Homework Assignment 3

CS/ECE 6810: Computer Architecture
October 31,2018
Name: Jake Pitkin
UID: u0891770

OoO and Cache Optimization

Due Date: November 11, 2018.
120 points

1. **OoO Processor and Cache Parameters.** Please specify each of the following statement is True or False and explain why.

   i Instructions read all source operands from the register file in the Tomasulos algorithm.**(2 points)**

   False. A source operand could also be read from a reservation station as Tomasulo's algorithm uses these for register renaming.

   ii In a processor with hardware speculation, a store can always be committed as soon as its effective address and source value are available in the load-store queue.**(2 points)**

   True. Effective address is required for dependence check and we check availability of operands every cycle to determine if we can execute the store operation.

   iii If cache block size remains same, but the number of cache blocks doubles, the compulsory misses remain the same.**(2 points)**

   False. A compulsory miss is the first access to a block. With twice as many cache blocks, we increase the number of possible compulsory misses. Increasing the cache block size instead could decrease the number of compulsory misses.

   iv If the cache capacity is fixed but the block size is increased, the miss rate will not change.**(2 points)**

   False. The number of compulsory misses will decrease as the block size is increasing. Additionally, changing the cache design will most likely change the conflict miss rate (depending on the access pattern of a given program) compared to the miss rate of the original design.

   v Write-through caches use write allocate mechanism to prevent writing dirty blocks to lower memory levels.**(2 points)**

   False. A write-through cache typically uses no-write allocate. No real benefit to cache what was written as a future matching write (when using write-through) will write through to memory regardless.

2. **Memory Hierarchy.** Consider a processor with the following memory organization: L1 Cache, L2 Cache, L3 Cache and main memory. Each cache stores both tags and data.

The data and tag arrays are going to be accessed with the same index value. Assume that the processor does serial tag/data look-up (first tag lookup and then data access) for L2 and L3 caches and parallel tag/data look-up for L1 cache. The table given below provides the time take to access the tag and data arrays in CPU cycles. Find the number of cycles required to complete 2000 load instructions accessing this hierarchy.**(15 points)**

Level	**Tag Access**	**Data Access**	**Hit Rate**
L1 (32 KB)	1	-	40%
L2 (1 MB)	4	12	50%
L3 (8 MB)	25	90	80%
Main Memory	-	500	-

$$
\begin{aligned}
L1\ hit\ cycles &= instructions * access\ time * L1_{hit} \\
&= 2,000 * 1 * 0.4 \\
&= 800 \\
L2\ hit\ cycles &= instructions * access\ time * L1_{miss} * L2_{hit} \\
&= 2,000 * (1 + 4 + 12) * 0.6 * 0.5 \\
&= 10,200 \\
L3\ hit\ cycles &= instructions * access\ time * L1_{miss} * L2_{miss} * L3_{hit} \\
&= 2,000 * (1 + 4 + 25 + 90) * 0.6 * 0.5 * 0.8 \\
&= 57,600 \\
Main\ Memory\ Cycles &= instructions * access\ time * L1_{miss} * L2_{miss} * L3_{miss} \\
&= 2,000 * (1 + 4 + 25 + 500) * 0.6 * 0.5 * 0.2 \\
&= 63,600 \\
Total\ cycles &= L1_{cycles} + L2_{cycles} + L3_{cycles} + Memory_{cycles} \\
&= 800 + 10,200 + 57,600 + 63,600 \\
&= 132,200\ cycles
\end{aligned}
$$

Answer: 132,200 cycles to complete 2000 load instructions.

3. **Cache Hit Rate.** Consider a 256K main memory system with a direct mapped cache that stores up to 4 blocks. Each cache block comprises 4 words. The replacement policy is MRU and at the beginning cache is empty. Compute the hit rate for the following stream of addresses generated by the processor.(All memory addresses are in decimal format and each address refer to a word).**(10 points)**

   `170 , 257, 168, 246, 176, 175, 176, 177, 175, 176, 177, 175, 176, 177, 176, 175, 174, 173, 172, 171, 170, 169, 168, 167, 168, 165, 164`

   Assumptions: Memory is zero-indexed. The replacement policy is unused as this is a direct mapped cache. We fetch a block of 4 contiguous words at a time when accessing memory and caching.

   To determine the location where a block of memory will be stored in the cache we can first calculate its block address and then mod that by the number of blocks in the cache. This is similar to modulo hashing done in the slides but we first break memory up into 4 word blocks.

$$Cache\ Location = Block\ Address\ \%\ Cache\ Size$$
$$Cache\ Location = \lfloor address/4 \rfloor\ \%\ 4$$

   1. **Address: 170, Block: 168-171, Cache Location: 2, Hit/Miss: Miss.**

	Word 1	**Word 2**	**Word 3**	**Word 4**
Block 0				
Block 1				
Block 2	168	169	170	171
Block 3				

   2. **Address: 257, Block: 256-259, Cache Location: 0, Hit/Miss: Miss.**

	Word 1	**Word 2**	**Word 3**	**Word 4**
Block 0	256	257	258	259
Block 1				
Block 2	168	169	170	171
Block 3				

   3. **Address: 168, Hit/Miss: Hit.**
   4. **Address: 246, Block: 244-247, Cache Location: 1, Hit/Miss: Miss.**

	Word 1	**Word 2**	**Word 3**	**Word 4**
Block 0	256	257	258	259
Block 1	244	245	246	247
Block 2	168	169	170	171
Block 3				

   5. **Address: 176, Block: 176-179, Cache Location: 0, Hit/Miss: Miss.**

	Word 1	Word 2	Word 3	Word 4
Block 0	176	177	178	179
Block 1	244	245	246	247
Block 2	168	169	170	171
Block 3				

6. Address: 175, Block: 172-175, Cache Location: 3, Hit/Miss: Miss.

	Word 1	Word 2	Word 3	Word 4
Block 0	176	177	178	179
Block 1	244	245	246	247
Block 2	168	169	170	171
Block 3	172	173	174	175

7. Address: 176, Hit/Miss: Hit.

8. Address: 177, Hit/Miss: Hit.

9. Address: 175, Hit/Miss: Hit.

10. Address: 176, Hit/Miss: Hit.

11. Address: 177, Hit/Miss: Hit.

12. Address: 175, Hit/Miss: Hit.

13. Address: 176, Hit/Miss: Hit.

14. Address: 177, Hit/Miss: Hit.

15. Address: 176, Hit/Miss: Hit.

16. Address: 175, Hit/Miss: Hit.

17. Address: 174, Hit/Miss: Hit.

18. Address: 173, Hit/Miss: Hit.

19. Address: 172, Hit/Miss: Hit.

20. Address: 171, Hit/Miss: Hit.

21. Address: 170, Hit/Miss: Hit.

22. Address: 169, Hit/Miss: Hit.

23. Address: 168, Hit/Miss: Hit.

24. Address: 167, Hit/Miss: Hit.

25. Address: 168, Hit/Miss: Hit.

26. Address: 165, Block: 164-167, Cache Location: 1, Hit/Miss: Miss.

	Word 1	Word 2	Word 3	Word 4
Block 0	176	177	178	179
Block 1	164	165	166	167
Block 2	168	169	170	171
Block 3	172	173	174	175

27. Address: 164, Hit/Miss: Hit.

Lookups: 27 Hits: 21 Misses: 6 Hit rate: 21/27 or 77.78%

4. **Cache Performance.** Consider the following pseudo code; suppose that **X** and **Y** are allocated as contiguous integer arrays in memory and are aligned to the 4KB boundaries. Assume **k** and **l** are allocated in the register file with no need for memory accesses. We execute the code on a machine where the integer type (**int**) is 4 bytes wide.

```
#define M 1024
int X[M*M];
int Y[M*M];
int k, l;
  for (k = 0; k < M; k++) do
    for (l = 0; l < M; l++) do
      Y[l*M+k] = X[k*M+l];
    end for
  end for
```

i Consider a 4KB 2-way set-associative cache architecture while cache block size is 32-byte (8-word) and the replacement policy is LRU. Compute the hit rate of data accesses generated by `loads` and `stores` to **X** and **Y**?**(15 points)**

Assumptions: Assume all the required data has been placed in the main memory prior to executing the code (from Canvas discussion). X and Y are referring to contiguous parts of the main memory each one starting from a multiple of 4K address (from Canvas discussion).

Given that M is 1024 and X and Y are both integer arrays containing M * M 4 byte integers, we can determine the size of X and Y and assign them locations in memory.

Each array contains M * M = 1,024 * 1,024 = 1,048,576 = 2^{20} integers that are each 4 bytes wide.

Thus X and Y are each $2^{20} * 4$ bytes = 2^{22} bytes = 4,096 KB wide in memory. We will let X start at memory address 0x0 and Y will start at memory address 0x400000. X and Y are contiguous in memory and both start on a multiple of a 4KB address (as 4,096 KB is a multiple of 4 KB).

Assumption: Assume that the processor can address up to 8,192KB of main memory (the memory allocated for the X and Y arrays).

We can now calculate the address bits, byte offset bits, index bits, and tag bits:

$$\begin{aligned} \text{address bits} &= 23 \text{ bits } (8,192KB = 2^{23}B) \\ \text{byte offset bits} &= 5 \text{ bits } (32B = 2^5) \\ \text{index bits} &= 4\text{KB}/(2 * 32\text{B}) = 6 \text{ bits } (2^6) \\ \text{tag bits} &= 23 - 5 - 6 = 12 \text{ bits} \end{aligned}$$

Something to note: given X and Y are contiguous in memory, they both start on a multiple of a 4KB address, and our cache "resets" its indexing to 0 every 2KB (64

rows in the cache with a 32B block size) then the memory location X[a] and Y[a], where a is in the range $[0, 1048575]$, will be mapped to the same line in the cache.

Now we can consider the access pattern of the `load` and `store` operations. Given the arrangement of the code we will be making alternating `load` (to get a value out of X) and `store`s (to place it in Y).

Given `X[k*M+l]` and the arrangement of the for-loops, each `load` operation will be sequential in memory (X[0], X[1], X[2], ... X[1,048,575]).

Given `Y[l*M+k]`, the `store` operations will not be sequential in memory (Y[0], Y[1,024], Y[2,048], ... Y[1,047,552], Y[1], Y[1,025], Y[2,049] ... Y[1,047,553], ...) but rather make 1,024 size hops until the end of the array is reached, increase by 1, and repeat 1,024 times.

For the accesses to the X array, we will have a compulsory miss and bring a 32-bytes (8-words) into a cache block. This will provide a hit for the proceeding 7 words as X is stored and accessed contiguously. Each integer in X is only used once so we only get spatial locality from the cache for X.

For the accesses to the Y array, we will not gain any spatial locality. As our sequences of accesses to Y are always 1024 addresses away from each other but Y is stored contiguously. We make stores that will hit all 64 lines of the cache multiple times in the 1024 access that will occur until we could take advantage of what is cached. Similar to X, each location in Y is only accessed once. Thus we will miss every time with Y's stores.

The accesses to Y won't interrupt the spatial locality we gain from X because we are using a 2-way set-associative cache with LRU. LRU is important as we will always evict the Y block (as the X will be the most recently used).

In summary:

X array `load` operation hit ratio: 7/8
Y array `store` operation hits: 0
`load` ratio of total operations: 1/2
`store` ratio of total operations: 1/2
Overall hit rate: 1/2 * 7/8 + 1/2 * 0 = 43.75%

ii Consider a 4KB fully associative cache architecture with 32-byte blocks. The replacement policy is LRU. Rewrite the code to remove all of the non-compulsory misses. (You need to ensure the new code generate the exact same output in the main memory.You are allowed to add a nested for loop to the code if necessary.) Please provide explanation on how the new code can remove those misses.**(15 points)**

The problem with the above code isn't the `load` operations on the X array as we are achieving optimal spatial locality for blocks of size 8 words. The problem is the `store` operations on the Y array. We are bringing in 8 words, only using one of them, and then the block gets overwritten before the other 7 are accessed.

One way to resolve this would be to add a third for-loop (as hinted by the problem) and adjusting the indexing into X and Y:

```
#define M 1024
int X[M*M];
int Y[M*M];
int k, l;
  for (k = 0; k < M; k = k + 8) do
    for (l = 0; l < M; l++) do
      for (m = 0; m < 8; m++) do
        Y[l*M+(k+m)] = X[(k+m)*M+l];
      end for
    end for
  end for
```

Now we will access Y[a], Y[a+1], ... Y[a+7] sequentially which will allow us to take advantage of spatial locality similar to how we are gaining spatial locality for the array X as explained in part i.

We will no longer be accessing X sequentially as we were in part i. But rather the new m for-loop will bring 8 blocks (of size 8 words) into the cache as it loops 8 times.

But notice when the m for-loop ends and the l for-loop increments we will be using these same 8 blocks that are already in the cache for the 8 accesses to the X array inside the m for-loop.

As such, for both the X and Y array we will always use all 8 words brought into the cache. The only time we will miss is when we inititally bring new unseen blocks of 8 words into the cache.

5. **Cache Addressing.** Consider a processor using 3 cache levels. Level-1 cache is a 32KB direct mapped cache with 16B blocks used for both instructions and data. Level-2 is a 256KB, 2-way set associative cache with 64B cache lines. Level-3 is a 1MB, 4-way set associative cache with 64B cache lines. Assume that the processor can address up to 16GB of main memory. Compute the size of the tag arrays in KB for each level.**(15 points)**

Level-1 Cache

First we need to calculate the tag bits to know the size of the tag arrays for level. This can be done by considering the address bits and how those are distributed to the byte offset, index, and tag bits:

$$\begin{aligned}
\text{address bits} &= 34 \text{ bits } (16GB = 2^{34}B) \\
\text{byte offset bits} &= 4 \text{ bits } (16B = 2^{4}) \\
\text{index bits} &= 32\text{KB}/16\text{B} = 11 \text{ bits } (2^{11}) \\
\text{tag bits} &= 34 - 4 - 11 = 19 \text{ bits}
\end{aligned}$$

For each block there will be a tag so we multiply the number of tag bits by the number of blocks:

$$\text{tag array size for L1} = 19 \text{ bits} * 2^{11} = 38,912\text{bits} = 4.75 \text{ KB}$$

Level-2 Cache

Assumption: "Cache line" means cache block and not cache row (as per Canvas discussion). Therefore a cache row is 2 * 64B = 128B wide.

Level two will have the same address bits but we will recalculate the distribution of the other bits:

$$\begin{aligned}
\text{address bits} &= 34 \text{ bits } (16GB = 2^{34}B) \\
\text{byte offset bits} &= 6 \text{ bits } (64B = 2^{6}) \\
\text{index bits} &= 256\text{KB}/(2 * 64\text{B}) = 11 \text{ bits } (2^{11}) \\
\text{tag bits} &= 34 - 6 - 11 = 17 \text{ bits}
\end{aligned}$$

For each block there will be a tag so we multiply the number of tag bits by the number of blocks:

$$\text{tag array size for L2} = 17 \text{ bits} * 2 * 2^{11} = 69,632\text{bits} = 8.5 \text{ KB}$$

Level-3 Cache

Assumption: "Cache line" means cache block and not cache row (as per Canvas discussion). Therefore a cache row is 4 * 64B = 256B wide.

Level three will have the same address bits but we will recalculate the distribution of the other bits:

$$\begin{aligned}
\text{address bits} &= 34 \text{ bits } (16GB = 2^{34}B) \\
\text{byte offset bits} &= 6 \text{ bits } (64B = 2^{6}) \\
\text{index bits} &= 1\text{MB}/(4 * 64\text{B}) = 12 \text{ bits } (2^{12}) \\
\text{tag bits} &= 34 - 6 - 12 = 16 \text{ bits}
\end{aligned}$$

For each block there will be a tag so we multiply the number of tag bits by the number of blocks:

$$\text{tag array size for L3} = 16 \text{ bits} * 4 * 2^{12} = 262,144 \text{bits} = 32 \text{ KB}$$

L1: $4.75KB$
L2: $8.5KB$
L3: $32KB$
Total: $45.25KB$

6. **Cache and Memory Model using CACTI**

   CACTI (`http://www.cs.utah.edu/~rajeev/cacti7/`) is an integrated cache and memory model for access time, cycle time, area, leakage, and dynamic power. You are asked to use CACTI 7 for investigating the impact of small and simple caches using a 22nm CMOS technology node. Consider a processor using 2 cache levels. Level-1 cache is a 32 KB direct mapped cache with 16B blocks used for both instructions and data. Level-2 is a 1MB, 4-way set associative cache with 64B cache lines. Assume that the processor can address up to 2GB of main memory. Assume that both of the Level-1 and Level-2 caches are single bank.

   i Compare the access time, energy, leakage power, and area of the caches mentioned above.**(10 points)**

   Assumption: The question doesn't state which energy and leakage power to report. So I will assume it's asking for the "Total dynamic read energy per access", "Total dynamic write energy per access", "Total leakage power of a bank", and "Total gate leakage power of a bank". I could not see Voltage in the CACTI output to calculate energy.

   Level-1 cache

$$\begin{aligned}
\text{access time} &= 0.248287 \text{ ns} \\
\text{Total dynamic read energy per access} &= 0.00453646 \text{ nJ} \\
\text{Total dynamic write energy per access} &= 0.00225639 \text{ nJ} \\
\text{Total leakage power of a bank} &= 13.5459 \text{ mW} \\
\text{Total gate leakage power of a bank} &= 0.0283335 \text{ mW} \\
\text{area} &= 0.190421 * 0.223231 \\
&= 0.04251 \text{ mm}^2
\end{aligned}$$

   Level-2 cache

$$\begin{aligned}
\text{access time} &= 1.23985 \text{ ns} \\
\text{Total dynamic read energy per access} &= 0.0135198 \text{ nJ} \\
\text{Total dynamic write energy per access} &= 0.0113968 \text{ nJ} \\
\text{Total leakage power of a bank} &= 388.533 \text{ mW} \\
\text{Total gate leakage power of a bank} &= 0.82578 \text{ mW} \\
\text{area} &= 1.37311 * 0.758194 \\
&= 1.0411 \text{ mm}^2
\end{aligned}$$

ii Investigate the impact of varying associativity for 1, 2, and 8 on the access time, energy, leakage, and area of the Level-2 cache.**(10 points)**

Assumption: I will take the sum of "Total dynamic read energy per access" and "Total dynamic write energy per access" for the energy use and the sum of "Total leakage power of a bank" and "Total gate leakage power of a bank" for the leakage. We are comparing associativity here so I feel it's most reasonable to consider overall energy and leakage.

Associativity	**Access Time**	**Energy**	**Leakage**	**Area**
1	1.2161	0.0236	388.753	1.0334
2	1.2404	0.0239	388.7873	1.0357
4	1.23985	0.0249	389.3588	1.0411
8	1.24245	0.02735	391.2359	1.0489

Ranking by Associativity for each metric:

$$\text{Access Time} = 1, 4, 2, 8$$
$$\text{Energy} = 1, 2, 4, 8$$
$$\text{Leakage} = 1, 2, 4, 8$$
$$\text{Area} = 1, 2, 4, 8$$