**NYU Tandon School of Engineering**

***Summer 2019, ECE 6913***

*Instructor: Azeez Bhavnagarwala, email: ajb20@nyu.edu*

***Please fill in your name:***

**HW Assignment 6:**

[released **Wednesday July 17th 2019**] [due **Wednesday July 24th 2019, *before* 11:55 PM**]

You are allowed to discuss HW assignments *only with other colleagues taking the class with you*. You are *not* allowed to share your solutions with other colleagues in the class. Please feel free to reach out to the Instructor during office hours or by appointment if you need any help with the HW. **please show all of your work**

Please enter your responses in this Word document after you download it from NYU Classes. *Please use the NYU Classes portal to send in your completed HW. If you are having difficulty doing this before the deadline, please convert it to PDF when you are done and email it to ajb20@nyu.edu before 11:55 PM on Wednesday July 24th 2019.*

**1.** Caches are important to providing a high-performance memory hierarchy to processors. Below is a list of 64-bit memory address references, given as word addresses.

`0x03, 0xb4, 0x2b, 0x02, 0xbf, 0x58, 0xbe, 0x0e, 0xb5, 0x2c, 0xba, 0xfd`

*1.1* For each of these references, identify the binary word address, the tag, and the index given a direct-mapped cache with 16 one-word blocks. Also list whether each reference is a hit or a miss, assuming the cache is initially empty.

16 words in cache, a block requested from the cache is 1 word, each memory reference is for a given word.

- Because there are 16 words in the cache, an address X maps to the direct-mapped cache word X modulo 16.
- That is, the low-order $\log_2 16=4$ bits : So, **low order 4 bits are used as the cache index.**
- Since the cache is assumed initially empty, assume there is no valid data in it
- *Since none of the memory references repeat with identical* ***tag and index***, all of them will ***miss***
- ***Each Block now has 2 Words***, with a total of ***8 Blocks***, larger Block, fewer Cache entries

- The low order log $_2 8$ = ***3 bits are used as the Cache index*** with **a one-bit field as the offset to *select a Word in the block***
- Cache again assumed to be initially empty – first access to a block is a miss with data from lower level memory written into it
- The Tag and cache index bits repeat thrice – identified in pairs with identical colors (green, brown and blue in Table below)
- A miss results in *both words* being fetched from lower level memory and written into cache so even though the same block is not requested at a later time, it registers as a hit – primary advantage of having multi-word blocks or lines
- Fewer Cache entries translates into shorter critical path to decode an entry improving Cache cycle time

| Hex Memory Reference | Binary Reference | Tag | Index | Hit / miss |
|---|---|---|---|---|
| 0x03 | 0000 0011 | 0 | 3 | M |
| 0xb4 | 1011 0100 | b | 4 | M |
| 0x2b | 0010 1011 | 2 | b | M |
| 0x02 | 0000 0010 | 0 | 2 | M |
| 0xbf | 1011 1111 | b | f | M |
| 0x58 | 0101 1000 | 5 | 8 | M |
| 0xbe | 1011 1110 | b | e | M |
| 0x0e | 0000 1110 | 0 | e | M |
| 0xb5 | 1011 0101 | b | 5 | M |
| 0x2c | 0010 1100 | 2 | c | M |
| 0xba | 1011 1010 | b | a | M |
| 0xfd | 1111 1101 | f | d | M |

*1.2* For each of these references, identify the binary word address, the tag, the index, and the offset given a direct-mapped cache with two-word blocks and a total size of eight blocks. Also list if each reference is a hit or a miss, assuming the cache is initially empty.

**larger blocks => lower miss rate, smaller cycle time for same cache size in words**

| Hex Memory Reference | Binary Reference | Tag | Cache Index | Block Offset | Hit / miss |
|---|---|---|---|---|---|
| 0x03 | 0000 0011 | ***0*** | ***1*** | ***1*** | ***M*** |
| 0xb4 | 1011 0100 | *b* | *2* | *0* | *M* |
| 0x2b | 0010 1011 | 2 | 5 | 1 | M |
| 0x02 | 0000 0010 | ***0*** | ***1*** | ***0*** | ***H*** |
| 0xbf | 1011 1111 | ***b*** | ***7*** | ***1*** | ***M*** |
| 0x58 | 0101 1000 | 5 | 4 | 0 | M |
| 0xbe | 1011 1110 | ***b*** | ***7*** | ***0*** | ***H*** |
| 0x0e | 0000 1110 | 0 | 7 | 0 | M |
| 0xb5 | 1011 0101 | *b* | *2* | *1* | *H* |
| 0x2c | 0010 1100 | 2 | 6 | 0 | M |
| 0xba | 1011 1010 | b | 5 | 0 | M |
| 0xfd | 1111 1101 | f | 6 | 1 | M |

*1.3* You are asked to optimize a cache design for the given references. There are three direct-mapped cache designs possible, all with a total of eight words of data:
C1 has 1-word blocks,
C2 has 2-word blocks, and
C3 has 4-word blocks.

- Total cache size is 8 Words
- same set of memory references
- Optimize for the number of words in a Block
- 3 possible Direct Mapped cache designs
- With 1 word, 2 words, 4 words per Block:
- Cache has 8 entries (Cache index 3 bits), 4 entries (Cache index 2 bits), 2 entries (Cache index 1 bit)

| Word Address | Binary Address | Tag (5 bits in hex) | Cache 1 block size = 1 word | | Cache 2 block size = 2 word | | Cache 3 block size = 4 word | |
|---|---|---|---|---|---|---|---|---|
| | | | Index (3 bits) | Hit/Miss | Index (2 bits) | Hit/Miss | Index (1 bit) | Hit/Miss |
| 0x03 | 0 0000 011 | 0x00 | 3 | M | 1 | M | 0 | M |
| 0xb4 | 1 0110 100 | 0x16 | 4 | M | 2 | M | 1 | M |
| 0x2b | 0 0101 011 | 0x05 | 3 | M | 1 | M | 0 | M |
| 0x02 | 0 0000 010 | 0x00 | 2 | M | 1 | M | 0 | M |
| 0xbf | 1 0111 111 | 0x17 | 7 | M | 3 | M | 1 | M |
| 0x58 | 0 1011 000 | 0x0b | 0 | M | 0 | M | 0 | M |
| 0xbe | 1 0111 110 | 0x17 | 6 | M | 3 | H | 1 | H |
| 0x0e | 0 0001 110 | 0x01 | 6 | M | 3 | M | 1 | M |
| 0xb5 | 1 0110 101 | 0x16 | 5 | M | 2 | H | 1 | M |
| 0x2c | 0 0101 100 | 0x05 | 4 | M | 2 | M | 1 | M |
| 0xba | 1 0111 010 | 0x17 | 2 | M | 1 | M | 0 | M |
| 0xfd | 1 1111 101 | 0x1F | 5 | M | 2 | M | 1 | M |

- The 2-word and 4-word columns with entries marked in green are misses even though the tag bits and the index bits match to a previous word access (in light blue). Since an access previous to the miss (highlighted in green) corresponding to the same cache line (but different tag bits) was a miss (in yellow) that cache line was replaced and no longer holds the data required by the miss (in green)
- Note that in 1.2, we saw an increase in the Block size (in Words) lower the miss rate. However, as we increase in Word size to 4 Words, the miss rate rises. This is because as the Block size (4 Words in Cache 3) becomes comparable to the Cache size (8 Words), the competition for Blocks increases with the entire Block replaced if a single entry misses. Limits on increasing Block size to improve hit rate as seen in 1.2 are imposed by the size

of the Cache itself. Fig 5.11 in text demonstrates this observation as well:

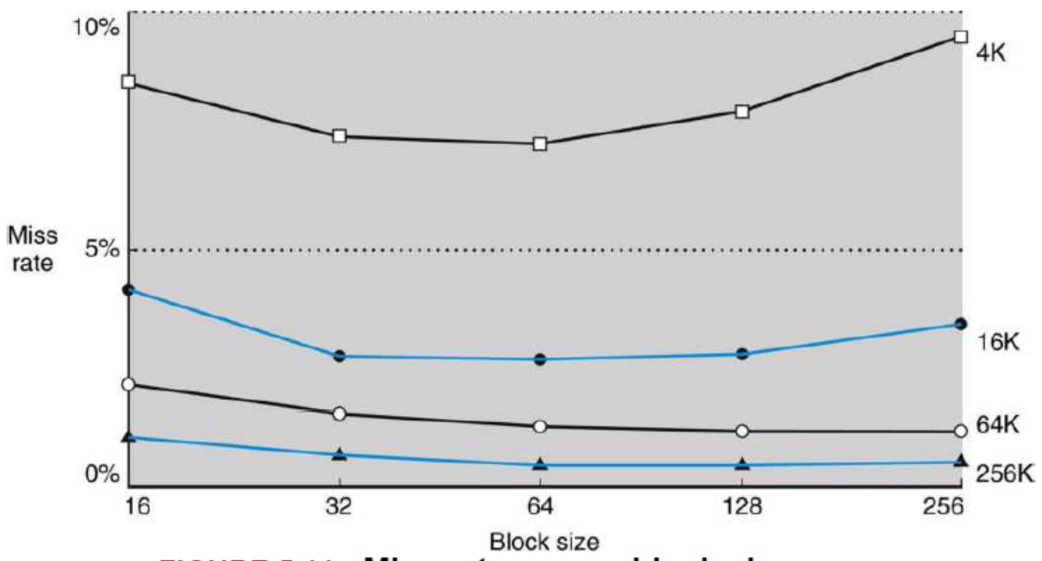


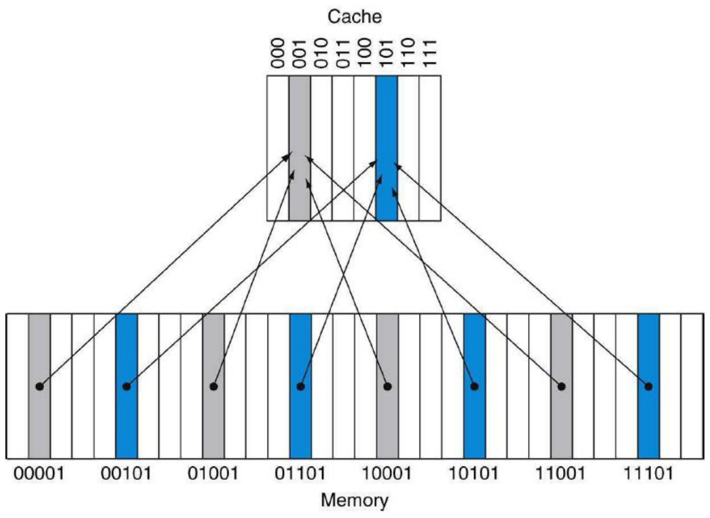
**FIGURE 5.11** **Miss rate versus block size.** Note that the miss rate actually goes up if the block size is too large relative to the cache size. Each line represents a cache of different size. (This figure is independent of associativity, discussed soon.) Unfortunately, SPEC CPU2000 traces would take too long if block size were included, so these data are based on SPEC92.

**2.** *Section 5.3* shows the typical method to index a direct-mapped cache, specifically (Block address) modulo (Number of blocks in the cache). Assuming a 64-bit address and 1024 blocks in the cache, consider a different indexing function, specifically (Block address[63:54] XOR Block address[53:44]). Is it possible to use this to index a direct-mapped cache? If so, explain why and discuss any changes that might need to be made to the cache. If it is not possible, explain why.

In any direct mapped cache – with a 10-bit index in this given problem,

The 10-bit cache index must be unique to any given block address. In other words, a given block address cannot map to more than one cache index – as shown in the color-coded figure (5.8 from text) below. If this were not the case, then a given block address could map to multiple cache locations.

So, at a minimum, any function that produces a unique 10-bit output corresponding to the 10-bit cache index and which can cover all possible cache blocks, is sufficient.



Clearly, [block address modulo number of blocks in cache] satisfies this minimum requirement

Assuming a 10-bit cache index that identifies one of 1024 cache entries in a direct mapped cache are given by bits [53:44] in a 64 bit memory address – lets assume for simplicity that bits [43:0] correspond to block and byte offsets and that the 20 most significant bits of the 64 bit address [63:44] correspond to the Memory address space of 1M Blocks

Let’s use, for any given 64 bit Memory address M:
Proposed Indexing Function: M[63:54] XOR M[53:44]
Observation A: For each unique set of bits in M[63:54], there are exactly 1024 possible combinations of M[53:44] in the 64 bit address provided corresponding to the opportunity to map the unique M[63:54] bits to any of exactly 1024 cache entries addressed by M[53:44]

Observation B: For each unique set of bits in [63:54] there is exactly only ONE result of the XOR function for each of the 1024 combinations in [53:44] satisfying the unique cache index for any given memory address vector of 64 bits

From the above 2 observations, the proposed XOR function to index the cache in a direct mapped cache is sufficient

**3.** For a direct-mapped cache design with a 64-bit address, the following bits of the address are used to access the cache.

| Tag | Index | Offset |
|---|---|---|
| 63–10 | 9–5 | 4–0 |

*3.1* What is the cache block size (in words)?

*3.2* How many blocks does the cache have?

*3.3* What is the ratio between total bits required for such a cache implementation over the data storage bits?

*Beginning from power on, the following byte-addressed cache references are recorded.*

| Address | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Hex | 00 | 04 | 10 | 84 | E8 | A0 | 400 | 1E | 8C | C1C | B4 | 884 |
| Dec | 0 | 4 | 16 | 132 | 232 | 160 | 1024 | 30 | 140 | 3100 | 180 | 2180 |

*3.4* For each reference, list (1) its tag, index, and offset, (2) whether it is a hit or a miss, and (3) which bytes were replaced (if any).

*3.5* What is the hit ratio?

*3.6* List the final state of the cache, with each valid entry represented as a record of <*index, tag, data*>. For example,

**<0, 3, Mem[0xC00]-Mem[0xC1F]>**

**Each Block has 32 Bytes** (offset is 5 bits wide) or 4 64-bit words or 4 8-Byte words (total of 32 Bytes).

- 2 bits determine one of 4 (64-bit) Words in the Block, 3 least significant bits determine the byte in each 64-bit word
- 5 bits in the index field indicate **32 Blocks or 32 lines in the cache**
- The cache stores 32 Blocks x 4 Words/Block x 8 Bytes/word = 1024 Bytes = *8192* bits
- In addition to data, 53 bits for Tag and 1 valid bit
  Total bits required = 8192 + 53x32 + 1 x 32 = *9920* bits

**9920 / 8192 = 1.21**

| **Byte Address** | **Binary Address** | **Tag** | **Index** | **Offset** | **Line replaced** | **Hit/Miss** |
|---|---|---|---|---|---|---|
| 0x00 | 00 0 0000 0 0000 | 0x0 | 0x00 | 0x00 | No | M |
| 0x04 | 00 0 0000 0 0100 | 0x0 | 0x00 | 0x04 | No | H |
| 0x10 | 00 0 0000 1 0000 | 0x0 | 0x00 | 0x10 | No | H |
| 0x84 | 00 0 0100 0 0100 | 0x0 | 0x04 | 0x04 | No | M |
| 0xe8 | 00 0 0111 0 1000 | 0x0 | 0x07 | 0x08 | No | M |
| 0xa0 | 00 0 0101 0 0000 | 0x0 | 0x05 | 0x00 | No | M |
| 0x400 | 01 0 0000 0 0000 | 0x1 | 0x00 | 0x00 | **Yes** | M |
| 0x1e | 00 0 0000 1 1110 | 0x0 | 0x00 | 0x1e | **Yes** | M |
| 0c8c | 00 0 0100 0 1100 | 0x0 | 0x04 | 0x0c | No | H |
| 0xc1c | 11 0 0000 1 1100 | 0x3 | 0x00 | 0x1c | **Yes** | M |
| 0xb4 | 00 0 0101 1 0100 | 0x0 | 0x05 | 0x14 | No | H |
| 0x884 | 10 0 0100 0 0100 | 0x2 | 0x04 | 0x04 | **Yes** | M |

The Cache line is replaced only when the tag bits of the memory reference change. All 32 bytes in that cache line are replaced.

Hit Ratio = 4/12 = 33.33%

## Brief description of Write Policies

| Write hit policy | Write miss policy |
|---|---|
| Write Through | Write Allocate |
| Write Through | No Write Allocate |
| Write Back | Write Allocate |
| Write Back | No Write Allocate |

*Possible combinations of interaction policies with main memory on write.*

Write Through with Write Allocate:

on Write **hits** it writes to cache and main memory

on Write **misses (tag mismatch)** it *updates the block in main memory* and *brings the block to the cache* (so that the next Write or Read to the same cache line will not miss)

*The benefit of bringing the updating the block in cache following Write misses is that the data is available in the cache for a subsequent Read access.*

The disadvantage of bringing updating the block in the cache following a write miss is that in a subsequent Write hit, this data in the cache that costed bandwidth and performance with a Write allocate miss policy is overwritten anyways and the memory is still updated on this subsequent Write. So, *Bringing the block to cache from updated memory on a write miss would not make a lot of sense.*

Write Through with No Write Allocate:

on **hits** it writes to cache and main memory;

on **misses** it *updates the block in main memory* not bringing that block to the cache;

Subsequent writes to the block will *update main memory because Write Through policy is employed*. So, some *time is saved not bringing the block in the cache on a miss* because it appears useless anyway.

However, *subsequent Reads to this address will report a miss because the cache line was not updated with a no write allocate policy.* These Read misses likely evict the cache line (since the cache line is inconsistent with memory) requiring the memory to be updated

Write Back with Write Allocate:

on **hits** it writes to cache setting dirty bit for the block, main memory is not updated;

on **misses** it *updates the block in main memory* **and** *brings the block to the cache*;

Subsequent writes to the same block, if the block originally caused a miss, *will hit in the cache next time*, setting dirty bit for the block. That *will eliminate extra memory accesses and result in very efficient execution* compared with Write Through with Write Allocate combination.

**Write Back with No Write Allocate:**

on **hits** it writes to cache setting dirty bit for the block, main memory is not updated;
on **misses** it *updates the block in main memory* **not** *bringing that block to the cache*;

Subsequent writes to the same block, if the block originally caused a miss, *will generate misses* all the way and *result in very inefficient execution.*

**4.** Recall that we have two write policies and two write allocation policies, and their combinations can be implemented either in L1 or L2 cache. Assume the following choices for L1 and L2 caches:

| **L1** | **L2** |
|---|---|
| Write through, non-write allocate | Write back, write allocate |

*4.1* Buffers are employed between different levels of memory hierarchy to reduce access latency. For this given configuration, list the possible buffers needed between L1 and L2 caches, as well as L2 cache and memory.

*4.2* Describe the procedure of handling an L1 write-miss, considering the components involved and the possibility of replacing a dirty block.

*4.3* For a multilevel exclusive cache configuration (a block can only reside in one of the L1 and L2 caches), describe the procedures of handling an L1 write-miss and an L1 read-miss, considering the components involved and the possibility of replacing a dirty block.

**4.1** The L1 cache has a low write miss penalty while the L2 cache has a high write miss penalty since the latency between RAM and L2 is much higher than the latency between L1 and L2.

A write buffer between the L1 and L2 cache would effectively pipeline the write to the L2 cache enabling it *to require only one cycle for the Write*. Since the buffer would hold data to be written into L2 from L1 and can be designed to be deep enough, it can prevent stalls from subsequent write misses in L1.

The L2 cache would benefit from write buffers between L1 and L2 when replacing a dirty block in L2, *since the new block would be read into and held by the buffer between L1 and L2 before* the dirty block is physically written to memory, only after which it could be overwritten in the L2 by the new block.

**4.2** On an L1 write miss, the word is written directly to L2 *without bringing its block into the L1 cache [write through/write back with no write allocate policy]*. If this results in an L2 miss, its block must be brought into the L2 cache from Memory, possibly replacing a dirty block, which must first be written to memory

**4.3** After an L1 write miss, the block will reside in L2 but not in L1.

A subsequent read miss on the same block will require that the block in L2 be written back to memory, transferred to L1, and invalidated in L2.

**5.** Consider the following program and cache behaviors.

| Data Reads per 1000 Instructions | Data Writes per 1000 Instructions | Instruction Cache Miss Rate | Data Cache Miss Rate | Block Size (bytes) |
|---|---|---|---|---|
| 250 | 100 | 0.30% | 2% | 64 |

*5.1* Suppose a CPU with a write-through, writeallocate cache achieves a CPI of 2. What are the read and write bandwidths (measured by bytes per cycle) between RAM and the cache? (Assume each miss generates a request for one block.)

*5.2* For a write-back, write-allocate cache, assuming 30% of replaced data cache blocks are dirty, what are the read and write bandwidths needed for a CPI of 2?

5.1 We can determine the bandwidths for Read and Write between cache and RAM by calculating the components for instruction memory and data memory arrays:

**Instruction bandwidth:**

When the CPI is 2, there are, on average, 0.5 instruction accesses per cycle.
**0.5 instructions read from Instruction memory per cycle**
0.3% of these *instruction* accesses cause a cache ***Read*** miss (and subsequent memory request).
**[0.5 instr/cycle] x [0.003 misses/instruction] = missed instructions/cycle**
Assuming each miss requests one block and each block is 64 bytes [8 words with 8 bytes (64 bits) per word] , instruction accesses generate an average of
**[0.5 instr/cycle] x [0.003 misses/instruction] x[64 bytes/block] =**

**= 0.096 bytes/cycle of *read traffic***

**Read Data bandwidth:**
25% of instructions generate a ***read*** request.
**[0.5 instr/cycle] x [0.25 Read Data Accesses/instruction] = [0.125 Read Data Accesses / cycle]**
2% of these generate a cache miss;
**[0.125 Read Data Accesses / cycle] x [0.02 misses / Read Data Access] = 0.0025 Read Misses/cycle**
Assuming each miss requests one block and each block is 64 bytes [8 words with 8 bytes (64 bits) per word] ,
**[0.0025 Read Misses/cycle] x [64 Bytes/block] x [1 block/miss] = 0.0025 x 64 Bytes/cycle = 0.16 Bytes/cycle**

**Write Data bandwidth:**
10% of instructions generate a ***write*** request.
**[0.5 instr/cycle] x [0.10 Write Data Accesses/instruction] = [0.05 Write Data Accesses / cycle]**
All of the words written to the cache must be written into Memory:
**[0.05 Write Data Accesses / cycle] x [8 bytes/word] x [1 word/write-through] = 0.4 Bytes/cycle**

For a *Write-allocate policy*, a Write miss also makes *a* ***read*** *request* to RAM

**[0.5 inst/cycle] x [0.10 Write Data Accesses/instruction] x [0.02 misses/Write Data Access] x [64 Bytes/miss]**
**= 0.064 Bytes/cycle**
Assuming each miss requests one Word (8 bytes ) since this is a write-through cache with only 1 word written per miss into memory ,
**[0.001 Write Misses/cycle] x [8 Bytes/word] x [1 word/miss] = 0.001 x 8 Bytes/cycle =** 0.008 **Bytes/cycle**

<u>**Total Read Bandwidth:**</u>
**0.096** (Instruction memory) + **0.16** (data memory) + **0.064** (Write-miss in Write-through cache with Write Allocate) **Bytes/cyle =** 0.32 **Bytes/cycle**
<u>**Total Write Bandwidth:**</u>
0.4 **Bytes/cycle**

**5.2**

With a write-back, write allocate cache, data are only written to memory on a cache miss. But, it is written to memory on every cache miss (both read and write), because any line could have dirty data when evicted, even if the eviction is caused by a read request

assuming 30% of replaced data cache blocks are dirty what are the read and write bandwidths needed assuming the same CPI?

the data write bandwidth requirement becomes

**[0.5 inst/cycle] x [0.10 Write Data Accesses/instruction + 0.25 Read Data Accesses/instruction]**
***= 0.175 Accesses/cycle***
**[0.175 Accesses/cycle] x [0.02 misses /Access]**
***= 0.0035 misses/cycle***
**[0.0035 misses/cycle] x [0.3 blocks/miss]**
***= 0.00105 blocks/cycle***
**[0.00105 blocks/cycle] x [64 bytes/block] =**
***=0.0672 bytes/cycle***

**6.** Media applications that play audio or video files are part of a class of workloads called "streaming" workloads (i.e., they bring in large amounts of data but do not reuse much of it). Consider a video streaming workload that accesses a 512 KiB working set sequentially with the following word address stream:

**0, 1, 2, 3, 4, 5, 6, 7, 8, 9 …**

*6.1* Assume a 64 KiB direct-mapped cache with a 32-byte block. What is the miss rate for the address stream above? How is this miss rate sensitive to the size of the cache or the working set? How would you categorize the misses this workload is experiencing, based on the 3C model?

*6.2* Re-compute the miss rate when the cache block size is 16 bytes, 64 bytes, and 128 bytes. What kind of locality is this workload exploiting?

*6.3* "*Prefetching*" is a technique that leverages predictable address patterns to speculatively bring in additional cache blocks when a particular cache block is accessed. One example of prefetching is a stream buffer that prefetches sequentially adjacent cache blocks into a separate buffer when a particular cache block is brought in. If the data are found in the prefetch buffer, it is considered as a hit, moved into the cache, and the next cache block is prefetched. Assume a two-entry stream buffer; and, assume that the cache latency is such that a cache block can be loaded before the computation on the previous cache block is completed. What is the miss rate for the address stream above?

**6.1** Since the addresses stream are word addresses and each 32-byte block contains four words. Thus, every fourth access will be a miss (i.e., a miss rate of 1/4). All misses are compulsory misses since the cache miss has been caused by the first access to a block that has never been in the cache. The miss rate is not sensitive to the size of the cache or the size of the working set. It is, however, sensitive to the access pattern and block size.

**6.2** The miss rates double if the block size halves since the 16 byte block contains only 2 words with every 2nd access becoming a miss ( miss rate of ½). Similarly, for a cache block size of 64 bytes (8 words), the miss rate drops to 1/8 since every 8th access is a miss. For a cache block size of 128 bytes (16 words) every 16th access becomes a miss with a miss rate of 1/16.

**6.3** The miss rate is 0: The pre-fetch buffer always has the next request ready.

**7.** Cache block size (B) can affect both miss rate and miss latency. Assuming a machine with a base CPI of 1, and an average of 1.35 references (both instruction and data) per instruction, find the block size that minimizes the total miss latency given the following miss rates for various block sizes.

| | | | | |
|---|---|---|---|---|
| 8: 4% | 16: 3% | 32: 2% | 64: 1.5% | 128: 1% |

*7.1* What is the optimal block size for a miss latency of 20 ×B cycles?

Average Memory Access Time (AMAT) as a linear function of Miss Latency is given by

AMAT = Miss Rate x Miss Latency

```
AMAT for B =     8:          0.040 × (20 × 8) = 6.40
AMAT for B =     16:         0.030 × (20 × 16) = 9.60
AMAT for B =     32:         0.020 × (20 × 32) = 12.80
AMAT for B =     64:         0.015 × (20 × 64) = 19.20
AMAT for B =     128: 0.010 × (20 × 128) = 25.60
```
*B = 8 is optimal.*

*The smallest Block size yields the lowest AMAT* since the miss latency is the lowest with the fewest Bytes that need to be transferred on a Miss

*7.2* What is the optimal block size for a miss latency of 24 +B cycles?

For cases where the Miss Latency begins to increase noticeably only for Block sizes larger than some threshold size, it is modeled below. For cases as these, while the Miss Latency does not increase as rapidly with Block size until it reaches some threshold, the Miss rate drops steadily with increasing Block size yielding an optimal Block size for minimum AMAT:

AMAT = Miss Rate x (24 + B)

```
AMAT for B = 8: 0.040 × (24 + 8) = 1.28
AMAT for B = 16: 0.030 × (24 + 16) = 1.20
AMAT for B = 32: 0.020 × (24 + 32) = 1.12
AMAT for B = 64: 0.015 × (24 + 64) = 1.32
AMAT for B = 128: 0.010 × (24 + 128) = 1.52
```
*B = 32 is optimal*

*7.3* For constant miss latency, what is the optimal block size?

B = *128 is optimal:* Minimizing the miss rate minimizes the total miss latency.

**8.** In this exercise, we will look at the different ways capacity affects overall performance. In general, cache access time is proportional to capacity. Assume that main memory accesses take 70 ns and that 36% of all instructions access data memory. The following table shows data for L1 caches attached to each of two processors, P1 and P2.

| | L1 Size | L1 Miss Rate | L1 Hit Time |
|---|---|---|---|
| P1 | 2 KiB | 8.0% | 0.66 ns |
| P2 | 4 KiB | 6.0% | 0.90 ns |

*8.1* Assuming that the L1 hit time determines the cycle times for P1 and P2, what are their respective clock rates?

```
Cycle times for P1 and P2 when L1 hit time determines cycle
time:

P1: 1.515 GHz; P2: 1.11 GHz
```

*8.2* What is the Average Memory Access Time for P1 and P2 (in cycles)?

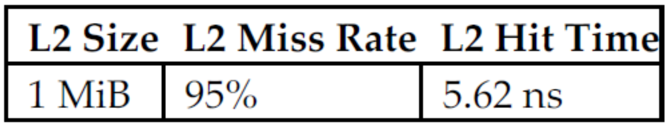
```
Average Memory Access Time (in cycles) = # of cycles for a hit +
Miss Rate x Miss Penalty

Miss penalty for P1 = 70ns/0.66ns = 107 cycles
Miss penalty for P2 = 70ns/0.90ns = 78 cycles

P1: AMAT = 1 + 0.08 x 107 cycles = 9.56 cycles or 6.31 ns
P2: AMAT = 1 + 0.06 x 78   cycles = 5.68 cycles or 5.11 ns
```

*8.3* Assuming a base CPI of 1.0 without any memory stalls, what is the total CPI for P1 and P2? Which processor is faster? (When we say a "base CPI of 1.0", we mean that instructions complete in one cycle, unless either the instruction access or the data access causes a cache miss.)

*we will now consider the addition of an L2 cache to P1 (to presumably make up for its limited L1 cache capacity). Use the L1 cache capacities and hit times from the previous table when solving these problems. The L2 miss rate indicated is its local miss rate.*



| L2 Size | L2 Miss Rate | L2 Hit Time |
|---|---|---|
| 1 MiB | 95% | 5.62 ns |

Total CPI for P1 and P2 are determined by calculating:

**For P1:**
Each instruction requires 1 cycle for a hit

1 cycle

If the instruction fetch from the Instruction memory misses at Miss Rate of 8%, the instruction incurs a 107 cycle delay

+ 0.08 x 107 cycles = 8.56

36% of the instructions also require access to Data Memory at a Miss rate of 8% which incur an additional delay of

+ 0.36 x 0.08 x 107 cycles = 3.08

So, 1 + 0.08 x 107 + 0.36 x 0.08 x 107 = 12.64 cycles @ 0.66ns/cycle = 8.34ns

**For P2:**

Each instruction requires 1 cycle for a hit

1 cycle

If the instruction fetch from the Instruction memory misses at Miss Rate of 8%, the instruction incurs a 107 cycle delay

+ 0.06 x 78 cycles = 4.68

36% of the instructions also require access to Data Memory at a Miss rate of 8% which incur an additional delay of

+ 0.36 x 0.06 x 78 cycles = 1.68

So, 1 + 0.08 x 78 + 0.36 x 0.08 x 78 = 7.36 cycles @ 0.66ns/cycle = 6.63ns

*8.4* What is the AMAT for P1 with the addition of an L2 cache? Is the AMAT better or worse with the L2 cache?

An L2 access requires 9 cycles (5.62ns/0.66ns).

All memory accesses require at least one cycle.

8% of memory accesses miss in the L1 cache and make an L2 access, which takes 9 cycles.

95% of all L2 access are misses and require a 107 cycle memory lookup.

AMAT = 1 + .08[9 + 0.95*107] = **9.85 cycles** – worse than without the L2 (9.56 cycles)

*8.5* Assuming a base CPI of 1.0 without any memory stalls, what is the total CPI for P1 with the addition of an L2 cache?

CPI for P1 with an L2 cache is [AMAT] + [%DataMemory Accesses/cycle] * [AMAT-1]

9.85 * 0.36*8.85 = 13.04 cycles

*8.6* What would the L2 miss rate need to be in order for P1 with an L2 cache to be faster than P1 without an L2 cache?

AMAT with L2 = 1 + .08[9 + MR_L2*107] = AMAT without L2 = 1 + 0.08 x 107 cycles = **9.56**

So, When the MR_L2 drops to below

[(9.56 -1)/0.08 -9 ]/107 = 91.5%, the AMAT with L2 will become faster than without the L2

*8.7* What would the L2 miss rate need to be in order for P1 with an L2 cache to be faster than P2 without an L2 cache?

We want P1's average time per instruction to be less than 6.63 ns. This means that we want

(CPI_P1 * 0.66) < 6.63. Th us, we need CPI_P1 < 10.05

$CPI\_P1 = AMAT\_P1 + 0.36(AMAT\_P1 - 1)$

$AMAT\_P1 + 0.36(AMAT\_P1-1) < 10.05$

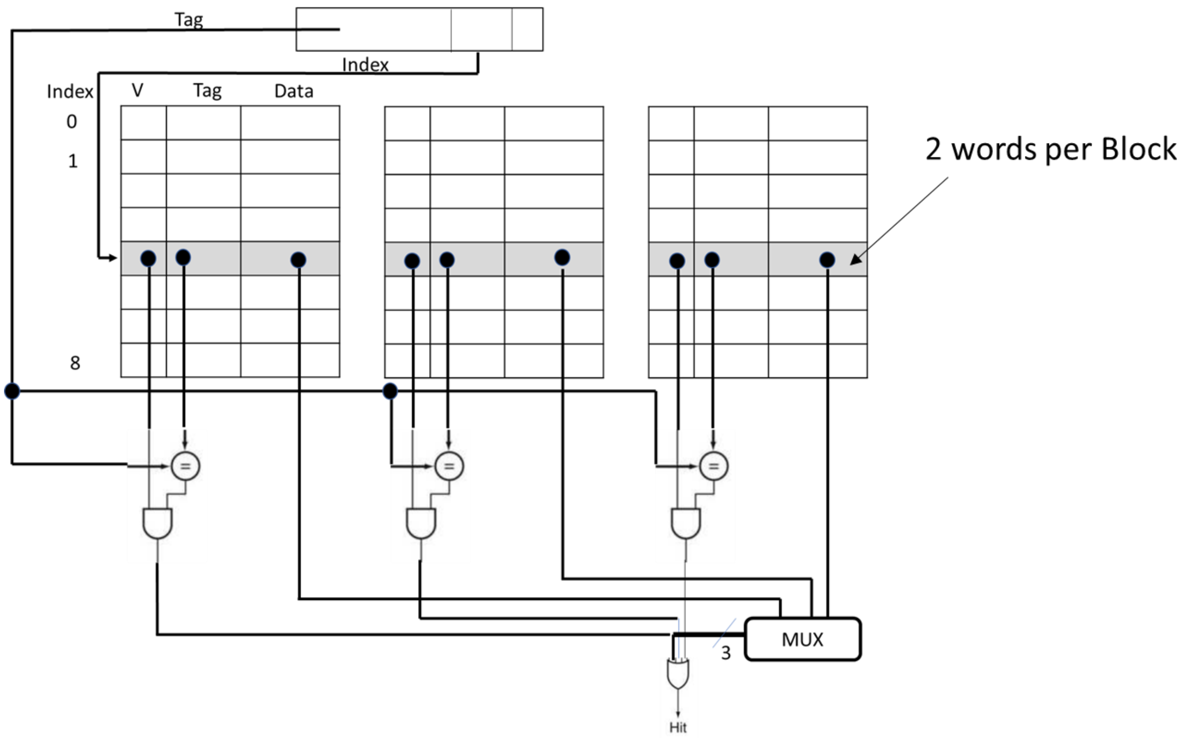
$AMAT\_P1 < 7.65$.

or
$1 + 0.08[9 + MR\_L2*107] < 7.65$
or
$MR\_L2 < 0.693$
or
**69.3%**.

**9.** This exercise examines the effect of different cache designs, specifically comparing associative caches to the direct-mapped caches from *Section 5.4*. For these exercises, refer to the sequence of word address shown below.

**`0x03, 0xb4, 0x2b, 0x02, 0xbe, 0x58, 0xbf, 0x0e, 0x1f, 0xb5, 0xbf, 0xba, 0x2e, 0xce`**

*9.1* Sketch the organization of a three-way set associative cache with two-word blocks and a total size of 48 words. Your sketch should have a style similar to *Figure 5.18*, but clearly show the width of the tag and data fields.



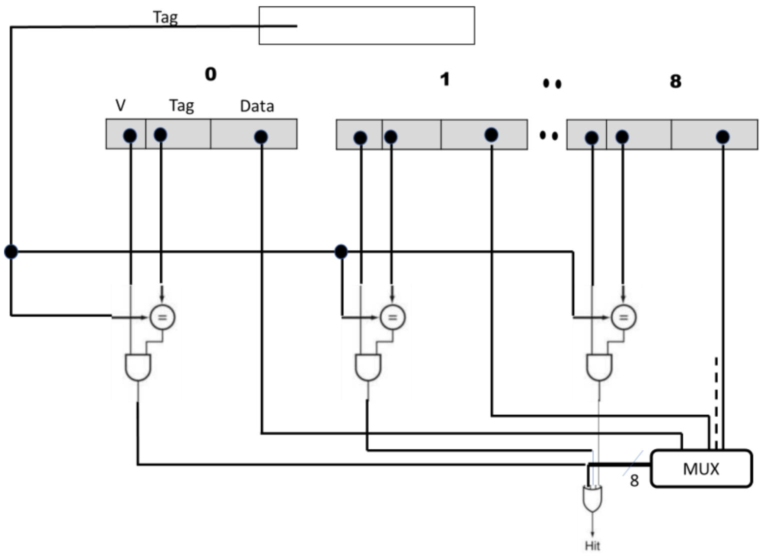
9.2 Trace the behavior of the cache from Exercise 9.1 Assume a true LRU replacement policy. For each reference, identify

- *the binary word address,*
- *the tag,*
- *the index,*
- *the offset*
- *whether the reference is a hit or a miss, and*
- *which tags are in each way of the cache after the reference has been handled.*

| Hex | Binary | Tag | Index | Offset | Hit / Miss |
|---|---|---|---|---|---|
| ***0x03*** | ***0000 001 1*** | ***0x0*** | ***1*** | ***1*** | ***M*** |
| ***0xb4*** | ***1011 010 0*** | ***0xb*** | ***2*** | ***0*** | ***M*** |
| 0x2b | 0010 101 1 | 0x2 | 5 | 1 | M |
| **0x02** | **0000 001 0** | **0x0** | **1** | **0** | **H** |
| ***0xbe*** | ***1011 111 0*** | ***0xb*** | ***7*** | ***0*** | ***M*** |
| 0x58 | 0101 100 0 | 0x5 | 4 | 0 | M |
| **0xbf** | **1011 111 1** | **0xb** | **7** | **1** | **H** |
| 0x0e | 0000 111 0 | 0x0 | 7 | 0 | M |
| 0x1f | 0001 111 1 | 0x1 | 7 | 1 | M |
| **0xb5** | **1011 010 1** | **0xb** | **2** | **1** | **H** |
| **0xbf** | **1011 111 1** | **0xb** | **7** | **1** | **H** |
| 0xba | 1011 101 0 | 0xb | 5 | 0 | M |
| 0x2e | 0010 111 0 | 0x2 | 7 | 0 | M |
| 0xce | 1100 111 0 | 0xc | 7 | 0 | M |

*9.3* Sketch the organization of a fully associative cache with one-word blocks and a total size of eight words. Your sketch should have a style similar to Figure 5.18, but clearly show the width of the tag and data fields.

```
Note that a fully associative cache does not have an index or offset fields
in the address - all of the bits in the cache are a single bit for the Valid
bit (V), Tag bits (63) and Data bits (64)
```



*9.4* Trace the behavior of the cache from Exercise 9.3. Assume a true LRU replacement policy. For each reference, identify

- *the binary word address,*
- *the tag,*
- *the index,*
- *the offset,*
- *whether the reference is a hit or a miss*
- *the contents of the cache after each reference has been handled.*

Cache has a total size of **eight words.** Because this cache is **fully associative** and has **one-word blocks,** *there is no index and no offset.* Consequently, the *word address is equivalent to the tag.*

| **Hex** | **Binary** | **Tag** | **Hit/Miss** | **Content** |
|---|---|---|---|---|
| 0x03 | 0000 0011 | 0x03 | M | 3 |
| 0xb4 | 1011 0100 | 0xb4 | M | 3, b4 |
| 0x2b | 0010 1011 | 0x2b | M | 3, b4, 2b |
| 0x02 | 0000 0010 | 0x02 | M | 3, b4, 2b, 2 |
| 0xbe | 1011 1110 | 0xbe | M | 3, b4, 2b, 2, be |
| 0x58 | 0101 1000 | 0x58 | M | 3, b4, 2b, 2, be, 58 |
| **0xbf** | 1011 1111 | 0x**bf** | M | 3, b4, 2b, 2, be, 58, **bf** |
| 0x0e | 0000 1110 | 0x0e | M | 3, b4, 2b, 2, be, 58, **bf**, e |
| 0x1f | 0001 1111 | 0x1f | M | b4, 2b, 2, be, 58, **bf**, e, 1f |
| 0xb5 | 1011 0101 | 0xb5 | M | 2b, 2, be, 58, **bf**, e, 1f, b5 |
| **0xbf** | 1011 1111 | 0x**bf** | **H** | 2b, 2, be, 58, e, 1f, b5, **bf** |
| 0xba | 1011 1010 | 0xba | M | 2, be, 58, e, 1f, b5, bf, ba |
| 0x2e | 0010 1110 | 0x2e | M | be, 58, e, 1f, b5, bf, ba, 2e |
| 0xce | 1100 1110 | 0xce | M | 58, e, 1f, b5, bf, ba, 2e, ce |

*9.5* Sketch the organization of a fully associative cache with two-word blocks and a total size of eight words. Your sketch should have a style similar to Figure 5.18, but clearly show the width of the tag and data fields.

Similar to 9.3 except the data field is twice as large holding 2 words

*9.6* Trace the behavior of the cache from Exercise 9.5. Assume an LRU replacement policy. For each reference, identify

- *the binary word address,*
- *the tag,*
- *the index,*
- *the offset,*
- *whether the reference is a hit or a miss,*
- *the contents of the cache after each reference has been handled.*

| **Hex** | **Binary** | **Tag** | **Offset** | **Hit/Miss** | **Content** |
|---|---|---|---|---|---|
| 0x03 | **000 0001** 1 | **0x01** | 1 | M | **[2,3]** |
| 0xb4 | 101 1010 0 | 0x5a | 0 | M | **[2,3],** [b4,b5] |
| 0x2b | 001 0101 1 | 0x15 | 1 | M | **[2,3],** [b4,b5], [2a,2b] |
| 0x02 | **000 0001** 0 | **0x01** | 0 | **H** | [b4,b5], [2a,2b], **[2,3]** |
| 0xbe | 101 1111 0 | 0x5f | 0 | M | [b4,b5], [2a,2b] [2,3], **[be,bf]** |
| 0x58 | 010 1100 0 | 0x2c | 0 | M | [2a,2b] [2,3], **[be,bf],** [58,59] |
| 0xbf | 101 1111 1 | 0x5f | 1 | **H** | [2a,2b] [2,3], [58,59], **[be,bf]** |
| 0x0e | 000 0111 0 | 0x07 | 0 | M | [2,3], [58,59], **[be,bf],** [e,f] |
| 0x1f | 000 1111 1 | 0x0f | 1 | M | [58,59], **[be,bf],** [e,f], [1e,1f] |
| 0xb5 | 101 1010 1 | 0xb5 | 1 | M | **[be,bf],** [e,f], [1e,1f], [b4,b5] |
| 0xbf | 101 1111 1 | 0xbf | 1 | **H** | [e,f], [1e,1f], [b4,b5], **[be,bf]** |
| 0xba | 101 1101 0 | 0xba | 0 | M | [1e,1f], [b4,b5], [be,bf], [ba,bb] |
| 0x2e | 001 0111 0 | 0x2e | 0 | M | [b4,b5], [be,bf], [ba,bb], [2e,2f] |
| 0xce | 110 0111 0 | 0xce | 0 | M | [be,bf], [ba,bb], [2e,2f], [ce,cf] |

*9.7* Repeat Exercise 9.6 using MRU (most recently used) replacement.

| Hex | Binary | Tag | Offset | Hit/Miss | Content |
|---|---|---|---|---|---|
| 0x03 | **000 0001** 1 | **0x01** | 1 | M | [2,3] |
| 0xb4 | **101 1010 0** | **0x5a** | 0 | M | [2,3], [b4,b5] |
| 0x2b | 001 0101 1 | 0x15 | 1 | M | **[2,3],** [b4,b5], [2a,2b] |
| 0x02 | **000 0001** 0 | **0x01** | 0 | **H** | [b4,b5], [2a,2b], **[2,3]** |
| 0xbe | 101 1111 0 | 0x5f | 0 | M | [b4,b5], [2a,2b] [2,3], *[be,bf]* |
| 0x58 | 010 1100 0 | 0x2c | 0 | M | [b4,b5], [2a,2b] [2,3], *[58,59]* |
| 0xbf | 101 1111 1 | 0x5f | 1 | M | [b4,b5], [2a,2b] [2,3], *[be,bf]* |
| 0x0e | 000 0111 0 | 0x07 | 0 | M | [b4,b5], [2a,2b] [2,3], *[e,f]* |
| 0x1f | 000 1111 1 | 0x0f | 1 | M | **[b4,b5],** [2a,2b] [2,3], [1e,1f] |
| 0xb5 | **101 1010 1** | **0x5a** | 1 | **H** | [2a,2b] [2,3], [1e,1f], ***[b4,b5]*** |
| 0xbf | 101 1111 1 | 0x5f | 1 | M | [2a,2b] [2,3], [1e,1f], *[be,bf]* |
| 0xba | 101 1101 0 | 0x5d | 0 | M | [2a,2b] [2,3], [1e,1f], *[ba,bb]* |
| 0x2e | 001 0111 0 | 0x17 | 0 | M | [2a,2b] [2,3], [1e,1f], *[2e,2f]* |
| 0xce | 110 0111 0 | 0x67 | 0 | M | [2a,2b] [2,3], [1e,1f], *[ce,cf]* |

- For *the first memory reference in Hex: 0x03*, if we invert the last bit in this address string (to get the address of the second word in the Block) *we would have the memory reference in Hex: 0x02*. Thus these 2 memory references are identified in the Content column – for this pair and for following pairs.
- In the Content column, *the pair of memory references in italics correspond to the candidate words that are replaced in the next cycle with an MRU replacement policy*. Content column **entries in bold** correspond to entries that are responsive to a hit – that were loaded into the cache from previous memory requests and are requested again

*9.8* Repeat Exercise 9.6 using the optimal replacement policy (i.e., the one that gives the lowest miss rate).

**note:** a **fully associative** cache **with two-word blocks** and a **total size of eight words**

| | **Hex** | **Binary** | **Tag** | **Offset** | **Hit/Miss** | **Content** |
|---|---|---|---|---|---|---|
| 1 | 0x03 | **000 0001** 1 | **0x01** | 1 | M | [2,3] |
| 2 | 0xb4 | **101 1010 0** | **0x5a** | 0 | M | [2,3], [b4,b5] |
| 3 | 0x2b | 001 0101 1 | 0x15 | 1 | M | [2,3], [b4,b5], [2a,2b] |
| 4 | 0x02 | **000 0001** 0 | **0x01** | 0 | **H** | [2,3], [b4,b5], [2a,2b] |
| 5 | 0xbe | 101 1111 0 | 0x5f | 0 | M | *[2,3],* [b4,b5], [2a,2b] [be,bf] |
| 6 | 0x58 | 010 1100 0 | 0x2c | 0 | M | [58,59], [b4,b5], [2a,2b], [be,bf] |
| 7 | 0xbf | 101 1111 1 | 0x5f | 1 | **H** | *[58,59],* [b4,b5], [2a,2b], [be,bf] |
| 8 | 0x0e | 000 0111 0 | 0x07 | 0 | M | *[e,f],* [b4,b5], [2a,2b], [be,bf] |
| 9 | 0x1f | 000 1111 1 | 0x0f | 1 | M | [1e,1f], [b4,b5], [2a,2b], [be,bf] |
| 10 | 0xb5 | **101 1010 1** | **0x5a** | 1 | **H** | [1e,1f], **[b4,b5],** [2a,2b], [be,bf] |
| 11 | 0xbf | 101 1111 1 | 0x5f | 1 | **H** | [1e,1f], *[b4,b5],* [2a,2b], **[be,bf]** |
| 12 | 0xba | 101 1101 0 | 0x5d | 0 | M | [1e,1f], [ba,bb], *[2a,2b],* [be,bf] |
| 13 | 0x2e | 001 0111 0 | 0x17 | 0 | M | [1e,1f], [ba,bb], [2e,2f], *[be,bf]* |
| 14 | 0xce | 110 0111 0 | 0x67 | 0 | M | [1e,1f], [ba,bb], [2e,2f], [ce,cf] |

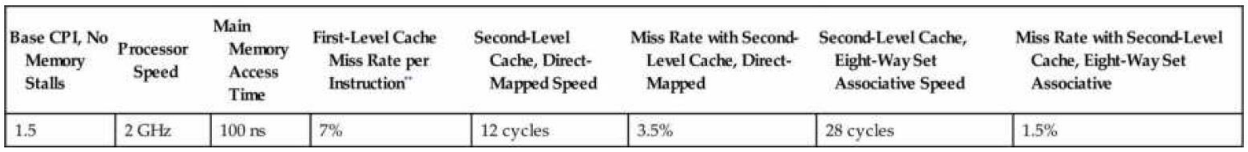
For the 6th memory reference, 0x58, the **LRU** policy used

For the 8th memory reference, 0x0e, the **MRU** policy used

For the 9th memory reference, 0x1f, the **MRU** policy used

For the 12-14th memory reference, 0xbf-0xce, the **LRU** policy used

**10.** Multilevel caching is an important technique to overcome the limited amount of space that a first-level cache can provide while still maintaining its speed. Consider a processor with the following parameters:



| Base CPI, No Memory Stalls | Processor Speed | Main Memory Access Time | First-Level Cache Miss Rate per Instruction** | Second-Level Cache, Direct-Mapped Speed | Miss Rate with Second-Level Cache, Direct-Mapped | Second-Level Cache, Eight-Way Set Associative Speed | Miss Rate with Second-Level Cache, Eight-Way Set Associative |
|---|---|---|---|---|---|---|---|
| 1.5 | 2 GHz | 100 ns | 7% | 12 cycles | 3.5% | 28 cycles | 1.5% |

***First Level Cache miss rate is per instruction. Assume the total number of L1 cache misses*
*(instruction and data combined) is equal to 7% of the number of instructions.*

*10.1* Calculate the CPI for the processor in the table using: 1) only a first-level cache, 2) a second-level direct mapped cache, and 3) a second-level eight-way set associative cache. How do these numbers change if main memory access time doubles? (Give each change as both an absolute CPI and a percent change.) Notice the extent to which an L2 cache can hide the effects of a slow memory.

Standard memory time: Each cycle on a 2- Ghz machine takes 0.5 ps. Thus, a main memory access requires 100/0.5 = 200 cycles

- L1 only: 1.5 + 0.07*200 = 15.5
- Direct mapped L2: 1.5 + .07 × (12 + 0.035 × 200) = 2.83
- 8-way set associated L2: 1.5 + .07 × (28 + 0.015 × 200) = 3.67.

Doubled memory access time (thus, a main memory access requires 400 cycles)

- L1 only: 1.5 + 0.07*400 = 29.5 (90% increase)
- Direct mapped L2: 1.5 + .07 × (12 + 0.035 × 400) = 3.32 (17% increase)
- 8-way set associated L2: 1.5 + .07 × (28 + 0.015 × 400) = 3.88 (5% increase).

*10.2* It is possible to have an even greater cache hierarchy than two levels? Given the processor above with a second-level, direct-mapped cache, a designer wants to add a third-level cache that takes 50 cycles to access and will have a 13% miss rate. Would this provide better performance? In general, what are the advantages and disadvantages of adding a third-level cache?

1.5 + 0.07 × (12 + 0.035 × (50 + 0.013 × 100)) = 2.47

Adding the L3 cache does reduce the overall memory access time, which is the main advantage of having an L3 cache. Th e disadvantage is that the L3 cache takes real estate away from having other types of resources, such as functional units.

*10.3* In older processors, such as the Intel Pentium or Alpha 21264, the second level of cache was external (located on a different chip) from the main processor and the first-level cache. While this allowed for large second-level caches, the latency to access the cache was much higher, and the bandwidth was typically lower because the second-level cache ran at a lower frequency. Assume a 512 KiB off-chip second level cache has a miss rate of 4%. If each additional 512 KiB of cache lowered miss rates by 0.7%, and the cache had a total access time of 50 cycles, how big would the cache have to be to match the performance of the second-level direct-mapped cache listed above?

```
We want the CPI of the CPU with an external L2 cache to be at most 2.83.
Let x be the necessary miss rate.
1.5 + 0.07*(50 + x*200) < 2.83

Solving for x gives that x < - 0.155. This means that even if the miss
rate of the L2 cache was 0, a 50- ns access time gives a CPI of 1.5 +
0.07*(50 + 0*200) = 5, which is greater than the 2.83 given by the on-chip
L2 caches. As such, no size will achieve the performance goal.
```