作业(3)： Memory

截至时间：2022.12.9/周五 23:59:59
提交方式：超算习堂（https://easyhpc.net/course/157）

Q1: Assume that L1 is 4-way associative, hit latency is 1 cycle; L2 is 8-way associative, hit latency is 10 cycles. 90% accesses to L1 are hits, and 80% L1 misses are hitting on L2. Both L1 and L2 are of 8-word block size. On a L2 miss, a block is fetched from memory, and 92 cycles for the first word to reach on L2 and 4 cycles for each of the 7 following words to reach L2. Suppose there is no delay to transfer blocks (or words) from L2 to L1 and to the CPU.

a. What's the L2 miss penalty?

b. Compute the average memory access time (AMAT).

c. Suppose *critical word first and early restart* is being applied when a miss occurs in L2 and a block is fetched from memory. Re-compute the AMAT.

d. For L1, suppose that *way prediction* is being used (no *critical word first and early restart*). With correct prediction, L1 hit time is still 1 cycle, but miss prediction incurs an overhead of 1 cycle. What's the AMAT if the way prediction rate is 60%?

e. Compare the AMAT in *b* and *d*. Is way-prediction still a useful technique for general cases? Why?

Q2: Consider the following cache configurations, A, B, C and D. All the caches use the write-back policy and thus each block has a 'dirty' bit, as well as a 'valid' bit. Assume that address length is 64b, and there is no virtual memory.

a. Calculate which address bits are used for indexing and tagging.
Use the address notation *A[N:M]* to describe which address bits are used for indexing or tagging. For example, *A[63:0]* denotes the whole 64-bit address. *A[12:4]* shows the address bits from the bit position 4 (LSB) to 12 (MSB), 9 bits total.

b. The size of tag and data arrays for each configuration.
Include these two bits when computing # of bits in the tag portion of the cache.

Configuration	**A**	**B**	**C**	**D**
Cache size	32KB	64KB	16KB	8KB
Block size	16B	128B	32B	64B
Associativity	Direct-mapped	8-way associative	16-way associative	Fully associative
Address bits used for indexing				
Address bits used for tagging				
total # of bits in all tag arrays				
total # of bits in all data arrays				

Q3: Consider a system that has 4 memory channels, each has 64b interface width and can accommodate up to four ranks. Assume that you are to purchase DDR memory chips with a capacity of 2Gb, 4Gb or 8Gb, and they respectively have data output width of 4, 8 and 16.

a. What's the maximum capacity can be supported for this system?

b. What's the aggregated bandwidth if each memory channel works at a frequency of 1200 MHz.

c. Suppose a sequence of reads (denoted by the mapped row, as listed in the able) are accessing the same memory bank. Note that the bank is already precharged at time 0. Further assume that precharge takes 20ns, row activate (i.e., open a row) takes 20ns, and cache line transfers to output pins (i.e., column read) takes 20ns. For the access pattern, estimate when each access completes for open-page and close-page policies.

Row being accessed	**Arrival time at memory controller**	**Open-page**	**Close-page**
X	10ns		
Y	75ns		
X	150ns		
X	210ns		
X	250ns		
Y	300ns		

Q4: (2.4, P152) <2.6> Using the sample program results in Figure 2.33:

a. What are the overall size and block size of the second-level cache?

b. What is the miss penalty of the second-level cache?

c. What is the associativity of the second-level cache?

d. What is the size of the main memory?

e. What is the paging time if the page size is 4 KB?

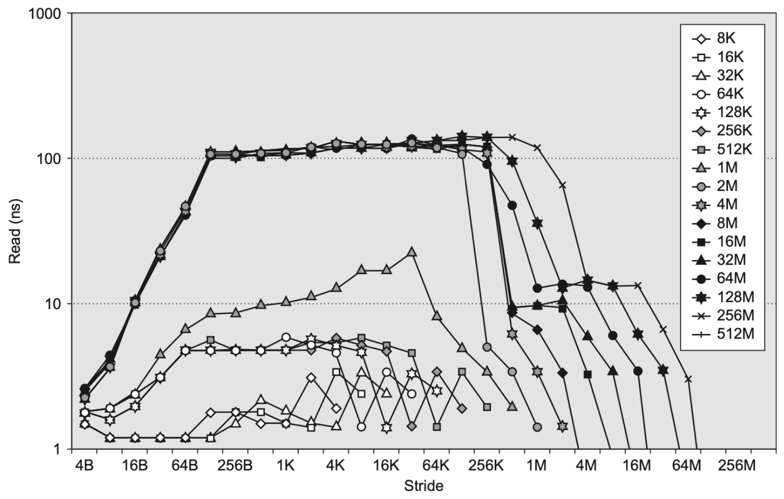

Figure 2.33 Sample results from program in Figure 2.32.

```
#include "stdafx.h"
#include <stdio.h>
#include <time.h>
#define ARRAY_MIN (1024) /* 1/4 smallest cache */
#define ARRAY_MAX (4096*4096) /* 1/4 largest cache */
int x[ARRAY_MAX]; /* array going to stride through */

double get_seconds() { /* routine to read time in seconds */
   __time64_t ltime;
   _time64( &ltime );
   return (double) ltime;
}
int label(int i) {/* generate text labels */
   if (i<1e3) printf("%1dB,",i);
   else if (i<1e6) printf("%1dK,",i/1024);
   else if (i<1e9) printf("%1dM,",i/1048576);
   else printf("%1dG,",i/1073741824);
   return 0;
}
int _tmain(int argc, _TCHAR* argv[]) {
int register nextstep, i, index, stride;
int csize;
double steps, tsteps;
double loadtime, lastsec, sec0, sec1, sec; /* timing variables */

/* Initialize output */
printf(" ,");
for (stride=1; stride <= ARRAY_MAX/2; stride=stride*2)
   label(stride*sizeof(int));
printf("\n");

/* Main loop for each configuration */
for (csize=ARRAY_MIN; csize <= ARRAY_MAX; csize=csize*2) {
   label(csize*sizeof(int)); /* print cache size this loop */
   for (stride=1; stride <= csize/2; stride=stride*2) {

      /* Lay out path of memory references in array */
      for (index=0; index < csize; index=index+stride)
         x[index] = index + stride; /* pointer to next */
      x[index-stride] = 0; /* loop back to beginning */

      /* Wait for timer to roll over */
      lastsec = get_seconds();
       sec0 = get_seconds(); while (sec0 == lastsec);

      /* Walk through path in array for twenty seconds */
      /* This gives 5% accuracy with second resolution */
      steps = 0.0; /* number of steps taken */
      nextstep = 0; /* start at beginning of path */
      sec0 = get_seconds(); /* start timer */
         { /* repeat until collect 20 seconds */
            (i=stride;i!=0;i=i-1) { /* keep samples same */
               nextstep = 0;
               do nextstep = x[nextstep]; /* dependency */
               while (nextstep != 0);
         }
         steps = steps + 1.0; /* count loop iterations */
         sec1 = get_seconds(); /* end timer */
      } while ((sec1 - sec0) < 20.0); /* collect 20 seconds */
      sec = sec1 - sec0;

      /* Repeat empty loop to loop subtract overhead */
      tsteps = 0.0; /* used to match no. while iterations */
      sec0 = get_seconds(); /* start timer */
         { /* repeat until same no. iterations as above */
            (i=stride;i!=0;i=i-1) { /* keep samples same */
               index = 0;
               do index = index + stride;
               while (index < csize);
         }
         tsteps = tsteps + 1.0;
         sec1 = get_seconds(); /* - overhead */
      } while (tsteps<steps); /* until = no. iterations */
      sec = sec - (sec1 - sec0);
      loadtime = (sec*1e9)/(steps*csize);
      /* write out results in .csv format for Excel */
      printf("%4.1f,", (loadtime<0.1) ? 0.1 : loadtime);
     }; /* end of inner for loop */
     printf("\n");
   }; /* end of outer for loop */
   return 0;
}
```

Figure 2.32 C program for evaluating memory system.