Instruction-Level Parallelism and Its Exploitation (Week#14b)

Speculation: Implementation Issues and Extensions

Speculation Support: Register Renaming Versus Reorder Buffers

- ✓ In Tomasulo's algorithm, architecturally visible registers (x0, . . . r31 and f0, . . . f31) are contained in combination of register set and reservation stations.
- ✓ In register-renaming approach, an extended set of physical registers is used to hold both the architecturally visible registers as well as temporary values.
- ✓ extended registers replace most of the function of ROB and reservation stations; only a queue to ensure that instructions complete in order is needed.
- ✓ A renaming process maps the names of architectural registers to physical register numbers in the extended register set, allocating a new unused register for the destination.
- ✓ WAW and WAR hazards are avoided by renaming of destination register, and speculation recovery is handled because a physical register holding an instruction destination does not become architectural register until the instruction commits.

Speculation: Implementation Issues and Extensions

- ✓ Renaming map supplies physical register number that currently corresponds to specified architectural register, a function performed by the register status table in Tomasulo's algorithm.
- ✓ When an instruction commits, renaming table is permanently updated to indicate that a physical register corresponds to actual architectural register, effectively finalizing the update to the processor state.
- ✓ ROB is not necessary with register renaming, hardware must track instructions in a queue-like structure and update the renaming table in order.
- ✓ **An advantage of the renaming approach versus the ROB approach:** instruction commit is slightly simplified because it requires only two simple actions:
  - i. record that the mapping between an architectural register number and physical register number is no longer speculative,
  - ii. free up any physical registers being used to hold "older" value of architectural register. In a design with reservation stations, a station is freed up when instruction using it completes execution and a ROB entry is freed up when corresponding instruction commits.

Speculation: Implementation Issues and Extensions

- ✓ Both register renaming and reorder buffers continue to be used in high-end processors, which has the ability to have as many as 100 or more instructions.
- ✓ A strategy for instruction issue with register renaming similar to multiple issue with reorder buffers can be deployed, as follows:
  - i. Issue logic reserves enough physical registers for the entire issue bundle (four registers for a four-instruction bundle with at most one register result per instruction).
  - ii. Issue logic determines what dependences exist within the bundle. If a dependence does not exist within the bundle, register renaming structure is used to determine the physical register that holds result on which instruction depends. When no dependence, result is from an earlier issue bundle and the register renaming table will have correct register number.
  - iii. If an instruction depends on an instruction that is earlier in the bundle, then the pre-reserved physical register in which result will be placed is used to update the information for issuing instruction.

Speculation: Implementation Issues and Extensions

The Challenge of More Issues per Clock

✓ Figure 3.29 shows a six-instruction code sequence and what the issue step must do

Instr. #	Instruction	Physical register assigned or destination	Instruction with physical register numbers	Rename map changes
1	add x1,x2,x3	p32	add p32,p2,p3	**x1-> p32**
2	sub x1,x1,x2	p33	sub p33,p32,p2	**x1->p33**
3	add x2,x1,x2	p34	add p34,p33,x2	**x2->p34**
4	sub x1,x3,x2	p35	sub p35,p3,p34	**x1->p35**
5	add x1,x1,x2	p36	add p36,p35,p34	**x1->p36**
6	sub x1,x3,x1	p37	sub p37,p3,p36	**x1->p37**

Figure 3.29 An example of six instructions to be issued in the same clock cycle and what has to happen. The instructions are shown in program order: 1–6; they are, however, issued in 1 clock cycle! The notation pi is used to refer to a physical register; the contents of that register at any point is determined by the renaming map. For simplicity, we assume that the physical registers holding the architectural registers x1, x2, and x3 are initially p1, p2, and p3 (they could be any physical register). The instructions are issued with physical register numbers, as shown in column four. The rename map, which appears in the last column, shows how the map would change if the instructions were issued sequentially. The difficulty is that all this renaming and replacement of architectural registers by physical renaming registers happens effectively in 1 cycle, not sequentially. The issue logic must find all the dependences and "rewrite" the instruction in parallel.

Speculation: Implementation Issues and Extensions

How Much to Speculate:

- ✓ Advantage of speculation is its ability to uncover events that stall the pipeline such as cache misses.
- ✓ Disadvantage is that speculation is not free. It takes time,energy and recovery of incorrect speculation further reduces performance.
- ✓ To support higher instruction execution rate needed to benefit from speculation, processor must have additional resources which take silicon area and power.
- ✓ if speculation causes an exceptional event to occur such as a cache or translation lookaside buffer (TLB) miss.Potential for significant performance loss increases.
- ✓ most pipelines with speculation allow low-cost exceptional events to be handled in speculative mode.
- ✓ If an expensive exceptional event occurs such as a second-level cache miss or a TLB miss, processor will wait until the instruction causing the event is no longer speculative before handling the event.
- ✓ Degrade the performance of some programs.

Speculation: Implementation Issues and Extensions

Speculating Through Multiple Branches: Three different situations can benefit from speculating on multiple branches simultaneously:

(1) a very high branch frequency

(2) significant clustering of branches

(3) long delays in functional units.

- ✓ In the first two cases, achieving high performance mean multiple branches are speculated.
- ✓ Database programs and other integer computations exhibit these properties making speculation on multiple branches important.
- ✓ Long delays in functional units can raise the importance of speculating on multiple branches to avoid stalls from longer pipeline delays.

Speculation: Implementation Issues and Extensions

Speculation and the Challenge of Energy Efficiency: whenever speculation is wrong, it consumes excess energy in two ways:

i. Instructions that are speculated and whose results are not needed, generate excess work for the processor, wasting energy.

ii. Undoing speculation and restoring the state of the processor to continue execution at appropriate address consumes additional energy that would not be needed without speculation.

- ✓ speculation raise the power consumption and if we control speculation, it would be possible to measure the cost.
- ✓ if speculation lowers the execution time by more than it increases the average power consumption, then the total energy consumed may be less.

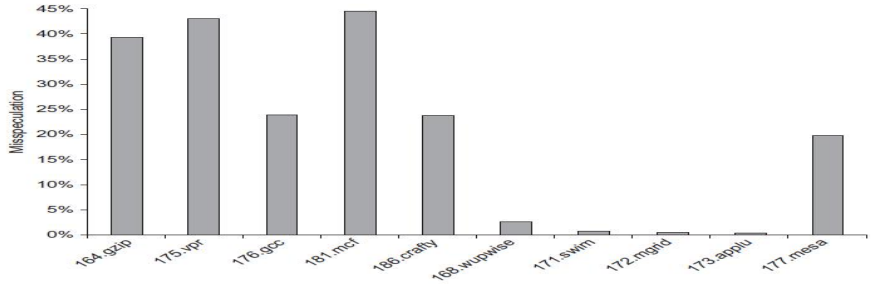

Figure 3.30 The fraction of instructions that are executed as a result of misspeculation is typically much higher for integer programs (the first five) versus FP programs (the last five).

Speculation: Implementation Issues and Extensions

Address Aliasing Prediction:

- ✓ Address aliasing prediction is a technique that predicts whether two stores or a load and store refer to the same memory address.
- ✓ If two such references do not refer to same address, then they may be safely interchanged.
- ✓ Address prediction relies on the ability of a speculative processor to recover after a misprediction;
- ✓ If the actual addresses that were predicted to be different turn out to be the same, processor simply restarts the sequence,
- ✓ Address value speculation has been used in several processors .
- ✓ Address prediction is a simple and restricted form of value prediction, which attempts to predict the value that will be produced by an instruction.
- ✓ Value prediction were highly accurate, eliminate data flow restrictions and achieve higher rates of ILP.

Cross Cutting Issues

Hardware versus Software Speculation:

a) Hardware-based speculation works better when control flow is unpredictable and when hardware-based branch prediction is superior to software-based branch prediction at compile time.

b) Hardware-based speculation maintains a completely precise exception model even for speculated instructions. Recent software-based approaches have added special support to allow this as well.

c) Hardware-based speculation does not require compensation or bookkeeping code, which is needed by software speculation mechanisms.

d) Compiler-based approaches benefit from the ability to see further in code sequence, resulting in better code scheduling than hardware driven approach.

e) Hardware-based speculation with dynamic scheduling does not require different code sequences to achieve good performance for different implementations of an architecture.

Cross Cutting Issues

Speculative Execution and the Memory System:

- ✓ processors that support speculative execution or conditional instructions possibly generate invalid addresses that would not occur without speculative execution.
- ✓ memory system must identify speculatively executed instructions and conditionally executed instructions and suppress the corresponding exception.
- ✓ cannot allow such instructions to cause the cache to stall on a miss because unnecessary stalls overwhelm the benefits of speculation.
- ✓ penalty of a cache miss that goes to DRAM is so large that speculated misses are handled only when the next level is on-chip cache (L2 or L3).

Multithreading: Exploiting Thread-Level Parallelism to Improve Uniprocessor Throughput

Multithreading allows multiple threads to share functional units of a single processor in an overlapping fashion.There are three main hardware approaches to multithreading:

1. Fine-grained multithreading: switches between threads on each clock cycle causing the execution of instructions from multiple threads to be interleaved. This interleaving is often done in a round-robin fashion skipping any threads that are stalled at that time.

Advantage of finegrained multithreading: it can hide the throughput losses that arise from both short and long stalls because instructions from other threads can be executed when one thread stalls, even if the stall is only for a few cycles.

Disadvantage of fine-grained multithreading: it slows down the execution of an individual thread because a thread that is ready to execute without stalls will be delayed by instructions from other threads.

increase in multithreaded throughput for a loss in the performance (as measured by latency) of a single thread.

Multithreading: Exploiting Thread-Level Parallelism to Improve Uniprocessor Throughput

2.Coarse-grained multithreading:

- ✓ switches threads only on costly stalls such as level two or three cache misses.
- ✓ instructions from other threads will be issued only when a thread encounters a costly stall,
- ✓ coarse-grained multithreading relieves the need to have thread-switching to slow down the execution of any one thread.
- ✓ **Coarse-grained multithreading suffers from a major drawback:** it is limited in its ability to overcome throughput losses from shorter stalls. This limitation arises from the pipeline start-up costs of coarse-grained multithreading.
- ✓ processor with coarse-grained multithreading issues instructions from a single thread, when a stall occurs the pipeline will see a bubble before the new thread begins executing.
- ✓ coarse-grained multithreading is useful for reducing the penalty of very high-cost stalls, where pipeline refill is negligible compared to stall time.

Multithreading: Exploiting Thread-Level Parallelism to Improve Uniprocessor Throughput

3. Simultaneous multithreading:

- ✓ variation on fine-grained multithreading that arises when fine-grained multithreading is implemented on top of a multiple-issue dynamically scheduled processor.
- ✓ SMT uses thread-level parallelism to hide long-latency events in a processor, increasing the usage of the functional units.
- ✓ register renaming and dynamic scheduling allow multiple instructions from independent threads to be executed without regard to the dependences among them;
- ✓ resolution of dependences can be handled by dynamic scheduling capability.

Figure 3.31 illustrates differences in a processor's ability to exploit resources of a superscalar for following processor configurations:

- ✓ A superscalar with no multithreading support
- ✓ A superscalar with coarse-grained multithreading
- ✓ A superscalar with fine-grained multithreading
- ✓ A superscalar with simultaneous multithreading

Multithreading: Exploiting Thread-Level Parallelism to Improve Uniprocessor Throughput

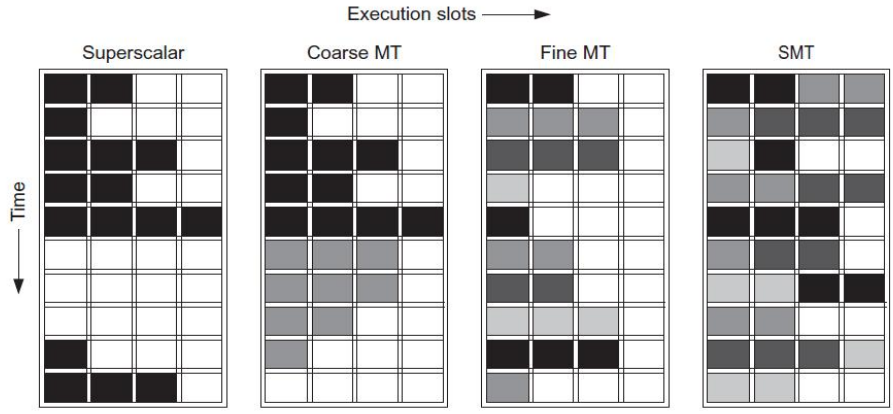

Figure 3.31 How four different approaches use the functional unit execution slots of a superscalar processor. The horizontal dimension represents the instruction execution capability in each clock cycle. The vertical dimension represents a sequence of clock cycles. An empty (white) box indicates that the corresponding execution slot is unused in that clock cycle. The shades of gray and black correspond to four different threads in the multithreading processors. Black is also used to indicate the occupied issue slots in the case of the superscalar without multithreading support. The Sun T1 and T2 (aka Niagara) processors are fine-grained, multithreaded processors, while the Intel Core i7 and IBM Power7 processors use SMT. The T2 has 8 threads, the Power7 has 4, and the Intel i7 has 2. In all existing SMTs, instructions issue from only one thread at a time. The difference in SMT is that the subsequent decision to execute an instruction is decoupled and could execute the operations coming from several different instructions in the same clock cycle.

Putting It All Together: Intel Core i7 6700 and ARM Cortex-A53

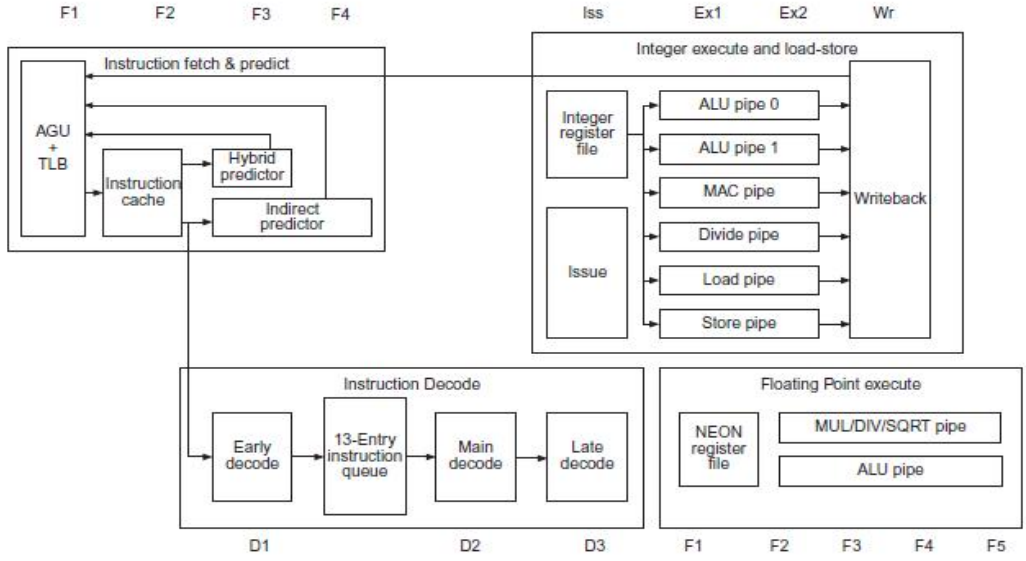

Figure 3.34 The basic structure of the A53 integer pipeline is 8 stages: F1 and F2 fetch the instruction, D1 and D2 do the basic decoding, and D3 decodes some more complex instructions and is overlapped with the first stage of the execution pipeline (ISS). After ISS, the Ex1, EX2, and WB stages complete the integer pipeline. Branches use four different predictors, depending on the type. The floating-point execution pipeline is 5 cycles deep, in addition to the 5 cycles needed for fetch and decode, yielding 10 stages in total.

Putting It All Together: Intel Core i7 6700 and ARM Cortex-A53

Four cycles of instruction fetch include an address generation unit that produces the next PC either by incrementing the last PC or from one of four predictors:

1) A single-entry branch target cache containing two instruction cache fetches. This target cache is checked during the first fetch cycle, if it hits then the next two instructions are supplied from target cache. In case of a hit and a correct prediction, branch is executed with no delay cycles.
2) A 3072-entry hybrid predictor used for all instructions that do not hit in the branch target cache, and operating during F3. Branches handled by this predictor incur a 2-cycle delay.
3) A 256-entry indirect branch predictor that operates during F4; branches predicted by this predictor incur a three-cycle delay when predicted correctly.
4) A 8-deep return stack, operating during F4 and incurring a three-cycle delay.

Putting It All Together: Intel Core i7 6700 and ARM Cortex-A53

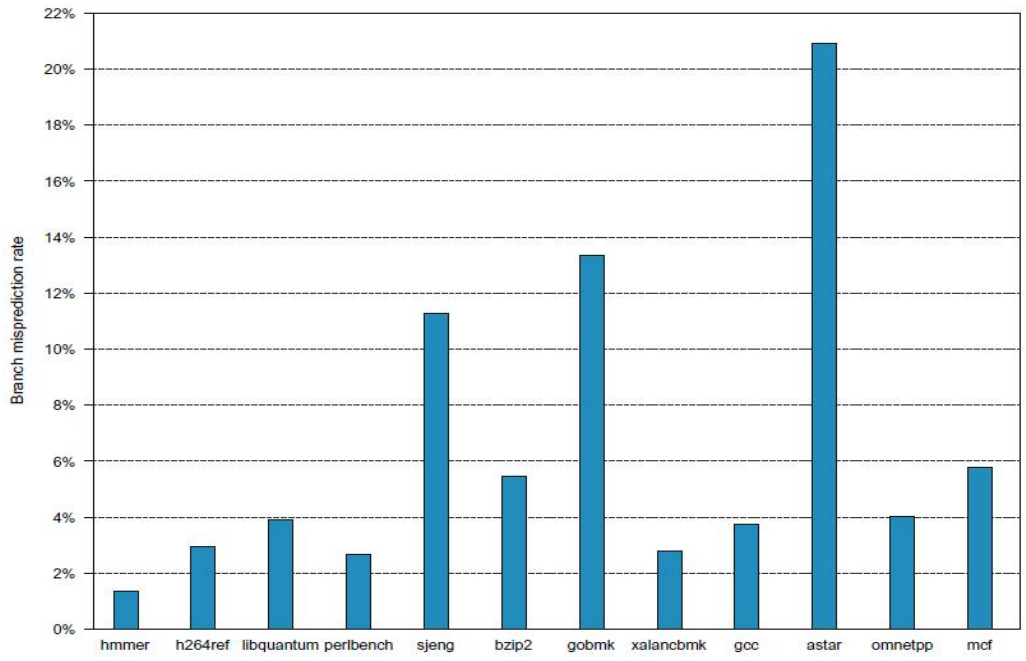

Figure 3.35 Misprediction rate of the A53 branch predictor for SPECint2006.

Putting It All Together: Intel Core i7 6700 and ARM Cortex-A53

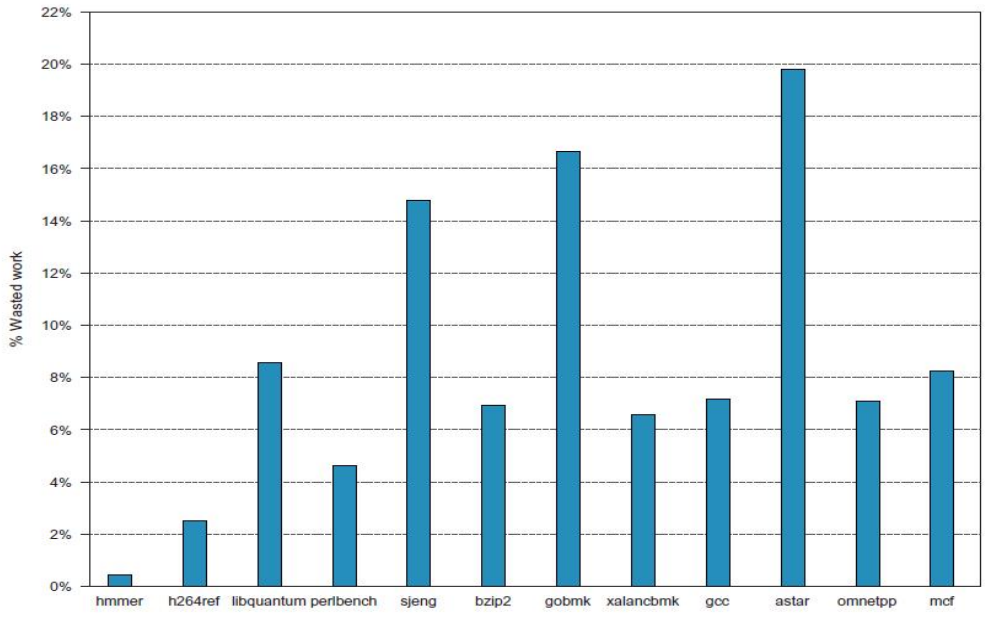

Figure 3.36 Wasted work due to branch misprediction on the A53. Because the A53 is an in-order machine, the amount of wasted work depends on a variety of factors, including data dependences and cache misses, both of which will cause a stall.

Putting It All Together: Intel Core i7 6700 and ARM Cortex-A53

Performance of A53 Pipeline A53 has an ideal CPI of 0.5 because of its dual-issue structure. Pipeline stalls can arise from three sources:

1) Functional hazards occur because two adjacent instructions selected for issue simultaneously use the same functional pipeline. A53 is statically scheduled, compiler should try to avoid such conflicts. When such instructions appear sequentially, they will be serialized at the beginning of the execution pipeline.

2) Data hazards are detected in the pipeline and may stall both instructions . Compiler should try to prevent such stalls.

3) Control hazards arise only when branches are mispredicted.

Putting It All Together: Intel Core i7 6700 and ARM Cortex-A53

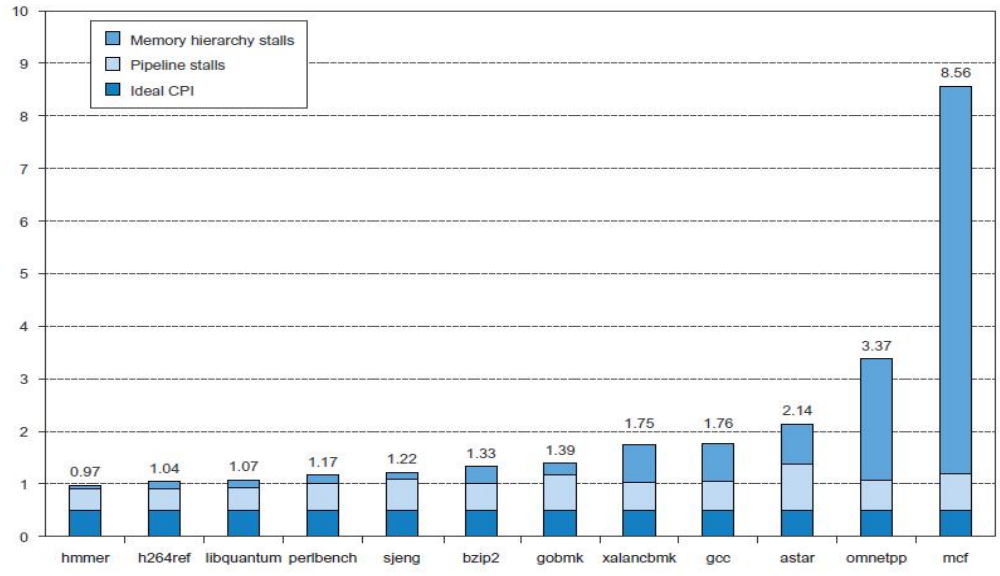

Figure 3.37 The estimated composition of the CPI on the ARM A53 shows that pipeline stalls are significant but are outweighed by cache misses in the poorest performing programs. This estimate is obtained by using the L1 and L2 miss rates and penalties to compute the L1 and L2 generated stalls per instruction. These are subtracted from the CPI measured by a detailed simulator to obtain the pipeline stalls. Pipeline stalls include all three hazards.

Intel Core i7

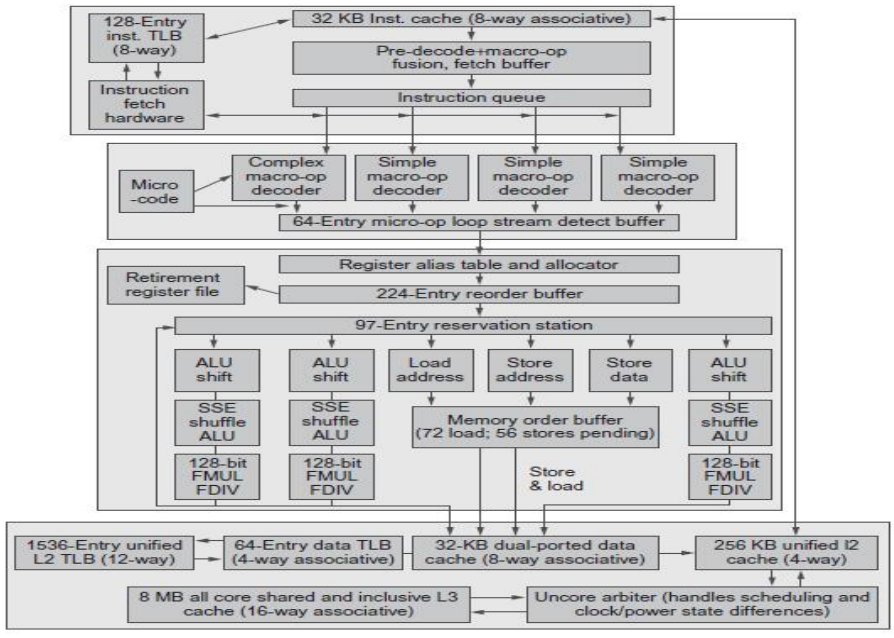

Figure 3.38 The Intel Core i7 pipeline structure shown with the memory system components. The total pipeline depth is 14 stages, with branch mispredictions typically costing 17 cycles, with the extra few cycles likely due to the time to reset the branch predictor. The six independent functional units can each begin execution of a ready micro-op in the same cycle. Up to four micro-ops can be processed in the register renaming table.

Intel Core i7

1. Instruction fetch: processor uses a sophisticated multilevel branch predictor to achieve a balance between speed and prediction accuracy. . Mispredictions cause a penalty of about 17 cycles.

2. 6 bytes are placed in the predecode instruction buffer: Macro-op fusion takes instruction combinations such as compare followed by a branch and fuses them into a single operation, which can issue and dispatch as one instruction.

3. Micro-op decode: Individual x86 instructions are translated into micro-ops. Micro-ops are simple RISC-V instructions that can be executed directly by pipeline. Three of decoders handle x86 instructions that translate directly into one micro-op.

4.Micro-op buffer preforms loop stream detection and microfusion: If there is a small sequence of instructions (less than 64 instructions) that comprises a loop, loop stream detector directly issue micro-ops from buffer.

Microfusion combines instruction pairs such as ALU operation and a dependent store and issues them to a single reservation station increasing usage of buffer.

Core i7 has much larger number of reorder buffer entries.

Intel Core i7

5. Perform the basic instruction issue: register location in the register tables, renaming the registers, allocating a reorder buffer entry, and fetching any results from the registers or reorder buffer before sending the micro-ops to the reservation stations. Up to four micro-ops can be processed every clock cycle.

6. The i7 uses a centralized reservation station shared by six functional units. Up to six micro-ops may be dispatched to the functional units every clock cycle.

7. Micro-ops are executed by the individual function units and then results are sent back to any waiting reservation station as well as to the register retirement unit, where they will update the register state once it is known that the instruction is no longer speculative. The entry corresponding to the instruction in the reorder buffer is marked as complete.

8. When one or more instructions at the head of the reorder buffer marked as complete, the pending writes in the register retirement unit are executed, and the instructions are removed from the reorder buffer.

Performance of Intel Core i7

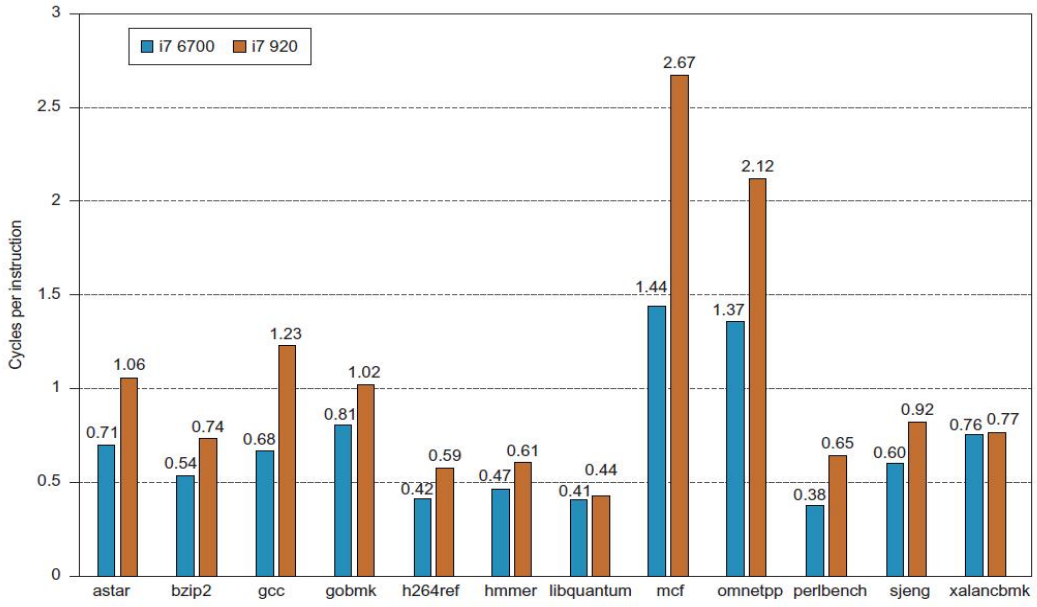

Figure 3.40 The CPI for the SPECCPUint2006 benchmarks on the i7 6700 and the i7 920. The data in this section were collected by Professor Lu Peng and PhD student Qun Liu, both of Louisiana State University.

Performance of Intel Core i7

Benchmark	CPI ratio (920/6700)	Branch mispredict ratio (920/6700)	L1 demand miss ratio (920/6700)
ASTAR	1.51	1.53	2.14
GCC	1.82	2.54	1.82
MCF	1.85	1.27	1.71
OMNETPP	1.55	1.48	1.96
PERLBENCH	1.70	2.11	1.78

Figure 3.41 An analysis of the five integer benchmarks with the largest performance gap between the i7 6700 and 920. These five benchmarks show an improvement in the branch prediction rate and a reduction in the L1 demand miss rate.

Review of Memory Hierarchy

Introduction to Cache

- ✓ Cache is highest or first level of the memory hierarchy and principle of locality applies at many levels.
- ✓ When the processor finds a requested data item in the cache, it is called a cache hit. When the processor does not find a data item it needs in the cache, a cache miss occurs.
- ✓ **Block or Line run:** A fixed-size collection of data containing the requested word, called a block or line run, is retrieved from the main memory and placed into the cache.
- ✓ **Temporal locality:** tells us that we are likely to need this word again in the near future, so it is useful to place it in the cache where it can be accessed quickly.
- ✓ **spatial locality:** there is a high probability that the other data in the block will be needed soon.
- ✓ Time required for the cache miss depends on both the latency and bandwidth of the memory.
- ✓ **Latency:** determines the time to retrieve the first word of the block
- ✓ **Bandwidth:** determines the time to retrieve the rest of this block.

Introduction to Cache

- ✓ A cache miss is handled by hardware and causes processors using in-order execution to pause or stall until data are available.
- ✓ With out-of-order execution, an instruction using the result must wait, but other instructions may proceed during the miss.
- ✓ **Page:** Virtual address space is broken into fixed-size blocks, called pages.
- ✓ Each page resides either in main memory or on disk. When processor references an item within a page that is not present in the cache or main memory, a fault occurs and the entire page is moved from the disk to main memory.
- ✓ Processor switches to some other task while the disk access occurs.

Introduction to Cache

- ✓ Figure B.1 shows the range of sizes and access times of each level in the memory hierarchy for computers ranging from high-end desktops to low-end servers.

Level	1	2	3	4
Name	Registers	Cache	Main memory	Disk storage
Typical size	<4 KiB	32 KiB to 8 MiB	<1 TB	>1 TB
Implementation technology	Custom memory with multiple ports, CMOS	On-chip CMOS SRAM	CMOS DRAM	Magnetic disk or FLASH
Access time (ns)	0.1–0.2	0.5–10	30–150	5,000,000
Bandwidth (MiB/sec)	1,000,000–10,000,000	20,000–50,000	10,000–30,000	100–1000
Managed by	Compiler	Hardware	Operating system	Operating system
Backed by	Cache	Main memory	Disk or FLASH	Other disks and DVD

Figure B.1 The typical levels in the hierarchy slow down and get larger as we move away from the processor for a large workstation or small server. Embedded computers might have no disk storage and much smaller memories and caches. Increasingly, FLASH is replacing magnetic disks, at least for first level file storage. The access times increase as we move to lower levels of the hierarchy, which makes it feasible to manage the transfer less responsively. The implementation technology shows the typical technology used for these functions. The access time is given in nanoseconds for typical values in 2017; these times will decrease over time. Bandwidth is given in megabytes per second between levels in the memory hierarchy. Bandwidth for disk/FLASH storage includes both the media and the buffered interfaces.

Cache Performance Review

- ✓ **Memory Stall Cycles:** Number of cycles during which the processor is stalled waiting for a memory access, which we call the memory stall cycles.
- ✓ Performance is the product of clock cycle time and the sum of the processor cycles and the memory stall cycles:

CPUexecution time = (CPUclock cycles +Memory stall cycles) * clock cycle time

- ✓ This equation assumes that the CPU clock cycles include the time to handle a cache hit and that the processor is stalled during a cache miss.
- ✓ The number of memory stall cycles depends on both the number of misses and the cost per miss, which is called the miss penalty:

$$\begin{aligned}\text{Memory stall cycles} &= \text{Number of misses} \times \text{Miss penalty}\\ &= \text{IC} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}\\ &= \text{IC} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss rate} \times \text{Miss penalty}\end{aligned}$$

Cache Performance Review

- ✓ The preceding formula is an approximation because the miss rates and miss penalties are often different for reads and writes. Memory stall clock cycles could then be defined in terms of the number of memory accesses per instruction, miss penalty (in clock cycles) for reads and writes, and miss rate for reads and writes:

$$\begin{aligned}\text{Memory stall clock cycles} = {} & \text{IC} \times \text{Reads per instruction} \times \text{Read miss rate} \times \text{Read miss penalty}\\ & + \text{IC} \times \text{Writes per instruction} \times \text{Write miss rate} \times \text{Write miss penalty}\end{aligned}$$

We usually simplify the complete formula by combining the reads and writes and finding the average miss rates and miss penalty for reads *and* writes:

$$\text{Memory stall clock cycles} = \text{IC} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss rate} \times \text{Miss penalty}$$

Some designers prefer measuring miss rate as *misses per instruction* rather than misses per memory reference. These two are related:

$$\frac{\text{Misses}}{\text{Instruction}} = \frac{\text{Miss rate} \times \text{Memory accesses}}{\text{Instruction count}} = \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}}$$

Cache Performance Review

Example Assume we have a computer where the cycles per instruction (CPI) is 1.0 when all memory accesses hit in the cache. The only data accesses are loads and stores, and these total 50% of the instructions. If the miss penalty is 50 clock cycles and the miss rate is 1%, how much faster would the computer be if all instructions were cache hits?

Answer First compute the performance for the computer that always hits:

$$\begin{aligned}\text{CPU execution time} &= (\text{CPU clock cycles} + \text{Memory stall cycles}) \times \text{Clock cycle}\\ &= (\text{IC} \times \text{CPI} + 0) \times \text{Clock cycle}\\ &= \text{IC} \times 1.0 \times \text{Clock cycle}\end{aligned}$$

Now for the computer with the real cache, first we compute memory stall cycles:

$$\begin{aligned}\text{Memory stall cycles} &= \text{IC} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss rate} \times \text{Miss penalty}\\ &= \text{IC} \times (1 + 0.5) \times 0.01 \times 50\\ &= \text{IC} \times 0.75\end{aligned}$$

where the middle term (1 + 0.5) represents one instruction access and 0.5 data accesses per instruction. The total performance is thus

$$\begin{aligned}\text{CPU execution time}_{\text{cache}} &= (\text{IC} \times 1.0 + \text{IC} \times 0.75) \times \text{Clock cycle}\\ &= 1.75 \times \text{IC} \times \text{Clock cycle}\end{aligned}$$

The performance ratio is the inverse of the execution times:

$$\begin{aligned}\frac{\text{CPU execution time}_{\text{cache}}}{\text{CPU execution time}} &= \frac{1.75 \times \text{IC} \times \text{Clock cycle}}{1.0 \times \text{IC} \times \text{Clock cycle}}\\ &= 1.75\end{aligned}$$

The computer with no cache misses is 1.75 times faster.

Cache Performance Review

Example To show equivalency between the two miss rate equations, let's redo the preceding example, this time assuming a miss rate per 1000 instructions of 30. What is memory stall time in terms of instruction count?

Answer Recomputing the memory stall cycles:

$$\begin{aligned}\text{Memory stall cycles} &= \text{Number of misses} \times \text{Miss penalty}\\ &= \text{IC} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}\\ &= \text{IC}/1000 \times \frac{\text{Misses}}{\text{Intruction} \times 1000} \times \text{Miss penalty}\\ &= \text{IC}/1000 \times 30 \times 25\\ &= \text{IC}/1000 \times 750\\ &= \text{IC} \times 0.75\end{aligned}$$

Four Memory Hierarchy Questions

We continue our introduction to caches by answering the four common questions for the first level of the memory hierarchy:

Q1: Where can a block be placed in the upper level? (block placement)

Q2: How is a block found if it is in the upper level? (block identification)

Q3: Which block should be replaced on a miss? (block replacement)

Q4: What happens on a write? (write strategy)

Four Memory Hierarchy Questions

Q1: Where Can a Block be Placed in a Cache?

Figure B.2 shows restrictions where a block is placed create three categories of cache organization:

Direct Mapped: If each block has only one place it can appear in the cache, the cache is said to be direct mapped. The mapping is usually:

$$(\text{Block address}) \bmod (\text{Number of blocks in cache})$$

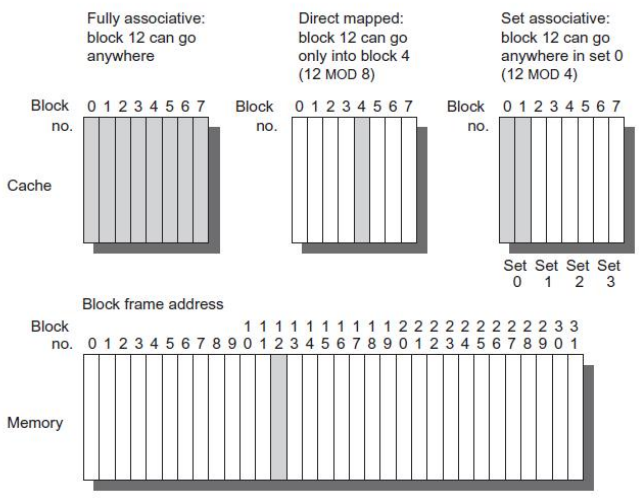

Figure B.2 **This example cache has eight block frames and memory has 32 blocks.** The three options for caches are shown left to right. In fully associative, block 12 from the lower level can go into any of the eight block frames of the cache. With direct mapped, block 12 can only be placed into block frame 4 (12 modulo 8). Set associative, which has some of both features, allows the block to be placed anywhere in set 0 (12 modulo 4). With two blocks per set, this means block 12 can be placed either in block 0 or in block 1 of the cache. Real caches contain thousands of block frames, and real memories contain millions of blocks. The set associative organization has four sets with two blocks per set, called *two-way set associative*. Assume that there is nothing in the cache and that the block address in question identifies lower-level block 12.

Four Memory Hierarchy Questions

Q1: Where Can a Block be Placed in a Cache?

Fully Associative: If a block can be placed anywhere in the cache, the cache is said to be fully associative.

Set Associative:

- ✓ If a block can be placed in a restricted set of places in the cache, the cache is set associative.
- ✓ A set is a group of blocks in the cache.
- ✓ A block is first mapped ona set, and then the block can be placed anywhere within that set. The set is usually chosen by bit selection: .

(*Block address*) MOD (*Number of sets in cache*

- ✓ **n-way set Associative:** If there are n blocks in a set, the cache placement is called n-way set associative.

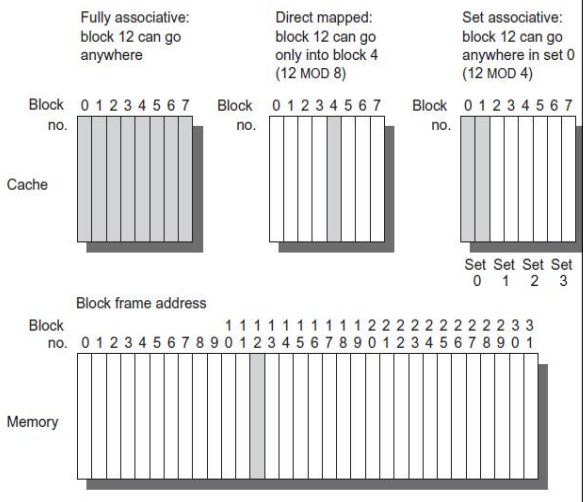

Figure B.2 This example cache has eight block frames and memory has 32 blocks. The three options for caches are shown left to right. In fully associative, block 12 from the lower level can go into any of the eight block frames of the cache. With direct mapped, block 12 can only be placed into block frame 4 (12 modulo 8). Set associative, which has some of both features, allows the block to be placed anywhere in set 0 (12 modulo 4). With two blocks per set, this means block 12 can be placed either in block 0 or in block 1 of the cache. Real caches contain thousands of block frames, and real memories contain millions of blocks. The set associative organization has four sets with two blocks per set, called *two-way set associative*. Assume that there is nothing in the cache and that the block address in question identifies lower-level block 12.

Four Memory Hierarchy Questions

- ✓ Caches from direct mapped to fully associative is a time of levels of set associativity.
- ✓ Direct mapped is one-way set associative,
- ✓ **m-way Set Aasociative:** fully associative cache with m blocks could be called “m-way set associative.”
- ✓ direct mapped can be thought of as having m sets, and fully associative as having one set.
- ✓ The vast majority of processor caches today are direct mapped, two-way set associative, or four-way set associative.

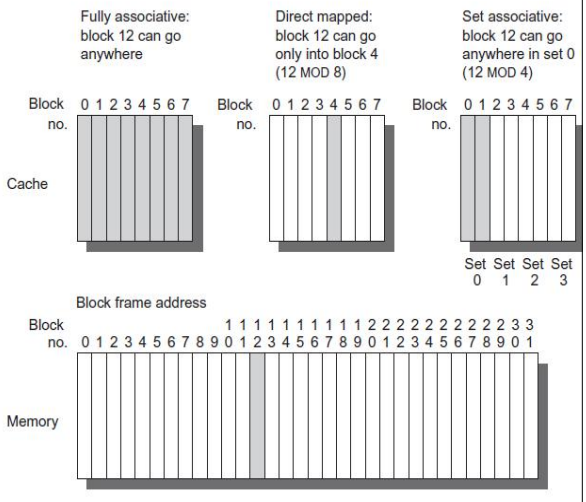

Figure B.2 This example cache has eight block frames and memory has 32 blocks. The three options for caches are shown left to right. In fully associative, block 12 from the lower level can go into any of the eight block frames of the cache. With direct mapped, block 12 can only be placed into block frame 4 (12 modulo 8). Set associative, which has some of both features, allows the block to be placed anywhere in set 0 (12 modulo 4). With two blocks per set, this means block 12 can be placed either in block 0 or in block 1 of the cache. Real caches contain thousands of block frames, and real memories contain millions of blocks. The set associative organization has four sets with two blocks per set, called *two-way set associative*. Assume that there is nothing in the cache and that the block address in question identifies lower-level block 12.

Four Memory Hierarchy Questions

Q2: How Is a Block Found If It Is in the Cache?

- ✓ Caches have an address tag on each block frame that gives the block address.
- ✓ Tag of every cache block contain the desired information is checked to see if it matches the block address from the processor.
- ✓ all possible tags are searched in parallel because speed is critical.
- ✓ To add a valid bit to the tag indicates this entry contains a valid address. If the bit is not set, there cannot be a match on this address.
- ✓ Figure B.3 shows how an address is divided.
- ✓ First division is between the **block address and the block offset** .
- ✓ **Block frame address** can be further divided into the tag field and the index field.
- ✓ **Block offset field** selects the desired data from the block, the index field selects the set, and the tag field is compared against it for a hit.

Four Memory Hierarchy Questions

Comparison could be made on more of the address than the tag, there is no need because of the following:

1) offset should not be used in the comparison, because the entire block is present or not, and hence all block offsets result in a match.

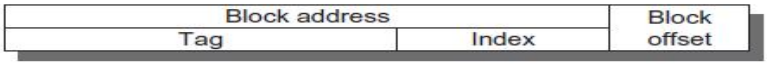

Figure B.3 The three portions of an address in a set associative or direct-mapped cache. The tag is used to check all the blocks in the set, and the index is used to select the set. The block offset is the address of the desired data within the block. Fully associative caches have no index field.

2) Checking the index is redundant, because it was used to select the set to be checked. An address stored in set 0, for example, must have 0 in the index field or it couldn't be stored in set 0; set 1 must have an index value of 1; and so on. This optimization saves hardware and power by reducing the width of memory size for the cache tag.

If total cache size is kept the same, increasing associativity increases the number of blocks per set, thereby decreasing the size of index and increasing the size of tag.

Four Memory Hierarchy Questions

Q3: Which Block Should be Replaced on a Cache Miss?

- ✓ When a miss occurs, cache controller must select a block to be replaced with desired data.
- ✓ Benefit of direct-mapped placement is that only one block frame is checked for a hit, and only that block can be replaced.
- ✓ With fully associative or set associative placement, there are many blocks to choose from on a miss.

There are three strategies employed for selecting which block to replace:

Random: To spread allocation uniformly, candidate blocks are randomly selected. Some systems generate pseudorandom block numbers to get reproducible behavior, which is particularly useful when debugging hardware.

Least recently used (LRU): To reduce the chance of throwing out information that will be needed soon, accesses to blocks are recorded. Relying on the past to predict the future, the block replaced is the one that has been unused for the longest time.

LRU relies on locality: if recently used blocks are likely to be used again, then a good candidate for disposal is the least recently used block.

First in, first out (FIFO): Replacement is based on FIFO order.

Four Memory Hierarchy Questions

- ✓ Random replacement is built in hardware. As the number of blocks increases, LRU becomes increasingly expensive and is usually only approximated.
- ✓ pseudo- LRU has a set of bits for each set in the cache with each bit corresponding to a single way (a way is bank in a set associative cache; there are four ways in four-way set associative cache) in the cache.
- ✓ When a set is accessed, the bit corresponding to the way containing the desired block is turned on; they are reset with the exception of the most recently turned on bit.
- ✓ When a block must be replaced, the processor chooses a block from the way whose bit is turned off, often randomly if more than one choice is available.
- ✓ This approximates LRU, because the block that is replaced will not have been accessed since the last time that all the blocks in the set were accessed.

Four Memory Hierarchy Questions

Figure B.4 shows the difference in miss rates between LRU, random, and FIFO replacement.

	Associativity								
	Two-way			Four-way			Eight-way		
Size	LRU	Random	FIFO	LRU	Random	FIFO	LRU	Random	FIFO
16 KiB	114.1	117.3	115.5	111.7	115.1	113.3	109.0	111.8	110.4
64 KiB	103.4	104.3	103.9	102.4	102.3	103.1	99.7	100.5	100.3
256 KiB	92.2	92.1	92.5	92.1	92.1	92.5	92.1	92.1	92.5

Figure B.4 Data cache misses per 1000 instructions comparing least recently used, random, and first in, first out replacement for several sizes and associativities. There is little difference between LRU and random for the largest size cache, with LRU outperforming the others for smaller caches. FIFO generally outperforms random in the smaller cache sizes. These data were collected for a block size of 64 bytes for the Alpha architecture using 10 SPEC2000 benchmarks. Five are from SPECint2000 (gap, gcc, gzip, mcf, and perl) and five are from SPECfp2000 (applu, art, equake, lucas, and swim). We will use this computer and these benchmarks in most figures in this appendix.

Four Memory Hierarchy Questions

Q4: What Happens on a Write?

- ✓ Blocks can be read from cache at the same time tag is read and compared, so the block read begins as soon as the block address is available.
- ✓ If the read is a hit, requested part of the block is passed on to the processor immediately. If it is a miss no harm except more power in desktop and server computers.
- ✓ Modifying a block cannot begin until the tag is checked to see if the address is a hit. Tag checking cannot occur in parallel, writes usually take longer than reads.
- ✓ Another complexity with Write is that the processor also specifies the size of the write between 1 and 8 bytes; only that portion of a block can be changed.
- ✓ There are two basic options when writing to the cache:

Write through: information is written to both the block in the cache and to the block in the lower-level memory.

Write back: information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.

Four Memory Hierarchy Questions

- ✓ To reduce the frequency of writing back blocks on replacement, a feature called the dirty bit is used.
- ✓ Dirty bit indicates whether the block is dirty (modified while in the cache) or clean (not modified). If it is clean, block is not written back on a miss, because identical information to the cache is found in lower levels.
- ✓ Both write back and write through have their advantages:
  - i. With write back, writes occur at the speed of cache memory and multiple writes within a block require only one write to the lower-level memory.
  - ii. write back uses less memory bandwidth, making write back attractive in multiprocessors.
  - iii. write back uses rest of the memory hierarchy and memory interconnect less than write through, it also saves power making it attractive for embedded applications.
  - iv. Write through is easier to implement than write back. The cache is always clean.
  - v. Write through also has the advantage that the next lower level has the most current copy of the data, which simplifies data coherency.

Four Memory Hierarchy Questions

- ✓ When the processor must wait for writes to complete during write through, the processor is said to write stall.
- ✓ To reduce write stalls is a write buffer, which allows the processor to continue as soon as the data are written to the buffer, thereby overlapping processor execution with memory updating.

Data are not needed on a write, there are two options on a write miss:

1) **Write allocate:** A write-allocate cache means that upon a write-miss, the relevant data is loaded from memory into the cache and further writes will operate on the cached data.
2) **No-write allocate:** A write-no-allocate cache means that upon a write-miss, the write occurs on the data in main memory with no loading to cache.**h**

Four Memory Hierarchy Questions

Example Assume a fully associative write-back cache with many cache entries that starts empty. Following is a sequence of five memory operations (the address is in square brackets):

```
Write Mem[100];
Write Mem[100];
Read  Mem[200];
Write Mem[200];
Write Mem[100].
```

What are the number of hits and misses when using no-write allocate versus write allocate?

Answer For no-write allocate, the address 100 is not in the cache, and there is no allocation on write, so the first two writes will result in misses. Address 200 is also not in the cache, so the read is also a miss. The subsequent write to address 200 is a hit. The last write to 100 is still a miss. The result for no-write allocate is four misses and one hit.

For write allocate, the first accesses to 100 and 200 are misses, and the rest are hits because 100 and 200 are both found in the cache. Thus, the result for write allocate is two misses and three hits.

Four Memory Hierarchy Questions

An Example: The Opteron Data Cache

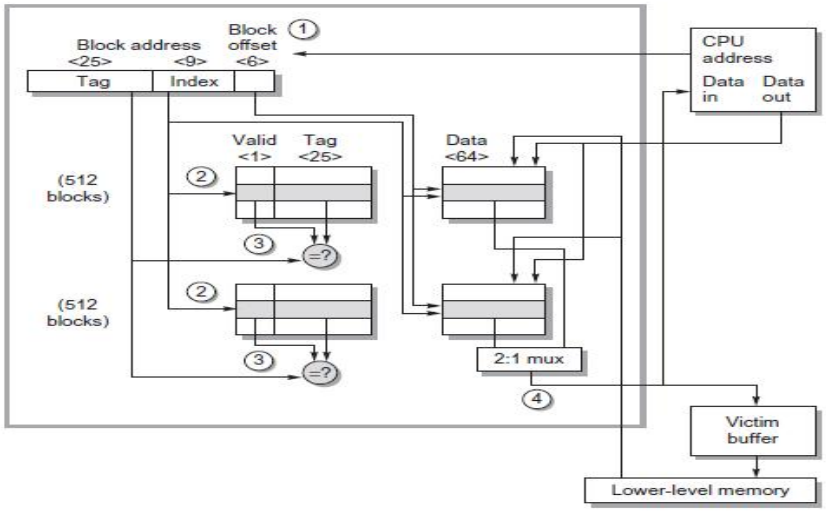

Figure B.5 The organization of the data cache in the Opteron microprocessor. The 64 KiB cache is two-way set associative with 64-byte blocks. The 9-bit index selects among 512 sets. The four steps of a read hit, shown as circled numbers in order of occurrence, label this organization. Three bits of the block offset join the index to supply the RAM address to select the proper 8 bytes. Thus, the cache holds two groups of 4096 64-bit words, with each group containing half of the 512 sets. Although not exercised in this example, the line from lower-level memory to the cache is used on a miss to load the cache. The size of address leaving the processor is 40 bits because it is a physical address and not a virtual address. Figure B.24 on page B-47 explains how the Opteron maps from virtual to physical for a cache access.

Four Memory Hierarchy Questions

An Example: The Opteron Data Cache

- ✓ Figure B.5 shows the organization of the data cache in the AMD Opteron microprocessor.
- ✓ Cache contains 65,536 (64 K) bytes of data in 64-byte blocks with two-way set associative placement, LRUreplacement, write back, and write allocate on a write miss. Let's trace a cache hit through the steps of a hit.
- ✓ 48-bit virtual address of cache used for tag comparison, which is simultaneously translated into a 40-bit physical address.
- ✓ Physical address coming into the cache is divided into two fields: the 34-bit block address and the 6-bit block offset (64=2^6 and 34+6=40).

Step1:

- ✓ Block address is further divided into an address tag and cache index. Step1 shows this division.
- ✓ Cache index selects the tag to be tested to see if the desired block is in the cache.
- ✓ Size of the index depends on cache size, block size, and set associativity.

Four Memory Hierarchy Questions

Step 2:

- ✓ set associativity is set to two, and we calculate the index as follows:

$$2^{\text{Index}} = \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativity}} = \frac{65,536}{64 \times 2} = 512 = 2^9$$

- ✓ the index is 9 bits wide, and the tag is 34-9 or 25 bits wide.
- ✓ to organize the data portion of the cache memory 8 bytes wide, which is the data word of the 64- bit Opteron processor.
- ✓ 3 more bits from the block offset are used to index the proper 8 bytes.

Step 3:

- ✓ After reading two tags from cache, they are compared with tag portion of the block address from processor.
- ✓ Valid bit must be set or else the results of the comparison are ignored.

Step 4:

- ✓ Assuming one tag does match, signal is sent to processor to load the proper data from the cache by using input from a 2:1 multiplexor.

Allows 2 clock cycles for these four steps, so the instructions in the following 2 clock cycles would wait if they tried to use the result of the load.

Four Memory Hierarchy Questions

Figure B.6 shows that instruction caches have lower miss rates than data caches. Separating instructions and data removes misses due to conflicts between instruction blocks and data blocks, but the split also fixes the cache space devoted to each type.

Size (KiB)	Instruction cache	Data cache	Unified cache
8	8.16	44.0	63.0
16	3.82	40.9	51.0
32	1.36	38.4	43.3
64	0.61	36.9	39.4
128	0.30	35.3	36.2
256	0.02	32.6	32.9

Figure B.6 Miss per 1000 instructions for instruction, data, and unified caches of different sizes. The percentage of instruction references is about 74%. The data are for two-way associative caches with 64-byte blocks for the same computer and benchmarks as Figure B.4.

Cache Performance

- ✓ Corresponding temptation for evaluating memory hierarchy performance is to concentrate on miss rate because it too, is independent of the speed of the hardware.
- ✓ A better measure of memory hierarchy performance is the average memory access time:

Average memory access time=Hit time + Miss rate x Miss penalty

Cache Performance

Example Which has the lower miss rate: a 16 KiB instruction cache with a 16 KiB data cache or a 32 KiB unified cache? Use the miss rates in Figure B.6 to help calculate the correct answer, assuming 36% of the instructions are data transfer instructions. Assume a hit takes 1 clock cycle and the miss penalty is 100 clock cycles. A load or store hit takes 1 extra clock cycle on a unified cache if there is only one cache port to satisfy two simultaneous requests. Using the pipelining terminology of Chapter 3, the unified cache leads to a structural hazard. What is the average memory access time in each case? Assume write-through caches with a write buffer and ignore stalls due to the write buffer.

Answer First let's convert misses per 1000 instructions into miss rates. Solving the preceding general formula, the miss rate is

$$\text{Miss rate} = \frac{\frac{\text{Misses}}{1000\ \text{Instructions}} / 1000}{\frac{\text{Memory accesses}}{\text{Instruction}}}$$

Because every instruction access has exactly one memory access to fetch the instruction, the instruction miss rate is

$$\text{Miss rate}_{16\ \text{KB instruction}} = \frac{3.82/1000}{1.00} = 0.004$$

Because 36% of the instructions are data transfers, the data miss rate is

$$\text{Miss rate}_{16\ \text{KB data}} = \frac{40.9/1000}{0.36} = 0.114$$

The unified miss rate needs to account for instruction and data accesses:

$$\text{Miss rate}_{32\ \text{KB unified}} = \frac{43.3/1000}{1.00 + 0.36} = 0.0318$$

Figure B6

Cache Performance

As stated herein, about 74% of the memory accesses are instruction references. Thus, the overall miss rate for the split caches is

$$(74\% \times 0.004) + (26\% \times 0.114) = 0.0326$$

Thus, a 32 KiB unified cache has a slightly lower effective miss rate than two 16 KiB caches.

The average memory access time formula can be divided into instruction and data accesses:

$$\begin{aligned}&\text{Average memory access time}\\ &= \%\,\text{instructions} \times (\text{Hit time} + \text{Instruction miss rate} \times \text{Miss penalty})\\ &\quad + \%\,\text{data} \times (\text{Hit time} + \text{Data miss rate} \times \text{Miss penalty})\end{aligned}$$

Therefore, the time for each organization is

$$\begin{aligned}&\text{Average memory access time}_{\text{split}}\\ &= 74\% \times (1 + 0.004 \times 200) + 26\% \times (1 + 0.114 \times 200)\\ &= (74\% \times 1.80) + (26\% \times 23.80) = 1.332 + 6.188 = 7.52\\ &\text{Average memory access time}_{\text{unified}}\\ &= 74\% \times (1 + 0.0318 \times 200) + 26\% \times (1 + 1 + 0.0318 \times 200)\\ &= (74\% \times 7.36) + (26\% \times 8.36) = 5.446 + 2.174 = 7.62\end{aligned}$$

Hence, the split caches in this example—which offer two memory ports per clock cycle, thereby avoiding the structural hazard—have a better average memory access time than the single-ported unified cache despite having a worse effective miss rate.

Cache Performance

Average Memory Access Time and Processor Performance Cache Performance:

- ✓ Designers often assume that all memory stalls are due to cache misses, because the memory hierarchy typically dominates other reasons for stalls.

$$\text{CPU time} = (\text{CPU execution clock cycles} + \text{Memory stall clock cycles}) \times \text{Clock cycle time}$$

- ✓ Processor stalls during misses and the memory stall time is strongly correlated to average memory access time.

Cache Performance

Example Let's use an in-order execution computer for the first example. Assume that the cache miss penalty is 200 clock cycles, and all instructions usually take 1.0 clock cycles (ignoring memory stalls). Assume that the average miss rate is 2%, there is an average of 1.5 memory references per instruction, and the average number of cache misses per 1000 instructions is 30. What is the impact on performance when behavior of the cache is included? Calculate the impact using both misses per instruction and miss rate.

Answer

$$\text{CPU time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Memory stall clock cycles}}{\text{Instruction}}\right) \times \text{Clock cycle time}$$

The performance, including cache misses, is

$$\begin{aligned}\text{CPU time}_{\text{with cache}} &= \text{IC} \times [1.0 + (30/1000 \times 200)] \times \text{Clock cycle time} \\ &= \text{IC} \times 7.00 \times \text{Clock cycle time}\end{aligned}$$

Now calculating performance using miss rate:

$$\text{CPU time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss penalty}\right) \times \text{Clock cycle time}$$

$$\begin{aligned}\text{CPU time}_{\text{with cache}} &= \text{IC} \times [1.0 + (1.5 \times 2\% \times 200)] \times \text{Clock cycle time} \\ &= \text{IC} \times 7.00 \times \text{Clock cycle time}\end{aligned}$$

The clock cycle time and instruction count are the same, with or without a cache. Thus, CPU time increases sevenfold, with CPI from 1.00 for a "perfect cache" to 7.00 with a cache that can miss. Without any memory hierarchy at all the CPI would increase again to $1.0 + 200 \times 1.5$ or 301—a factor of more than 40 times longer than a system with a cache!

Cache Performance

Example What is the impact of two different cache organizations on the performance of a processor? Assume that the CPI with a perfect cache is 1.0, the clock cycle time is 0.35 ns, there are 1.4 memory references per instruction, the size of both caches is 128 KiB, and both have a block size of 64 bytes. One cache is direct mapped and the other is two-way set associative. Figure B.5 shows that for set associative caches we must add a multiplexor to select between the blocks in the set depending on the tag match. Because the speed of the processor can be tied directly to the speed of a cache hit, assume the processor clock cycle time must be stretched 1.35 times to accommodate the selection multiplexor of the set associative cache. To the first approximation, the cache miss penalty is 65 ns for either cache organization. (In practice, it is normally rounded up or down to an integer number of clock cycles.) First, calculate the average memory access time and then processor performance. Assume the hit time is 1 clock cycle, the miss rate of a direct-mapped 128 KiB cache is 2.1%, and the miss rate for a two-way set associative cache of the same size is 1.9%.

Answer Average memory access time is

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

Thus, the time for each organization is

$$\text{Average memory access time}_{\text{1-way}} = 0.35 + (.021 \times 65) = 1.72\,\text{ns}$$

$$\text{Average memory access time}_{\text{2-way}} = 0.35 \times 1.35 + (.019 \times 65) = 1.71\,\text{ns}$$

The average memory access time is better for the two-way set-associative cache. The processor performance is

$$\begin{aligned}\text{CPU time} &= \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}\right) \times \text{Clock cycle time} \\ &= \text{IC} \times \left[\left(\text{CPI}_{\text{execution}} \times \text{Clock cycle time}\right) \right. \\ &\quad \left. + \left(\text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss penalty} \times \text{Clock cycle time}\right)\right]\end{aligned}$$

Cache Performance

Substituting 65 ns for (Miss penalty × Clock cycle time), the performance of each cache organization is

$$\text{CPU time}_{\text{1-way}} = \text{IC} \times [1.0 \times 0.35 + (0.021 \times 1.4 \times 65)] = 2.26 \times \text{IC}$$

$$\text{CPU time}_{\text{2-way}} = \text{IC} \times [1.0 \times 0.35 \times 1.35 + (0.019 \times 1.4 \times 65)] = 2.20 \times \text{IC}$$

and relative performance is

$$\frac{\text{CPU time}_{\text{2-way}}}{\text{CPU time}_{\text{1-way}}} = \frac{2.26 \times \text{Instruction count}}{2.20 \times \text{Instruction count}} = 1.03$$

Cache Performance

Miss Penalty and Out-of-Order Execution Processors

Let's redefine memory stalls to lead to a new definition of miss penalty as nonoverlapped latency:

$$\frac{\text{Memory stall cycles}}{\text{Instruction}} = \frac{\text{Misses}}{\text{Instruction}} \times (\text{Total miss latency} - \text{Overlapped miss latency})$$

Cache Performance

Example Let's redo the preceding example, but this time we assume the processor with the longer clock cycle time supports out-of-order execution yet still has a direct-mapped cache. Assume 30% of the 65 ns miss penalty can be overlapped; that is, the average CPU memory stall time is now 45.5 ns.

Answer Average memory access time for the out-of-order (OOO) computer is

$$\text{Average memory access time}_{\text{1-way,OOO}} = 0.35 \times 1.35 + (0.021 \times 45.5) = 1.43\text{ns}$$

The performance of the OOO cache is

$$\text{CUP time}_{\text{1-way,OOO}} = \text{IC} \times [1.6 \times 0.35 \times 1.35 + (0.021 \times 1.4 \times 45.5)] = 2.09 \times \text{IC}$$

Hence, despite a much slower clock cycle time and the higher miss rate of a direct-mapped cache, the out-of-order computer can be slightly faster if it can hide 30% of the miss penalty.

Cache Performance

$$2^{\text{index}} = \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativity}}$$

$$\text{CPU execution time} = (\text{CPU clock cycles} + \text{Memory stall cycles}) \times \text{Clock cycle time}$$

$$\text{Memory stall cycles} = \text{Number of misses} \times \text{Miss penalty}$$

$$\text{Memory stall cycles} = \text{IC} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

$$\frac{\text{Misses}}{\text{Instruction}} = \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}}$$

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

$$\text{CPU execution time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Memory stall clock cycles}}{\text{Instruction}} \right) \times \text{Clock cycle time}$$

$$\text{CPU execution time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty} \right) \times \text{Clock cycle time}$$

$$\text{CPU execution time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss penalty} \right) \times \text{Clock cycle time}$$

$$\frac{\text{Memory stall cycles}}{\text{Instruction}} = \frac{\text{Misses}}{\text{Instruction}} \times (\text{Total miss latency} - \text{Overlapped miss latency})$$

$$\text{Average memory access time} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times (\text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2})$$

$$\frac{\text{Memory stall cycles}}{\text{Instruction}} = \frac{\text{Misses}_{L1}}{\text{Instruction}} \times \text{Hit time}_{L2} + \frac{\text{Misses}_{L2}}{\text{Instruction}} \times \text{Miss penalty}_{L2}$$

Figure B.7 Summary of performance equations in this appendix. The first equation calculates the cache index size, and the rest help evaluate performance. The final two equations deal with multilevel caches, which are explained early in the next section. They are included here to help make the figure a useful reference.

Six Basic Cache Optimizations

- ✓ The average memory access time formula gave us a framework to present cache optimizations for improving cache performance:

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

- ✓ Hence, we organize six cache optimizations into three categories:
  - ✓ **Reducing the miss rate:** larger block size, larger cache size, and higher associativity
  - ✓ **Reducing the miss penalty:** multilevel caches and giving reads priority over writes
  - ✓ **Reducing the time to hit in the cache:** avoiding address translation when indexing the cache

Six Basic Cache Optimizations

The classical approach to improving cache behavior is to reduce miss rates, and there are three techniques to do so.

1. Compulsory: The very first access to a block cannot be in the cache, so the block must be brought into the cache. These are also called cold-start misses or first-reference misses.

2. Capacity: If the cache cannot contain all the blocks needed during execution of a program, capacity misses (in addition to compulsory misses) will occur because of blocks being discarded and later retrieved.

3.Conflict: If the block placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory and capacity misses) will occur because a block may be discarded and later retrieved if too many blocks map to its set. These misses are also called collision misses. The idea is that hits in a fully associative cache that become misses in an n-way set-associative cache are due to more than n requests on some popular sets.

Cache size (KiB)	Degree associative	Total miss rate	Miss rate components (relative percent) (sum = 100% of total miss rate)					
			Compulsory		Capacity		Conflict	
4	1-way	0.098	0.0001	0.1%	0.070	72%	0.027	28%
4	2-way	0.076	0.0001	0.1%	0.070	93%	0.005	7%
4	4-way	0.071	0.0001	0.1%	0.070	99%	0.001	1%
4	8-way	0.071	0.0001	0.1%	0.070	100%	0.000	0%
8	1-way	0.068	0.0001	0.1%	0.044	65%	0.024	35%
8	2-way	0.049	0.0001	0.1%	0.044	90%	0.005	10%
8	4-way	0.044	0.0001	0.1%	0.044	99%	0.000	1%
8	8-way	0.044	0.0001	0.1%	0.044	100%	0.000	0%
16	1-way	0.049	0.0001	0.1%	0.040	82%	0.009	17%
16	2-way	0.041	0.0001	0.2%	0.040	98%	0.001	2%
16	4-way	0.041	0.0001	0.2%	0.040	99%	0.000	0%
16	8-way	0.041	0.0001	0.2%	0.040	100%	0.000	0%
32	1-way	0.042	0.0001	0.2%	0.037	89%	0.005	11%
32	2-way	0.038	0.0001	0.2%	0.037	99%	0.000	0%
32	4-way	0.037	0.0001	0.2%	0.037	100%	0.000	0%
32	8-way	0.037	0.0001	0.2%	0.037	100%	0.000	0%
64	1-way	0.037	0.0001	0.2%	0.028	77%	0.008	23%
64	2-way	0.031	0.0001	0.2%	0.028	91%	0.003	9%
64	4-way	0.030	0.0001	0.2%	0.028	95%	0.001	4%
64	8-way	0.029	0.0001	0.2%	0.028	97%	0.001	2%
128	1-way	0.021	0.0001	0.3%	0.019	91%	0.002	8%
128	2-way	0.019	0.0001	0.3%	0.019	100%	0.000	0%
128	4-way	0.019	0.0001	0.3%	0.019	100%	0.000	0%
128	8-way	0.019	0.0001	0.3%	0.019	100%	0.000	0%
256	1-way	0.013	0.0001	0.5%	0.012	94%	0.001	6%
256	2-way	0.012	0.0001	0.5%	0.012	99%	0.000	0%
256	4-way	0.012	0.0001	0.5%	0.012	99%	0.000	0%
256	8-way	0.012	0.0001	0.5%	0.012	99%	0.000	0%
512	1-way	0.008	0.0001	0.8%	0.005	66%	0.003	33%
512	2-way	0.007	0.0001	0.9%	0.005	71%	0.002	28%
512	4-way	0.006	0.0001	1.1%	0.005	91%	0.000	8%
512	8-way	0.006	0.0001	1.1%	0.005	95%	0.000	4%

Figure B.8 Total miss rate for each size cache and percentage of each according to the three C's. Compulsory misses are independent of cache size, while capacity misses decrease as capacity increases, and conflict misses decrease as associativity increases. Figure B.9 shows the same information graphically. Note that a direct-mapped cache of size *N* has about the same miss rate as a two-way set-associative cache of size *N*/2 up through 128 K. Caches larger than 128 KiB do not prove that rule. Note that the Capacity column is also the fully associative miss rate. Data were collected as in Figure B.4 using LRU replacement.

Six Basic Cache Optimizations

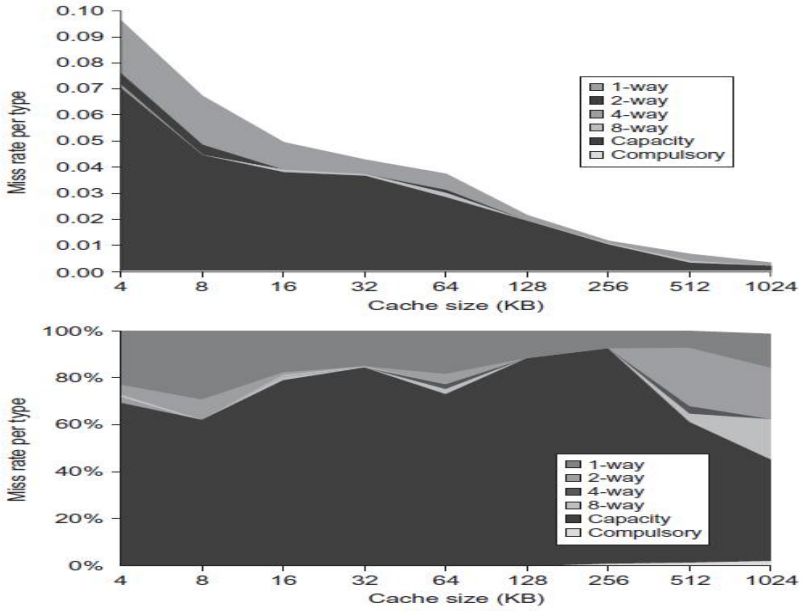

Figure B.9 Total miss rate (top) and distribution of miss rate (bottom) for each size cache according to the three C's for the data in Figure B.8. The top diagram shows the actual data cache miss rates, while the bottom diagram shows the percentage in each category. (*Space allows* the graphs to show one extra cache size than can fit in Figure B.8.)