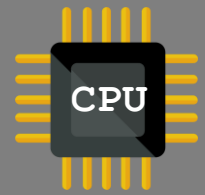


Topic 5

Memories II

Xinfei Guo
xinfei.guo@sjtu.edu.cn

June 15th, 2022

T5 learning goals

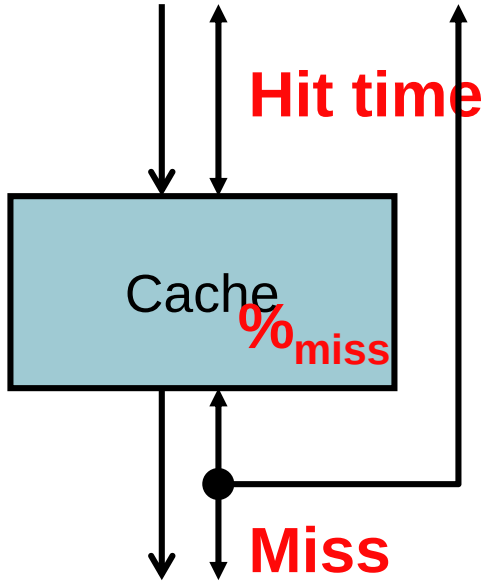
- Memories
  - Memories I: Memory Overview
  - Memories II: Cache (Advanced Cache Optimizations)
  - Memories III: Memory Management

A lot of new concepts and theories!!!

Review: Cache performance

- CPI_{ALUOps} does not include memory instructions
- AMAT = Average Memory Access Time

$$AMAT = HitTime + MissRate \times MissPenalty$$
$$= (HitTime_{Inst} + MissRate_{Inst} \times MissPenalty_{Inst}) +$$
$$(HitTime_{Data} + MissRate_{Data} \times MissPenalty_{Data})$$



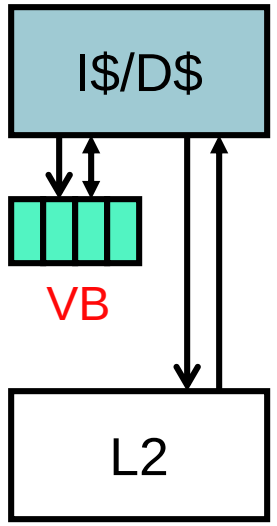
$$CPUtime = IC \times \left(\frac{AluOps}{Inst} \times CPI_{AluOps} + \frac{MemAccess}{Inst} \times AMAT \right) \times CycleTime$$

Review: Miss Rate ABC

- Associativity
  - Decrease conflict misses
  - Increase hit latency
- Block size
  - Increase conflict misses (fewer entries)
  - Decrease capacity misses (spatial locality)
  - Decrease Compulsory (cold) misses
- Capacity
  - Decrease capacity misses
  - Increase hit latency
- More …

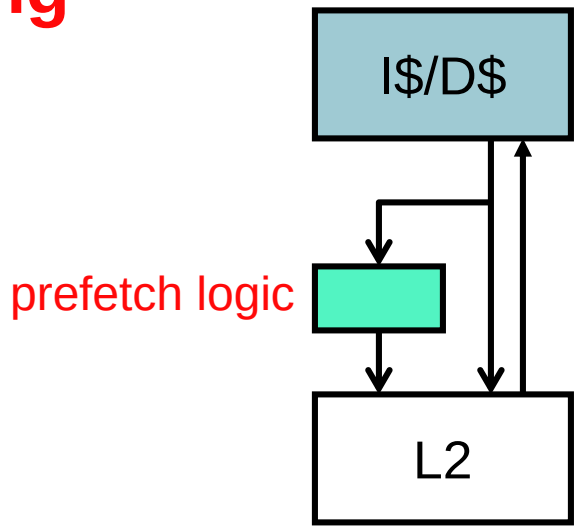
Reducing Conflict Misses: Victim Buffer

- Conflict misses: not enough associativity
  - High associativity is expensive, but also rarely needed
    - 3 blocks mapping to same 2-way set and accessed (XYZ)+
- **Victim buffer (VB)**: small fully-associative cache
  - Sits on I$/D$ miss path
  - Small (e.g., 8 entries) so very fast
  - Blocks kicked out of I$/D$ placed in VB
  - On miss, check VB: hit? Place block back in I$/D$
  - 8 extra ways, shared among all sets
    - Only a few sets will need it at any given time
  - Very effective in practice



Hardware Prefetching

- Bring data into cache proactively/**speculatively**
  - If successful, reduces number of caches misses
- Key: anticipate upcoming miss addresses accurately
  - Can do in software or hardware
- Simple hardware prefetching: **next block prefetching**
  - Miss on address **X** → anticipate miss on **X+block-size**
  - \+ Works for insns: sequential execution
  - \+ Works for data: arrays
- Table-driven hardware prefetching
  - Use **predictor** to detect strides, common patterns
- Effectiveness determined by:
  - **Timeliness**: initiate prefetches sufficiently in advance
  - **Coverage**: prefetch for as many misses as possible
  - **Accuracy**: don't pollute with unnecessary data



Hardware Prefetching

- Instruction prefetching
  - Typically, CPU fetches 2 blocks on miss: requested and next
  - Requested block goes in instruction cache, prefetched goes in instruction stream buffer
- Data prefetching
  - Prefetching invoked if 2 successive L2 cache misses to a page, if distance between those cache blocks is < 256 bytes
  - The Intel Core i7 supports hardware prefetching into both L1 and L2 with the most common case of prefetching being accessing the next line.
- Relies on utilizing memory bandwidth that otherwise would be unused
  - When prefetched data are not used or useful data are displaced, prefetching will have a very negative impact on power
  - If it interferes with demand misses it can actually lower performance.

Hardware Prefetching

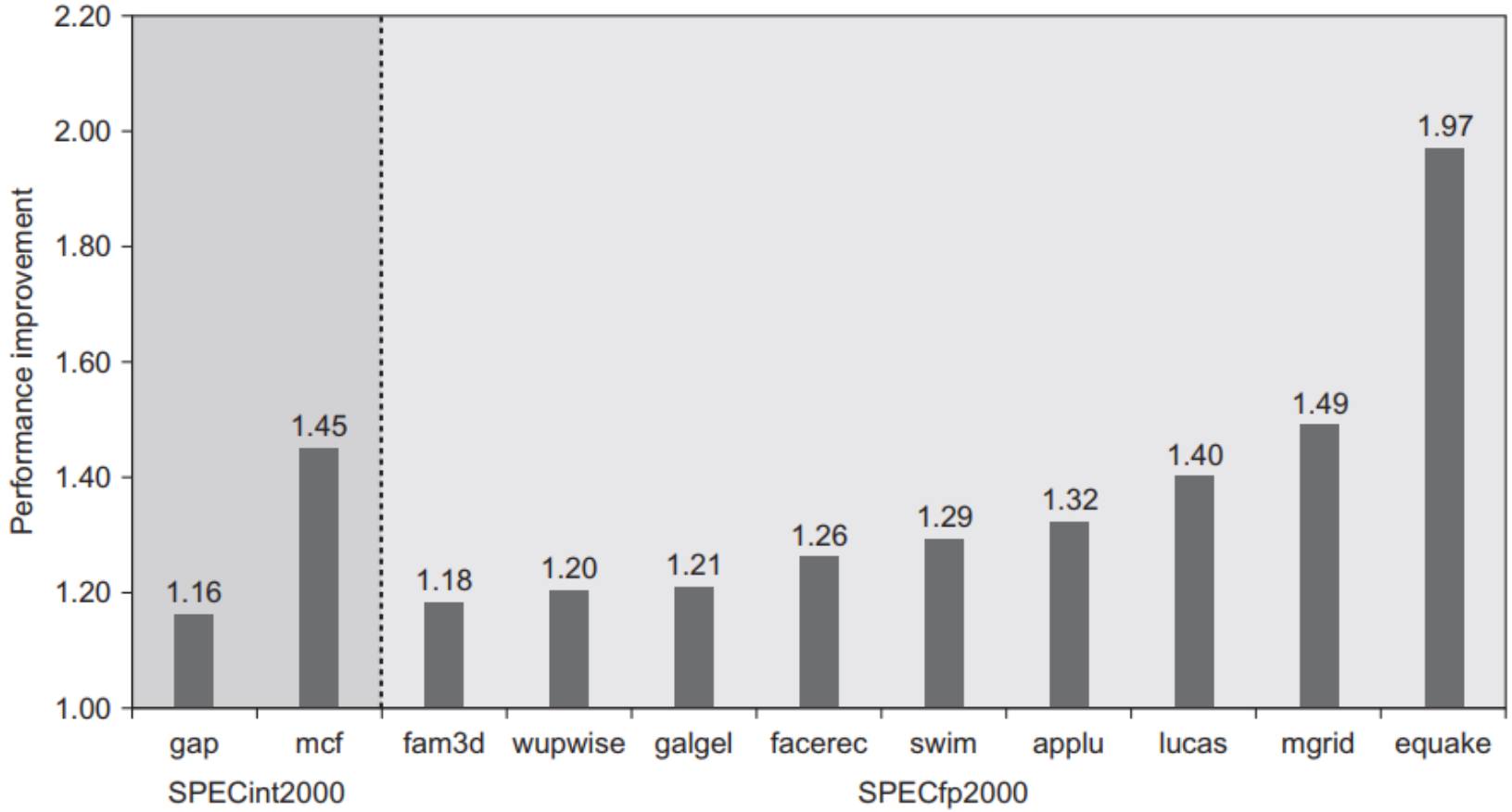


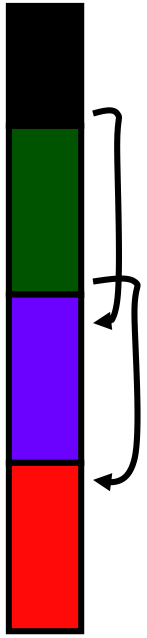
Figure 2.15 Speedup because of hardware prefetching on Intel Pentium 4 with hardware prefetching turned on for 2 of 12 SPECint2000 benchmarks and 9 of 14 SPECfp2000 benchmarks. Only the programs that benefit the most from prefetching are shown; prefetching speeds up the missing 15 SPECCPU benchmarks by less than 15% (Boggs et al., 2004).

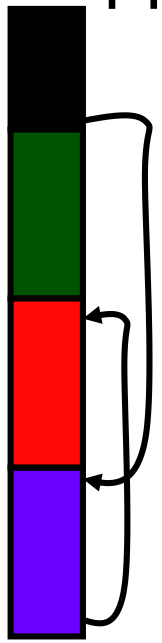
Compiler Optimizations to Reduce Miss Rate

- McFarling [1989] reduced misses by 75% in software on 8KB direct-mapped cache, 4 byte blocks
- Instructions
  - Reorder procedures in memory to reduce conflict misses
  - Profiling to look at conflicts (using tools they developed)
- Data
  - Loop interchange: Change nesting of loops to access data in memory order
  - Blocking: Improve temporal locality by accessing blocks of data repeatedly vs. going down whole columns or rows
  - 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 variable overlap

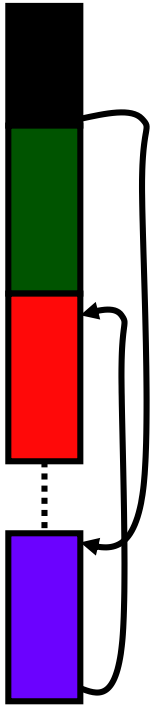
Software Restructuring: Code

- Compiler can lay out code for temporal and spatial locality
  - If (a) { **code1;** } else { **code2;** } **code3;**
  - But, **code2** case never happens (say, error condition)





+ Better locality
+ Fewer taken branches

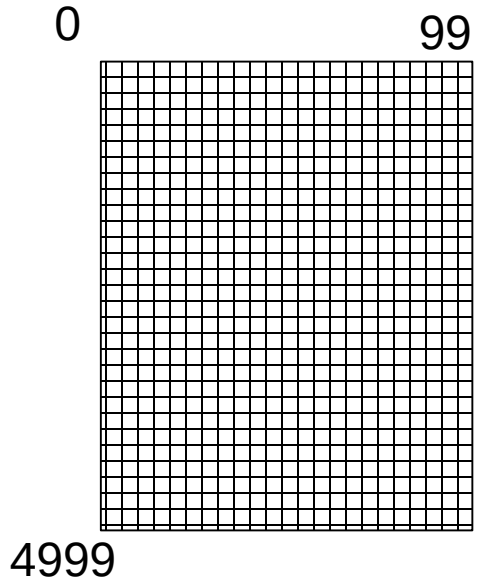


+ Better locality for code after **code3**
+ Fewer taken branches

Loop interchange

```
/* 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];
```

Sequence of access: X[0][0], X[1][0], X[2][0], …



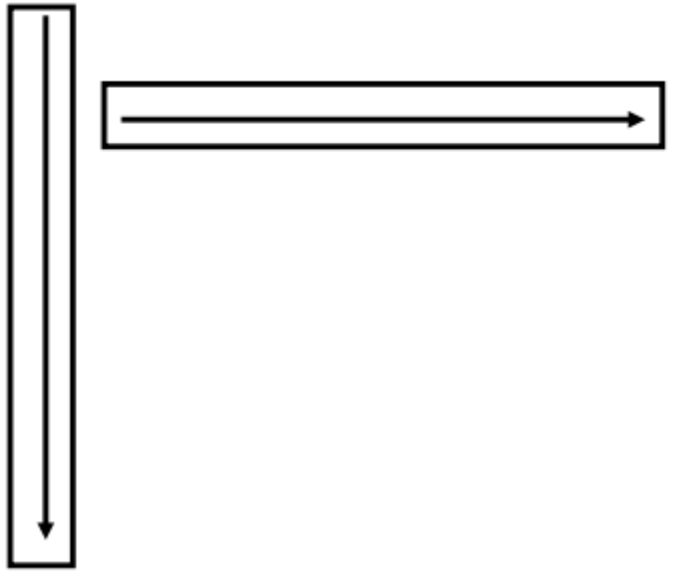
the array is laid out by rows

```
/* 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];
```

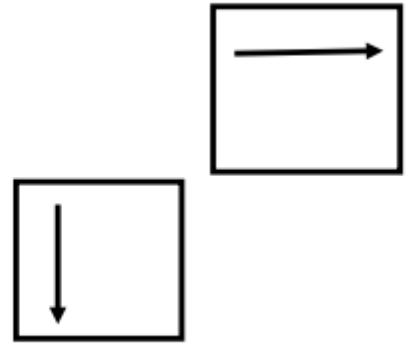
Sequence of access: X[0][0], X[0][1], X[1][2], …

Blocking

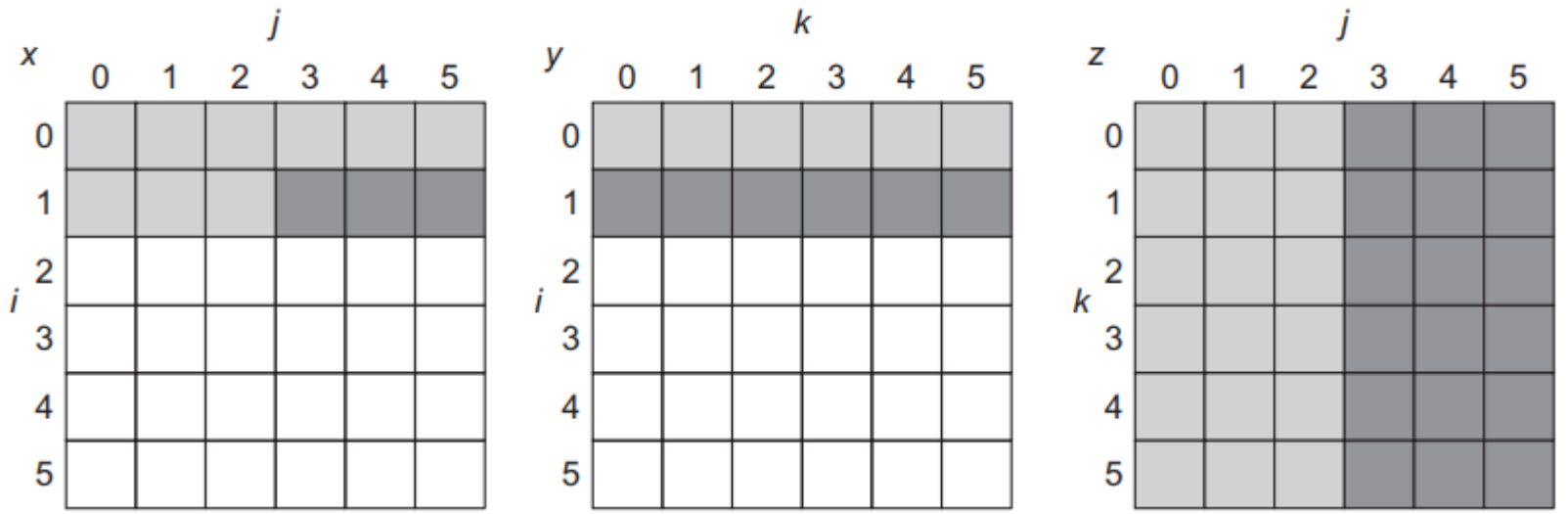
```
/* Before */
for (i = 0; i < N; i = i+1)
   for (j = 0; j < N; j = j+1)
      {r = 0;
       for (k = 0; k < N; k = k+1){
          r = r + y[i][k]*z[k][j];};
       x[i][j] = r;
      };
```



- 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:
  - $2N^3 + N^2$ => (assuming no conflict; otherwise …)
- Idea: compute on BxB submatrix that fits

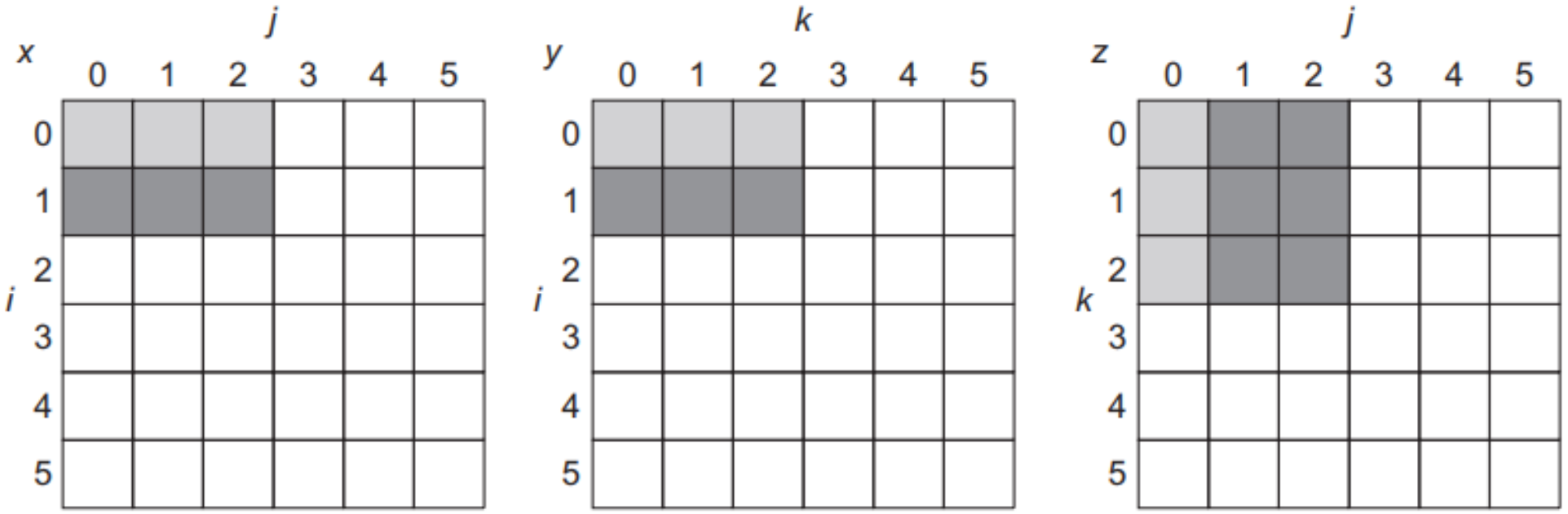


Blocking



A snapshot of the three arrays x, y, and z when N=6 and i=1. The age of accesses to the array elements is indicated by shade: white means not yet touched, light means older accesses, and dark means newer accesses. The elements of y and z are read repeatedly to calculate new elements of x. The variables i, j, and k are shown along the rows or columns used to access the arrays.

Array Access for Blocking/Tiling Transformation



The age of accesses to the arrays x, y, and z when B=3. Note that, in contrast to previous slide, a smaller number of elements is accessed.

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;
           };
```

- B called Blocking Factor
- Capacity misses from $2N^3 + N^2$ to $2N^3/B + N^2$

Merging/Splitting Arrays

- Reduce conflicts between val & key; improve spatial locality

```
/* 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];
```

Loop Fusion

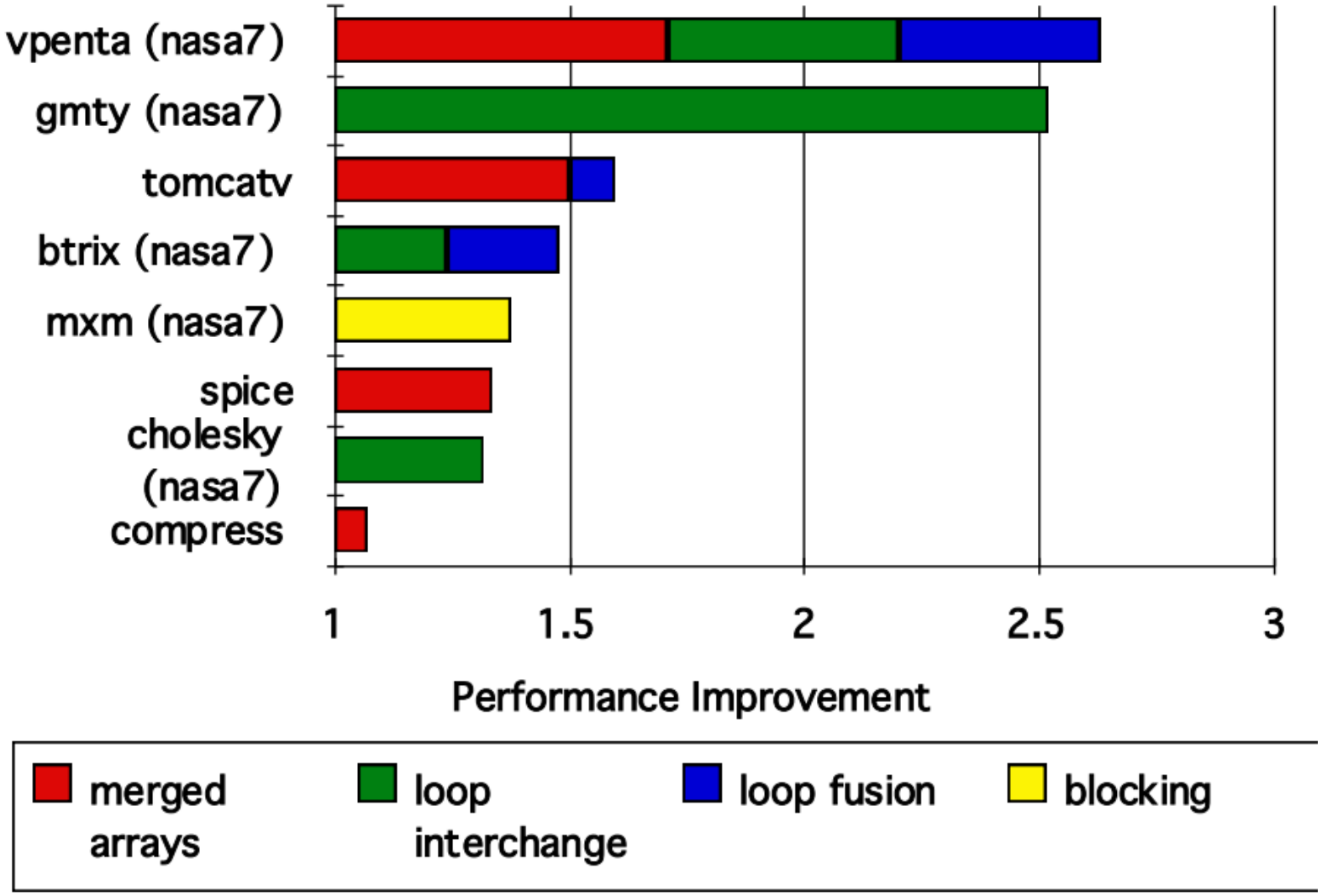
```
/* Before */
for (i = 0; i < N; i = i+1)
   for (j = 0; j < N; j = j+1)
    a[i][j] = 1/b[i][j] * c[i][j];

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];}
```

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

Summary of Compiler Optimizations to Reduce Cache Misses



source: https://passlab.github.io/CSCE513/

Software Prefetching

- Use a special “prefetch” instruction
  - Tells the hardware to bring in data
  - Just a hint
- Inserted by programmer or compiler
- Example

```
int tree_add(tree_t* t) {
  if (t == NULL) return 0;
  __builtin_prefetch(t->left);
  __builtin_prefetch(t->right);
  return t->val + tree_add(t->right) + tree_add(t->left);
}
```

- Multiple prefetches bring multiple blocks in parallel
  - More “Memory-level” parallelism (MLP)

Book Example (P112 -113)

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]);
        /* b(j,0) for 7 iterations later */
        prefetch(a[0][j+7]);
        /* a(0,j) for 7 iterations later */
        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) for +7 iterations */
                a[i][j] = b[j][0] * b[j+1][0];}
```

Prefetching in GCC Compiler

Built-in Function: void **__builtin_prefetch** (*const void *addr, ...*)

This function is used to minimize cache-miss latency by moving data into a cache before it is accessed. You can insert calls to `__builtin_prefetch` into code for which you know addresses of data in memory that is likely to be accessed soon. If the target supports them, data prefetch instructions are generated. If the prefetch is done early enough before the access then the data will be in the cache by the time it is accessed.

The value of *addr* is the address of the memory to prefetch. There are two optional arguments, *rw* and *locality*. The value of *rw* is a compile-time constant one or zero; one means that the prefetch is preparing for a write to the memory address and zero, the default, means that the prefetch is preparing for a read. The value *locality* must be a compile-time constant integer between zero and three. A value of zero means that the data has no temporal locality, so it need not be left in the cache after the access. A value of three means that the data has a high degree of temporal locality and should be left in all levels of cache possible. Values of one and two mean, respectively, a low or moderate degree of temporal locality. The default is three.

```
for (i = 0; i < n; i++)
  {
    a[i] = a[i] + b[i];
    __builtin_prefetch (&a[i+j], 1, 1);
    __builtin_prefetch (&b[i+j], 0, 1);
    /* ... */
  }
```

Data prefetch does not generate faults if *addr* is invalid, but the address expression itself must be valid. For example, a prefetch of `p->next` does not fault if `p->next` is not a valid address, but evaluation faults if `p` is not a valid address.

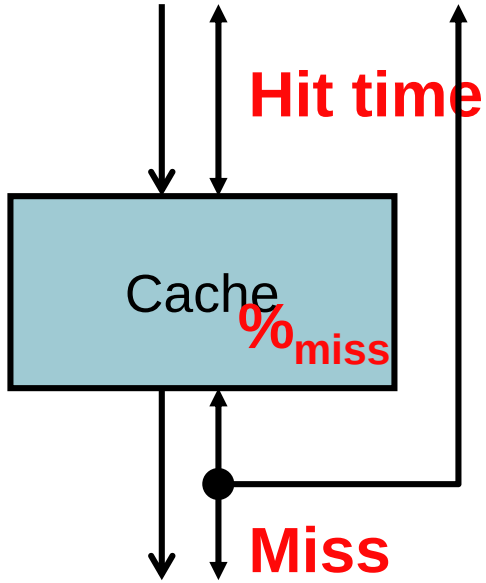
If the target does not support data prefetch, the address expression is evaluated if it includes side effects but no other code is generated and GCC does not issue a warning.

https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html

Review: Cache performance

- CPI_{ALUOps} does not include memory instructions
- AMAT = Average Memory Access Time

$$AMAT = HitTime + MissRate \times MissPenalty$$
$$= (HitTime_{Inst} + MissRate_{Inst} \times MissPenalty_{Inst}) +$$
$$(HitTime_{Data} + MissRate_{Data} \times MissPenalty_{Data})$$



$$CPUtime = IC \times \left(\frac{AluOps}{Inst} \times CPI_{AluOps} + \frac{MemAccess}{Inst} \times AMAT \right) \times CycleTime$$

Tag/Data Access

- Reads: read tag and data in parallel
  - Tag mis-match → data is wrong (OK, just stall until good data arrives)
- Writes: read tag, write data in parallel? No! Why not?
  - Tag mis-match → clobbered data (oops!)
  - For associative caches, which way was written into?
- Writes are a two step (multi-cycle) process
  - Step 1: match tag
  - Step 2: write to matching way
  - Bypass (with address check) to avoid load stalls
  - May introduce structural hazards

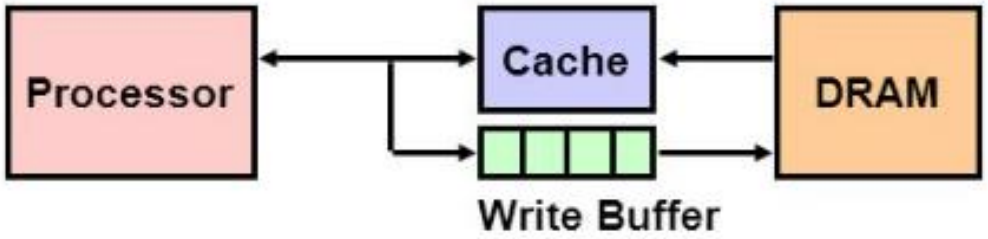
Write Propagation

- When to propagate new value to lower-level caches/memory?
- **Option #1: Write-through**: immediately
  - On hit, update cache
  - Immediately send the write to the next level
  - memory (or other processors) always have latest data
  - Simpler management of cache
- **Option #2: Write-back**: when block is replaced
  - Now we have multiple versions of the same block in various caches and in memory!
  - Requires additional "**dirty**" bit per block (updated with new values)
  - much lower bandwidth, since data often overwritten multiple times
  - Better tolerance to long-latency memory?

Write Allocate vs Non-Allocate

- What happens on write-miss?
- Write allocate: allocate new cache line in cache
  - Allocate cache block on miss by fetching corresponding memory block
  - Update cache block and then memory block
  - Commonly used (especially with write-back caches)
- Write non-allocate (or “write-around”)
  - Simply send write data through to underlying memory/cache - don’t allocate new cache line!
  - potential more read miss

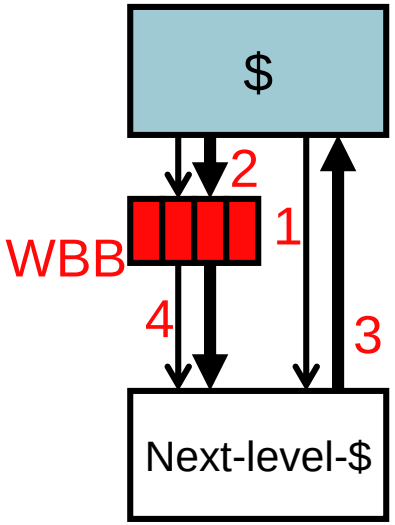
Write Buffer for Write-Through



- Solution to time consuming write through technique (for both hit and miss)
  - CPU: write date into buffer and write buffer
  - Memory controller: write contents of the buffer to memory
  - CPU proceeds to next step, while letting buffer to complete write through
  - Write buffer is just a FIFO (Typical 4 entries)
  - Frees buffer when completing write to memory
  - CPU stalls if buffer is full

Optimizing Writebacks

- **Writeback-buffer (WBB)**:
  - Hide latency of writeback (keep them off critical path)
  - Step#1: Send “fill” request to next-level
  - Step#2: While waiting, write dirty block to buffer
  - Step#3: When new blocks arrives, put it into cache
  - Step#4: Write buffer contents to next-level

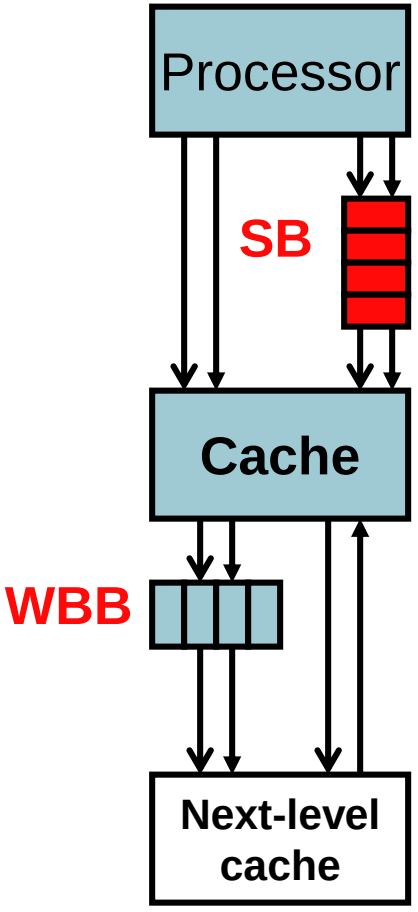


Write Misses and Store Buffers

- Read miss?
  - Load can't go on without the data, it must stall
- Write miss?
  - Technically, no instruction is waiting for data, why stall?

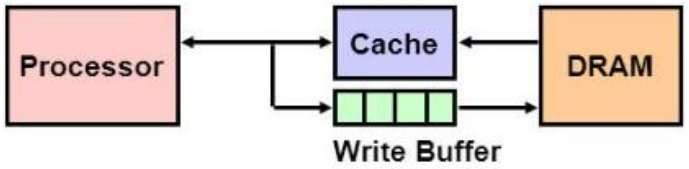
- **Store buffer**: a small buffer
  - Stores put address/value to store buffer, **keep going**
  - Store buffer writes stores to D$ in the background
  - Loads must search store buffer (in addition to D$)
  - \+ Eliminates stalls on write misses (mostly)
  - – Creates some problems for multicore (later)

- Store buffer vs. writeback-buffer
  - Store buffer: "in front" of D$, for hiding store misses
  - Writeback buffer: "behind" D$, for hiding writebacks



Reducing Miss Penalty: Giving Priority to Read Misses over Writes

```
SW R3, 512(R0) ; 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)
```



- Problem: write through with write buffers offer RAW conflicts with main memory reads on cache misses. (value in R2 is not equal to R3)
  - If simply wait for write buffer to empty, might increase read miss penalty
  - Check write buffer contents before read; if no conflicts, let the memory access continue

- Write-back also want buffer to hold misplaced blocks
  - Read miss replacing dirty block
  - Normal: Write dirty block to memory, and then do the read
  - Instead copy the dirty block to a write buffer, then do the read, and then do the write
  - CPU stall less since restarts as soon as do read

Reducing Miss Penalty: Early Restart and Critical Word First

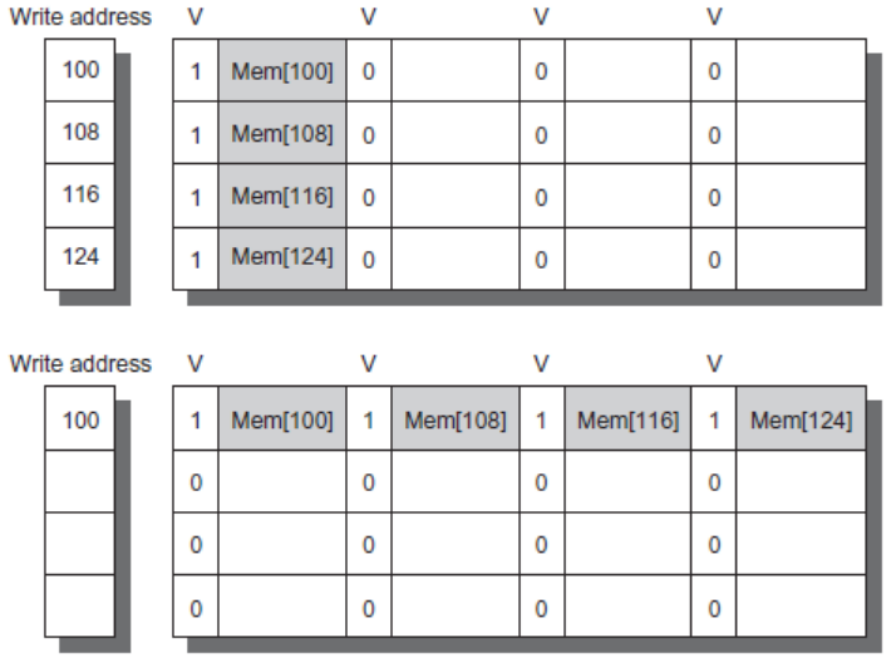
- Don’t wait for full block to be loaded before restarting CPU
  - *Early restart*—As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution
  - *Critical Word First*—Request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block. Also called *wrapped fetch* and *requested word first*
- Generally useful only in large blocks,
- Spatial locality a problem; tend to want next sequential word, so not clear if benefit by early restart

block

Reducing Miss Penalty: Merging Write Buffer to Reduce Miss Penalty

- When storing to a block that is already pending in the write buffer, update write buffer
- If buffer contains modified blocks, addresses can be checked to see if new data matches that of some write buffer entry
- For sequential writes in write-through caches, increases block size of write (more efficient)
- Reduces stalls due to full write buffer
- Assume we had four entries in the write buffer, and each entry could hold four 64-bit words.

Write buffer



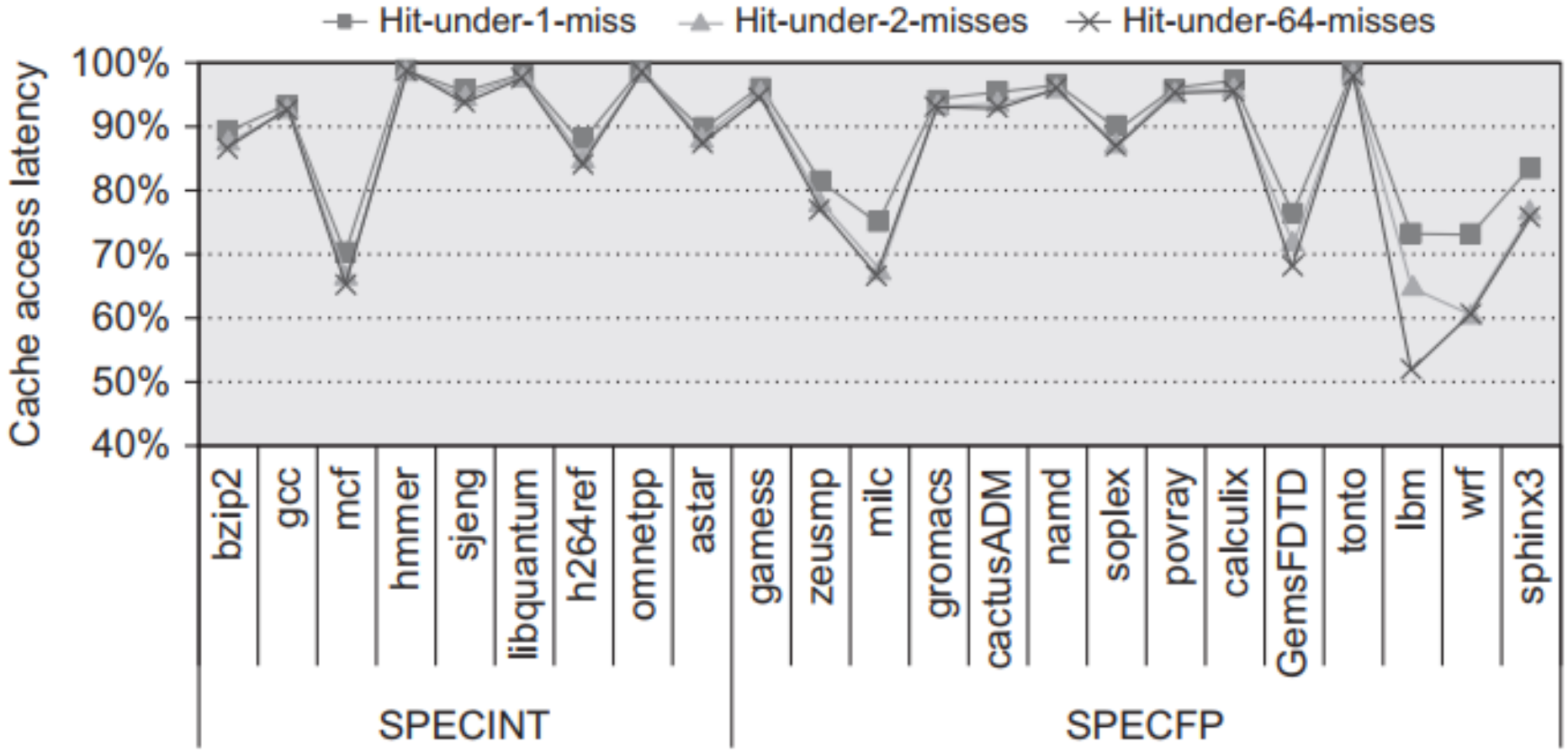
No write merging

Write merging

Reduce Miss Penalty: Non-blocking Caches to reduce stalls on misses

- *Non-blocking cache* or *lockup-free cache* allow data cache to continue to supply cache hits during a miss
  - requires F/E bits on registers or out-of-order execution
  - requires multi-bank memories
- "*hit under miss*" reduces the effective miss penalty by working during miss vs. ignoring CPU requests
- "*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 as there can be multiple outstanding memory accesses
  - Requires *multiple memory banks* (otherwise cannot support)
  - Penium Pro allows 4 outstanding memory misses

The effectiveness of a nonblocking cache

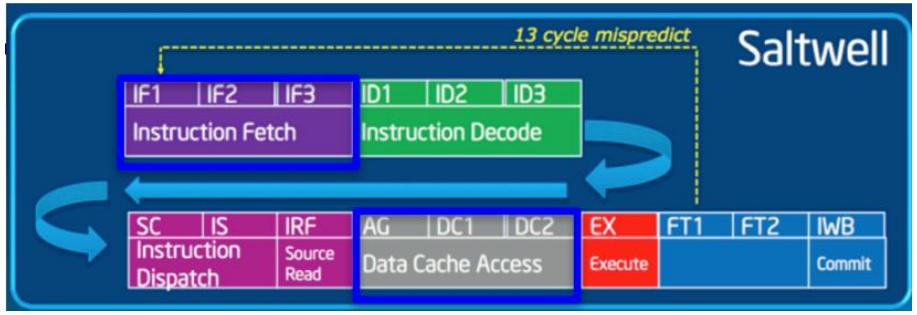


Performance Evaluation of Non-Blocking Caches

- A cache miss does not necessarily stall the processor
  - difficult to judge the impact of any single miss and hence to calculate the average memory access time.
- The effective miss penalty is the non-overlapped time that the processor is stalled.
  - not the sum of the misses
- The benefit of nonblocking caches is complex, depends on
  - the miss penalty when there are multiple misses
  - the memory reference pattern
  - how many instructions the processor can execute with a miss outstanding

Reduce Miss Penalty: Pipelined Access and Multibanked Caches

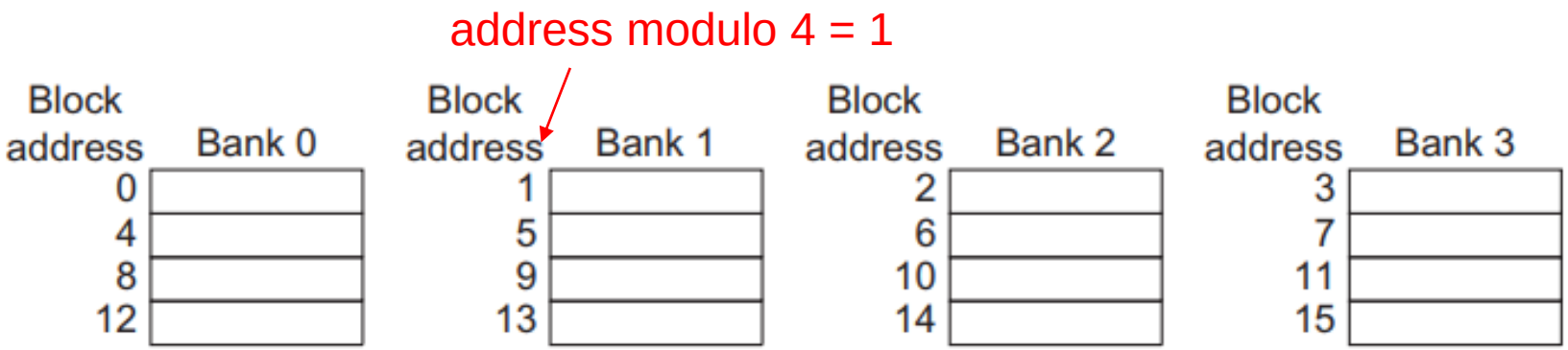
- Pipeline cache access to improve bandwidth
  - Examples:
    - Pentium: 1 cycle
    - Pentium Pro – Pentium III: 2 cycles
    - Pentium 4 – Core i7: 4 cycles
- Advantage:
  - fast cycle time and slow hit
  - Makes it easier to increase associativity



- Drawback: Increasing the number of pipeline stages leads to
  - Increases branch mis-prediction penalty
  - more clock cycles between the issue of the load and the use of the data
  - It is easier to pipeline the instruction cache than the data cache because the processor can rely on high performance branch prediction to limit the latency effects.
- Note that it increases the bandwidth of instructions rather than decreasing the actual latency of a cache hit

Reduce Miss Penalty: Pipelined Access and Multibanked Caches

- To handle multiple data cache accesses per clock, we can divide the cache into independent banks, each supporting an independent access.
- Works best when accesses naturally spread across banks
  - mapping of addresses to banks affects behavior of memory system
- sequential interleaving:
  - Spread block addresses sequentially across banks
  - E,g, bank i has all blocks with address i modulo n

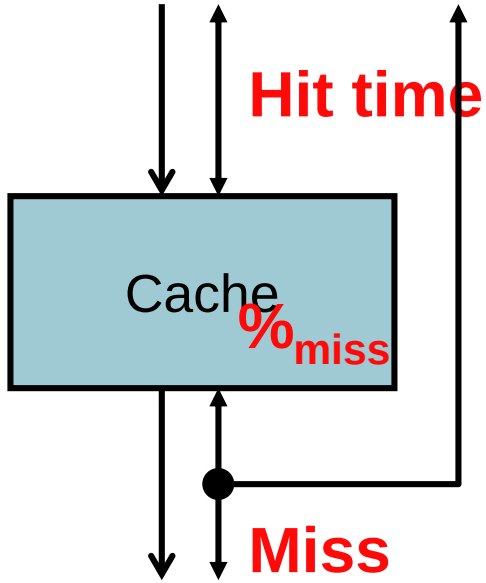


Four-way interleaved cache banks using block addressing. Assuming 64 bytes per block, each of these addresses would be multiplied by 64 to get byte addressing.

Review: Cache performance

- CPI_{ALUOps} does not include memory instructions
- AMAT = Average Memory Access Time

$$AMAT = HitTime + MissRate \times MissPenalty$$
$$= (HitTime_{Inst} + MissRate_{Inst} \times MissPenalty_{Inst}) +$$
$$(HitTime_{Data} + MissRate_{Data} \times MissPenalty_{Data})$$

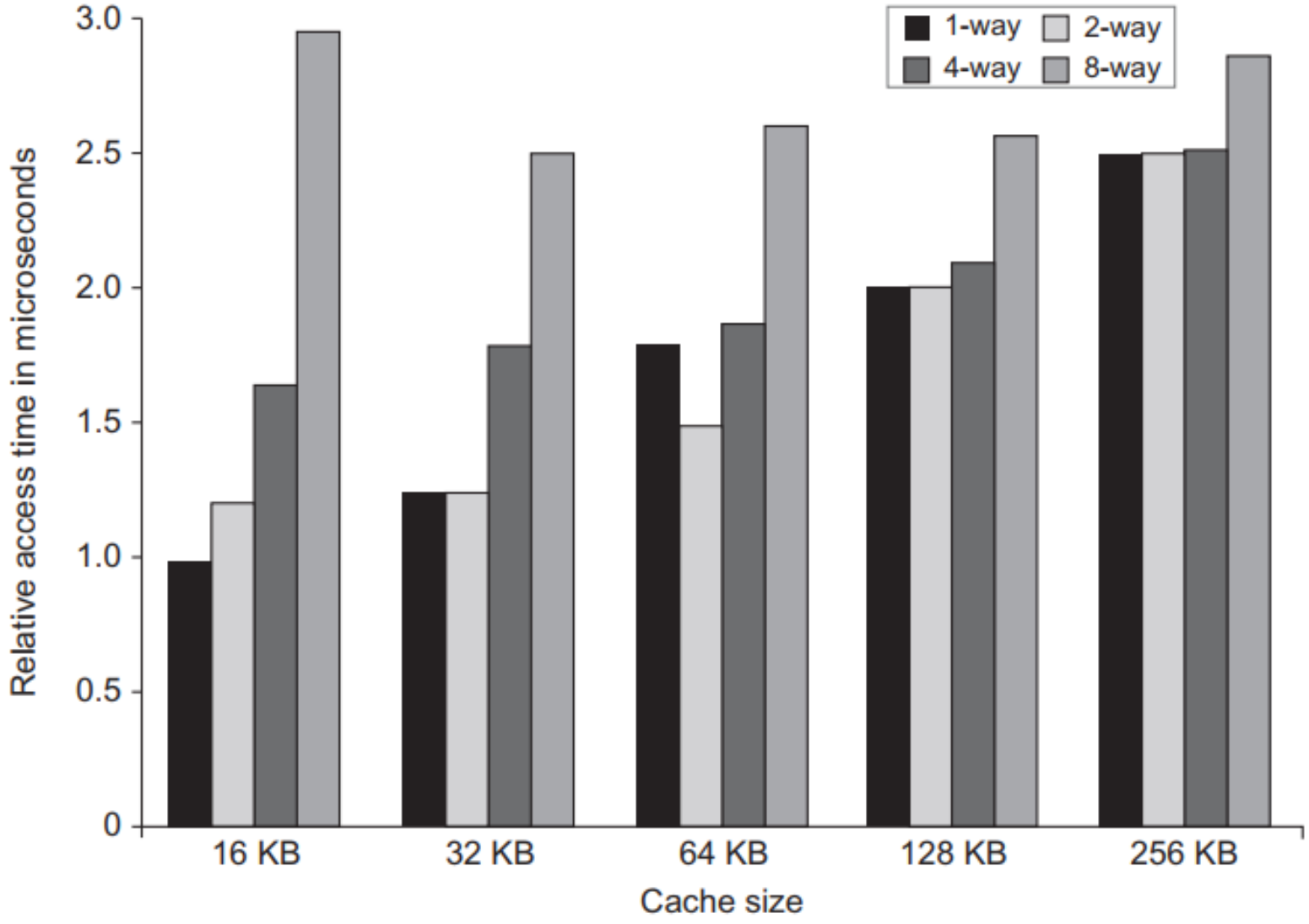


$$CPUtime = IC \times \left(\frac{AluOps}{Inst} \times CPI_{AluOps} + \frac{MemAccess}{Inst} \times AMAT \right) \times CycleTime$$

Reduce Hit Time: Small and Simple First-Level Caches

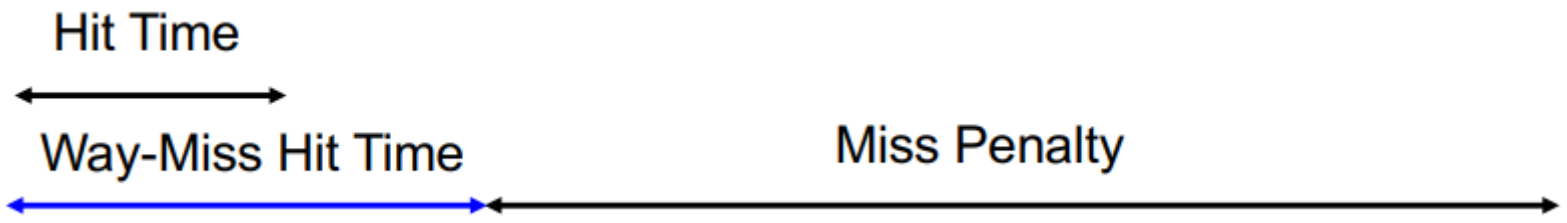
- Cache-hit critical path, three steps:
  - addressing the tag memory using the index portion of the address
  - comparing the read tag value to the address, and
  - setting the multiplexor to choose the correct data item if the cache is set associative.
- Guideline: smaller hardware is faster, Small data cache and thus fast clock rate
  - size of the L1 caches has recently increased either slightly or not at all.
  - Also L2 cache small enough to fit on chip with processor ⇒ avoids time penalty of going off chip
- Guideline: simpler hardware is faster
  - Direct-mapped caches can overlap the tag check with the transmission of the data, effectively reducing hit time.
  - Lower levels of associativity will usually reduce power because fewer cache lines must be accessed.
- General design: small and simple cache for 1st-level cache
  - One emphasis is on fast clock time while hiding L1 misses with dynamic execution and using L2 caches to avoid going to memory

Cache Size vs. Hit Time



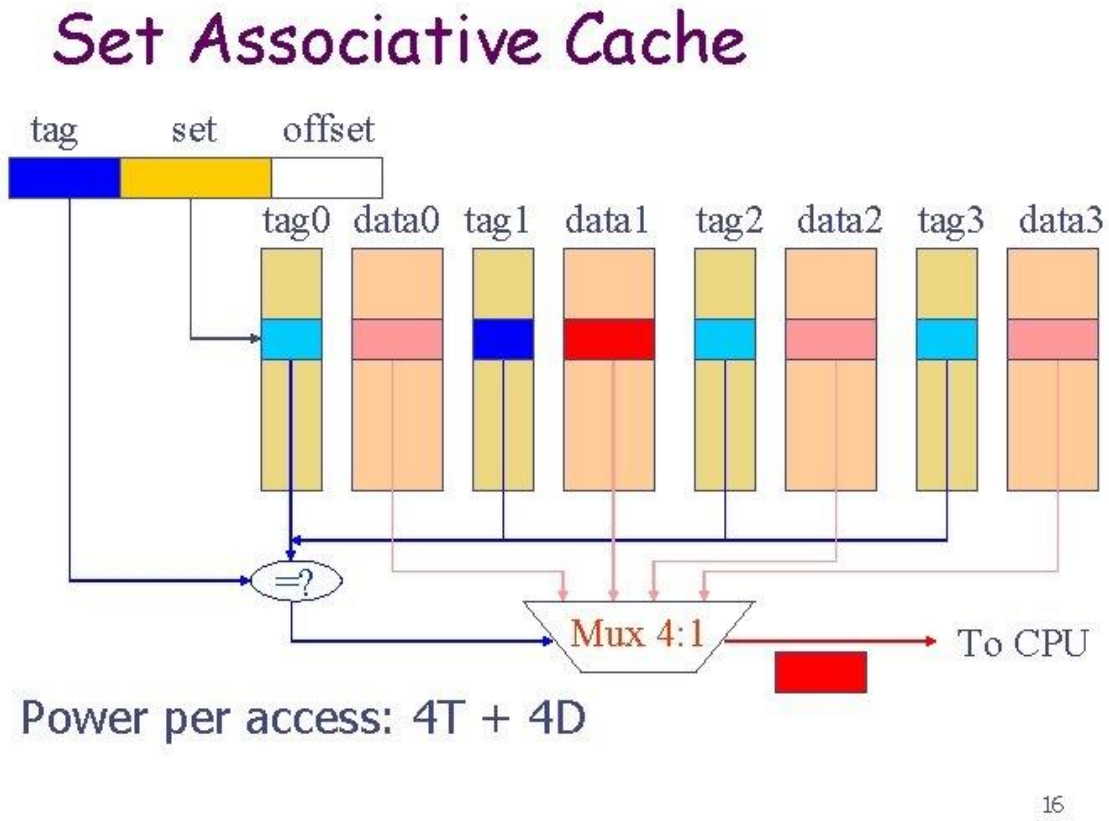
Reduce Hit Time: Via Way Prediction

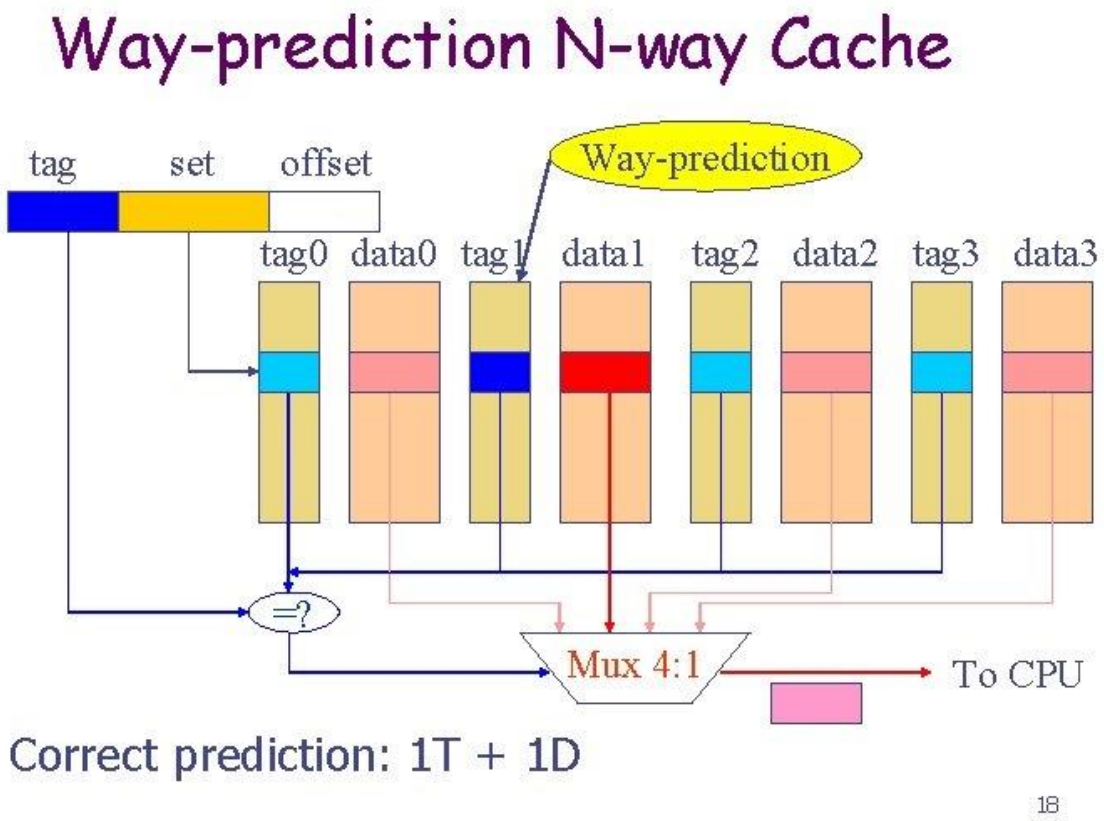
- How to combine fast hit time of direct-mapped with lower conflict misses of 2-way set associative cache?
- Way prediction: keep extra bits in cache to predict “ way ” (block within set) of next access
  - Multiplexer set early to select desired block; only 1 tag comparison done that cycle (in parallel with reading data)
  - Miss -> check other blocks for matches in next cycle, one cycle overhead



  - Can be extended to do way selection for saving power
  - Drawback: CPU pipeline harder if hit time is variable-length

Power Savings

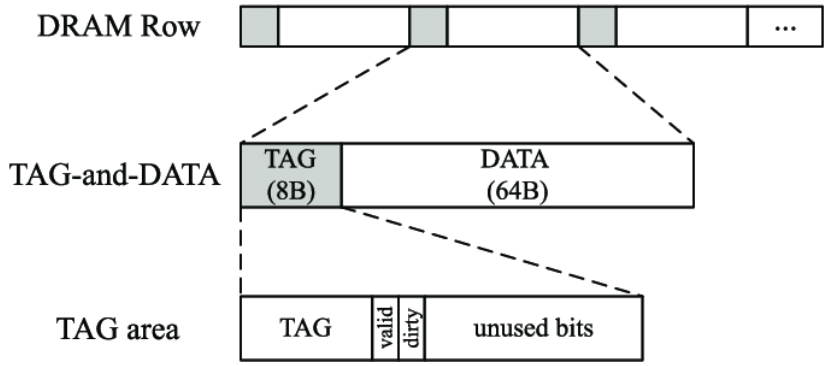




source: Z. Zhu, HPCA'01 Low power tutorial

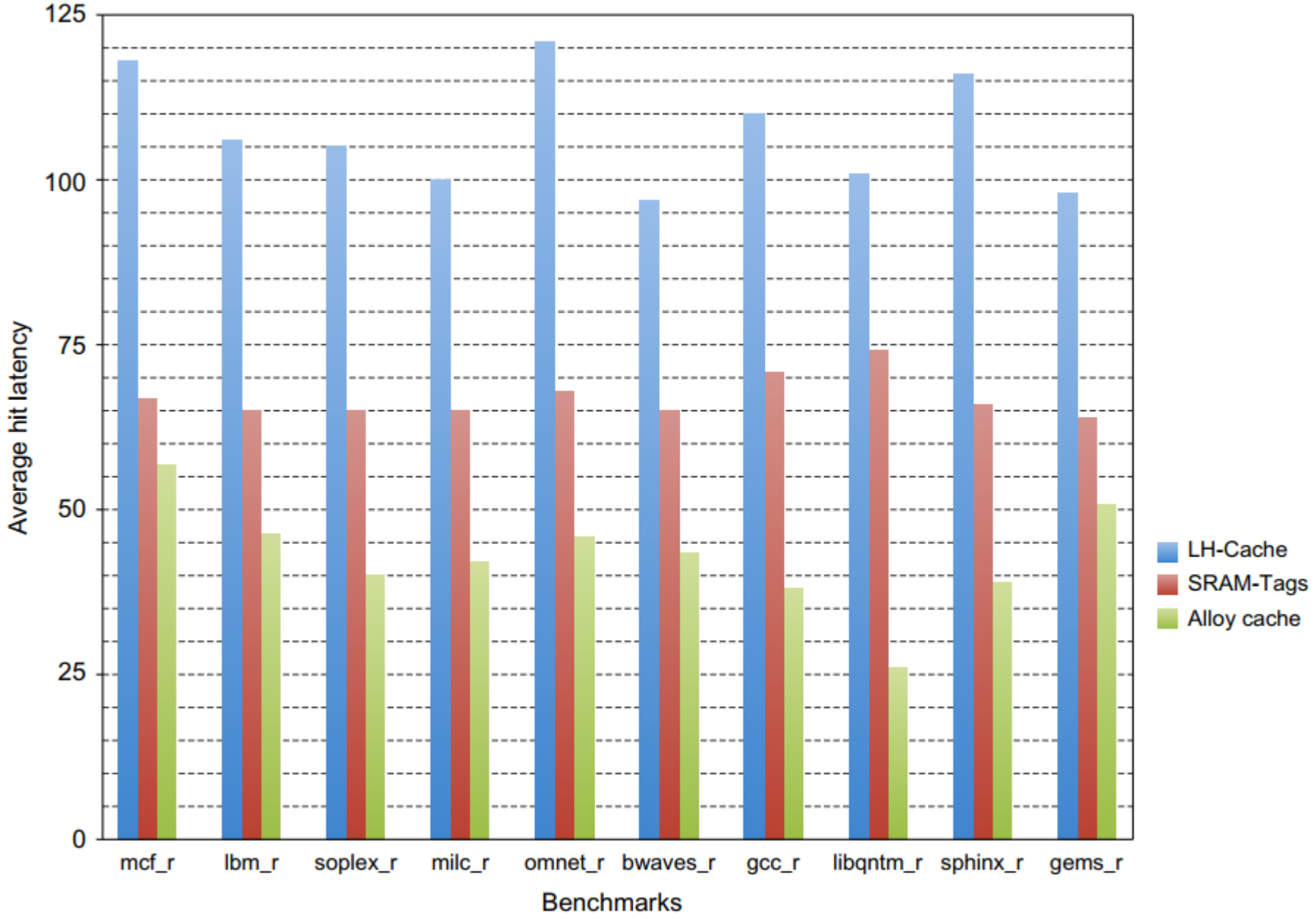
Reduce Hit Time: Increase Memory Bandwidth
Using HBM to Extend the Memory Hierarchy

- Inpackage DRAMs or EDRAM can be used to build massive L4 caches
  - 128 MiB to 1 GiB (more than current L3)
  - Smaller blocks require substantial tag storage
  - Larger blocks are potentially inefficient (more misses)



- One approach (Loh and Hill: L-H):
  - place the tags and the data in the same row in the HBM
  - Each SDRAM row is a block index
  - Each row contains set of tags and 29 data segments
  - 29-set associative
  - Hit requires a CAS: is the number of cycles required to read out a column in a row.

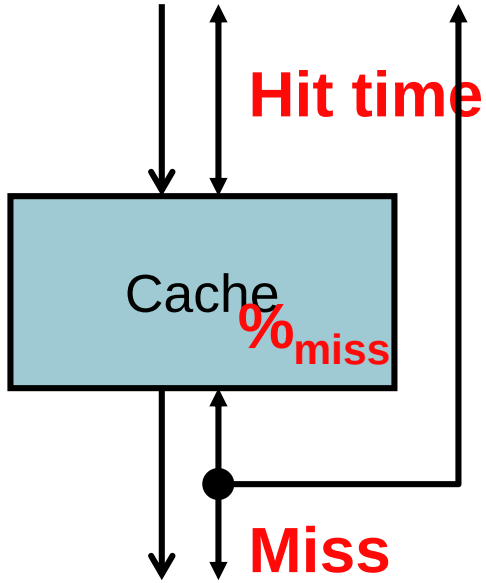
- Alternative approach (Alloy cache)
  - Mold tag and data together
  - Use direct mapped


125
100
75
50
25
0
Average hit latency
mcf_r
lbm_r
soplex_r
milc_r
omnet_r
bwaves_r
gcc_r
libqntm_r
sphinx_r
gems_r
Benchmarks
LH-Cache
SRAM-Tags
Alloy cache

Review: Cache performance

- CPI_{ALUOps} does not include memory instructions
- AMAT = Average Memory Access Time

$$AMAT = HitTime + MissRate \times MissPenalty$$
$$= (HitTime_{Inst} + MissRate_{Inst} \times MissPenalty_{Inst}) +$$
$$(HitTime_{Data} + MissRate_{Data} \times MissPenalty_{Data})$$



$$CPUtime = IC \times \left(\frac{AluOps}{Inst} \times CPI_{AluOps} + \frac{MemAccess}{Inst} \times AMAT \right) \times CycleTime$$

Summary (Basic Cache Optimization)

Technique	Hit time	Miss penalty	Miss rate	Hardware complexity	Comment
Larger block size		–	+	0	Trivial; Pentium 4 L2 uses 128 bytes
Larger cache size	–		+	1	Widely used, especially for L2 caches
Higher associativity	–		+	1	Widely used
Multilevel caches		+		2	Costly hardware; harder if L1 block size ≠ L2 block size; widely used
Read priority over writes		+		1	Widely used
Avoiding address translation during cache indexing	+			1	Widely used

Figure B.18 Summary of basic cache optimizations showing impact on cache performance and complexity for the techniques in this appendix. Generally a technique helps only one factor. + means that the technique improves the factor, – means it hurts that factor, and blank means it has no impact. The complexity measure is subjective, with 0 being the easiest and 3 being a challenge.

Summary (Advanced Cache Optimization)

Technique	Hit time	Band-width	Miss penalty	Miss rate	Power consumption	Hardware cost/ complexity	Comment
Small and simple caches	+			–	+	0	Trivial; widely used
Way-predicting caches	+				+	1	Used in Pentium 4
Pipelined & banked caches	–	+				1	Widely used
Nonblocking caches		+	+			3	Widely used
Critical word first and early restart			+			2	Widely used
Merging write buffer			+			1	Widely used with write through
Compiler techniques to reduce cache misses				+		0	Software is a challenge, but many compilers handle common linear algebra calculations
Hardware prefetching of instructions and data			+	+	–	2 instr., 3 data	Most provide prefetch instructions; modern high-end processors also automatically prefetch in hardware
Compiler-controlled prefetching			+	+		3	Needs nonblocking cache; possible instruction overhead; in many CPUs
HBM as additional level of cache		+/–	–	+	+	3	Depends on new packaging technology. Effects depend heavily on hit rate improvements

Review of Cache Optimizations

- 3 C's of cache misses
  - Compulsory, Capacity, Conflict
- Write policies
  - Write back, write-through, write-allocate, no write allocate
- Multi-level cache hierarchies reduce miss penalty
  - L1 – L4 (HBM, eDRAM)
  - Can change design tradeoffs of L1 cache if known to have L2
- Prefetching: retrieve memory data before CPU request
  - Hardware vs. Software
  - Prefetching can waste bandwidth and cause cache pollution
- Software memory hierarchy optimizations
  - Loop interchange, loop fusion, cache tiling

Design Guideline

- Cache block size: 32 or 64 bytes
  - Fixed size across cache levels
- Cache sizes (per core):
  - L1: Small and fastest for low hit time, 2K to 62K each for D$ and I$ separated
  - L2: Large and faster for low miss rate, 256K – 512K for combined D$ and I$ combined
  - L3: Large and fast for low miss rate: 1MB – 8MB for combined D$ and I$ combined
- Associativity
  - L1: directed, 2/4 way
  - L2: 4/8 way
- Banked, pipelined and no-blocking access

For Software Development

- Explicit or compiler-controlled prefetching
  - Insert prefetching call.
- Explicit or compiler-assisted code optimization for cache performance
  - Loop transformation (interchange, blocking, etc)

Where are we Heading?

- T5: Memories III

Acknowledgement

Slides in this topic are inspired in part by material developed and copyright by:

- ARM Courseware
- Prof. Yonghong Yan @ University of South Carolina
- Prof. Joe Devietti @ Upenn, CIS 571
- Prof. Ron Dreslinski @ UMich, EECS 470
- Prof. Hakim Weatherspoon @ Cornell, CS 3410
- Prof. Krste Asanovic @ UCB, CS252
- Xinfei Guo @ JI, VE370 2021 SU

Action Items

- HW #3 is out
- Lab #4 is out
- Reading Materials
  - Ch. 2.1 – 2.3
  - Ch. Appendix B.1-B.3