
EEC 581 Computer Architecture

Lec 12 – Memory Hierarchy (Ch. 5)

Chansu Yu
Electrical and Computer Engineering
Cleveland State University

Acknowledgement ...

- **Part of class notes are from**
 - David Patterson
 - Electrical Engineering and Computer Sciences
 - University of California, Berkeley
 - <http://www.eecs.berkeley.edu/~pattsn>
 - <http://www-inst.eecs.berkeley.edu/~cs252>

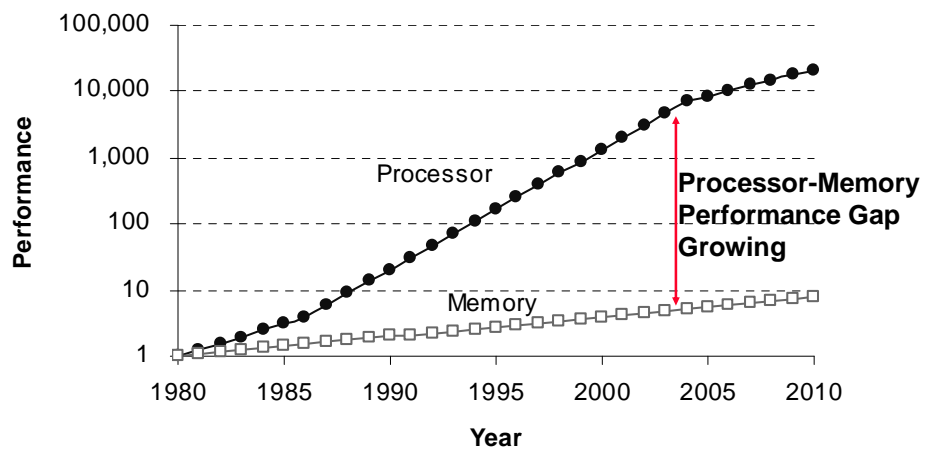
Outline

- 11 Advanced Cache Optimizations (5.2)
- Memory Technology and DRAM optimizations (5.3)
- Virtual Machines (5.4)
- AMD Opteron Memory Hierarchy (5.6)
- Fallacies and Pitfalls
- Conclusion

11/27/2007

3

Why More on Memory Hierarchy?

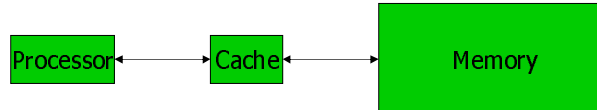


11/27/2007

4

Caches: The Basic Idea

- **A smaller set of storage locations storing a subset of information from a larger set.**
 - Typically, SRAM cache for DRAM main memory:

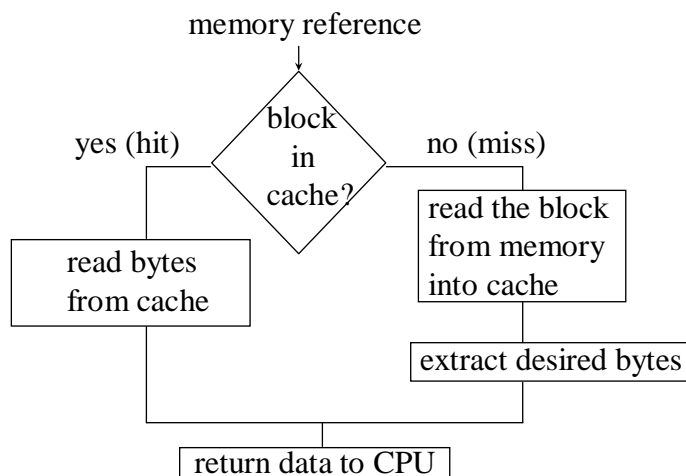


- **Goal:** Decrease average time for data access.
- **Use:** Look in cache for data. Look in larger storage only if not found in cache.
- Invisible to programmer.
- **Multiple ways to organize – we will see several.**
- **For simplicity, assume all memory accesses are word-sized.**

11/27/2007

5

Caches: Flowchart



11/27/2007

6

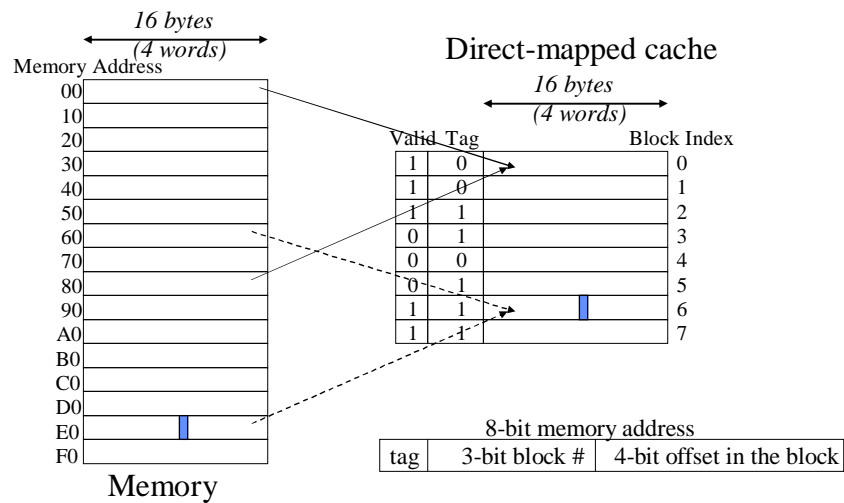
Decoding Memory Address

- Memory address [376]
 - ↓ offset within the block
 - ↓ cache block index
 for identifying the original memory block
- Given the memory address (e.g. 376)
 - Extract "7"
 - Compare 7th tag in cache with "3"
 - Extract 6th byte in the block

11/27/2007

7

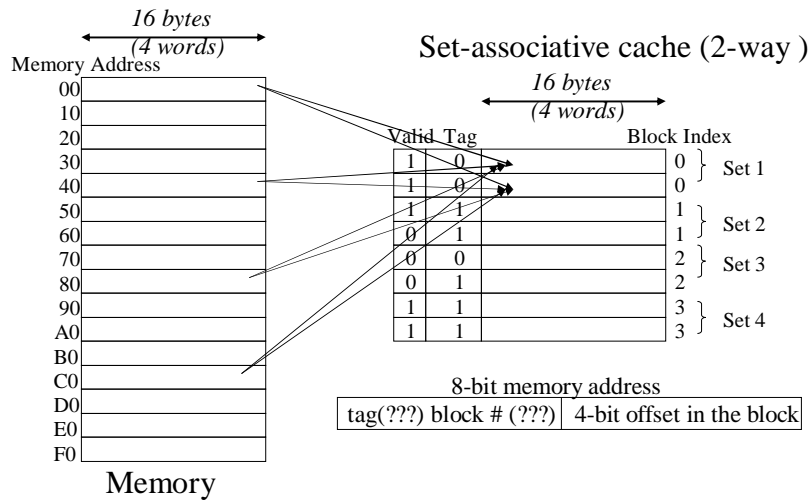
Direct-mapped Block Placement



11/27/2007

8

2-Way Set-associative Block Placement



11/27/2007

9

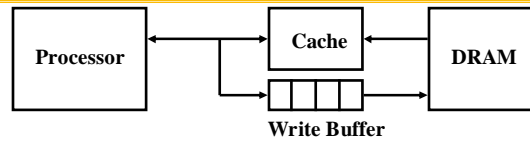
Review: 6 Basic Cache Optimizations

- Reducing hit time
 1. Giving Reads Priority over Writes
 - E.g., Read complete before earlier writes in write buffer
 2. Avoiding Address Translation during Cache Indexing
- Reducing Miss Penalty
 3. Multilevel Caches
- Reducing Miss Rate
 4. Larger Block size (Compulsory misses)
 5. Larger Cache size (Capacity misses)
 6. Higher Associativity (Conflict misses)

11/27/2007

10

Write Buffer



- **A Write Buffer is needed between the Cache and Memory**
 - Used for every write in write-through cache
 - Used when block replacement in write-back cache

 - Processor: writes data into the cache and the write buffer
 - Memory controller: write contents of the buffer to memory
- **Write buffer is just a FIFO:**
 - Typical number of entries: 4
 - Works fine if: Store frequency (w.r.t. time) $\ll 1 / \text{DRAM write cycle}$

11/27/2007

11

11 Advanced Cache Optimizations

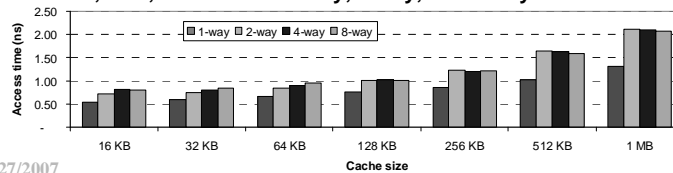
- Reducing hit time
 1. Small and simple caches
 2. Way prediction
 3. Trace caches
- Increasing cache bandwidth
 4. Pipelined caches
 5. Multibanked caches
 6. Nonblocking caches
- Reducing Miss Penalty
 7. Critical word first
 8. Merging write buffers
- Reducing Miss Rate
 9. Compiler optimizations
- Reducing miss penalty or miss rate via parallelism
 10. Hardware prefetching
 11. Compiler prefetching

11/27/2007

12

1. Fast Hit times via Small and Simple Caches

- Index tag memory and then compare takes time
- \Rightarrow Small cache can help hit time since smaller memory takes less time to index
 - E.g., L1 caches same size for 3 generations of AMD microprocessors: K6, Athlon, and Opteron
 - Also L2 cache small enough to fit on chip with the processor avoids time penalty of going off chip
- Simple \Rightarrow direct mapping
 - Can overlap tag check with data transmission since no choice
- Access time estimate for 90 nm using CACTI model 4.0
 - Median ratios of access time relative to the direct-mapped caches are 1.32, 1.39, and 1.43 for 2-way, 4-way, and 8-way caches

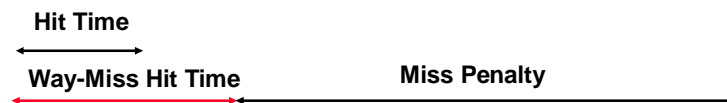


11/27/2007

13

2. Fast Hit times via Way Prediction

- How to combine fast hit time of Direct Mapped and have the lower conflict misses of 2-way SA cache?
- Way prediction: keep extra bits in cache to predict the “way,” or block within the set, of next cache access.
 - Multiplexor is set early to select desired block, only 1 tag comparison performed that clock cycle in parallel with reading the cache data
 - Miss \Rightarrow 1st check other blocks for matches in next clock cycle

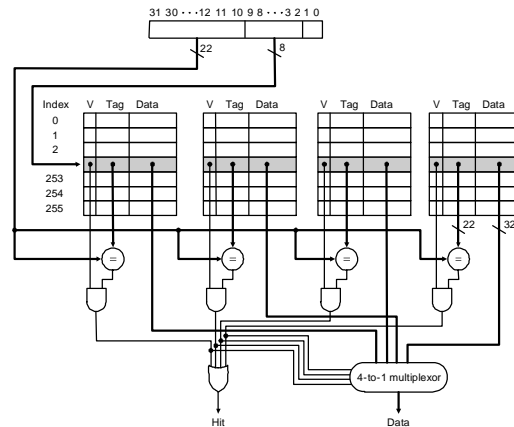


- Accuracy \approx 85%
- Pentium 4

11/27/2007

14

4-way SA Cache



11/27/2007

15

3. Fast Hit times via Trace Cache

- Find more instruction level parallelism?
How avoid translation from x86 to microops?
- Trace cache in Pentium 4
 1. Dynamic traces of the executed instructions vs. static sequences of instructions as determined by layout in memory
 2. Cache the micro-ops vs. x86 instructions
 - Decode/translate from x86 to micro-ops on trace cache miss
- + 1. ⇒ better utilize long blocks (don't exit in middle of block, don't enter at label in middle of block)
- 1. ⇒ complicated address mapping since addresses no longer aligned to power-of-2 multiples of word size
- 1. ⇒ instructions may appear multiple times in multiple dynamic traces due to different branch outcomes

11/27/2007

16

Microprogramming

- **Instruction**
 - is not the smallest software entity
 - consists of a number of micro-instructions
- **Microprogramming = microinstructions + sequencing**
- In the '60s and '70s microprogramming was very popular
- In the '80s RISC processors based on pipelining became popular
- Implementations of IA-32 architecture processors since 486 use:
 - “hardwired control” for simpler instructions
 - “microcoded control” for more complex instructions
- **The IA-64 architecture uses a RISC-style ISA**

11/27/2007

17

Microprogramming

Instructions from memory are compiled into micro-instructions

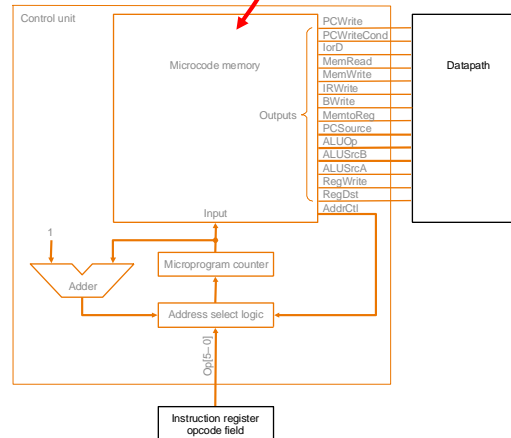
- **Microprogram approach simplifies control signals generation**
 - via grouping a number of signals when they are used together, and
 - naming them for easy programming
- **For example, register – related operations are**
 - Read (for all instructions),
 - Write ALU (for R-type), and
 - Write MDR (for lw)

They mean

- No control signals
- RegWrite, MemtoReg=0, RegDst=1
- RegWrite, MemtoReg=1, RegDst=0

11/27/2007

18



Microprogramming

- **Fields (microinstruction format):**
 - Label, ALU Control, SRC1, SRC2, Reg Control, Memory, PCWrite Control, Sequencing

- **Example: state 0**

- RTL: $PC = PC + 4$, $IR = Memory[PC]$
- Add PC by 4, Read Memory[PC], PCWrite with ALU, next

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshft	Read			Dispatch 1
Mem1	Add	A	Extend				Dispatch 2
LW2					Read ALU		Seq
				Write MDR			Fetch
SW2					Write ALU		Fetch
Rformat1	Func code	A	B				Seq
				Write ALU			Fetch
BEQ1	Subt	A	B			ALUOut-cond	Fetch
JUMP1						Jump address	Fetch

Microinstruction format

Field name	Value	Signals active	Comment
ALU control	Add	ALUOp = 00	Cause the ALU to add.
	Subt	ALUOp = 01	Cause the ALU to subtract; this implements the compare for branches.
	Func code	ALUOp = 10	Use the instruction's function code to determine ALU control.
SRC1	PC	ALUSrcA = 0	Use the PC as the first ALU input.
	A	ALUSrcA = 1	Register A is the first ALU input.
	B	ALUSrcB = 00	Register B is the second ALU input.
SRC2	4	ALUSrcB = 01	Use 4 as the second ALU input.
	Extend	ALUSrcB = 10	Use output of the sign extension unit as the second ALU input.
	Extshft	ALUSrcB = 11	Use the output of the shift-by-two unit as the second ALU input.
Register control	Read		Read two registers using the rs and rt fields of the IR as the register numbers and putting the data into registers A and B.
	Write ALU	RegWrite, RegDst = 1, MemtoReg = 0	Write a register using the rd field of the IR as the register number and the contents of the ALUOut as the data.
Memory	Write MDR	RegWrite, RegDst = 0, MemtoReg = 1	Write a register using the rt field of the IR as the register number and the contents of the MDR as the data.
	Read PC	MemRead, lcrD = 0	Read memory using the PC as address; write result into IR (and the MDR).
PC write control	Read ALU	MemRead, lcrD = 1	Read memory using the ALUOut as address; write result into MDR.
	Write ALU	MemWrite, lcrD = 1	Write memory using the ALUOut as address, contents of B as the data.
Sequencing	ALU	PCSource = 00, PCWrite	Write the output of the ALU into the PC.
	ALUOut-cond	PCSource = 01, PCWriteCond	If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.
Sequencing	jump address	PCSource = 10, PCWrite	Write the PC with the jump address from the instruction.
	Seq	AddrCtl = 11	Choose the next microinstruction sequentially.
	Fetch	AddrCtl = 00	Go to the first microinstruction to begin a new instruction.
Sequencing	Dispatch 1	AddrCtl = 01	Dispatch using the ROM 1.
	Dispatch 2	AddrCtl = 10	Dispatch using the ROM 2.

4: Increasing Cache Bandwidth by Pipelined Cache Access

- Pipeline cache access to maintain bandwidth, but higher latency
 - Instruction cache access pipeline stages:
 - 1 cc: Pentium
 - 2 cc's: Pentium Pro through Pentium III
 - 4 cc's: Pentium 4
- ⇒ greater penalty on mispredicted branches
- ⇒ more clock cycles between the issue of the load and the use of the data

11/27/2007

21

5. Increasing Cache Bandwidth: Non-Blocking Caches

- Non-blocking cache or lockup-free cache allow data cache to continue to supply cache hits during a miss
 - Useful in OOO completion processors
 - reduces the effective miss penalty by working during miss
- “hit under miss”
- “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
 - Pentium Pro allows 4 outstanding memory misses

11/27/2007

22

6: Increasing Cache Bandwidth via Multibanked Caches

- Rather than treat the cache as a single monolithic block, divide into independent banks that can support simultaneous accesses
 - E.g., T1 (“Niagara”) L2 has 4 banks
- Banking works best when accesses naturally spread themselves across banks \Rightarrow mapping of addresses to banks affects behavior of memory system
- Simple mapping that works well is “sequential interleaving”
 - Spread block addresses sequentially across banks
 - E.g, if there 4 banks, Bank 0 has all blocks whose address modulo 4 is 0; bank 1 has all blocks whose address modulo 4 is 1; ...

11/27/2007

23

7. Reduce Miss Penalty: Early Restart and Critical Word First

- Don't wait for full block 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
 - Spatial locality \Rightarrow tend to want next sequential word, so not clear size of benefit of just early restart
- **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
 - Long blocks more popular today \Rightarrow Critical Word 1st Widely used



block

11/27/2007

24

8. Merging Write Buffer to Reduce Miss Penalty

- Write buffer to allow processor to continue while waiting to write to memory
- If buffer contains modified blocks, the addresses can be checked to see if address of new data matches the address of a valid write buffer entry
- If so, new data are combined with that entry
- Increases block size of write for write-through cache of writes to sequential words, bytes since multiword writes more efficient to memory
- The Sun T1 (Niagara) processor, among many others, uses write merging

11/27/2007

25

9. Reducing Misses by Compiler Optimizations

- McFarling [1989] reduced caches misses by 75% on 8KB direct mapped cache, 4 byte blocks in software
- Instructions
 - Reorder procedures in memory so as to reduce conflict misses
 - Profiling to look at conflicts (using tools they developed)
- Data
 - *Merging Arrays*: improve spatial locality by single array of compound elements vs. 2 arrays
 - *Loop 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

11/27/2007

26

Merging Arrays Example

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

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

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


11/27/2007

27

Loop Interchange Example

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

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



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

11/27/2007

28

Loop Fusion Example

```

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

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

```

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

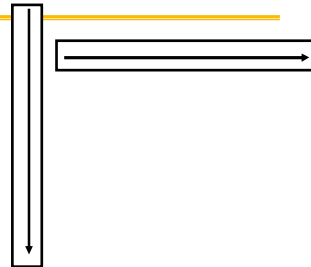
29

Blocking Example

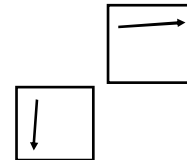
```

/* 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:**
 - $2N^3 + N^2 \Rightarrow$ (assuming no conflict; otherwise ...)
- **Idea: compute on BxB submatrix that fits**



11/27/2007

30

Blocking Example

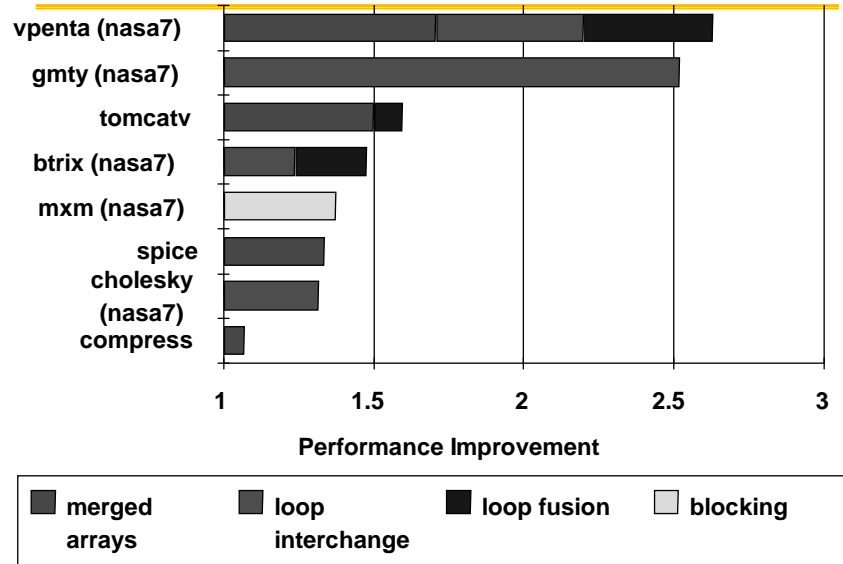
```
/* After */
for (jj = 0; jj < N; jj = jj+B)
for (kk = 0; kk < N; kk = kk+B)
for (i = 0; i < N; i = i+1)
  for (j = jj; j < min(jj+B-1,N); j = j+1)
    {r = 0;
     for (k = kk; k < min(kk+B-1,N); k = k+1) {
       r = r + y[i][k]*z[k][j];};
     x[i][j] = x[i][j] + r;
    };
```

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

11/27/2007

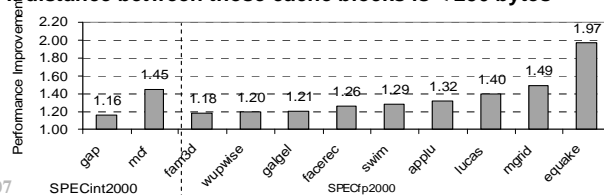
31

Summary of Compiler Optimizations to Reduce Cache Misses



10. Reducing Misses by Hardware Prefetching of Instructions & Data

- Prefetching relies on having extra memory bandwidth that can be used without penalty
- Instruction Prefetching
 - Typically, CPU fetches 2 blocks on a miss: the requested block and the next consecutive block.
 - Requested block is placed in instruction cache when it returns, and prefetched block is placed into instruction stream buffer
- Data Prefetching
 - Pentium 4 can prefetch data into L2 cache from up to 8 streams from 8 different 4 KB pages
 - Prefetching invoked if 2 successive L2 cache misses to a page, if distance between those cache blocks is < 256 bytes



11/27/2007

SPECint2000

SPECfp2000

33

11. Reducing Misses by Software Prefetching Data

- Data Prefetch
 - Register prefetch: Load data into register (HP PA-RISC loads)
 - Cache prefetch: load into cache (MIPS IV, PowerPC, SPARC v. 9)
 - Special prefetching instructions cannot cause faults; a form of speculative execution
- Issuing Prefetch Instructions takes time
 - Is cost of prefetch issues < savings in reduced misses?
 - Higher superscalar reduces difficulty of issue bandwidth

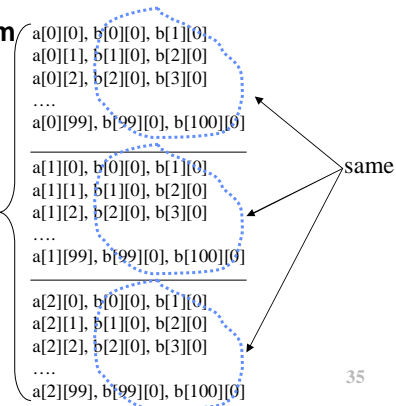
11/27/2007

34

Prefetch - Example

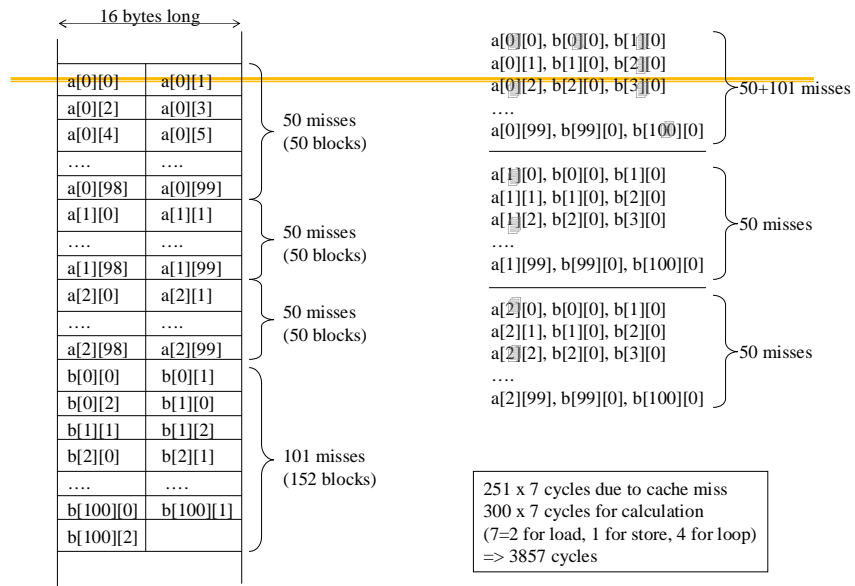
- 8KB direct-mapped data cache (16-byte block) with write-back and write-allocate policy
- $a[3][100]$ & $b[101][3]$ are 8 bytes long floating point numbers
- Consider the following program

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



11/27/2007

35



11/27/2007

36

```

for (i=0; i<3; i++)
  for (j=0; j<100; j+=2)
    prefetch (a[i][j]);
for (j=0; j<100; j++)
  prefetch (b[j][0]);
for (i=0; i<3; i++)
  for (j=0; j<100; j++)
    a[i][j] = b[j][0] * b[j+1][0];

```

Concurrent execution
(no need to wait)

150 x 5 cycles due to prefetch loop
 100 x 5 cycles due to prefetch loop
 300 x 7 cycles for calculation
 => 3350 cycles : small benefit !!!

11/27/2007

37

```

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

```

100 x (1+1+2+1+4) for a[0][*]
 200 x (1+2+1+4) for a[1,2][*]
 => 2500 cycles : more benefit !!!

11/27/2007

38

Compiler Optimization vs. Memory Hierarchy Search

- **Compiler tries to figure out memory hierarchy optimizations**
- **New approach: “Auto-tuners” 1st run variations of program on computer to find best combinations of optimizations (blocking, padding, ...) and algorithms, then produce C code to be compiled for *that* computer**
- **“Auto-tuner” targeted to numerical method**
 - E.g., PHIPAC (BLAS), Atlas (BLAS), Sparsity (Sparse linear algebra), Spiral (DSP), FFT-W

11/27/2007

39

Technique	Hit Time	Band-width	Miss penalty	Miss rate	HW cost/complexity	Comment
Small and simple caches	+			-	0	Trivial; widely used
Way-predicting caches	+				1	Used in Pentium 4
Trace caches	+				3	Used in Pentium 4
Pipelined cache access	-	+			1	Widely used
Nonblocking caches		+	+		3	Widely used
Banked caches		+			1	Used in L2 of Opteron and Niagara
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; some computers have compiler option
Hardware prefetching of instructions and data			+	+	2 instr., 3 data	Many prefetch instructions; AMD Opteron prefetches data
Compiler-controlled prefetching			+	+	3	Needs nonblocking cache; in many CPUs