Appendix B Solutions

B.1 a. Average access time = (1 – miss rate) × hit time + miss rate × miss time = .95 × 1 + 0.05 × 105 = 6.2 cycles.

b. Because of the randomness of the accesses, the probability and access will be a hit is equal to the size of the cache divided by the size of the array. Hit rate = 64 Kbytes/256 Mbytes ≈1/4000 = 0.00025.

Therefore

average access time = 0.00025 × 1 + (1 – 0.00025) × 105 = 104.974 cycles.

c. The access time when the cache is disabled is 100 cycles which is less than the average access time when the cache is enabled and almost all the accesses are misses. If there is no locality at all in the data stream then the cache memory will not only be useless but also it will be a liability.

d. Assuming the memory access time with no cache is T_{off}, with cache is T_{on} and the miss rate is m, the average access time (with cache on)

$T_{on} = (1 - m)(T_{off} - G) + m(T_{off} + L)$

The cache becomes useless when the miss rate is high enough to make T_{off} less than or equal to T_{on} .

At this point we have

$T_{off} <= (1 - m)(T_{off} - G) + m(T_{off} + L)$

After some algebraic manipulation, the inequality reduces to $m \geq \left(\frac{G}{G+L}\right)$. For part (a), G=99 and L=5, a miss rate greater than or equal to 99/104 (~ .95) would render the cache useless.

B.2 a.

Cache Block	Set	Way	Possible Memory Blocks
0	0	0	M0, M8, M16, M24
1	1	0	M1, M9, M17, M25
2	2	0	M2, M10, M18, M26
3	3	0	….
4	4	0	….
5	5	0	….
6	6	0	….
7	7	0	M7, M15, M23, M31

Direct-mapped cache organization.

b.

Cache Block	Set	Way	Possible Memory Blocks
0	0	0	M0, M2, …., M30
1	0	1	M0, M2, …., M30
2	0	2	M0, M2, …., M30
3	0	3	M0, M2, …., M30
4	1	0	M1, M3, ….., M31
5	1	1	M1, M3, ….., M31
6	1	2	M1, M3, ….., M31
7	1	3	M1, M3, ….., M31

Four-way set-associative cache organization.

B.3

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

Estimate the power usage in power units for the following configurations. We assume the cache is 4-way associative.

a. Read Hit:

LRU: 4 accesses for arrays of data/tag/miscellaneous components ➔ $4 \times (20 + 5 + 1) = 104$ power units.

FIFO and Random: 4 accesses for arrays data/tag components ➔ $4 \times (20 + 5) = 100$ power units.

b. Read Miss:

LRU: 4 accesses for arrays of data/tag/miscellaneous components ➔ $4 \times (20 + 5 + 1) = 104$ power units.

FIFO: 4 accesses for arrays data/tag components + one access to FIFO pointer ➔ $4 \times (20 + 5) + 1 = 101$ power units.

Random: 4 accesses for arrays data/tag components ➔ $4 \times (20 + 5) = 100$ power units.

c. Read hit (split access):

LRU: 4 accesses for arrays of tag/miscellaneous components plus one access to hit data array ➔ $4 \times (5 + 1) + 20 = 44$ power units.

FIFO, Random: 4 accesses for arrays of tag components plus one access to hit data array ➔ $4 \times (5) + 20 = 40$ power units.

d. Read miss, split access (cost of line-fill ignored):

LRU: 4 accesses for arrays of tag/miscellaneous components ➔ $4 \times (5 + 1) = 24$ power units.

FIFO, Random: 4 accesses for arrays of tag components ➔ $4 \times (5) = 20$ power units.

e. Read hit, split access with way prediction hit:

LRU: one access to arrays of tag/miscellaneous components plus one access to data array ➔ $(5 + 1) + 20 = 26$ power units.

FIFO, Random: one access to arrays of tag components plus one access to data array ➔ $5 + 20 = 25$ power units.

f. Read hit, split access with way prediction miss:

LRU: one access to arrays of tag/miscellaneous components, plus 4 accesses of tag/miscellaneous components, plus one access to data array ➔ $(5 + 1) + 4 \times (5 + 1) + 20 = 50$ power units.

FIFO, Random: access to arrays of tag components, plus 4 accesses of tag components, plus one access to data array ➔ $5 + 4 \times (5) + 20 = 45$ power units.

g. Read miss, split access with way prediction miss (cost of line-fill ignored):

LRU: one access to arrays of tag/miscellaneous components, plus 4 accesses of tag/miscellaneous components ➔ $(5 +1) + 4 \times (5 +1) = 30$ power units.

FIFO: one access to arrays of tag components, plus 4 accesses to arrays of tag component, and one access to miscellaneous component ➔ $5 + 4 \times (5) + 1 = 26$ power units.

Random: one access to arrays of tag components, plus 4 accesses to arrays of tag component, ➔ $5 + 4 \times (5) = 25$ power units.

h. For every access:

P(way hit, cache hit) = .95

P(way miss, cache hit) = .02

P (way miss, cache miss) = .03

LRU = $(.95 \times 26 + .02 \times 50 + .03 \times 30)$ power units

FIFO = $(.95 \times 25 + .02 \times 45 + .03 \times 26)$ power units

Random = $(.95 \times 25 + .02 \times 45 + .03 \times 25)$ power units

B.4 a. Loop has only one iteration in which one word (4 bytes) would be written. According to the formula for the number of CPU cycles needed for writing B bytes on the bus $\left(10 + 5\left(\left\lceil \frac{B}{8} \right\rceil - 1\right)\right)$, the answer is 10 cycles.

b. For a write-back cache, 32 bytes will be written. According to the above formula the answer is 10 + 5(4 – 1) = 25 cycles.

c. When PORTION is eight, eight separate word writes will need to be performed by the write-through cache. Each (see part a) costs 10 cycles. So the total writes in the loop will cost 80 CPU cycles.

d. X array updates on the same cache line cost 10X cycles for the write-through cache and 25 cycles for the write-back cache (when the line is replaced). Therefore if the number of updates is equal to or larger than 3, the write-back cache will require fewer cycles.

e. Answers vary, but one scenario where the write-through cache will be superior is when the line (all of whose words are updated) is replaced out and back to the cache several times (due to cache contention) in between the different updates. In the worst case if a the line is brought in and out eight times and a single word is written each time, then the write-through cache will require $8 \times 10 = 80$ CPU cycles for writes, while the write-back cache will use $8 \times 25 = 200$ CPU cycles.

B.5 A useful tool for solving this type of problem is to extract all of the available information from the problem description. It is possible that not all of the information will be necessary to solve the problem, but having it in summary form makes it easier to think about. Here is a summary:

- CPU: 1.1 GHz (0.909ns equivalent), CPI of 0.7 (excludes memory accesses)
- Instruction mix: 75% non-memory-access instructions, 20% loads, 5% stores
- Caches: Split L1 with no hit penalty, (i.e., the access time is the time it takes to execute the load/store instruction
  - L1 I-cache: 2% miss rate, 32-byte blocks (requires 2 bus cycles to fill, miss penalty is 15ns + 2 cycles
  - L1 D-cache: 5% miss rate, write-through (no write-allocate), 95% of all writes do not stall because of a write buffer, 16-byte blocks (requires 1 bus cycle to fill), miss penalty is 15ns + 1 cycle
- L1/L2 bus: 128-bit, 266 MHz bus between the L1 and L2 caches
- L2 (unified) cache, 512 KB, write-back (write-allocate), 80% hit rate, 50% of replaced blocks are dirty (must go to main memory), 64-byte blocks (requires 4 bus cycles to fill), miss penalty is 60ns + 7.52ns = 67.52ns
- Memory, 128 bits (16 bytes) wide, first access takes 60ns, subsequent accesses take 1 cycle on 133 MHz, 128-bit bus

a. The average memory access time for instruction accesses:

- L1 (inst) miss time in L2: 15ns access time plus two L2 cycles (two = 32 bytes in inst. cache line/16 bytes width of L2 bus) = $15 + 2 \times 3.75 = 22.5$ns. (3.75 is equivalent to one 266 MHz L2 cache cycle)

- L2 miss time in memory: 60ns + plus four memory cycles (four = 64 bytes in L2 cache/16 bytes width of memory bus) = $60 + 4 \times 7.5 = 90$ns (7.5 is equivalent to one 133 MHz memory bus cycle).
- Avg. memory access time for inst = avg. access time in L2 cache + avg. access time in memory + avg. access time for L2 write-back.

  $= 0.02 \times 22.5 + 0.02 \times (1 - 0.8) \times 90 + 0.02 \times (1 - 0.8) \times 0.5 \times 90 = .99$ns (1.09 CPU cycles)

b. The average memory access time for data reads:

Similar to the above formula with one difference: the data cache width is 16 bytes which takes one L2 bus cycles transfer (versus two for the inst. cache), so

- L1 (read) miss time in L2: $15 + 3.75 = 18.75$ns
- L2 miss time in memory: 90ns
- Avg. memory access time for read $= .02 \times 18.75 + .02 \times (1 - 0.8) \times 90 + 0.02 \times (1 - 0.8) \times 0.5 \times 90 = .92$ns (1.01 CPU cycles)

c. The average memory access time for data writes:

Assume that writes misses are not allocated in L1, hence all writes use the write buffer. Also assume the write buffer is as wide as the L1 data cache.

- L1 (write) time to L2: $15 + 3.75 = 18.75$ns
- L2 miss time in memory: 90ns
- Avg. memory access time for data writes $= .05 \times 18.75 + 0.05 \times (1 - 0.8) \times 90 + .05 \times (1 - 0.8) \times 0.5 \times 90 = 2.29$ns (2.52 CPU cycles)

d. What is the overall CPI, including memory accesses:

- Components: base CPI, Inst fetch CPI, read CPI or write CPI, inst fetch time is added to data read or write time (for load/store instructions).

  CPI $= 0.7 + 1.09 + 0.2 \times 1.01 + .05 \times 2.52 = 2.19$ CPI.

B.6 a. "Accesses per instruction" represents an average rate commonly measured over an entire benchmark or set of benchmarks. However, memory access and instruction commit counts can be taken from collected data for any segment of any program's execution. Essentially, average accesses per instruction may be the only measure available, but because it is an average, it may not correspond well with the portion of a benchmark, or a certain benchmark in a suite of benchmarks that is of interest. In this case, getting exact counts may result in a significant increase in the accuracy of calculated values.

b. misses/instructions committed = miss rate × (memory accesses/instructions committed) = miss rate × (memory accesses/instructions fetched) × (instructions fetched/instructions committed)

c. The measurement "memory accesses per instruction fetched" is an average over all fetched instructions. It would be more accurate, as mentioned in the answer to part (a), to measure the exact access and fetch counts. Suppose we are interested in the portion alpha of a benchmark.

Misses/instructions committed = miss rate × (memory accesses$_{alpha}$/instruction committed$_{alpha}$)

B.7 The merging write buffer links the CPU to the write-back L2 cache. Two CPU writes cannot merge if they are to different sets in L2. So, for each new entry into the buffer a quick check on only those address bits that determine the L2 set number need be performed at first. If there is no match in this "screening" test, then the new entry is not merged. If there is a set number match, then all address bits can be checked for a definitive result.

As the associativity of L2 increases, the rate of false positive matches from the simplified check will increase, reducing performance.

B.8 a. A 64-byte loop will completely fit in the 128 byte cache and the asymptotic miss rate will be 0%.

b. Because the cache is fully associative and uses an LRU replacement policy, both 192-byte and 320-byte loops will asymptotically produce a100% miss rate.

c. The 64-byte loop does not have any misses and will not benefit from the most recently used replacement policy. On the other hand, both the 192-byte and 320-byte will have 124 bytes (31 instructions) occupying uncontested cache locations while the rest of the loop instructions will compete for one block (4 bytes). So for the 192-byte loop there will be 31 hits and 17 misses every iterations. For the 320-byte loop there will be 31 hit and 49 misses every iteration.

d. Solutions vary, but random replacement or any technique that breaks the LRU circular pattern replacement might be better than LRU.

B.9 Construct a trace of the form addr1, addr2, addr3, addr1, addr2, addr3, addr1, addr2, addr3, ……, such that all the three addresses map to the same set in the two-way associative cache. Because of the LRU policy, every access will evict a block and the miss rate will be 100%.

If the addresses are set such that in the direct mapped cache addr1 maps to one block while add2 and addr3 map to another block, then all addr1 accesses will be hits, while all addr2/addr3 accesses will be all misses, yielding a 66% miss rate.

Example for 32 word cache: consider the trace 0, 16, 48, 0, 16, 48, ……

When the cache is direct mapped address 0 will hit in set 0, while addresses 16, and 48 will keep bumping each other off set 16.

On the other hand if the 32 word cache is organized as 16 set of two ways each. All three addresses (0,16,48) will map to set 0. Because of LRU that stream will produce a 100% miss rate!

That behavior can happen in real code except that the miss rates would not be that high because of all the other hits to the other blocks of the cache.

B.10 a. L1 cache miss behavior when the caches are organized in an inclusive hierarchy and the two caches have identical block size:

- Access L2 cache.
- If L2 cache hits, supply block to L1 from L2, evicted L1 block can be stored in L2 if it is not already there.
- If L2 cache misses, supply block to both L1 and L2 from memory, evicted L1 block can be stored in L2 if it is not already there.
- In both cases (hit, miss), if storing an L1 evicted block in L2 causes a block to be evicted from L2, then L1 has to be checked and if L2 block that was evicted is found in L1, it has to be invalidated.

b. L1 cache miss behavior when the caches are organized in an exclusive hierarchy and the two caches have identical block size:

- Access L2 cache
- If L2 cache hits, supply block to L1 from L2, invalidate block in L2, write evicted block from L1 to L2 (it must have not been there)
- If L2 cache misses, supply block to L1 from memory, write evicted block from L1 to L2 (it must have not been there)

c. When L1 evicted block is dirty it must be written back to L2 even if an earlier copy was there (inclusive L2). No change for exclusive case.

B.11 a. Allocating space in the cache for an instruction that is used infrequently or just once means that the time taken to bring its block into the cache is invested to little benefit or no benefit, respectively. If the replaced block is heavily used, then in addition to allocation cost there will be a miss penalty that would not have been incurred if there were a no-allocate decision.

Code example

```
Begin Loop1
       C1
       Begin Loop2
              C2
       End Loop2
End Loop1
```

In the simple example above, it is better not to have code C1 enter the cache as it might eventually conflict with C which is executed more often.

b. A software technique to enforce exclusion of certain code blocks from the instruction cache is to place the code blocks in memory areas with non-cacheable attributes. That might force some reordering of the code. In the example above that can be achieved by placing C1 in a non-cacheable area and jumping to it. A less common approach would be to have instructions turning caching on and off surrounding a piece of code.

B.12 a.

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

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	miss	fault
5	hit	hit
9	miss	fault
14	miss	fault
10	hit	hit
6	miss	hit
15	hit	hit
12	miss	hit
7	hit	hit
2	miss	fault

B.13 a. We can expect software to be slower due to the overhead of a context switch to the handler code, but the sophistication of the replacement algorithm can be higher for software and a wider variety of virtual memory organizations can be readily accommodated. Hardware should be faster, but less flexible.

b. Factors other than whether miss handling is done in software or hardware can quickly dominate handling time. Is the page table itself paged? Can software implement a more efficient page table search algorithm than hardware? What about hardware TLB entry prefetching?

c. Page table structures that change dynamically would be difficult to handle in hardware but possible in software.

d. Floating-point programs often traverse large data structures and thus more often reference a large number of pages. It is thus more likely that the TLB will experience a higher rate of capacity misses.

B.14 Simulation.

B.15 Solutions vary.