Chapter 8

Memory and I/O Organization

מעבר למעבד הראשי, מחשב מורכב מרכיבי אחסון ומערכת קלט/פלט (I/O) שמחברת את המעבד לרשתות וממשקי משתמש. בפרק הזה נדון בארגון מערכת הזכרון והיבטים של גישה להתקנים נוספים.

Slides 2 – 4 Principle of Locality

The principle of locality states is another **rule of thumb** — for 90% of run time the next instruction or data access will come from a small volume of memory (about 10% of the program) closest to the current instruction or data. Given an access to address A in a program that occupies a volume B of memory, the next access will be to an address within $A \pm 0.05B$ for 90% of run time. Slide 2 shows statistics for programs from SPEC CINT showing the percentage of total memory accessed in 80% (dark gray) and 90% (light gray) of run time.

Amdahl's law suggests that these memory accesses should be made as fast as possible. The **memory hierarchy** shown on slide 3 divides memory into types from long-term storage to register. Long-term **storage** holds all programs and data in a complex system organized by the operating system with relatively long access times. **Main memory** holds running programs and data of a smaller size, closer to the CPU and with faster access. **Cache** copies a section of main memory into the CPU for faster access. Both main memory and cache are organized by byte address. **Registers** hold a small amount of data in the CPU with fastest access — organization is defined in the ISA for the CPU.

The table on slide 4 shows typical sizes and access times for various memory types. These numbers improve constantly and may already be out of date.

Slides 5 – 7 CPU and Memory Hierarchy

The diagram on slide 5 shows the relationship between memory elements for the DLX. The Instruction Memory in the IF stage is **instruction cache** and the Data Memory in stage MEM is the **data cache**. These caches are called **Level 1 cache (L1)** because they are the first level of access to memory by the CPU. Cache holds a **copy** of a limited section of main memory. The sections copied to cache are characterized by their RAM address locations.

The L1 instruction and data caches are connected to a **cache controller** that also connects to a larger **Level 2 cache (L2)** and to the **external bus interface** that connects the CPU to the rest of the world. The CPU is connected over the external bus to an **I/O controller** (often called a **chipset**) that connects to **main memory** (RAM), **long-term storage** and other devices such as the user interface and network.

Slide 6 shows the operation of the cache update mechanism in general terms. When the information required by the CPU has been previously copied to L1 cache, then the instruction fetch or data access in the CPU takes place in 1 clock cycle. If the required information is not in L1, we say that the **addresses are not present in L1** (referring to the data/instruction contents of those addresses, not the actual addresses). In that case, L1 requests the information from L2 cache and from main memory. If the addresses are present in L2 then it is copied to L1 and the request to main memory (which takes much more time) can be cancelled. If the information is not in L2 then it will be copied from main memory to both L1 and L2. Copying from L2 to L1 can take several clock cycles — copying from main memory to L1 takes many clock cycles.

Two special cases are shown on slide 7. Although cache and main memory are accessed by the byte address of the information, long-term storage and other devices are addressed by the OS at a specific **device address**. For example, to write k bytes to the hard drive, the OS performs k successive byte writes to the single address assigned to the hard disk. A similar process occurs for network access. Since the information at these device addresses changes constantly there is no reason to copy them to cache — the device addresses are marked **non-cacheable** by the CPU.

Another special case is virtual memory in which the total address space is larger than the physical memory present in the machine. For example, a 32-bit Pentium 4 CPU from 1999 was capable of addressing 64 GB of memory but even today few personal computers hold 64 GB of physical memory. With virtual memory the relationship of program address and physical address may change, and may affect the relationship of main memory to cache.

Slides 8 – 12 Memory Organization

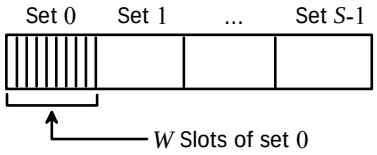
An n-bit address specifies the location of a specific byte in memory. The address space consists of 2^N byte locations from 0 to $2^n - 1$. The CPU must have an n-bit register to hold the address sent to the physical memory.

Information from main memory is copied to cache as needed (we refer to data to mean both data and instruction information). Working with cache presents three important questions:

How much data should be copied to cache?
Which data is located in cache at any given time?
How can the CPU find a particular address location in cache?

To simplify matters, bytes are copied to cache in **blocks** (called cache lines in Intel documents). A block is a collection of **B bytes** in memory (main or cache) and B is chosen so that the **page size** in virtual memory is an integer multiple of B (a page swap involves an integer number of blocks).

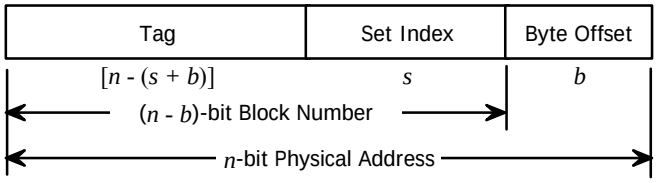
The cache is divided into **S sets** and each set is divided into **W slots**. Each slot holds exactly one block. The organizing rule is: **The block number determines the set number — any slot in the set is allowed to hold the block**.



S sets in cache

As shown on slide 10, the block size is chosen as a power of 2: $\mathbf{B = 2^b}$. The **block number** for the n-bit **address A** is the **upper n – b bits** of the address = int[A / B]. The lower b bits are called the **block offset** (the number of a certain address within the block from 0 to B – 1).

Similarly (slide 11) the number of sets is a power of 2: $\mathbf{S = 2^s}$. The set index is the lower s bits of the block number, and is found from **set number = block number % S**. A block copied to cache must be placed in the set numbered by the set index.



Parsing of n-bit address

The remaining upper n – (b + s) bits of the address are called the **tag**. The tag is written into a register associated with the cache to identify the blocks copied to the set. Writing the tag and set index provides the block number.

To **find a particular byte in cache**, the CPU calculates the block number for that address, then the set index for than block number, and then searches the tags of the blocks located in the set. If the tag is found, then the contents of the associated slot in the set is the desired block. The required byte is found by counting within the bytes stored in the slot.

The total cache size is

$$\text{total cache size} = S \times W \times B = \frac{\text{sets}}{\text{cache}} \times \frac{\text{blocks}}{\text{set}} \times \frac{\text{bytes}}{\text{block}} = \frac{\text{bytes}}{\text{cache}}$$

CPU documentation usually specifies the total size along with W and B.

Slides 13 – 15 Example of Memory Organization

A 32-bit address defines a 2^{32} = 4GB memory space.

Choose a block size B = 2^b with b = 4 and B = 16. The block number is 32 – 4 = 28 bits and defines 2^{28} = 256 Mblocks.

Choose S = 2^s with s = 8 and S = 256 sets. When block B is copied to cache it must be assigned to set B % 256. Set 0 may hold block 0, or 256, or 512, and so on. Set 1 may hold block 1, 257, 513, and so on.

The tag number is 32 – (4 + 8) = 20 bits.

Choose W = 2^2 = 4 slots per set. There are "4 ways" to choose the location of a block in an empty set and the number of "ways" is called **associativity**. This cache is **4-way associative**.

The total cache size = 256 sets/cache x 4 blocks/set x 16 bytes/block = 16 KB.

At some moment in run time set 1 may hold blocks 1025, 513, 1281, 257. The tag numbers that identify these blocks are int[block/256] = 4, 2, 5, 1.

If the program requires access to address 20509_{10} = 0x0000501D the cache controller parses this address as byte 0x0D = 13 in block 0x501 which — if it is in cache — must be located in set 01 with tag 5. The cache controller scans the tags for set 1 and finds 5 in slot 2 (of slots 0, 1, 2, 3).

The cache controller reads the 16 bytes of block 0x501 and sends byte 0x0D to IR or LMD.

Slide 16 Cache Definitions and Policies

Cache hit	The CPU requires a block for read or write and finds the block in cache.
Cache miss	The CPU requires a block not in cache. On a read miss the cache controller loads the block to cache.
Write allocate	A cache policy: on a write miss the cache controller loads the block to cache.
No write allocate	A cache policy: on a write miss the cache controller writes directly to RAM without loading the block.

Swapping out a cache block

The CPU accesses a block not in cache and must copy the block to cache, but all of the slots in the required set are full. The cache controller must remove (swap out) the block that is least recently used (LRU). There are two possible policies for cache / RAM consistency:

Write Through — the cache controller updates RAM on every write to cache

Write Back — the cache controller updates main memory when the block is swapped out

Slides 17 – 18 Cache Performance Issues

L1 Cache hit	The CPU reads/writes memory in 1 clock cycle.
L1 Cache Miss	The CPU stalls while L1 loads the missing block from L2 or main memory
Miss Rate	The number of cache misses per cache accesses (instruction fetch, data load or data store). This rate depends on cache size and organization.
Miss Penalty	The number of stall cycles while cache loads a missing block depends on cache hardware technology and organization

Cache misses contribute to CPI^{stall}.

Using the definitions of miss rate and miss penalty, we easily calculate CPI^{stall} for a **1 level unified cache** (not split into instruction and data):

$$\begin{aligned} CPI^{stall} &= \frac{\text{stall cycles}}{\text{stalls}} \times \frac{\text{stalls}}{\text{instruction type}} \times \frac{\text{instruction types}}{\text{instruction}} \\ &= \frac{\text{stall cycles}}{\text{stalls}} \times \frac{\text{stalls}}{\text{instruction type}} \times \frac{\text{memory accesses}}{\text{instruction}} \\ &= \frac{\text{stall cycles}}{\text{stalls}} \times \frac{\text{stalls}}{\text{instruction type}} \times \frac{\text{instruction accesses} + \text{load accesses} + \text{store accesses}}{\text{instruction}} \\ &= \text{miss penalty} \times \text{miss rate} \times \frac{IC + IC^{load} + IC^{store}}{IC} \end{aligned}$$

CPI^{stall} for a **1 level split cache** treats instruction cache misses and data cache misses separately:

$$\begin{aligned} CPI^{stall} &= \frac{\text{instruction stall cycles}}{\text{stalls}} \times \frac{\text{instruction stalls}}{\text{instruction access}} \times \frac{\text{instruction accesses}}{\text{instruction}} \\ &\quad + \frac{\text{data stall cycles}}{\text{stalls}} \times \frac{\text{data stalls}}{\text{data access}} \times \frac{\text{data accesses}}{\text{instruction}} \\ &= \text{instruction miss penalty} \times \text{instruction miss rate} \times \frac{1 \text{ instruction access}}{\text{instruction}} \\ &\quad + \text{data miss penalty} \times \text{data miss rate} \times \frac{IC^{load} + IC^{store}}{IC} \end{aligned}$$

For most programs the instruction cache miss rate is lower than the data cache miss rate because most instructions executed are located in loops — an entire loop may be loaded to cache on the first iteration and there will be no instruction cache misses while the loop runs. Access to a large data structure may be sequential with little return to previously accessed data.

For example, with

Instruction Miss Penalty = Data Miss Penalty = 50 cycles per stall
Instruction Miss Rate = 0.5%
Data Miss Rate = 5.0%
IC^{load} = 25%
IC^{store} = 15%

$$CPI^{stall} = 50 \times 0.005 \times 1 + 50 \times 0.05 \times 0.40 = 0.25$$

$CPI = 1.25 \Rightarrow 20\%$ degradation

This high performance degradation makes the addition of a 2nd level of cache worthwhile.

Slide 19 Two-Level Unified Cache — Definitions

We model a two-level cache as follows:

L1 cache has miss rate M_{L1} so that its hit rate is $1 - M_{L1}$. On a miss at L1, the controller accesses L2 and main memory.

L2 cache has miss rate M_{L2} so that its hit rate is $1 - M_{L2}$. On a miss at L2, the controller must wait for main memory.

On an L1 miss and an L2 hit the miss penalty is P_{L1}. On an L1 miss followed by an L2 miss an extra penalty P_{L2} is added — the total penalty is $P_{L1} + P_{L2}$.

$IC^{A} = IC^{load} + IC^{store}$ is the total number of data accesses. As always, the total number of instruction access is just the total number of instructions = IC.

Slides 20 – 23 Performance of Two-Level Unified Cache

Slide 20 shows an event tree that analyzes the probability of incurring each penalty. We derive the CPI^{stall} formally:

$$CPI^{stall} = \frac{\text{stall cycles}}{IC} = \sum_{\text{n = stall types}} \frac{\text{stall cycles}}{\text{stall type n}} \times \frac{\text{stalls of type n}}{IC}$$

$$\text{where stall types} = \{\text{hits, misses, data, instructions}\}$$

$$= \sum_{\text{i = \{data, instr\}}} \left[\frac{\text{stall cycles}}{(\text{L1 miss, L2 hit})_i} \times \frac{(\text{L1 miss, L2 hit})_i}{IC} + \frac{\text{stall cycles}}{(\text{L1 miss, L2 miss})_i} \times \frac{(\text{L1 miss, L2 miss})_i}{IC} \right]$$

$$= \sum_{\text{i = \{data, instr\}}} \left[\frac{\text{stall cycles}}{(\text{L1 miss, L2 hit})_i} \times \frac{(\text{L1 miss, L2 hit})_i}{\text{L1 access}_i} \times \frac{\text{L1 access}_i}{IC} + \frac{\text{stall cycles}}{(\text{L1 miss, L2 miss})_i} \times \frac{(\text{L1 miss, L2 miss})_i}{\text{L1 access}_i} \times \frac{\text{L1 access}_i}{IC} \right]$$

We make the (approximately valid) assumption that misses at L1 and L2 are statistically independent. Then

$$\frac{(\text{L1 miss, L2 hit})_i}{\text{L1 access}_i} = \frac{\text{L1 miss}_i \text{ followed by L2 hit}}{\text{L1 access}_i} = \frac{\text{L1 miss}_i}{\text{L1 access}_i} \times \underbrace{\frac{\text{L2 access}}{\text{L1 miss}_i}}_{1} \times \frac{\text{L2 hit}}{\text{L2 access}} = \frac{\text{L1 miss}_i}{\text{L1 access}_i} \times \frac{\text{L2 hit}}{\text{L2 access}}$$

$$\frac{(\text{L1 miss, L2 miss})_i}{\text{L1 access}_i} = \frac{\text{L1 miss}_i \text{ followed by L2 miss}}{\text{L1 access}_i} = \frac{\text{L1 miss}_i}{\text{L1 access}_i} \times \underbrace{\frac{\text{L2 access}}{\text{L1 miss}_i}}_{1} \times \frac{\text{L2 miss}}{\text{L2 access}} = \frac{\text{L1 miss}_i}{\text{L1 access}_i} \times \frac{\text{L2 miss}}{\text{L2 access}}$$

and so

$$CPI^{stall} = \sum_{\text{i = \{data, instr\}}} \left[\frac{\text{stall cycles}}{(\text{L1 miss, L2 hit})_i} \times \frac{\text{L1 miss}_i}{\text{L1 access}_i} \times \frac{\text{L2 hit}}{\text{L2 access}} \times \frac{\text{L1 access}_i}{IC} + \frac{\text{stall cycles}}{(\text{L1 miss, L2 miss})_i} \times \frac{\text{L1 miss}_i}{\text{L1 access}_i} \times \frac{\text{L2 miss}}{\text{L2 access}} \times \frac{\text{L1 access}_i}{IC} \right]$$

Using the definitions for miss rates and miss penalties, and isolating common factors this becomes

$$CPI^{stall} = P_{L1} \times M_{L1} \times (1 - M_{L2}) \times \frac{IC + IC^A}{IC} + (P_{L1} + P_{L2}) \times M_{L1} \times M_{L2} \times \frac{IC + IC^A}{IC}$$

$$= M_{L1} \times \left[1 + \frac{IC^A}{IC}\right] \times \left[P_{L1} \times (1 - M_{L2}) + (P_{L1} + P_{L2}) \times M_{L2}\right]$$

$$= M_{L1} \times \left[1 + \frac{IC^A}{IC}\right] \times \left[P_{L1} - P_{L1} \times M_{L2} + P_{L1} \times M_{L2} + P_{L2} \times M_{L2}\right]$$

$$CPI^{stall} = M_{L1} \times \left[1 + \frac{IC^A}{IC}\right] \times \left[P_{L1} + P_{L2} \times M_{L2}\right]$$

Notice that this expression can be read directly off the **event tree**:

CPI^{stall} = L1 miss rate × L1 access rate^{I+D} × (L1 penalty + extra penalty × L2 miss rate)

Slides 24 – 25 Split L1 Cache with Unified L2 Cache

For the unified cache we have

$$\frac{\text{misses at L1}}{IC} = \frac{\text{misses at L1}}{\text{accesses at L1}} \times \frac{\text{accesses at L1}}{IC} = M_{L1} \times \left[1 + \frac{IC^A}{IC}\right] = M_{L1} + M_{L1} \times \frac{IC^A}{IC}$$

To find the expression for a split cache (I + D in separate caches) we replace:

$$\frac{\text{misses at L1}}{IC} \rightarrow \frac{\text{data misses at L1}}{\text{data accesses at L1}} + \frac{\text{instruction misses at L1}}{\text{instruction accesses at L1}}$$

$$= M_{L1}{}^{\text{instructions}} \times 1 + M_{L1}{}^{\text{data}} \times \frac{IC^A}{IC} = M_{L1}{}^{I} + M_{L1}{}^{D} \times \frac{IC^A}{IC}$$

and so

$$CPI^{stall}{}_{\text{unified}} = M_{L1} \times \left[1 + \frac{IC^A}{IC}\right] \times \left[P_{L1} + P_{L2} \times M_{L2}\right]$$

$$\rightarrow \quad CPI^{stall}{}_{\text{split}} = \left[M_{L1}{}^{I} + M_{L1}{}^{D} \times \frac{IC^A}{IC}\right] \times \left[P_{L1} + P_{L2} \times M_{L2}\right]$$

Comparing the expressions for **1-level** (slide 18) and **2-level** cache, the miss rates are the same and the enhancement between is entirely in the miss penalty:

$$CPI^{stall}_{1-level} = \left[M^{I}_{L1} + M^{D}_{L1} \times \frac{IC^A}{IC}\right] \times P_{L1} \qquad CPI^{stall}_{2-level} = \left[M^{I}_{L1} + M^{D}_{L1} \times \frac{IC^A}{IC}\right] \times \left(P_{L1} + P_{L2} \times M_{L2}\right)$$

In 1-level cache, the penalty is L1 to RAM and so P_{L1} = 50 CC.

In 2-level cache, the penalty is P_{L1} = 5 CC for L1 to L2 and on an L2 miss an extra penalty of P_{L2} = 45 CC for L2 to RAM. With a 1% miss rate at L2, $P_{L1} + P_{L2} \times M_{L2}$ = 5 + 45 x 0.01 = 5.45 CC.

Slides 26 – 31 Issues Affecting Miss Rate

Cache misses can be divided into three basic types:

Compulsory miss

Since a block is not copied to cache until the first access to a byte in the block, there is always a first access to a block and the first access will always miss in cache. Compulsory misses are not affected by cache properties.

Capacity miss

The cache is smaller than main memory and some blocks must be removed from cache to make room for other required blocks. If the cache were larger it would be possible to hold more data with less swapping out. The capacity miss rate is lower for a larger cache size.

Conflict miss

A given block must be copied to specific set. There are times when some blocks must be removed from the set to make room for required blocks, even when there are empty slots in other sets. This is not caused by an overall capacity miss, but by the associativity of the cache organization. For larger W there is less likelihood that one set will be full while another set has space. The conflict miss rate is lower when block placement (associativity) is more flexible. For example, conflict misses can be caused by address aliasing — data structures from different sections of memory alias to the same set and conflict.

Slides 27 and 28 show statistics on cache misses as size and associativity vary. The compulsory miss rate stays constant as expected. For any given cache size the conflict miss rate drops as associativity goes up. The capacity miss rate drops as the cache size increases. These statistics are summarized in the graph on slide 29. Slide 30 shows the conflict miss rate by cache size and associativity. For standard cache sizes there is very little improvement above W = 4 and this is the usual choice. The reason for restricting associativity is explained on slide 31. For a fixed cache size, a larger W requires fewer sets $S = 2^s$ and so the tag length $n – (b + s)$ is longer. A longer tag requires a longer tag search, which slows down cache access.

Slides 32 – 39 Miss Examples

We begin with two examples at the extremes of cache behavior — locality and non-locality.

Extreme locality

The program

`int i,a;for (i = 0 ; i < 4096 ; i++){ a = i;}`with **`i`** stored in a register and **`a`** stored in memory performs 4096 memory writes. As in most cache systems, the write allocate policy means that the first access to **`a`** causes a compulsory miss and load of **`a`** to cache. There are no further misses and so the miss rate is just 1 miss / 4096 accesses = 2.44×10^{-4}.

Extreme non-locality

The program

```
int i,a[4096];
for (i = 0 ; i < 4096 ; i++){
      a[i] = i;
}
```

with **i** stored in a register and **a[]** stored in memory also performs 4096 memory writes, but now to 4096 different addresses **a[i]**. We take the block size = 16 bytes/block which means that each block holds 4 words = 4 array elements. This time there are 4096/4 = 1024 compulsory misses and loads to cache. There are no read misses and so the miss rate is 1024 misses / 4096 accesses = 0.25 = 25%.

More practical examples require more **details of cache parameters**: We take B = 16 bytes per block, S = 256 sets and W = 4 slots per set. Total cache size is 16 x 256 x 4 = 16 KB = 4 Kwords.

Good locality

The program

```
int a[4096],i,j;
for (i = 0 ; i < 10 ; i++){
      for (j = 0 ; j < 4096 ; j++){
      a[j] = i + j;
      }
}
```

with **i,j** stored in registers and **a[]** stored in memory performs 40960 memory writes. There are 4096/4 = 1024 compulsory misses and loads to cache for **i = 0**, but the 16 KB of the array fills the cache and there are no further misses on **i > 0**. The miss rate is now 1024 misses / 40960 accesses = 0.025 = 2.5%.

Extreme non-locality

Making the array **a[]** too large to fit into cache has a dramatic effect on the miss rate. The program

```
int a[5120],i,j;
for (i = 0 ; i < 10 ; i++){
      for (j = 0 ; j < 5120 ; j++){
      a[j] = i + j;
      }
}
```

with **i,j** stored in registers and **a[]** stored in memory performs 51200 memory writes, to 5120 different addresses **a[i]**. For **i = 0** there is a compulsory miss every 4 elements, a total of 5120 / 4 = 1280 misses. But unlike the smaller array, the first outer loop iteration does not finish with the full array in cache. Looking at set 0 for i = 0,

on j = 0, block 0 → slot 0
on j = 1024, block 256 → slot 1
on j = 2048, block 512 → slot 2
on j = 3072, block 768 → slot 3
on j = 4096, block 1024 → slot 0

Since block 1024 must enter slot 1024 % 256 = 0 it forces the **eviction** (swap out) of block 0 (the LRU). Set 0 now holds block 1024, 256, 512, 768.

On the second iteration of the outer loop, i = 1

on j = 0, block 0 → slot 1	cache **evicts** the LRU = block 256
on j = 1024, block 256 → slot 2	cache **evicts** the LRU = block 512
on j = 2048, block 512 → slot 3	cache **evicts** the LRU = block 768
on j = 3072, block 768 → slot 0	cache **evicts** the LRU = block 1024
on j = 4096, block 1024 → slot 1	cache **evicts** the LRU = block 256

This continues until the end of the program. Again there is a capacity miss every 4 accesses and so the miss rate is 25%.

Slide 36 shows that this high miss rate could be improved if it were possible to switch the cache controller to evict the most recently used (MRU) instead of LRU. In that case, slots 0, 1 and 2 would be stable after the first outer loop iteration, while slot 3 of set 0 alternates between the 768 and 1024. The miss rate falls from 25% to 11.5%.

The most effective way to prevent cache misses for this program is to split the loop into two pieces as

```
int a[5120],i,j;
for (i = 0 ; i < 10 ; i++){
      for (j = 0 ; j < 4096 ; j++){
            a[j] = i + j;
      }
for (i = 0 ; i < 10 ; i++){
      for (j = 4096 ; j < 5120 ; j++){
            a[j] = i + j;
      }
}
```

With this optimization both sections run with a miss rate of 2.5%. Unfortunately, this solution suffers the severe disadvantage of requiring that the programmer be concerned about performance.

Address aliasing

We consider the program (slide 38)

```
int a[512],b[512],c[512],i,j;
  for (i = 0 ; i < 20; i++){
    for (j = 0 ; j < 512 ; j++){
      c[j] = a[j] + b[j] + c[j];
    }
  }
```

where the arrays are placed at the addresses

```
a: 02000000 – 020007FF
b: 02001000 – 020017FF
c: 02002000 – 020027FF
```

Array **b[]** does not follow array **a[]** immediately in memory, but a compiler may do this for convenience.

We first consider the case that B = 16 and S = 256 with W =2. The block offset for each address is the last hex digit (in green), and the set number is the two next hex digits (in blue). We see that all 3 arrays must be placed in sets 00 – 7F. No blocks are assigned to sets 80 – FF which remain empty.

For the **`i = 0`** iteration of the outer loop, there is a compulsory miss for **`a[j]`** loaded to slot 0, then a compulsory miss for **`b[j]`** loaded to slot 1, then a compulsory miss for **`c[j]`** loaded to slot 0 (evicting **`a[j]`**) — there is no miss for **`c[j]`** on the write.

For the **`i > 0`** iteration of the outer loop, there is a capacity miss for **`a[j]`** loaded to slot 0 (evicting **`c[j]`**) and a capacity miss for **`c[j]`** (evicting **`a[j]`**) loaded to slot 0.

The miss rate is 12.8%.

If the cache we reorganized (slide 39) with fewer sets and more slots per set (S = 128 and W = 4) then all sets 00 – 7F will contain blocks. Each array will occupy one of the 128 sets and there will be no conflict misses — no misses for **`i > 0`**.

The miss rate will drop from 18,8% to 0.94%.

Slides 40 Workstation Layout with PCI

Slide 5 shows the CPU external bus connection to main memory and long-term storage through an I/O controller. Slide 40 shows a simplified layout of a workstation using the **PCI** standard I/O system. PCI (Peripheral Component Interconnect) is organized around an I/O controller called a **bridge** that determines communications protocols among the CPU, main memory, long-term storage, user interface, network interfaces and other I/O systems.

The CPU and RAM are directly connected to the bridge. A switching fabric (a data exchange protocol that may be an active switch as in a computer network, or a simple bus that permits only one transmission at a time) connects other device controllers to the bridge. Software drivers installed in the operating system provide an API (application programming interface) between the CPU and the device controllers.

Slides 41 – 46 PCI Services

In addition to permitting CPU access to main memory and long-term storage, the PCI system provides system services:

Boot services

The PCI enables access to BIOS — the basic input/output system. This is a ROM-based device holding software routines that enable system initiation and OS boot.

Timers

The PCI operates a collection of system timers, counters and real time clocks for managing, coordinating and synchronizing system activities.

Interrupt controllers

The PCI implements a programmable interrupt controller that enables hardware interrupts to be enabled and assigned to system activities via IRQ (interrupt requests). On modern systems an advanced programmable interrupt controller (APIC) also permits generation, transfer and interpretation of interrupt messages between processors.

Direct Memory Access (DMA)

The DMA system permits devices to define memory transfers that do not require CPU intervention for byte-by-byte read and write.

BIOS

BIOS is the BASIC INPUT/OUTPUT SYSTEM and provides code that implements the minimum functionality of the workstation at system start-up. When the hardware system is started or reset, the CPU performs a self-check (also known as a POST — power-on self-test — or a sanity check). If the CPU is OK, then it fetches an instruction from physical address **FFFF0h** (a defined constant address near the top of the physical memory space of the Intel 8086). Address FFFF0h contains a branch instruction — the branch target is a code section located in PCI BIOS ROM (Read Only Memory) called **firmware**. (The form of ROM is usually E^2PROM, Electrically Erasable Programmable ROM, that permits updating of the system code.)

The BIOS firmware permits location and operation of the keyboard, display, and a boot device (a long-term storage unit containing an operating system). Most workstations today contain an additional level of firmware above BIOS called **UEFI** (Unified Extensible Firmware Interface). UEFI is a small hardware-oriented operating system that performs system management including boot of the main OS. When UEFI is present, BIOS loads UEFI and UEFI loads the main OS from the boot device. In a non-UEFI system, the BIOS loads the OS from boot device directly.

Interrupt Handling

An **interrupt** is an asynchronous event generated outside the CPU that temporarily halts normal instruction flow in a controlled manner, directs the CPU to a handle task in need of attention and then restarts normal instruction flow in a controlled manner. A **trap** is a software instruction that causes behavior similar to an interrupt.

PCI-compatible CPUs contain a **local APIC** (Advanced Programmable Interrupt Controller) that enables software-defined interrupt handling. The APIC accepts **local interrupts** generated by an interrupt number placed on the processor external bus interface and triggered by an input signal on the bus called **INTR**. The APIC also sends and receives formatted **interrupt messages** on a CPU bus shared among processors in the system. The OS uses these messages to **assign threads to cores**, among other coordination tasks.

The **I/O APIC** behaves like a local APIC except that it belongs to the I/O system attached to the PCI bridge. This enables attached devices (user interface, long term storage, networks and so on) to exchange interrupt messages of the type exchanged by CPUs. In older workstations (20th century), a subset of the local interrupts was assigned to the IRQ (interrupt request) system. Each hardware device was assigned an **IRQ** number by the OS. In a PCI system with processor support, IRQs are emulated in the interrupt message system. All hardware devices share a single IRQ number — the actual system request is sent in an interrupt message.

The messages between processor cores are called **Interprocessor Interrupts** (IPI). The drawing on slide 44 shows the Processor System Bus as a common communication channel for the CPUs, also attached to the PCI bridge. The I/O APIC permits the I/O system to behave as another CPU on the system.

Direct Memory Access (DMA)

DMA permits a peripheral device (such as the user interface, storage and network) to access main memory directly without intervention by the CPU to execute transfer instructions. It is

used for large data transfers. DMA permits the CPU to continue working concurrently with the transfer and the CPU can preempt DMA to perform cache updates.

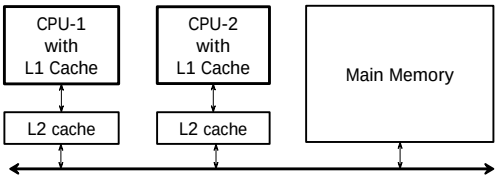
In **interrupt mode**, the CPU executes code that instructs the DMA controller to set up the transfer. The DMA request includes the start address in memory and the number of bytes to transfer. The DMA controller acts as master of the data path and actively transfers data between RAM and the peripheral device. The DMA controller executes an IRQ at end of the transfer and the CPU takes back control of the bus.

In **PCI arbitration**, the peripheral device requests control of the bus directly from the PCI bridge, including details of the required read or write memory access. The PCI bridge grants bus control of the system bus to the attached device and executes the data transfers.

Slides 47 – 48 Cache Coherency and Consistency

On slide 16 we saw two methods (**write back** and **write through**) to maintain consistency of the information stored in the CPU cache and the main memory. With multiple CPU cores in most computers, telephones and tablets, the consistency problem becomes more complex.

Suppose that CPU-1 and CPU-2 are running **concurrent threads** in the same process. Both CPUs read variable X to cache from main memory. If CPU-1 updates X in its cache copy, then this write to X invalidates the copies of X in CPU-2 cache and in main memory — the correct value of X is stored in CPU-1 cache and the old copies in CPU-2 cache and in main memory should not be used (another example of a RAW hazard).



Cache consistency is the case in which the copies of data in all locations are always identical, and is extremely difficult to obtain. **Cache coherence** is the case in which all reads and writes occur in the correct order — this is the goal of cache management in multicore systems.

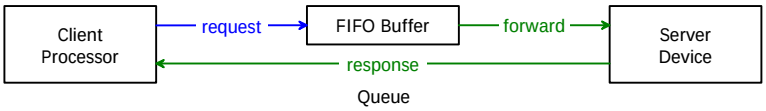
The most common method for maintaining cache coherence is **bus snooping**. All caches in the system use the write back policy. The CPU updates its private cache with no update to main memory. Main memory is updated only when a cache block is evicted from cache. But on every CPU write to a block in its cache the **CPU writes the destination addresses** on the shared processor bus (this is a short and fast bus cycle). All devices on the bus monitor these addresses, which allows each CPU and the PCI I/O device to see which CPUs are loading a particular cache block, and to see which CPUs are writing to a particular cache block.

The bus enforces **write synchronization** — bus arbitration prevents writes to multiple copies of a cache block. Only one CPU can place a target address on the memory bus per bus cycle, and only one CPU can write to any given cache block in a bus cycle.

When a CPU or the main memory sees a write to a certain address by another CPU, it marks its own copy of the block **invalid**. If two CPUs load X at the same time and try to write to X at the same time, only one CPU will be allowed to write by the write synchronization method. The other CPU will have to invalidate its copy of X and cancel all the work that led up to the attempted write of X. After invalidating X, the CPU that needs X will try to read the memory location from main memory. The read attempt will trigger the CPU that has the valid copy to evict the cache block and update main memory. The problem of cache coherence is one of the main sources of **complexity in parallel programming**.

Slides 49 – 52 Client/Server Model of I/O

It is useful to model the relationship between CPU and an I/O device (including storage) as a client/server system. The CPU operates at a much higher data transfer rate than I/O devices and data accesses must be stored in a FIFO (first-in, first out) buffer. The buffer stores read/write requests from the CPU (up to some maximum capacity) and forwards them to the I/O device at the lower rate.



Characteristics of a client/server system:

Latency The time between a client request and the buffer response

Throughput The number of services provided per unit time

Bandwidth The maximum data transfer rate of the I/O channel (including buffer)

Capacity The maximum throughput of the server through the buffer, which depends on the bandwidth and the service rate (server speed)

Utilization The request rate as a proportion of capacity

The FIFO buffer can handle the client/server rate disparity because client requests are bursty, appearing intermittently at high speed. The peak client request rate is much larger than the average client request rate. Generally,

Peak client rate > Server rate > Average client rate

The client sees a **minimum latency** determined by the maximum service response rate (which is related to the minimum time required to service one request). This creates a trade-off between utilization and latency. As more requests are stored in the buffer, the latency increases. If the client (CPU) makes more requests than can be held by the FIFO then there will be overflow and data will be lost.

Slides 53 – 55 Queuing Theory

Queuing theory provides a mathematical description of queues in client/server systems. The simplest results begin with the following assumptions:

Client requests arrive independently (by Poisson statistics) and have a random length (number of bytes to transfer). Over the long run, requests reach a steady state (small variations around a fixed value) characterized by the average request rate.

The **buffer** stores requests and forwards in order of arrival (FIFO) at service rate. In the steady state there is an average buffer level (number of stored requests).

The **server** provides services to each request independently (by Poisson statistics). In the steady state there is an average service rate.

The results of Markov chain analysis are:

$$\text{Utilization} = \frac{\text{Request Rate}}{\text{Service Rate}}$$

$$\text{Latency} = \frac{1}{\text{Service Rate} - \text{Request Rate}} = \frac{1}{\text{Service Rate}}\left(\frac{1}{1 - \text{Utilization}}\right)$$

$$\text{Buffer Level} = \text{Latency} \times \text{Request Rate} = \text{Utilization} \times \left(\frac{1}{1 - \text{Utilization}}\right)$$

Buffer level and latency rise sharply with increased utilization.

85% utilization is a reasonable trade-off to prevent long I/O waits.

As utilization rises, there is a greater chance of system noise, transmission errors and buffer overflows that must be treated by re-transmission of data. The re-transmission increases the volume of data transmitted and can make the buffer congestion worse. **Traffic shaping** is a method to prevent very long I/O latency. Clients are by allocated transmission quotas that prevent average utilization from rising above some safe value, usually about 85%.