CSCE 614 Computer Architecture
Final Review Sheet

N/A

May 3, 2018

1 Chapter 3

1.1 Hardware Speculation

Main idea:

1. Out-of-order issue, in-order **commit** (important!!)
2. Store won't access the main memory until the commit phase
3. Consecutive instructions no longer needs to wait for branch results, just go ahead and execute after issued
4. Can enter the execution phase before the actual execution begins (output result later)
5. Use reorder buffer. If wrong: rollback

Example 1. Assume all operations have 1 cycle latency. Two-issue processor, 2 CDB, branches single issue, at most 2 could issue on the same time, 1 functional unit for memory access.

Dependency graph:

$$\begin{tikzcd}
& \text{LD} \arrow[d, dash] & & & DADDUIR_1 \\
& \text{DADDUI } R_2 \arrow[dl, dash] \arrow[dr, dash] & & & \\
\text{SD} & & \text{BNE} & &
\end{tikzcd}$$

Time table: see below.

2 Chapter 2

2.1 Memory Hierarchy: Terminology

■ Hit: data appears in some block in the upper level (example: Block X)

	Issue	Execution	Memory Access	Write CDB	Commit	Note
LD $R_2, 0(R_1)$	1	2	3	4	5	
DADDUI $R_2, R_2, \#1$	1	5		6	7	Waiting on LD
SD $R_2, 0(R_1)$	2	3			7	Waiting on DADDUI
DADDUI $R_1, R_1, \#8$	2	3		4	8	
BNE R_2, R_3, loop	3	7			8	Waiting on DADDUI
LD $R_2, 0(R_1)$	4	5	6	7	9	No wait on BNE
DADDUI $R_2, R_2, \#1$	4	8		9	10	Waiting on LD
SD $R_2, 0(R_1)$	5	6			10	Waiting on DADDUI
DADDUI $R_1, R_1, \#8$	5	6		7	11	Commit wait to be in order
BNE R_2, R_3, loop	6	10			11	Waiting on DADDUI

Table 1: Timetable for Example 1.

★ Hit Rate: the fraction of memory access found in the upper level

★ Hit Time: Time to access the upper level which consists of RAM access time + Time to determine hit/miss

■ Miss: data needs to be retrieve from a block in the lower level (Block Y)

★ Miss Rate = 1 - (Hit Rate)

★ Miss Penalty: Time to replace a block in the upper level + Time to deliver the block the processor

♡ See HW5 problem 3 part a solution on how to calculate miss penalty.

■ Ideally, hit time should be significantly less than miss penalty. Otherwise, there will be no reason to build a memory hierarchy.

■ The Principle of Locality: Program access a relatively small portion of the address space at any instant of time.

★ Temporal locality: locality in time

★ Spatial locality: locality in space

2.2 Cache Measures

■ Hit rate: fraction found in that level

★ So high that usually talk about Miss rate

★ Miss rate fallacy: as MIPS to CPU performance, miss rate to average memory access time in memory

■ **Average memory-access time = Hit time + Miss rate × Miss penalty** (ns or clocks)

■ Miss penalty: time to replace a block from lower level, including time to replace in CPU

★ access time: time to lower level = f(latency to lower level)

★ transfer time: time to transfer block = f(BW between upper & lower levels)

■ Cache: SRAM, main memory: DRAM

2.3 Memory Hierarchy: 4 Questions

2.3.1 Block Placement

Type: fully associative (associativity = # of blocks in a cache, 1 set), directly mapped (1-way associativity, 1 block per set), n-way set associative

★ Fully associative: memory blocks can be placed anywhere within the cache, most disorganized

★ Directly mapped: one memory block has a designated place to go. e.g. Block 12 placed in a 8-block directly mapped cache, it would be placed under (12 mod 8) = set 4. (No replacement policy is needed)

★ n-way set associative: one memory block can be placed randomly within a designated set. e.g. Block 12 placed in a 8-block 2-way associative cache, it would be placed under (12 mod 4) = set 0.

As the number of ways increases, the number of bits used for index decreases. (Extreme case: fully associative cache has 0 bit for indexing)

2.3.2 Block Identification

To find a block within a cache: do tag matching ONLY.

Increasing associativity shrinks index, expands tag.

Block Address		Block
Tag	Index	Offset

Example 2. Suppose we have a 16KB of data in a direct-mapped cache with 4 word blocks. Determine the size of the tag, index and offset fields if we're using a 32-bit architecture.

- Offset: 1 block contains 4 words = 4*4 = 16 bytes = 2^4 bytes. Therefore 4 bits are reserved for offset.
- Index: # of blocks in the cache = $\frac{16KB}{16bytes} = \frac{2^{14}bytes}{2^4bytes} = 2^{10}$

  Since this is a directly mapped cache, associativity = 1.

  $\therefore$ 10 bits would be used for indexing.
- Tag: all the remaining bits are for tag. 32 - 4 - 10 = 18 bits.

Example 3. Assume the same cache structure as above. Now read the following 4 addresses: 0x00000014, 0x0000001C, 0x00000034, 0x00008014. Show the final content in the cache as well as the number of hits and misses.

First decompose the addresses above into different segments:

000000000000000000 0000000001 0100
000000000000000000 0000000001 1100
000000000000000000 0000000011 0100

000000000000000010 0000000001 0100

0x00000014: Miss, bring the block 0x00000010 - 0x0000001F to cache index block 1.
0x0000001C: Hit (tag matched)
0x00000034: Miss, bring the block 0x00000030 - 0x0000003F to cache index block 3.
0x00008014: Miss, tag not matching with the current block content, replace the cache index block 1 with new block 0x00008010 - 0x0000801F.

Summary: 1 hit, 3 misses, Final cache content:

Index	Block
0	
1	0x00008010 - 0x0000801F
2	
3	0x00000030 - 0x0000003F
...	

Table 2: Final cache content for Example 3.

Example 4. Consider the following loop:

```
for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
                x[i][j] += 3;
```

Assume that the base address for x[0][0] is 0x00000010 and each element has size 4 bytes. Using the same cache as in example 2, show the final cache content and count the number of misses and hits.

The array would go from x[0][0] to x[2][2] in this case. Assuming row major order, the following address would be accessed in sequential order:
0x00000010 (x[0][0])
0x00000014 (x[0][1])
0x00000018 (x[0][2])
0x0000001C (x[1][0])
0x00000020 (x[1][1])
0x00000024 (x[1][2])
0x00000028 (x[2][0])
0x0000002C (x[2][1])
0x00000030 (x[2][2])
∴ block with address 0x00000010 - 0x0000001F would be placed on cache index 1, address 0x00000020 - 0x0000002F would be placed on cache index 2, and 0x00000030 - 0x0000003F would be placed on cache index 3.
To sum, there would be 3 misses in total (compulsory - when first bringing in the cache blocks), 6 hits (consecutive accesses all hit in cache)
Final cache content:

Index	Block
0	
1	0x00000010 - 0x0000001F
2	0x00000020 - 0x0000002F
3	0x00000030 - 0x0000003F
...	

Table 3: Final cache content for Example 4.

2.3.3 Block Replacement

Different cache replacement policies: random, LRU, etc.

LRU. Evict the least frequently used block. Visualize it as a linked list, where the front is the most frequently used item that is the most unlikely to be evicted.

Algorithm for LRU:
On cache hit: move the block to MRU (i.e. to the front of the linked list)
On cache miss: replace the LRU block (i.e. the block at the end of the linked list). Move the newly replaced block to MRU.

SRRIP. There are two types of SRRIP: Hit Promotion (implemented in HW4), or Frequency Promotion.

Algorithm for n-bit SRRIP with hit promotion:
On cache hit: set RRPV of block to be 0.
On cache miss:

1. search for the first $2^n - 1$ RRPV from the left.
2. if $2^n - 1$ RRPV is found, go to step 5.
3. increment all RRPVs.
4. go back to step 1.
5. replace the block and set RRPV to $2^n - 2$.

n-bit SRRIP with frequency promotion: similar algorithm as above, but instead of setting RRPV of the block to be 0, change it to (current RRPV - 1, unless already 0) upon a cache hit.

FIFO. The first cache block would be evicted first. Purely based on timestamp.

Algorithm for FIFO:
On cache hit: No action.
On cache miss: Replace the block that has been in the cache for the longest time.

2.3.4 Write Strategy

	Write-Through	Write-Back
Policy	Data written to cache block, also written to lower-level memory	Write data only to the cache. Update lower level when a block falls out of the cache
Debug	Easy	Hard
Do read misses produce writes?	No	Yes
Do repeated writes make it to lower level?	Yes	No

Table 4: Comparison between the write through and write back strategies.

Write buffers are used in both strategies so that CPU won't stall. A *buffer* is used rather than a *register* since bursts of writes are common. RAW is still an issue for write buffer; to avoid it, drain the buffer before next read, or send read first after checking the write buffers.

Write strategies coupled with misses handling
Note: non-allocated = not bringing the block to the cache again.

$$\begin{cases} \text{hit: write through, write back} \\ \text{miss: non allocated, allocated} \end{cases}$$

2.4 Memory Hierarchy Basics

Causes of misses:
✓Compulsory: first reference to a block.
✓Capacity: blocks are discarded due to running out of space and later retrieved.
✓Conflict: Program makes repeated references to multiple addresses from different blocks that map to the same location in the cache
Miss per instruction $= \frac{Miss\_rate * Memory\_accesses}{Instruction\_count} = Miss\_rate * \frac{Memory\_accesses}{Instruction}$

Multi-level caches design:

- Inclusive: If all blocks in the higher level cache are also present in the lower level cache, then the lower level cache is said to be *inclusive* of the higher level cache.
- Exclusive: If the lower level cache contains blocks that are not present in the higher level cache, then the lower level cache is said to be *exclusive* of the higher level cache.
- Non-inclusive: there is no enforcement of either the cache inclusion property nor the cache exclusion property. A cache line in an inner cache may or may not be in an outer cache. Checking is necessary when responding to requests from other processors.

2.5 Basic Cache Optimization Techniques

2.5.1 Larger block size

✓Miss rate **decreases** due to spatial locality (more data can be packed into a block)
✓Compulsory misses **decreases** as the block is filled quicker.

× Hit time **increases** due to more data packed into the block, offset increases (i.e. takes longer time to find the requested data)
× Conflict misses **increases** as more blocks may be mapped to the same spot
× Miss penalty **increases** as it takes longer time to take the block.

2.5.2 Larger total cache capacity

✓Miss rate **decreases** as the cache can store more information, therefore more likely to be hit.

× Hit time **increases** as the search space is larger
× Power consumption **increases** to support operations in a larger cache (incur during disk I/Os)

2.5.3 Higher associativity

Extreme case: fully associative (blocks can be placed anywhere)
✓Conflict misses **decreases** as multiple memory addresses are less likely to be mapped to a same spot.

× Hit time **increases**
× Power consumption **increases** because more cache lines need to be accessed

2.5.4 Higher number of cache levels

No direct impact on AMAT.
✓Overall memory access time **decreases**. Can have one small cache to keep up with CCT of the fast processor and a larger cache to capture many accesses that would go to main memory.

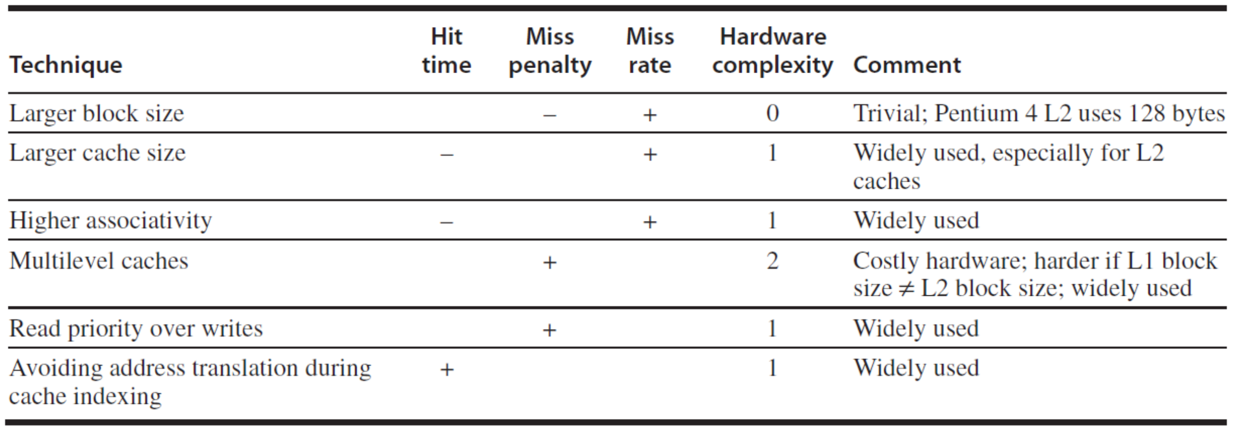
2.5.5 Giving priority to read misses over writes

✓Miss penalty **decreases**: instead of stall on a read miss, wait for the write buffer to clear, and then read from the memory, now only need to check the contents of the write buffer on a read miss, and if there are no conflicts and the memory system is available, let the read miss continue. (Basically the write buffer supplies the updated content)

2.5.6 Avoiding address translation in cache indexing

✓Hit time **decreases**

2.5.7 Summary



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

2.6 Advanced Cache Optimization Techniques

2.6.1 Small and simple first level caches

Critical timing path:

1. Addressing tag memory
2. comparing tags
3. selecting the correct set

If use direct mapped cache: can skip the 3rd step! Tag comparison and data transmission can take place on the same time.

✓Power **decreases** with lower associativity due to fewer cache lines are accessed.

2.6.2 Way Prediction

✓Hit time **decreases** if prediction is correct
Note: I-cache has better accuracy than D-cache.

× Miss penalty **increases** if prediction is wrong!

Example 5. Assume that a 64 KB four-way set associative single-banked L1 data cache is the cycle time limiter in a system. As an alternative cache organization you are considering a way-predicted cache modeled as a 64 KB direct-mapped cache with 80% prediction accuracy. Unless stated otherwise, assume that a mispredicted way access that hits in the cache takes one more cycle.

For a 4-way set associative cache, it has hit latency = 2 cycles, clock cycle = 2.53 ns, iss rate = 0.33%, and miss penalty = 4 cycles. For a direct mapped cache, it has hit latency = 2 cycles, clock cycle = 0.91 ns, miss rate = 2.2%, and miss penalty = 12 cycles.

Calculate AMAT for the way predicted cache.

AMAT = Hit time + Miss rate * Miss penalty
For way-predicted cache, 3 scenarios: miss, hit with way prediction correct, or hit with way prediction incorrect. It has cycle time and access time similar to a direct mapped cache and miss rate similar to a 4-way cache.
$\therefore$ AMAT(way-predicted) $= 0.33\% * 12 + [0.8 * 2 + 0.2 * 3] * (1 - 0.33\%) = 2.24$ cycles $=$ 2.03 ns.
`RLIN note:  use the miss rate as its direct comparison (4-way in this case) and others as its configuration setup (direct mapped in this case)`

Example 6. Discuss the difference between way prediction and value prediction.
Way prediction: : keep extra bits in cache to predict predict the "way", or block within the set, of next cache access.
Value prediction: The basic idea is to have the processor guess what value will be returned by a load instruction, and have it compute speculatively with this value. When the load completes, the actual value returned by the memory system is compared with the guess; if the guess is correct, the processor simply notes that fact and continues, but if the guess is incorrect, the processor rolls back the computation and recomputes with the actual value.

2.6.3 Pipelining Cache

This optimization is simply to pipeline cache access so that the effective latency of a first-level cache hit can be multiple clock cycles, **giving fast clock cycle time and high bandwidth but slow hits.**
✓Associativity **increases**
✓Throughput **increases**

× Branch miss prediction penalty **increases** if prediction is wrong (needs redo)

2.6.4 Nonblocking Caches

Previously: if there is a miss, consecutive requests must wait until the cache has fetch the result from the memory. (Definition of **blocking cache**)
Nonblocking cache: Allow hits before previous misses complete. Could be "Hit under miss" or "Hit under multiple miss".
★ L2 must support this
★ In general, processors can hide L1 miss penalty but not L2 miss penalty.
✓Throughput **increases** as consecutive memory requests no longer need to wait for a long time.

Miss Status Holding Registers (MSHR). A hardware design to handle miss-under-miss scenarios. When a miss occurs, the MSHRs are looked up to determine if the cache block is already being fetched. If there is a hit in MSHR, a secondary miss has occurred. It will be captured in MSHR and won't be sent to the memory again. (PC of the secondary request will be added to MSHR)

Compared with pipelined cache? Wait time happens inside the non-blocking cache, whereas it takes place outside the cache during the pipelining phase

2.6.5 Multibanked Caches

Organize cache as independent banks to support simultaneous access (✓for access time)
Note: use the **lower order bits** as bank index!!! If higher order bits (from the left) are used, consecutive blocks may be mapped to the same bank, incurring longer wait time. Also cannot exploit parallelism.

Each bank is treated as directly mapped cache

Tag	Cache Index	Bank Index	Offset

Table 5: A bank structure.

Example 7. Discuss the difference between multibank caches and n-way associative cache.

Banks are independent with each other. They have their own read and write port.

For n-way associative caches, it is possible that all requests might be mapped to 1 way, therefore cannot do n multiple operations at the same time.

2.6.6 Critical Word First, Early Restart

✓Miss penalty **decreases** as the wait time to get the requested word is shorter.

Request the missed word first from memory and send it to the processor ASAP; fetch the word in normal order, but send the requested word to the processor immediately when it arrives. Both techniques let the processor continues execution.

!: Effectiveness of these strategies depends on block size and likelihood of another access to the portion of the block that has not yet been fetched

2.6.7 Merging Write Buffer

Combine multiple write requests from the same block to a valid write buffer entry (multiword write)

✓Miss penalty **decreases** as it reduces stalls due to full write buffer.

Note: **DO NOT apply to I/O addresses.**

2.6.8 Compiler Optimizations

- ■ 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

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

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

- Loop Interchange: change nesting of loops to access data in order stored in memory. Improve *spatial locality.*

  Row-major order: array elements in the same row are stored consecutively in memory (a[1,1], a[1,2], a[1,3]...). In that sense, it's better to have the loop as followed:

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

  Example 8. Given the code block above and memory address at x[0][0] = a, compute the memory address of x[i][j].

  Assumption: row major order. a is a pointer to x[0][0].

  # of rows passed to i: $100 * 4 * i$

  # of columns passed to j: $4 * j$

  $\therefore$ x[i][j] would have address $a + 400i + 4j$.

  Column-major order: array elements in the same column are stored consecutively in memory (a[1,1], a[2,1], a[3,1]...). In that sense, it's better to have the loop as followed:

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

- Loop Fusion: Combine 2 independent loops that have same looping and some variables overlap. Improve *spatial locality.*

```
/* 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

- Blocking: Improve *temporal locality* by accessing "blocks" of data repeatedly vs. going down whole columns or rows

  Idea: compute on BxB submatrix that fits in cache

```
/* 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;
                };
/* 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 is called Blocking Factor

  Capacity Misses from $2N^3 + N^2$ to $2N^3/B + N^2$

2.6.9 Hardware Prefetching

Fetch two blocks on miss (include next sequential block). Better utilizing the memory bandwidth.

✓Miss penalty and miss rate **decrease**

Streaming: instead of 1 block, >1 blocks will be fetched.

Steaming vs. increased block sizes? Similar, but streaming doesn't have more conflicted misses, however maintained the same hit time; also, streaming is "hoping" that the other block will be used.

Downside: cache pollution (prefetch block is not useful but brought into the cache, replacing actual blocks that will be referenced and should stay there)

To avoid? Prefetch the guessing block in a buffer. Bring that block into cache only when it's actually requested by the processor.

2.6.10 Compiler Prefetching

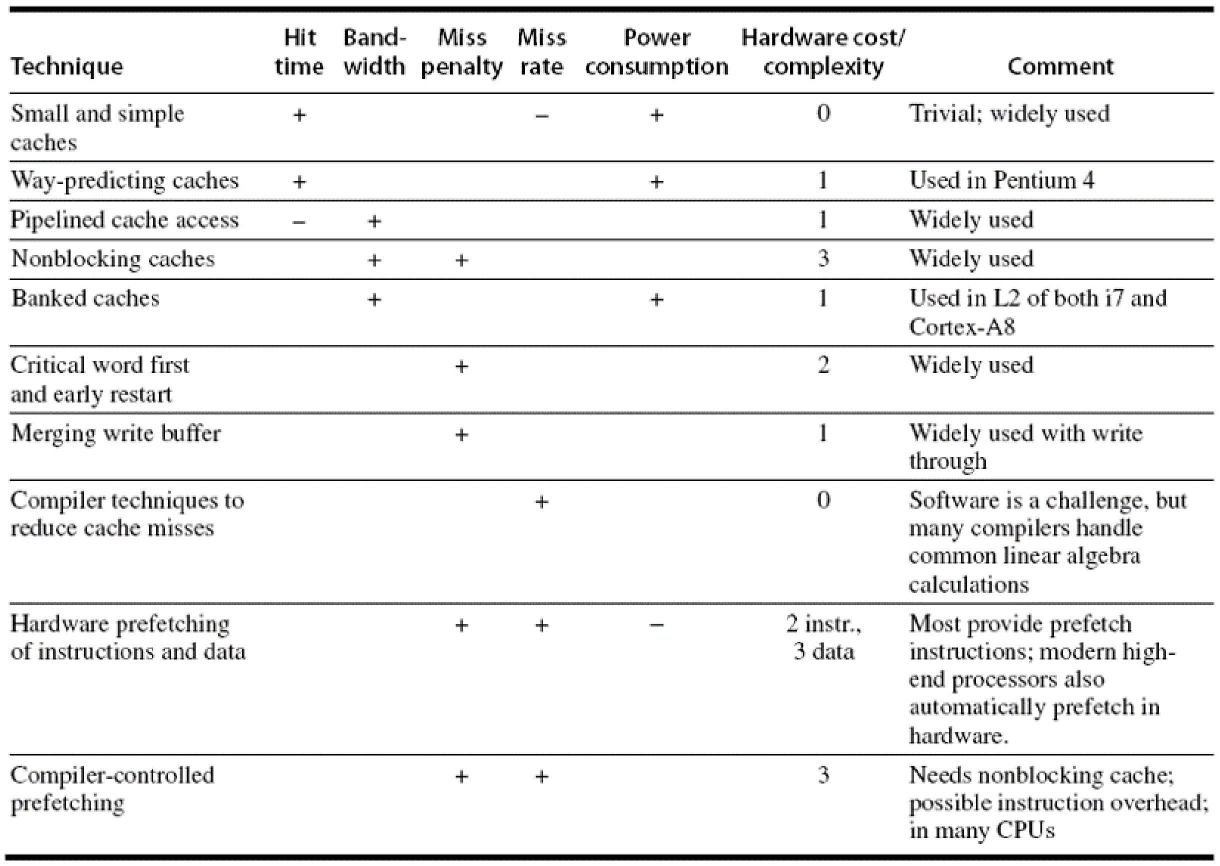
Insert prefetch instructions before data is needed

✓Miss penalty and miss rate **decrease**

Register prefetch: Loads data into register

Cache prefetch: Loads data into cache
Combine with loop unrolling and software pipelining
Note: non-blocking cache is required.
× Downside: ↑ instruction counts

2.6.11 Summary



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 cache access	–	+				1	Widely used
Nonblocking caches		+	+			3	Widely used
Banked caches		+			+	1	Used in L2 of both i7 and Cortex-A8
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

Figure 2.11 Summary of 10 advanced cache optimizations showing impact on cache performance, power consumption, and complexity. Although generally a technique helps only one factor, prefetching can reduce misses if done sufficiently early; if not, it can reduce miss penalty. + 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.

2.7 Memory Technology

Won't be covered in the final. Please refer to the slides.

2.8 Virtual Memory

Basic idea: treat the memory as a cache of the disk. Give applications an illusion that each of them occupies the full memory space (indeed no, all sharing the same memory + VM). Managed by the OS. (See HW5 P1 for example)

2.8.1 Four Questions of Memory Hierarchy

- Block placement: fully associative
- Block identification: use page tables
- Block replacement: replacement policy is the scope of OS. Not handled by the hardware.
- Write strategy: write back, as writing through disks takes a long time. That is, only the data in the memory is updated. Dirty bit is required!

2.8.2 Three Advantages of Virtual Memory

- Translation:
  - Program can be given consistent view of memory, even though physical memory is scrambled
  - Makes multithreading reasonable (now used a lot!)
  - Only the most important part of program (”Working Set”) must be in physical memory.
  - Contiguous structures (like stacks) use only as much physical memory as necessary yet still grow later.
- Protection:
  - Different threads (or processes) protected from each other.
  - Different pages can be given special behavior (Read Only, Invisible to user programs, etc).
  - Kernel data protected from User programs
  - Very important for protection from malicious programs
- Sharing:
  - Can map same physical page to multiple users (”Shared memory”)

Page: A virtual address space, divided into blocks of memory.

A page table is indexed by a **virtual address**. Page table maps virtual page numbers to physical frames (”PTE” = Page Table Entry)

Each process has its own page table and cannot access each other’s page tables.

A valid page table entry codes **physical memory ”frame”** address for the page

If the page table is too large to fit into the main memory: make it several levels.

Page replacement policy: hardware only supports setting dirty and used bits (page maintained by OS)

2.8.3 Translation Look-Aside Buffer (TLB)

TLB: A small fully-associative cache of mappings from virtual to physical addresses. Important for fast translation. (Could have misses; distinguish from the branch history table, which is a table and could not have misses but might be wrong)

TLB misses are significant in processor performance. If missed: needs to go through the page table.

TLB also contains protection bits for virtual address; a fast common case: Virtual address is in TLB, process has permission to read/write it.

Virtual addresses and physical addresses shared the same offset but different page numbers. TLB faciliates in translating the virtual page # to physical page #.

Overlapped TLB access. First lookup physical address using the offset from the virtual address. Once the offsets are matched, check against TLB to make sure tags are matching (hit!)

Overlapped access only works as long as the address bits used to index into the cache do not change as the result of VA translation. This usually limits things to small caches, large page sizes, or high n-way set associative caches if you want a large cache.

3 Chapter 5

Topic: thread level parallelism (MIMD model - multiple instruction stream, multiple data stream)

3.1 Types

- Symmetric multiprocessors (SMP): Small number of cores. Share single memory with uniform memory latency
- Distributed shared memory (DSM): Memory distributed among processors. Non-uniform memory access/latency (NUMA). Processors connected via direct (switched) and non-direct (multi-hop) interconnection networks

3.2 Cache Coherence

Coherence vs consistency.

■ Coherence: concerns about 1 variable. When a variable is updated by a processor, it should be synced on other processors as well.

■ Consistency: concerns about 2 or more variables. It focuses on the order. If a processor writes location A followed by location B, any processor that sees the new value of B must also see the new value of A.

Coherent caches provide:

★ Migration: movement of data

★ Replication: multiple copies of data

Coherence issue comes from writing.

Cache coherence protocol: directory based vs. snooping.

3.2.1 Snoopy Coherence Protocol

Snoopy protocol for invalidate + write through caches

$$\begin{tikzcd} V \arrow[loop above] \arrow[d, bend right] \\ I \arrow[u, bend right] \arrow[loop right] \end{tikzcd}$$

Basic idea:

- At valid:

  Stay if processor read, processor write then bus write, and bus read

  Become invalid if bus write

- At invalid:

  Stay if processor write then bus write, bus read, and bus write

  Become valid if processor read then bus read

Snoopy protocol for write back caches: MSI

$$\begin{tikzcd} I_{00} \arrow[r, shift right] \arrow[d, shift left] & S_{11} \arrow[l, shift right] \arrow[dl, shift left] \\ M_{01} \arrow[u, shift left] \arrow[ur, shift left] & \end{tikzcd}$$

Basic idea:

- At invalid:

Processor read: regardless of hit or miss, this cache block is invalid. Thus, the processor needs to place a read request on the bus and change the status to be "shared".

Processor write: regardless of hit or miss, this cache block is invalid. Thus, the processor needs to place a write request on the bus and change the status to be "modified".

Bus read: a read request is on the bus (generated by other processors) but my block is invalid, so no action.

Bus write: a write request is on the bus (generated by other processors) but my block is invalid, so I cannot offer it (no action).

- At shared:

  Processor read:

  Hit: nice! The current shared block has the requested info, no further action.

  Miss: it could be due to the processor having the shared block but tag does not match. Just need to place a read request on the bus, but stay in the shared status.

  Processor write:

  Hit: nice, I own the block. But now I need to write to the block. Due to the fact that this is a write-back cache, I need to own this block exclusively so that other processors can't touch it at the same time. Thus, the status has to change to modified with a write request placed on the bus (so that other processors know).

  Miss: similar to the hit case, may be due to tag not matching. Still needs to request the block from the memory (broadcast through the bus) and also prevent other writing to the block by setting the status to be modified.

  Bus read: somebody wants to read my block, but there is no change, so stay in the shared status.

  Bus write: somebody updated the shared block value. The current shared block is no longer valid. Change to invalid state.

- At modified:

  Processor read:

  Hit: nice, it's just here! No further action is required.

  Miss: the requested info is not here, has to place a request on the bus and change status to be shared. Also needs to write back the info to the memory (before placing the read miss on bus)

  Processor write:

Hit: just update the value. Nothing else would change.

Miss: similar to read miss, needs to place a request on bus. However status is not changing due to the fact that a write is going on.

Bus read: somebody wants to read my block. I (the processor that owns the modified block) have the latest status. Thus, I will offer it. I would write back to the memory & change my status to be shared.

Bus write: somebody wants to write on my modified block. I should update the memory immediately then make my copy invalid (latest modified copy at the new owner)

Complication. Operations might not be atomic.
Extension. Add exclusive state to indicate clean block in only one cache (MESI protocol) to prevents needing to write invalidate on a write; Owned state.

3.2.2 Coherence Misses

True sharing misses: two or more processors writing to the same variable at a shared block.
False sharing misses: two or more processor writing to different variables, but they happened to be on the same block.
To avoid false sharing misses? Reduce the block size
None of these would occur on uniprocessor.

3.2.3 Directory Protocol

There is no buses anymore; each processor has its own caches and memory. Can have secrets!
$\rightarrow$ Add a directory in front of the memory to keep track of every block (SMP) or multiple directories (DSM). Directory maintains block states and sends invalidation messages.
Directory needs to get acknowledgements so takes longer time.

Three states:

- Shared: One or more nodes have the block cached, value in memory is up-to-date. Directory keeps the set of node IDs. (Similar to SHARED in snoopy)
- Uncached: similar to INVALID in snoopy.
- Modified: Exactly one node has a copy of the cache block, value in memory is out-of-date. Directory keeps the owner node ID. (Similar to MODIFIED in snoopy)

Basic ideas:

- For uncached block:
  - Read miss: Requesting node is sent the requested data and is made the only sharing node, block is now shared

  - Write miss: The requesting node is sent the requested data and becomes the sharing node, block is now exclusive

- For shared block:
  - Read miss: The requesting node is sent the requested data from memory, node is added to sharing set
  - Write miss: The requesting node is sent the value, all nodes in the sharing set are sent invalidate messages, sharing set only contains requesting node, block is now exclusive

- For exclusive block:
  - Read miss: The owner is sent a data fetch message, block becomes shared, owner sends data to the directory, data written back to memory, sharers set contains old owner and requestor
  - Data write back: Block becomes uncached, sharer set is empty
  - Write miss: Message is sent to old owner to invalidate and send the value to the directory, requestor becomes new owner, block remains exclusive

4 Chapter 4

Topic: Data level parallelism in vector, SIMD, and GPU

SIMD (single instruction stream, multiple data stream) is more energy efficient than MIMD as it only needs to fetch one instruction per data operation. It allows programmers to continue to think sequentially.

4.1 Vector Architectures

Basic idea: Read sets of data elements into "vector registers"; Operate on those registers; Disperse the results back into memory.

Example architecture learned: VMIPS. Assume that each vector register holds a 64-element, 64 bits/element vector. All functional and load/store units are fully pipelined (that is: the first instruction takes the longest cycle; all preceding instruction just +1)

Vector execution time depends upon 3 factors:
★ Length of operand vectors
★ Structural hazards
★ Data dependencies

4.1.1 Some Terminology

Convoy. Set of vector instructions that could potentially execute together (i.e. no hazard) - dependency graph needed!

Chaining. Allows a vector operation to start as soon as the individual elements of its vector source operand become available.

Chime. Unit of time to execute one convey; m conveys executes in m chimes. For vector length of n, requires m * n clock cycles (a good estimation, but not accurate).

4.1.2 Improvements on Vectors

Multiple lanes. Break down a long vector register to small segments (parallel pipelines). More hardware to support >1 element per clock cycle.

Avoiding interlane communication.

Not doable in CMP due to hardware costs.

Vector Length Register (VLR). Using vector to perform an operation may be faster, but the length of the designated vector is not known at compile time. Break down the operation to be (n/MVL)+1 pieces, each piece has length MVL except for the last piece = n%MVL. (MVL = maximum vector length)

Note: VLR $\leqslant$ MVL

n greater than MVL? Use **Strip mining** (see example below).

Sample code: $Y = aX + Y$

```
low = 0;
VL = (n % MVL); /*find odd-size piece using modulo op % */
for (j = 0; j <= (n/MVL); j=j+1) { /*outer loop*/
        for (i = low; i < (low+VL); i=i+1) /*runs for length VL*/
                Y[i] = a * X[i] + Y[i] ; /*main operation*/
        low = low + VL; /*start of next vector*/
        VL = MVL; /*reset the length to maximum vector length*/
}
```

Vector Mask Register. For conditional branches, use a vector-mask register (boolean vector) to control the execution of a vector instruction. Any vector instructions executed operate only on the vector elements whose corresponding entries in the vector-mask register are 1. (energy $\uparrow$)

Rely on compilers to manipulate mask registers explicitly

USE VECTOR MASK REGISTER TO "DISABLE" ELEMENTS - GFLOPS RATE DECREASES!

Memory Banks. Memory system must be designed to support high bandwidth for vector loads and stores.

■ Spread accesses across multiple banks.

- ★ Control bank addresses independently
- ★ Load or store non sequential words
- ★ Support multiple vector processors sharing the same memory

Example 9. 32 processors, each generating 4 loads and 2 stores/cycle. Processor cycle time is 2.167 ns, SRAM cycle time is 15 ns. How many memory banks needed?

# of cycles = $\frac{15ns}{2.167ns}$ = 7 cycles.

Max. # of requests at the peak time: (4 loads + 2 stores) = 6 references per cycle

∴ For 1 processor, max # of requests = 6 * 7 = 42 references

Assume that 1 memory bank takes care of 1 reference. Total memory bank needed: 42 * 32 = 1344.

Stride. Stride = The distance separating elements to be gathered into a single register

Non-unit stride: a vector processor can handle strides greater than 1 using only vector load and vector store operations with stride capability.

Bank conflict = requests all coming in to the same bank, causing stalls

occurs if (Number of banks)/(Least common multiple (Stride, Number of banks) < Bank busy time

Example 10. Suppose we have 8 memory banks with a bank busy time of 6 clocks and a total memory latency of 12 cycles. How long will it take to complete a 64-element vector load with a stride of 1? With a stride of 32?

Since the number of banks is larger than the bank busy time, for a stride of 1 the load will take 12 + 64 = 76 clock cycles, or 1.2 clock cycles per element. The worst possible stride is a value that is a multiple of the number of memory banks, as in this case with a stride of 32 and 8 memory banks. Every access to memory (after the first one) will collide with the previous access and will have to wait for the 6-clock-cycle bank busy time. The total time will be 12 + 1 + 6 * 63 = 391 clock cycles, or 6.1 clock cycles per element.

Gather-Scatter. Accesses are still scattered around in main memory. Similar to vector processors, re-index the scattered access so that they are clustered and good for vector processing.

Programming Vector Architectures. Depends upon the compiler - programmer relationship to utilize the vectors.

4.2 SIMD Extensions

Media applications operate on data types narrower than the native word size.

SIMD is particularly good for energy (but not performance), esp. for loops. It doesn't have one core repetitively do all the work.

4.2.1 Limitations

1. Number of data operands encoded into op code
2. No sophisticated addressing modes (strided, scatter-gather)
3. No mask registers

4.3 GPU

4.3.1 Basic ideas

■ Heterogeneous execution model

★ CPU is the host, GPU is the device

■ Develop a C-like programming language for GPU

■ Unify all forms of GPU parallelism as CUDA thread

★ Nvidia GPU shines mainly because of the support from the CUDA library

■ Programming model is ”Single Instruction Multiple Thread”

4.3.2 Threads and Blocks

A thread is associated with each data element. Threads are organized into blocks, blocks are organized into a grid.

★ GPU hardware handles thread management, not applications or OS

★ No more registers available? Stop assigning more threads!

4.3.3 NVIDIA GPU Architecture

Compared with vector machines, similarity...

♡ Works well with data-level parallel problems

♡ Scatter-gather transfers (all loads gather, all stores scatter)

♡ Mask registers

♡ Large register files

Differences...

↗ No scalar processor

↗ Uses multithreading to hide memory latency

↗ Has many functional units, as opposed to a few deeply pipelined units like a vector processor

4.3.4 Conditional Branching

Note: GPU has many threads, each of them has no data dependency with each other. Could be good (when facing branches, just calculate all results and choose the matching one), or could be bad (when two threads are forced to communicate with each other due to code design).

GPU uses hardware to manipulate internal mask registers that are invisible to GPU software

Also used:

■ Branch synchronization stack

→ Entries consist of masks for each SIMD lane, i.e. which threads commit their results (all threads execute)

■ Instruction markers to manage when a branch diverges into multiple execution paths

→ Push on divergent branch

■ ... and when paths converge

→ Act as barriers, pops stack

Sidenote: GPU does context switch all the time!
If a thread is pending on data, switch to the next one

4.4 Loop-Level Parallelism

Focuses on determining whether data accesses in later iterations are dependent on data values produced in earlier iterations (i.e. Loop-carried dependence)

4.4.1 How to find if there is a loop level dependency?

Use GCD test!

Definition: If a dependency exists, GCD(c,a) must evenly divide (d-b). Assume that $X[a*i+b]$ and $X[c*i+d]$ are of interest.

Example 11. Is there a loop level dependency in the following statement?

```
for (i=0; i<100; i=i+1) {
        X[2*i+3] = X[2*i] * 5.0;
}
```

Use GCD test: a = 2, b = 3, c = 2, d = 0.

$\therefore$ GCD(a,c) = GCD(2,2) = 2. d - b = -3.

Since 2 does not divide -3 (-3/2 does not have remainder 0), no dependence is possible.

4.4.2 Finding dependencies

Note: in a loop structure, there could still be antidependencies (write after read (WAR)) and output dependencies (write after write (WAW)).

Example: see HW5 exercise 4.14.