Exercises by Amr Zaky

B.1 [10/10/10/15] <B.1> You are trying to appreciate how important the principle of locality is in justifying the use of a cache memory, so you experiment with a computer having an L1 data cache and a main memory (you exclusively focus on data accesses). The latencies (in CPU cycles) of the different kinds of accesses are as follows: cache hit, 1 cycle; cache miss, 105 cycles; main memory access with cache disabled, 100 cycles.

a. [10] <B.1> When you run a program with an overall miss rate of 5%, what will the average memory access time (in CPU cycles) be?

b. [10] <B.1> Next, you run a program specifically designed to produce completely random data addresses with no locality. Toward that end, you use an array of size 256 MB (all of it fits in the main memory). Accesses to random elements of this array are continuously made (using a uniform random number generator to generate the elements indices). If your data cache size is 64 KB, what will the average memory access time be?

c. [10] <B.1> If you compare the result obtained in part (b) with the main memory access time when the cache is disabled, what can you conclude about the role of the principle of locality in justifying the use of cache memory?

d. [15] <B.1> You observed that a cache hit produces a gain of 99 cycles (1 cycle vs. 100), but it produces a loss of 5 cycles in the case of a miss (105 cycles vs. 100). In the general case, we can express these two quantities as G (gain) and L (loss). Using these two quantities (G and L), identify the highest miss rate after which the cache use would be disadvantageous.

B.2 [15/15] <B.1> For the purpose of this exercise, we assume that we have 512-byte cache with 64-byte blocks. We will also assume that the main memory is 2 KB large. We can regard the memory as an array of 64-byte blocks: M0, M1, …, M31. Figure B.30 sketches the memory blocks that can reside in different cache blocks if the cache was fully associative.

a. [15] <B.1> Show the contents of the table if cache is organized as a direct-mapped cache.

b. [15] <B.1> Repeat part (a) with the cache organized as a four-way set associative cache.

B.3 [10/10/10/10/15/10/15/20] <B.1> Cache organization is often influenced by the desire to reduce the cache's power consumption. For that purpose we assume that the cache is physically distributed into a data array (holding the data), tag array (holding the tags), and replacement array (holding information needed by replacement policy). Furthermore, every one of these arrays is physically distributed into multiple sub-arrays (one per way) that can be individually accessed; for example, a four-way set associative least recently used (LRU) cache would have

Cache block	Set	Way	Memory blocks that can reside in cache block
0	0	0	M0, M1, M2, …, M31
1	0	1	M0, M1, M2, …, M31
2	0	2	M0, M1, M2, …, M31
3	0	3	M0, M1, M2, …, M31
4	0	4	M0, M1, M2, …, M31
5	0	5	M0, M1, M2, …, M31
6	0	6	M0, M1, M2, …, M31
7	0	7	M0, M1, M2, …, M31

Figure B.30 Memory blocks that can reside in cache block.

four data sub-arrays, four tag sub-arrays, and four replacement sub-arrays. We assume that the replacement sub-arrays are accessed once per access when the LRU replacement policy is used, and once per miss if the first-in, first-out (FIFO) replacement policy is used. It is not needed when a random replacement policy is used. For a specific cache, it was determined that the accesses to the different arrays have the following power consumption weights:

Array	Power consumption weight (per way accessed)
Data array	20 units
Tag	Array 5 units
Miscellaneous array	1 unit

Estimate the cache power usage (in power units) for the following configurations. We assume the cache is four-way set associative. Main memory access power—albeit important—is not considered here. Provide answers for the LRU, FIFO, and random replacement policies.

a. [10]<B.1> A cache read hit. All arrays are read simultaneously.

b. [10] <B.1> Repeat part (a) for a cache read miss.

c. [10] <B.1> Repeat part (a) assuming that the cache access is split across two cycles. In the first cycle, all the tag sub-arrays are accessed. In the second cycle, only the sub-array whose tag matched will be accessed.

d. [10] <B.1> Repeat part (c) for a cache read miss (no data array accesses in the second cycle).

e. [15] <B.1> Repeat part (c) assuming that logic is added to predict the cache way to be accessed. Only the tag sub-array for the predicted way is accessed in cycle one. A way hit (address match in predicted way) implies a cache hit. A way miss dictates examining all the tag sub-arrays in the second cycle. In

case of a way hit, only one data sub-array (the one whose tag matched) is accessed in cycle two. Assume there is way hit.

f. [10] <B.1> Repeat part (e) assuming that the way predictor missed (the way it chose is wrong). When it fails, the way predictor adds an extra cycle in which it accesses all the tag sub-arrays. Assume a cache read hit.

g. [15] <B.1> Repeat part (f) assuming a cache read miss.

h. [20] <B.1> Use parts (e), (f), and (g) for the general case where the workload has the following statistics: way-predictor miss rate = 5% and cache miss rate = 3%. (Consider different replacement policies.)

B.4 [10/10/15/15/15/20] <B.1> We compare the write bandwidth requirements of write-through versus write-back caches using a concrete example. Let us assume that we have a 64 KB cache with a line size of 32 bytes. The cache will allocate a line on a write miss. If configured as a write-back cache, it will write back the whole dirty line if it needs to be replaced. We will also assume that the cache is connected to the lower level in the hierarchy through a 64-bit-wide (8-byte-wide) bus. The number of CPU cycles for a B-bytes write access on this bus is

$$10 + 5\left(\left\lceil \frac{B}{8} \right\rceil - 1\right)$$

For example, an 8-byte write would take $10 + 5\left(\left\lceil \frac{B}{8} \right\rceil - 1\right)$ cycles, whereas using the same formula a 12-byte write would take 15 cycles. Answer the following questions while referring to the C code snippet below:

…

```
#define PORTION 1 … Base = 8*i; for (unsigned int j=base;
j < base+PORTION; j++) //assume j is stored in a register

data[j] = j;
```

a. [10] <B.1> For a write-through cache, how many CPU cycles are spent on write transfers to the memory for the all the combined iterations of the `j` loop?

b. [10] <B.1> If the cache is configured as a write-back cache, how many CPU cycles are spent on writing back a cache line?

c. [15] <B.1> Change `PORTION` to 8 and repeat part (a).

d. [15] <B.1> What is the minimum number of array updates to the same cache line (before replacing it) that would render the write-back cache superior?

e. [15] <B.1> Think of a scenario where all the words of the cache line will be written (not necessarily using the above code) and a write-through cache will require fewer total CPU cycles than the write-back cache.

B.5 [10/10/10/10/] <B.2> You are building a system around a processor with in-order execution that runs at 1.1 GHz and has a CPI of 0.7 excluding memory accesses. The only instructions that read or write data from memory are loads

(20% of all instructions) and stores (5% of all instructions). The memory system for this computer is composed of a split L1 cache that imposes no penalty on hits. Both the I-cache and D-cache are direct mapped and hold 32 KB each. The I-cache has a 2% miss rate and 32-byte blocks, and the D-cache is write-through with a 5% miss rate and 16-byte blocks. There is a write buffer on the D-cache that eliminates stalls for 95% of all writes. The 512 KB write-back, unified L2 cache has 64-byte blocks and an access time of 15 ns. It is connected to the L1 cache by a 128-bit data bus that runs at 266 MHz and can transfer one 128-bit word per bus cycle. Of all memory references sent to the L2 cache in this system, 80% are satisfied without going to main memory. Also, 50% of all blocks replaced are dirty. The 128-bit-wide main memory has an access latency of 60 ns, after which any number of bus words may be transferred at the rate of one per cycle on the 128-bit-wide 133 MHz main memory bus.

a. [10] <B.2> What is the average memory access time for instruction accesses?

b. [10] <B.2> What is the average memory access time for data reads?

c. [10] <B.2> What is the average memory access time for data writes?

d. [10] <B.2> What is the overall CPI, including memory accesses?

B.6 [10/15/15] <B.2> Converting miss rate (misses per reference) into misses per instruction relies upon two factors: references per instruction fetched and the fraction of fetched instructions that actually commits.

a. [10] <B.2> The formula for misses per instruction on page B-5 is written first in terms of three factors: miss rate, memory accesses, and instruction count. Each of these factors represents actual events. What is different about writing misses per instruction as miss rate times the factor *memory accesses per instruction?*

b. [15] <B.2> Speculative processors will fetch instructions that do not commit. The formula for misses per instruction on page B-5 refers to misses per instruction on the execution path, that is, only the instructions that must actually be executed to carry out the program. Convert the formula for misses per instruction on page B-5 into one that uses only miss rate, references per instruction fetched, and fraction of fetched instructions that commit. Why rely upon these factors rather than those in the formula on page B-5?

c. [15] <B.2> The conversion in part (b) could yield an incorrect value to the extent that the value of the factor references per instruction fetched is not equal to the number of references for any particular instruction. Rewrite the formula of part (b) to correct this deficiency.

B.7 [20] <B.1, B.3> In systems with a write-through L1 cache backed by a write-back L2 cache instead of main memory, a merging write buffer can be simplified. Explain how this can be done. Are there situations where having a full write buffer (instead of the simple version you've just proposed) could be helpful?

B.8 [20/20/15/25] <B.3> The LRU replacement policy is based on the assumption that if address A1 is accessed less recently than address A2 in the past, then A2 will be accessed again before A1 in the future. Hence, A2 is given priority over A1. Discuss how this assumption fails to hold when the a loop larger than the instruction cache is being continuously executed. For example, consider a fully

associative 128-byte instruction cache with a 4-byte block (every block can exactly hold one instruction). The cache uses an LRU replacement policy.

a. [20] <B.3> What is the asymptotic instruction miss rate for a 64-byte loop with a large number of iterations?

b. [20] <B.3> Repeat part (a) for loop sizes 192 bytes and 320 bytes.

c. [15] <B.3> If the cache replacement policy is changed to most recently used (MRU) (replace the most recently accessed cache line), which of the three above cases (64-, 192-, or 320-byte loops) would benefit from this policy?

d. [25] <B.3> Suggest additional replacement policies that might outperform LRU.

B.9 [20] < B.3> Increasing a cache's associativity (with all other parameters kept constant), *statistically* reduces the miss rate. However, there can be pathological cases where increasing a cache's associativity would increase the miss rate for a particular workload. Consider the case of direct mapped compared to a two-way set associative cache of equal size. Assume that the set associative cache uses the LRU replacement policy. To simplify, assume that the block size is one word. Now construct a trace of word accesses that would produce more misses in the two-way associative cache. (*Hint*: Focus on constructing a trace of accesses that are exclusively directed to a single set of the two-way set associative cache, such that the same trace would exclusively access two blocks in the direct-mapped cache.)

B.10 [10/10/15] <B.3> Consider a two-level memory hierarchy made of L1 and L2 data caches. Assume that both caches use write-back policy on write hit and both have the same block size. List the actions taken in response to the following events:

a. [10] <B.3> An L1 cache miss when the caches are organized in an inclusive hierarchy.

b. [10] <B.3> An L1 cache miss when the caches are organized in an exclusive hierarchy.

c. [15] <B.3> In both parts (a) and (b), consider the possibility that the evicted line might be clean or dirty.

B.11 [15/20] <B.2, B.3> Excluding some instructions from entering the cache can reduce conflict misses.

a. [15] <B.3> Sketch a program hierarchy where parts of the program would be better excluded from entering the instruction cache. (*Hint*: Consider a program with code blocks that are placed in deeper loop nests than other blocks.)

b. [20] <B.2, B.3> Suggest software or hardware techniques to enforce exclusion of certain blocks from the instruction cache.

B.12 [15] <B.4> A program is running on a computer with a four-entry fully associative (micro) translation lookaside buffer (TLB):

VP#	PP#	Entry valid
5	30	1
7	1	0
10	10	1
15	25	1

The following is a trace of virtual page numbers accessed by a program. For each access indicate whether it produces a TLB hit/miss and, if it accesses the page table, whether it produces a page hit or fault. Put an X under the page table column if it is not accessed.

Virtual page index	Physical page #	Present
0	3	Y
1	7	N
2	6	N
3	5	Y
4	14	Y
5	30	Y
6	26	Y
7	11	Y
8	13	N
9	18	N
10	10	Y
11	56	Y
12	110	Y
13	33	Y
14	12	N
15	25	Y

Virtual page accessed	TLB (hit or miss)	Page table (hit or fault)
1		
5		
9		
14		
10		
6		
15		
12		
7		
2		

B.13 [15/15/15/15/] <B.4> Some memory systems handle TLB misses in software (as an exception), while others use hardware for TLB misses.

a. [15] <B.4> What are the trade-offs between these two methods for handling TLB misses?

b. [15] <B.4> Will TLB miss handling in software always be slower than TLB miss handling in hardware? Explain.

c. [15] <B.4> Are there page table structures that would be difficult to handle in hardware but possible in software? Are there any such structures that would be difficult for software to handle but easy for hardware to manage?

d. [15] <B.4> Why are TLB miss rates for floating-point programs generally higher than those for integer programs?

B.14 [25/25/25/25/20] <B.4> How big should a TLB be? TLB misses are usually very fast (fewer than 10 instructions plus the cost of an exception), so it may not be worth having a huge TLB just to lower the TLB miss rate a bit. Using the SimpleScalar simulator (*www.cs.wisc.edu/~mscalar/simplescalar.html*) and one or more SPEC95 benchmarks, calculate the TLB miss rate and the TLB overhead (in percentage of time wasted handling TLB misses) for the following TLB configurations. Assume that each TLB miss requires 20 instructions.

a. [25] <B.4> 128 entries, two-way set associative, 4 KB to 64 KB pages (going by powers of 2).

b. [25] <B.4> 256 entries, two-way set associative, 4 KB to 64 KB pages (going by powers of 2).

c. [25] <B.4> 512 entries, two-way set associative, 4 KB to 64 KB pages (going by powers of 2).

d. [25] <B.4> 1024 entries, two-way set associative, 4 KB to 64 KB pages (going by powers of 2).

e. [20] <B.4> What would be the effect on TLB miss rate and overhead for a multitasking environment? How would the context switch frequency affect the overhead?

B.15 [15/20/20] <B.5> It is possible to provide more flexible protection than that in the Intel Pentium architecture by using a protection scheme similar to that used in the Hewlett-Packard Precision Architecture (HP/PA). In such a scheme, each page table entry contains a "protection ID" (key) along with access rights for the page. On each reference, the CPU compares the protection ID in the page table entry with those stored in each of four protection ID registers (access to these registers requires that the CPU be in supervisor mode). If there is no match for the protection ID in the page table entry or if the access is not a permitted access (writing to a read-only page, for example), an exception is generated.

a. [15] <B.5> How could a process have more than four valid protection IDs at any given time? In other words, suppose a process wished to have 10 protection IDs simultaneously. Propose a mechanism by which this could be done (perhaps with help from software).

b. [20] <B.5> Explain how this model could be used to facilitate the construction of operating systems from relatively small pieces of code that can't overwrite each other (microkernels). What advantages might such an operating system have over a monolithic operating system in which any code in the OS can write to any memory location?

c. [20] <B.5> A simple design change to this system would allow two protection IDs for each page table entry, one for read access and the other for either write or execute access (the field is unused if neither the writable nor executable bit is set). What advantages might there be from having different protection IDs for read and write capabilities? (*Hint*: Could this make it easier to share data and code between processes?)