ECE 570
High-Performance Computer Architecture
Winter 2015
Solutions Set #2

[25 pts]

1- Consider the following C program to perform matrix multiplication. You will need to use Visual Studio for this problem.

(a) Parallelize this code using pthreads with two threads. The code should be written so that the generation of result matric c[][] is subdivided across the rows, i.e., rows 0 – (N/2)-1 are allocated to Thread 0 and rows N/2 – N-1 are allocated to Thread 1. In order to run pthreads in Visual Studio, you will need to add the pthreads library and include files. See the following link on how this can be done:
http://web.cs.du.edu/~sturtevant/pthread.html

(b) Parallelize this code using OpenMP with two threads. Again, the code should be written so that the rows 0 – (N/2)-1 are allocated to Thread 0 and the rows N/2 – N-1 are allocated to Thread 1. In order to run OpenMP with Visual Studio, you need to turn on the "OpenMP Support" option. The following link shows how this can be done:
http://supercomputingblog.com/openmp/getting-started-with-openmp-on-visual-studio/

For both programs, include statements to print out when Thread 0 and 1 are created and what their thread IDs are. In addition, include statements to print out which part of the result matrix the threads are working on.

```c
#include <stdio.h>
#include <stdlib.h>

#define N 10                    /* N x N matrix */


int main(void)
{
   int i, j, k;
   double a[N][N], b[N][N], c[N][N];

   printf("Initializing matrices...\n");

   /*** Initialize matrices ***/
   for (i = 0; i<N; i++)
   for (j = 0; j<N; j++)
      a[i][j] = i + j;
   for (i = 0; i<N; i++)
   for (j = 0; j<N; j++)
      b[i][j] = i*j;
   for (i = 0; i<N; i++)
   for (j = 0; j<N; j++)
      c[i][j] = 0;

   for (i = 0; i<N; i++)
   {
      for (j = 0; j<N; j++)
         for (k = 0; k<N; k++)
            c[i][j] += a[i][k] * b[k][j];
   }


   /*** Print results ***/
   printf("******************************************************\n");
   printf("Result Matrix:\n");
   for (i = 0; i<N; i++)
   {
      for (j = 0; j<N; j++)
         printf("%6.2f   ", c[i][j]);
      printf("\n");
   }
```

```
    printf("*******************************************************\n");
    printf("Done.\n");
    getchar();    /* Keeps the command window open in Visual Studio*/
}
```

Solution

Here is my code.

Pthreads version

```
/** pthreads version of MMT **/
/*********************************************
Simple Multi-threaded matrix multiplication
*********************************************/

#include <stdio.h>
#include <pthread.h>

#define N 10
#define NTHREADS 2
int jmax = N / NTHREADS;

/* function prototypes */
void* matMult(void*);

/* global matrix data */
double a[N][N], b[N][N], c[N][N];
int count = 0;

int main(void)
{
    pthread_t thr[NTHREADS];
    int i, j;

    printf("Initializing matrices...\n");

    /*** Initialize matrices ***/
    for (i = 0; i<N; i++)
        for (j = 0; j<N; j++)
            a[i][j] = i + j;
    for (i = 0; i<N; i++)
        for (j = 0; j<N; j++)
            b[i][j] = i*j;
    for (i = 0; i<N; i++)
        for (j = 0; j<N; j++)
            c[i][j] = 0;

    for (i = 0; i < NTHREADS; ++i) {
        pthread_create(&thr[i], NULL, matMult, (void*)i);
        printf("Created thread %d, Thread ID = %u\n", i, thr[i]);
    }
    for (i = 0; i < NTHREADS; ++i) {
        printf("Joined thread %d, Thread ID = %u\n", i, thr[i]);
        pthread_join(thr[i], NULL);
    }

    /*** Print results ***/
    printf("*******************************************************\n");
    printf("Result Matrix:\n");
    for (i = 0; i<N; i++)
    {
        for (j = 0; j<N; j++)
            printf("%6.2f   ", c[i][j]);
```

```
            printf("\n");
        }
        printf("*******************************************************\n");
        printf("Done.\n");
        getchar();    /* Keeps the command window open in Visual Studio*/
    }

    void* matMult(void* thread_id)
    {
        int i, j, k;
        int offset, row;
        offset = jmax*(int)thread_id;


        /*** Display which thread is executing which part ***/
        printf("Thread %d performing matrix multiply with offset = %d\n", thread_id,
    offset);

        for (j = 0; j < jmax; j++) {
            row = j + offset;
            for (i = 0; i < N; i++) {
                for (k = 0; k < N; k++) {
                    c[row][i] = c[row][i] + a[row][k] * b[k][i];
                }
            }
        }
        getchar();    /* Keeps the command window open in Visual Studio*/
        return 0;
    }
```

OpenMP version

```
/** OpenMP version of MMT **/
/*********************************************
Simple Multi-threaded matrix multiplication
*********************************************/


#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define N 10                    /* N x N matrix */


int main(void)
{
    int i, j, k;
    double a[N][N], b[N][N], c[N][N];

    printf("Initializing matrices...\n");

    /*** Initialize matrices ***/
    for (i = 0; i<N; i++)
    for (j = 0; j<N; j++)
        a[i][j] = i + j;
    for (i = 0; i<N; i++)
    for (j = 0; j<N; j++)
        b[i][j] = i*j;
    for (i = 0; i<N; i++)
    for (j = 0; j<N; j++)
        c[i][j] = 0;

    #pragma omp parallel for num_threads(2) schedule(static) private(j, k)
```

```
    for (i = 0; i<N; i++)
    {
        int thread_ID = omp_get_thread_num();
        printf("Executing Thread %d and Row = %d\n", thread_ID, i);
        //c[i][j] = 0;
        for (j = 0; j < N; j++)
            for (k = 0; k<N; k++)
                c[i][j] += a[i][k] * b[k][j];
    }


    /*** Print results ***/
    printf("******************************************************\n");
    printf("Result Matrix:\n");
    for (i = 0; i<N; i++)
    {
        for (j = 0; j<N; j++)
            printf("%6.2f   ", c[i][j]);
        printf("\n");
    }
    printf("******************************************************\n");
    printf("Done.\n");
    getchar();    /* Keeps the command window open in Visual Studio*/
    return 0;
}
```

[25 pts]

2- Consider a dual-core (P1 and P2) SMP system with two-way, set-associative caches, write-back, and LRU replacement policy. Each cache has four blocks (i.e., lines) labeled 0, 1, 2, and 3. The shared main memory consists of 8 blocks labeled 0, 1, 2, …, 7. Assume the same clock drives the processors and the memory bus and the following:

- Each transaction (request/response) completes in one cycle.
- In case of simultaneous bus requests from both processors, the priority is given in a round-robin fashion, i.e., P1, P2, P1 and so on…, i.e., if the bus was last acquired by P1 then P2 will be given the priority.

For the following two asynchronous sequence of memory-access events, where boldface numbers are for writes and the remaining are for reads, trace the execution of these block accesses on the two processors using the MESI coherence protocol.

P1: 0, **0**, 0, **1**, 1, 4, 3, 3, 5, **5**, 5
P2: 2, **2**, 0, 0, **7**, 5, 5, 5, **7**, 7, 0

Clearly indicate all the operations performed in each cycle in the "Comments" column; i.e., processor request (`PrRd`-hit/miss or `PrWr`-hit/miss), bus request (`BusRd(S` or `S')`, `BusRdX`, `BusUpgr`), and whether a cache (`flush`) or main memory responds (MM responds) to complete the transaction. The first bus transaction is shown below.

T	P1	Set 0		Set 1		h	B	P2	Set 0		Set 1		h	Comments
		0	1	2	3				0	1	2	3		
1	0	E (0)					P1	2						P1 `PrRd`-miss, `BusRd(S')`, MM responds P2 `PrRd`-miss, P2 waits
2	**0**	M (0)				√	P2	2	E (2)					P1 `PrWr`-hit P2 `BusRd(S')`, MM responds
3	0	M (0)				√		**2**	M (2)				√	P1 `PrRd`-hit P2 `PrWr`-hit
4	**1**	S (0)					P2	0	M (2)	S (0)				P1 `PrWr`-miss, P1 waits, `flush` P2 `PrRd`-miss, `BusRd(S)`
5	**1**	S (0)		M (1)			P1	0	M (2)	S (0)			√	P1 `BusRdX`, MM responds P2 `PrRd`-hit
6	1	S (0)		M (1)		√	P2	**7**	M (2)	S (0)	M (7)			P1 `PrRd`-hit P2 `PrWr`-miss, `BusRdX`, MM responds
7	4	S (0)	E (4)	M (1)			P1	5	M (2)	S (0)	M (7)			P1 `PrRd`-miss, `BusRd(S')`, MM responds

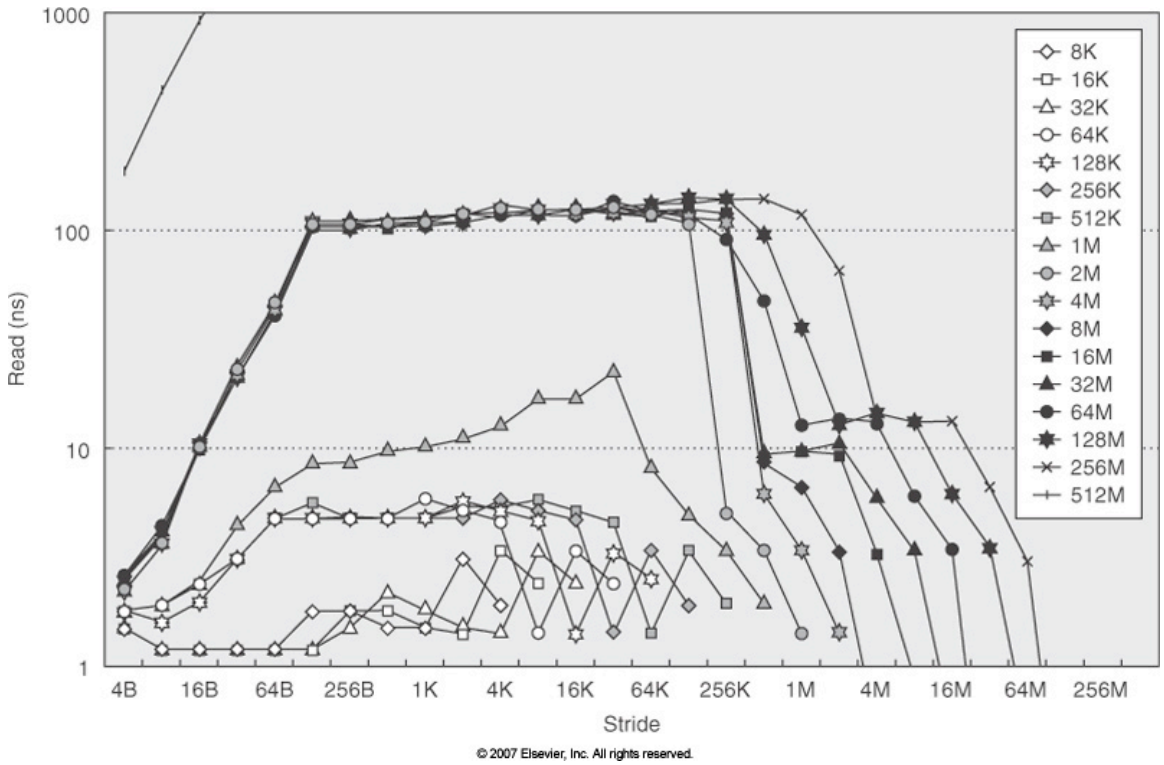
														P2 `PrRd`-miss, P2 waits
8	3	S (0)	E (4)	M (1)			P2	5	M (2)	S (0)	M (7)	E (5)		P1 `PrRd`-miss, P1 waits P2 `BusRd(S')`, MM responds
9	3	S (0)	E (4)	M (1)	E (3)		P1	5	M (2)	S (0)	M (7)	E (5)	√	P1 `BusRd(S')`, MM responds P2 `PrRd`-hit
10	3	S (0)	E (4)	M (1)	E (3)	√		5	M (2)	S (0)	M (7)	E (5)	√	P1 `PrRd`-hit P2 `PrRd`-hit
11	5	S (0)	E (4)	S (5)	E (3)		P1	**7**	M (2)	S (0)	M (7)	S (5)	√	P1 `PrRd`-miss, `BusRd(S)` P2 `PrWr`-hit, `flush`
12	**5**	S (0)	E (4)	M (5)	E (3)	√	P1	7	M (2)	S (0)	M (7)	I (5)	√	P2 `PrWr`-hit, `BusUpgr` P2 `PrRd`-hit
13	5	S (0)	E (4)	M (5)	E (3)	√		0	M (2)	S (0)	M (7)	I (5)	√	P1 `PrRd`-hit P2 `PrRd`-hit

T indicates the cycle. For P1 and P2 columns, b indicates when a request (read/write) is first issued for block b. 0-3 columns indicate the state of the block b (and cache block number indicated in parenthesis). B column indicates which processor has access to the bus and h column indicates whether there was a cache hit.

[25 pts]

3- Consider the graph shown below, which is the result of running a program that reads memory locations using different block (8 Kbytes - 512 Mbytes) and stride (4 bytes – 256 Mbytes) sizes on a hypothetical system. This allows for estimating latencies of different levels of the memory hierarchy.

(a) How many levels of caches are there?
(b) What are the overall size and block size of the L1-cache?
(c) What are the overall size and block size of the L2-cache, if there is one?
(d) What are the miss penalties for the L1-cache and (if present) L2-cache?
(e) What is the time for a page fault to secondary memory (i.e., hard disk)?



Solutions

(a) How many levels of caches are there?

From the graph, you can see that there is a small jump (few ns) in access time from 32 K to 64 K (you have to look at the values, since this is almost not visible in the graph) so there are definitely L1 and L2 caches. In addition, there are two groups of lines; one group with simulated cache sizes between 8K-32K and another group with cache sizes 64K-1M. This seems to indicate at first L1 is 3 2KB and L2 is 1 MB. Afterwards, there is a jump to next group of lines (2 MB – 256 MB). So it is difficult to tell if there is a L3 cache, but based on the latency experienced by these groups of lines (~100 ns), which is very large for L3, most likely there is no L3 cache.

(b) What are the overall size and block size of the L1-cache?

Based on the discussion in (a) L1 is 32KB. The block size is harder to determine from the graph, but we know that the block size of the cache impacts the number of hits a stream of references of a given stride will have on each line fill, i.e., if the stride is less than block size, the access time will improve. Once the stride reaches multiples of the block size, the access time will remain the same since there will be a miss for every stride. This is very difficult to tell from looking at group of lines for 8K - 32K. But if you look at the group of lines for 64K - 1M, there is a linear increase as function of stride and they level off at stride ~64B. Therefore, L1 block size is 64B.

(c) What are the overall size and block size of the L2-cache, if there is one?

Based on the discussion in (a) L2 is 1 MB. For similar reasons as in (b), the group of lines for 2 M - 256 M seems to indicate the L2 block size is 128 bytes.

(d) What are the miss penalties for the L1-cache and (if present) L2-cache?

Based on the group of lines represented by lines 64K-1M, the miss penalty on L1 is ~4-ns (note that this is in log scale). For L2, the group of lines for 2M - 256M represent miss penalty for L2 (i.e., main memory access time), which is ~100ns.

(e) What is the time for a page fault to secondary memory (i.e., hard disk)?

The line for 512M seems to indicate there was a page fault requiring access to the hard disk. Since this is way off the chart, we cannot determine the latency. However, hard disks usually have seek times in the range of ms.

[25 pts]

4- Suppose the base CPI with a perfect memory system is 1.5, i.e., CPU_{exec}=1.5. Determine which cache configuration is the best by computing the CPU_{time} for the three caches below:
   (a) 16-Kbyte direct-mapped unified cache using write-back and has a miss rate of 0.029.
   (b) 16-Kbyte 2-way, set-associative unified cache using write-back and has a miss rate of 0.022.
   (c) 32-Kbyte direct-mapped unified cache using write back and has a miss rate of 0.02.

   Assume the following:
   - The memory system has a unified cache (for instruction and data) and main memory (there is no L2 cache).
   - 64-bit wide main memory has an access latency of 100 clock cycles, the bandwidth between main memory and cache is 4 bytes per clock cycle and that 50% of the blocks in the cache are dirty.
   - There are 64 bytes per block and 25% of the instructions are load and store operations.
   - There is no write buffer, thus there is latency involved in writing back the evicted cache blocks to the main memory.
   - $CCT_{16K,\ 2\text{-}way} = 1.10 \times CCT_{16K,\ 1\text{-}way}$
   - $CCT_{32K,\ 1\text{-}way} = 1.30 \times CCT_{16K,\ 1\text{-}way}$

Solution

Consider the equation for CPU_{time}

$CPU_{time} = IC \times (CPI_{exe} + \text{Memory stall cycles/instruction}) \times CCT$

Where Memory Stall Cycles/Instruction = Memory Accesses/Instruction × Miss Rate × Miss Penalty

The Memory stall cycles take into account two sources of stall:
- Those cause by fetching instructions from memory
- Those caused by load or store instructions that access data.

For fetch stalls, the number of memory accesses per instruction is one since all instructions require one memory access to fetch each instruction. In the case of data stalls, only load and store instructions access memory access, which implies that the number of memory accesses per instruction for data accesses is $f_{loads}+f_{stores}$ (essentially, we are evaluating this expression: Average Accesses =$\sum$ (Frequency × Accesses). Substituting this information into the above equation yields

$$\text{Memory Stall Cycles/Instruction} = (R_iP_i) + [(f_{stores} + f_{loads})\ R_dP_d],$$

where *R* is the miss rate, *P* is the miss penalty, and *f* is the frequency of load/store. The subscripts indicate the specific miss rate and miss penalty (i for instruction access and d for data access). However, since all three configurations are unified, this implies that the miss rate and penalties are independent whether they are data or instruction accesses. Since it's a unified cache, the miss rate and the miss penalty are the same for instruction and data accesses and $f_{stores} + f_{loads} = 0.2$. Thus, we have

$$\text{Memory Stall Cycles/Instruction} = (RP) + [f_{loads/stores}RP] = 1.25 \times RP$$

With these equations we can examine the relative performance of the three cache configurations under a variety of circumstances.

For a write-back cache, the miss penalty *P* depends on the time required to flush and fill a cache block. Each block flush or block fill requires 100 clock cycles for memory latency plus 8 clocks to fill the 64-byte block at 8 bytes (64 bits) per clock. While a block fill must always occur on a miss, block flush only occurs if the block being replaced is dirty, which happens 50% of the time. Thus, the miss penalty *P* for a block fill is 1.5×(100+8) = 162 clock cycles. Using these values, we get the following:

$CPUtime_{16K,\ 1\text{-}way} = IC \times (1.5 + 1.25 \times 0.029 \times 162) \times CCT = 7.37 \times IC \times CCT$

$CPUtime_{16K,\ 2\text{-}way} = IC \times (1.5 + 1.25 \times 0.022 \times 162) \times 1.1CCT = 6.55 \times IC \times CCT$

$CPUtime_{32K,\ 1\text{-}way} = IC \times (1.5 + 1.25 \times 0.020 \times 162) \times 1.3CCT = 7.215 \times IC \times CCT$

As can be seen 16-KB, 2-way, set-associative is the fastest!