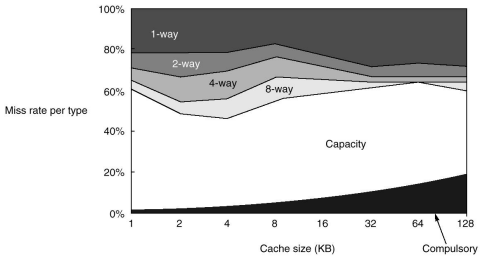
Reducing Miss Rate

- **Sources of Cache misses**
  - **Compulsory misses**
    - » **Misses resulting from first access to a block**
    - » **Also called cold start misses or first reference misses**
  - **Capacity misses**
    - » **Misses resulting from finite or limited cache size**
  - **Conflict misses**
    - » **Misses resulting from finite or limited set associativity**
    - » **Also called collision misses or interference misses**

Miss rate per type



Miss rate per type



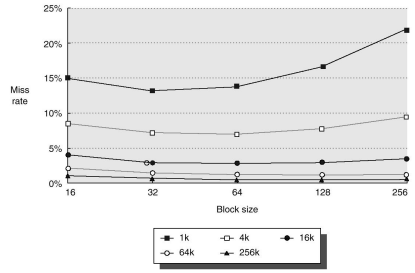
Back to Reducing Miss Rate

1. Larger block size
2. Higher set associativity
3. Victim cache
4. Pseudo-associative cache
5. Hardware prefetching of instruction and data
6. Compiler-controlled prefetching
7. Compiler optimization

First Approach
Large Block Size

- reduces compulsory misses - principle of locality
- reduces number of blocks in cache
  - increasing conflict misses

Larger Block Size



Larger Block Size

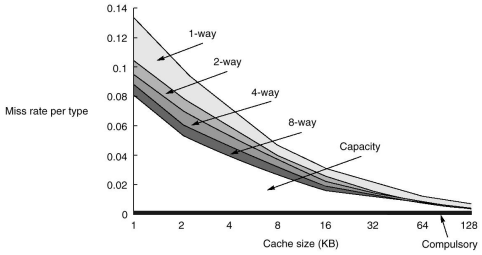
Miss rate

Block size	Cache size 1K	4K	16K	64K	256K
16	15.05%	8.57%	3.94%	2.04%	1.09%
32	13.34%	7.24%	2.87%	1.35%	0.70%
64	13.76%	7.00%	2.64%	1.06%	0.51%
128	16.64%	7.78%	2.77%	1.02%	0.49%
256	22.01%	9.51%	3.29%	1.15%	0.49%

Average memory access time

Block size	Miss penalty	Cache size 1K	4K	16K	64K	256K
16	42	7.321	4.599	2.655	1.857	1.458
32	44	**6.870**	**4.186**	**2.263**	1.594	1.308
64	48	7.605	4.360	2.267	**1.509**	**1.245**
128	56	10.318	5.357	2.551	1.571	1.274
256	72	16.847	7.847	3.369	1.828	1.353

Second Approach Higher Associativity



Higher Set Associativity

Sacrifice : increased hit time

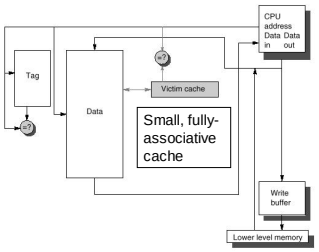
Cache size (KB)	Associativity One-way	Two-way	Four-way	Eight-way
1	7.65	6.60	6.22	5.44
2	5.90	4.90	4.62	4.09
4	4.60	3.95	3.57	3.19
8	3.30	3.00	2.87	2.59
16	2.45	2.20	2.12	2.04
32	2.00	1.80	1.77	**1.79**
64	1.70	1.60	1.57	**1.59**
128	1.50	1.45	1.42	**1.44**

Assumptions: clock cycle time (2-way) = 1.10 x clock cycle time (1-way)
clock cycle time (4-way) = 1.12 x clock cycle time (1-way)
clock cycle time (8-way) = 1.14 x clock cycle time (1-way)

Third Approach Victim Cache

- **Add a small, fully associative cache between a cache and its refill path**
- **Check victim cache upon a miss**
- **A four-entry victim cache removed 20% to 95% of the conflict misses in a 4-KB direct mapped cache**

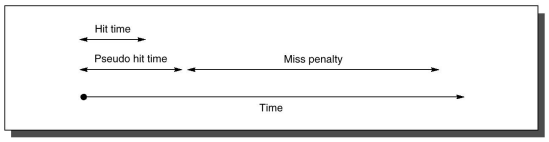
Victim Cache



Fourth Approach Pseudo-Associative Caches

- **Pseudo-Associative or Column Associative**
  - **get miss rate of set associative and the hit speed of direct mapped**
- **Cache access proceeds just as in the direct mapped cache for a hit**
- **On a miss, before going to the memory, another cache entry is checked to see if it matches**
  - **E.g., invert the MSB of the index field to find the other block in the "Pseudo Set"**
- **Have one fast and one slow hit time**

Pseudo-associative Cache



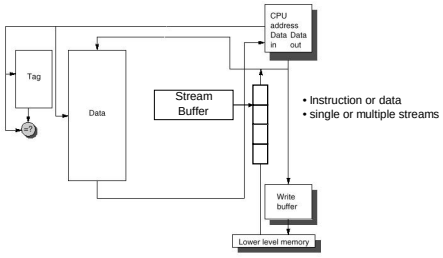
Two types of hits: regular (first access) and pseudo (second access) hits

Questions: 1. Methods for second access
2. Replacement policy

Fifth Approach Hardware Prefetching

- **Prefetch instructions and/or data before it is requested by the processor**
  - **Prefetched into the cache or an external buffer**
- **E.g. AXP 21064 fetches two (instruction) blocks on a miss**
  - **requested block,**
    - **» Placed in the instruction cache**
  - **next consecutive block**
    - **» Placed into the instruction stream buffer**
  - **If requested block is in stream buffer**
    - **» Get it from there and issue next prefetch request**

Hardware Prefetching



Hardware Prefetching

- Relies on utilizing unused memory bandwidth
- Could interfere with demand misses
  - Memory bus
  - Memory unit

Sixth Approach Compiler-Controlled Prefetching

- Compiler inserts prefetch instructions
- Several flavors of prefetch
  - Register Prefetch
    - » load the value in a register
  - Cache Prefetch
    - » load data only into the cache and not the register
  - Either can be faulting or non-faulting (nonbinding)
    - » Address does or does not cause virtual address fault and protection violation
- Normal load instruction
  - Faulting register prefetch instruction
- Non faulting instructions turn into no-op if they were to cause a fault

Prefetching

- Prefetch is made semantically invisible to a program
- Lockup-free (nonblocking) Cache
  - Processor can proceed while the prefetched data are being fetched
  - Cache continues to supply instructions and data while waiting for the prefetched data to return

Compiler-controlled Prefetching

```
/* Before */
for ( i = 0; i < 3; i = i + 1 )
  for ( j = 0; j < 100; j = j + 1 )
    a [ i ][ j ] = b [ j ][ 0 ] * b [ j + 1 ][ 0 ];

/* After */
for ( j = 0; j < 100; j = j + 1 ) {
  prefetch (b [ j + 7 ][ 0 ]);
  prefetch (a [ 0 ][ j + 7 ]);
  a [ 0 ][ j ] = b [ j ][ 0 ] * b [ j + 1 ][ 0 ];};
for ( i = 1; i < 3; i = i + 1 )
  for ( j = 0; j < 100; j = j + 1 ) {
    prefetch (a [ i ][ j + 7 ]);
    a [ i ][ j ] = b [ j ][ 0 ] * b [ j + 1 ][ 0 ];}
```

Seventh Approach Compiler Optimizations

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

Merging Arrays

```
/* Before */
int val[SIZE];
int key[SIZE];

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

- Reducing conflicts between val & key and improving spatial locality

Loop Interchange

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

Loop Fusion

```
/* 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];}
```

Blocking

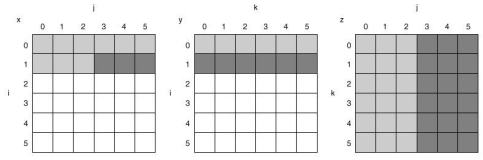
- Reduce misses via improved temporal locality
- Divide operations to blocks (submatrices)

```
/* 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[]**

Blocking

Before blocking



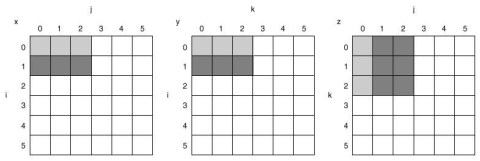
x[ ] y[ ] z[ ]

Blocking

```
/* 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;
     };
```

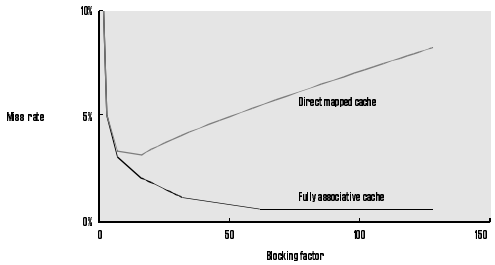
Blocking

After blocking



x[ ] y[ ] z[ ]

Blocking factor



Summary

$$CPUtime = IC \times \left(CPI_{Execution} + \frac{Memory\ \ accesses}{Instruction} \times \textbf{Miss rate} \times Miss\ \ penalty \right) \times Clock\ \ cycle\ \ time$$

- **Reducing Miss Rate**
  1. **Larger Block Size**
  2. **Higher Set Associativity**
  3. **Victim Cache**
  4. **Pseudo-Associative Cache**
  5. **Hardware Prefetching**
  6. **Compiler-controlled Prefetching**
  7. **Compiler Optimizations**

Improving Cache Performance

- **Average memory-access time**
  = Hit time + Miss rate x Miss penalty
- **Improve performance by:**
  1. **Reduce the miss rate,**
  2. **Reduce the miss penalty, or**
  3. **Reduce the time to hit in the cache.**

Reducing Cache Miss Penalty

1. **Giving priority to read misses over writes**
2. **Sub-block placement for reduced miss penalty**
3. **Early restart and critical work first**
4. **Nonblocking caches to reduce stalls on cache misses**
5. **Second-level caches**

First Technique: Giving priority to read misses over writes

- **Give priority to reads due to read misses over writes from the write buffer in accessing main memory**
- **Problem - example**

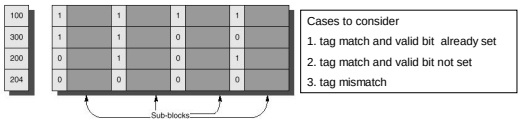
```
SW 512(R0), R3      ; M[512] <- R3 (cache index 0)
LW R1, 1024(R0)     ; R1 <- M[1024](cache index 0)
LW R2, 512(R0)      ; R2 <- M[512] (cache index 0)
```

Solution: (1) Wait until the write buffer becomes empty

*(2) Check the addresses of the words in the write buffer

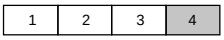
Second Technique: Sub-block placement for reduced miss penalty

• Tag is associated with block consisting of a number of sub-blocks, each of which has a valid bit - reduced tag storage & miss penalty



Third Technique: Early restart and critical word first

- **Early restart**



1 -> 2 -> 3 -> 4

Requested word: word 3

Processing performed background

- **Critical word first**

3 -> 4 -> 1 -> 2

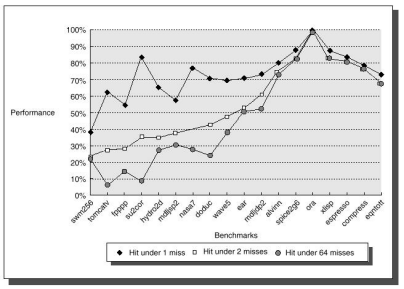
- **Large sized blocks reap most benefit**

Fourth Technique: Nonblocking caches to reduce stalls on cache misses

- **Nonblocking cache - does not block on a miss**
- **Possibility**
  - **Hit under miss (requires at least out-of-order completion capability)**
  - **Hit under multiple misses (requires in addition a memory system that can service multiple misses simultaneously)**

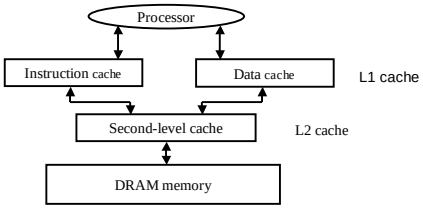
→Significantly increases complexity of cache controller

Nonblocking caches to reduce stalls on cache misses



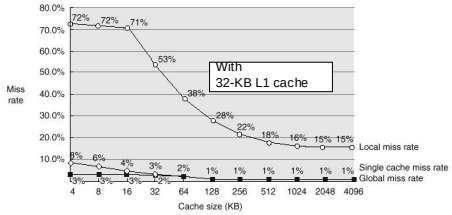
- 8-KB direct-mapped
- 32-bytes blocks
- 16-clock-cycle miss penalty

Fifth Technique: Second-level caches



- L1 (first-level) cache: Optimized for fast hit time
- L2 (second-level) cache: Optimized for high hit rate
- Important concern: Inclusion property

Second-level caches



Average memory access time = Hit time (L1) +
Miss rate (L1) x (Hit time (L2) +
Miss rate (L2) x Miss penalty (L2))
Local Miss Rate : # of misses in cache divided by total # of accesses to this cache
Global Miss Rate : # of misses divided by the total # of accesses generated by CPU

How to Design Second-level Caches

- Size
- Associativity
- Block size
- Inclusion property

Summary

- Techniques for reducing cache miss penalty
  - Giving priority to read misses over writes
  - Sub-block placement for reduced miss penalty
  - Early restart and critical work first
  - Nonblocking caches to reduce stalls on cache misses
  - Second-level caches

Improving Cache Performance

- Average memory-access time
  = Hit time + Miss rate x Miss penalty
- Improve performance by:
  1. Reduce the miss rate,
  2. Reduce the miss penalty, or
  3. Reduce the time to hit in the cache.

Reducing Hit Time

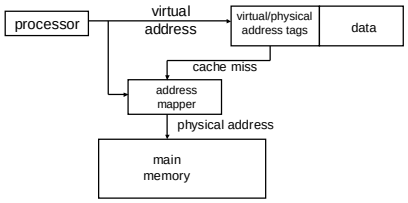
1. Small and Simple Caches
2. Avoiding Address Translation During Indexing of the Cache
   - Using virtual caches
   - Accessing physical caches without address translation
3. Pipelining Writes for Write Hits

First Technique: Small and Simple Caches

- Smaller is faster → smaller caches
- Simple is better → direct-mapped
- Alpha AXP 21064 has
  - Direct-mapped 8-KB (256 32-byte blocks) L1 instruction cache and date cache
  - Direct-mapped 128-KB to 8-MB L2 cache
- Becomes increasing important due to the pressure of a fast clock cycle

Second Technique: Avoid Address Translation

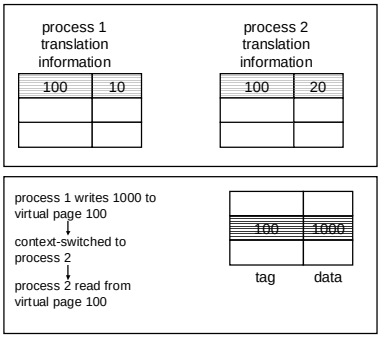
- Virtual cache vs Physical cache
- Cache is indexed and/or tagged with the virtual address (what is virtual address?)
- Cache access and MMU translation/validation done in parallel



Using virtual caches

- Problems with virtual caches
  - homonym problem
  - synonym problem

Homonym problem

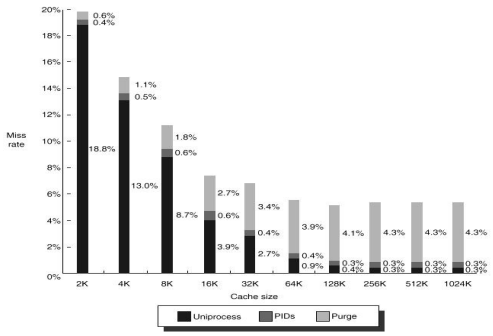


Homonym problem

Solutions to homonym problem

1. **Cache purging at each context switch**
2. **Using PID (process id) as an additional tag**

Homonym problem



Synonym problem

process 1 translation information

100	10
200	10

process 1 reads from virtual page 100

100	5

process 1 reads from virtual page 200

100	5

process 1 writes 10 to virtual page 100

100	5
200	5

process 1 reads from virtual page 200

100	10
200	5

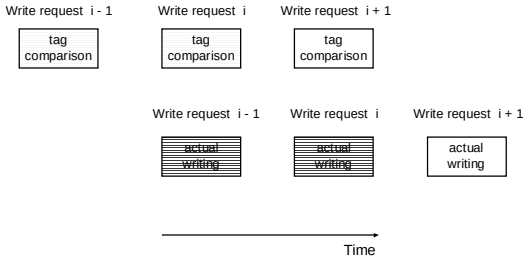
tag data

Synonym problem

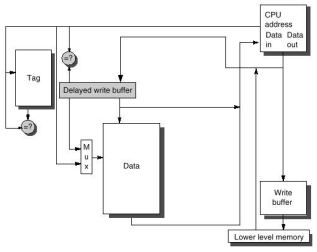
Solutions to synonym problem

1. **Hardware anti-aliasing**
2. **Alignment of synonyms (require all the synonyms to be identical in the lower bits of their virtual addresses assuming a direct-mapped cache)**

Third Technique: Pipelining Writes for Fast Write Hits



Pipelining Writes for Fast Write Hits



Summary of Cache Optimizations

Technique	Miss rate	Miss penalty	Hit time	Hardware complexity	Comment
Larger block size	+	–		0	Trivial; RS/6000 550 uses 128
Higher associativity	+		–	1	e.g., MIPS R10000 is 4-way
Victim caches	+			2	Similar technique in HP 7200
Pseudo-associative caches	+			2	Used in L2 of MIPS R10000
Hardware prefetching of instructions and data	+			2	Data are harder to prefetch; tried in a few machines; Alpha 21064
Compiler-controlled prefetching	+			3	Needs nonblocking cache too; several machines support it

Summary of Cache Optimizations

Technique	Miss rate	Miss penalty	Hit time	Hardware complexity	Comment
Compiler techniques to reduce cache misses	+			0	Software is challenge; some machines give compiler option
Giving priority to read misses over writes		+		1	Trivial for uniprocessor, and widely used
Subblock placement		+		1	Used primarily to reduce tags
Early restart and critical word first		+		2	Used in MIPS R10000, IBM 620
Nonblocking caches		+		3	Used in Alpha 21064, R10000
Second-level caches		+		2	Costly hardware; harder if block size L1 ≠ L2; widely used
Small and simple caches	–		+	0	Trivial; widely used
Avoiding address translation during indexing of the cache			+	2	Trivial if small cache; used in Alpha 21064
Pipelining writes for fast write hits			+	1	Used in Alpha 21064