Student Name: Ashna Mittal

Student Number: 251206758

CS 3350B: Assignment 1

Solution 1:

```
int factRec(int n)
{
	if (n <1)
	{
		return 1;
	}
	int n1 = factRec(n-1);
	return n1 * n;
}

int factIter(int n)
{
	int fact = 1;
	for (int i = 1; i <= n; ++i)
	{
		fact = fact * i;
	}
	return fact;
}
```

The recursive function and the iterative function yield the same result when n = 5, which is 120.

Analyzing factRec() Recursive function, it is clear that this recursive function has poor temporal locality because it involves function calls that keep adding new frames to the call stack until the base case is reached. Each call to factRec() is a jump to the same set of instructions, but because of the recursive nature, it's not repeatedly executing the same

instruction in a short period of time. Instead, it's preparing to execute a new instance of the same instructions, which involves different memory locations for each call's local variables. It also has poor spatial locality because it doesn't make use of data stored in nearby memory locations. Each recursive call creates a new stack frame with its own local variables, which are not necessarily close in memory to those of the previous call.

Analyzing factIter() Iterative function, it is clear that this iterative function has good temporal locality because it involves a loop that executes the same block of instructions repeatedly (5 times) in a short amount of time (Temporal locality in access to sum, i, fact). It also has a good special locality since it's using a small set of variables (‘fact’ and ‘i’) that are stored close to each other in memory and are reused throughout the loop.

When n=5, the execution of both the functions would go like this:

- The factRec() function calls itself with n = 4,3,2,1 each time creating a new stack frame and then unwinding as the calls return.
- The factIter() function simply loops from i = 1 to i = 5, multiplying the fact variable by i at each iteration and storing the result in the same memory location for fact.

In terms of which function has better locality, factIter() has better temporal and spatial locality because of its for loop that reuses a small number of memory locations and instructions, predicting the next instruction and fetching it to the cache memory. The recursive version has poorer locality compared to the iterative version because each recursive call creates 5 new stack frame and works with different memory locations.

Solution 2:

a) The simulation of the cache access sequence produced the following results:

Address	Index	Block Offset	Hit/Miss	Type of Miss
8	10	00	Miss	Cold
11	10	11	Hit	-

9	10	01	Hit	-
7	01	11	Miss	Cold
15	11	11	Miss	Cold
0	00	00	Miss	Cold
22	01	10	Miss	Capacity
2	00	10	Hit	-
6	01	10	Miss	Conflict
3	00	11	Hit	-

b) Drawing the table outline as per the cache's specifications which has 4 byte per line (2 bits for block offset), 4 sets (2 bites for each index), 2 line each and a total capacity of 32 bytes-

00				
01				
10				
11				

$(8)_{10} = (0...01000)_2$ -> set index is 10 and block offset is 00.

- Cache is initially empty, so access to 8 is a miss (cold miss because a cold miss occurs at level *k* for a block *b* when this block is missing).
- Its cache line gets installed in the cache.
- It gets installed in set with index 10, and 8 goes in block offset 0. The rest of the cache line is filled in: 8,9,10,11.

00				
01				
10	8	9	10	11
11				

Then 9 and 11 are a hit.

Then 7, 15 and 0 are a cold miss at 01, 11 and 00 set indexes respectively and so their cache lines get installed in the cache.

00	0	1	2	3

01	4	5	6	7
10	8	**9**	10	**11**
11	12	13	14	15

For 22, it is a Capacity miss at set index 01 and block offset 10 (because the set of active blocks (that is, the data set with which the program is actively working on is larger than the size of the cache at level *k*). The rest of the cache line is filled in: 20,21,22,23.

00	0	1	2	3
01	4	5	6	7
	20	21	22	23
10	8	9	10	11
11	12	13	14	15

Then 2 is a hit.

For 6, it is a Conflict miss at set index 01(because at level *k* multiple data items from level *k* + 1 all map to the same position at level *k*).

Then 3 is a hit.

The final table would look like this:

00	0	1	**2**	**3**
01	4	5	6	7
	20	21	22	23
10	8	**9**	10	**11**
11	12	13	14	15

	L1 size	L1 miss rate	L1 hit time
P1	64 KB	3.6%	1.26ns
P2	128 KB	3.1%	2.17ns

Solution 3:

a) Average Memory Access Time (AMAT) = Time for a Hit + Miss Rate × Miss Penalty

For P1:

- Hit Time = 1.26 ns
- Miss Rate = 3.6%
- Miss Penalty = Time to access the main memory = 100 ns

AMAT(p1) = 1.26ns + (0.036×100ns) = 1.26ns + 3.6ns
= 4.86 ns

For P2:

- Hit Time = 2.17 ns
- Miss Rate = 3.1%
- Miss Penalty = Time to access the main memory = 100 ns

AMAT(p2) = 2.17 ns + (0.031×100 ns) = 2.17 ns + 3.1 ns
=5.27 ns

b) CPIstall = CPIideal + (Average memory stall cycles/Hit time)
Where Average memory stall cycles = Access Rate × Miss Rate × Miss Penalty

For P1:

- Access Rate = 0.5 (or 50%)
- Miss Rate = 3.6%
- Miss Penalty = Time to access the main memory = 100 ns

Average Memory stall cycle (p1) = 0.5 × 0.036 × 100 = 1.8

CPIstall(p1) = 2.0 + 1.8 = 3.42857 = 3.43

For P2:

- Access Rate = 0.5 (or 50%)
- Miss Rate = 3.1%
- Miss Penalty = Time to access the main memory = 100 ns

Average Memory stall cycle (p2) = 0.5 × 0.031 × 100 = 1.55

CPIstall(p2) = 2.0 + 1.55 = 2.71428 = 2.71

With these values, we can see that p2 is faster in terms of CPIstall compared to p1.

c) After adding the following:

L2 size	L2 Miss Rate	L2 Hit Time
16 MB	42%	21.24 ns

Average Memory Access Time (AMAT) = L1 Hit Time + L1 Miss Rate × L1 Miss Penalty
Where L1 Miss Penalty = L2 Hit Time + L2 Miss Rate × L2 Miss Penalty

L2 Miss Penalty = L2 Hit Time+(L2 Miss Rate × Memory Access Time)
= 21.24 ns+(0.42 × 100 ns)
= 63.24 ns

AMAT = 1.26 ns + (0.036 × 63.24 ns) = 1.26 ns + 2.27664 ns
= 3.53664 ns = 3.54 ns

So, after considering the L2 cache, the new AMAT for p1 is approximately 3.54 ns. This represents the average time it takes to access memory, considering the hit times and miss rates of both the L1 and L2 caches.

d) CPIstall with L2 for P1 = CPIideal + (Average memory stall cycles with L2/Hit time)
= 2 + (L1 Miss Rate × (L2 Hit Time + (L2 MissRate × Memory Access Time)) * access rate
= 2 + 0.036(21.24 ns + (0.42 × 100 ns)) × 0.5
= 2 + (0.036 × 63.24 ns)/1.26 × 0.5
= 2 + 1.80686 × 0.5
= 2.90343
= 2.90

e) Processor P2 exhibits superior speed even after P1 is equipped with L2 cache as per the comparison Cycle Per Instruction (CPI) stall for each processor.
Given that P2 is the faster processor, to match P2's performance, P1 would need to achieve a miss rate in its L1 cache that results in the same AMAT as P2.

2.90 = 2.0 + (0.5× MissRate×100/2.17))
MissRate = 3.906%

This means if P1 could improve its L1 cache to have a miss rate of about 3.90%, it would be able to match the performance of P2, assuming all other factors remain constant.

Solution 4:

(a)

i. A cache block can store 8 words, and each integer is 4 bytes (or 1 word since a word in a 32-bit system is 4 bytes). Thus, each cache block can store 8 integers. The loop

for (int i = 0; i < N; i += 2) accesses every other integer(i and i+1 pair). Since the cache is direct mapped, each new block of the array A that is accessed will result in a cache miss. Since we access every second integer, we will access N/2 integers, but since each cache block can store 8 integers, we will access N/(2*8) blocks. f N is not a multiple of 16, we need to account for the last block that may not be completely filled.

Number of cache misses= 32859/(8*2) = 32859/16 = 2053.6875 = 2054

ii. Cache miss rate = Number of Cache misses/ Total number of Cache references
= 2054/32859
= 0.0625 = 6.25%

iii. In a direct-mapped cache, the first access to a memory location that maps to an empty cache line is always a compulsory miss, also known as a cold miss. Therefore, 2054 cache blocks are for all the integer values but since we have 128 sets only, there will be 128 cold cache missed and then every block accessed thereon will result in a conflict miss(1925 conflict misses in total). Hence, 1925 + 128 = 2054 cache misses.

(b)

i. This time, the loop for (int i = 0; i < N; ++i) accesses every integer in array A and writes it to array B. Every read from A will be a miss since it's the first access (compulsory miss). Because of write-allocate, every first write to a block in B will also bring the block into the cache, hence also a miss. The number of memory references will be 2 per loop. Therefore, we have N misses from reading A and N misses from writing to B.

Number of cache misses= 32859 + 32859 = 65718

ii. Cache miss rate = Number of Cache misses/ Total number of Cache references
= 65718/ 65718
= 1 = 100%

iii. The misses will be compulsory for the first access to each cache block in both arrays A and B. Therefore there will be 128 cold misses and 65718-128 = 65590 conflict misses.

Solution 5:

a) Normalize1's execution is faster than Normalize2 because of its sequential data access pattern. Normalize1 accesses data in row-major order and Normalize accesses

data in column-major order. Normalize1 leads to a better spatial locality and higher cache hit rates. This means that once a cache line is loaded due to a cache miss, subsequent elements are likely to be in the same cache line, resulting in fewer cache misses. On the contrary, Normalize2 jumps over entire rows, which can lead to more cache misses because it does not make as effective use of the cache lines that have been loaded. By comparing the LLC-load-misses and CPU-cycles between Normalize1 and Normalize2, we can see that Normalize1 likely has fewer LLC-load-misses and lower CPU-cycles compared to Normalize2 for the same matrix size, which would indicate it is indeed faster due to more efficient cache usage.

b) The miss rate increases for N > 512 because of the increased working set size, which leads to increased cache pressure and higher cache miss rates. The cache line size is 64 bytes. Size of double in C is typically 8 bytes. So, each cache line can thus hold 64 / 8 = 8 double values. For a matrix of size N x N, each row will contain N double values, which will occupy N * 8 bytes of memory. When N <= 512, each row occupies 512 * 8 = 4,096 bytes. Given the cache line size, the processor can load several contiguous double values from a row into the cache with each cache line fill. This means fewer cache misses due to spatial locality, as accessing one element in the row makes it likely that subsequent elements are also in the cache. However, as N exceeds 512, the size of each row in the matrix becomes larger than what can be held in the cache at once. The cache is unable to hold all the data needed for large matrix rows, which leads to evictions and subsequent cache misses as the processor constantly needs to fetch data from the main memory that was previously evicted from the cache. The increase in cache miss rate is not a sharp "jump" at N = 512 but has an intermediate effect at N = 1024 because the processor's prefetching behaviour might still manage to keep some of the needed data in the cache, smoothing out the increase in cache misses as N grows larger than 512. Even the L3 cache might still be able to hold a significant portion of the data even when N slightly exceeds 512, due to its relatively large size. When N reaches 1024, each row is now 1024 * 8 = 8,192 bytes, which is much larger than the size of the L3 cache lines, exacerbating cache misses since even more data needs to be evicted and re-fetched from main memory.

c) Normalize2 accesses the matrix in a column-wise fashion, which means it accesses memory with a stride of N doubles between each access. This access pattern does not utilize spatial locality effectively because the elements accessed consecutively do not reside next to each other in memory.

or N = 256, the matrix size in bytes is 256 * 256 * 8 which equals 512 KB. This size can still mostly fit into the L3 cache of the processor. Therefore, even with poor spatial locality, the entire matrix can reside in the L3 cache after it has been fully read once, resulting in a low miss rate after the initial population of the cache.

For N = 1024, the matrix size is 1024 * 1024 * 8, which equals about 8 MB. This size nearly fits into the L3 cache, but given that the cache is not exclusively dedicated to

this matrix, but also is shared with other processes and the operating system, not all of the matrix can be stored. Thus, even though the cache can hold a large portion of the matrix, cache misses will occur when the cache has to replace parts of the matrix with other data.

The miss rates for N=256 and N=1024 are misleading because:

- On the first pass through the matrix, the cache is being populated, which incurs compulsory misses and after the matrix is in the cache, if it fits, the miss rate will drop for subsequent accesses.
- Even if parts of the matrix are evicted, an intelligent cache replacement policy might keep frequently accessed data in the cache, which could reduce the miss rate.
- Modern processors have prefetchers that attempt to load data into the cache before it is needed. If the prefetcher incorrectly predicts the data needed for Normalize2, it could lead to an artificially low miss rate.

Assuming perf is capable of reporting any possible hardware event, a few additional performance metrics to get a more precise understanding of the data locality of Normalize2 would be:

- Translation Lookaside Buffer Misses: TLB misses would indicate how often the processor has to walk the page table to find data, which can be affected by large stride access patterns.
- Context Switches: Frequent context switches could lead to cache being flushed or repopulated, affecting cache miss rates.
- Cache Replacement Rules: Even if parts of the matrix are evicted, an efficient cache replacement policy might keep frequently accessed data in the cache, which could reduce the miss rate and increase the efficiency.
- DRAM Accesses: Direct measurements of accesses to DRAM would reflect how often the processor has to go out to main memory, which would be indicative of cache misses and low efficiency.
- Page Faults: Page faults can occur if the working set size exceeds the available physical memory, causing additional delays.

These metrics would offer insights into how to optimize the function for better cache utilization and performance of Normalize2.
