# CpSc 418: Homework 3

Due Nov. 21, 2002, 9:30am., in class.

| log2(nbytes) | log2(stride) | log2(nrefs) | time |
|---:|---:|---:|---:|
| 10 | 6 | 28 | 2.0 seconds |
| 11 | 6 | 28 | 2.0 seconds |
| 12 | 6 | 28 | 2.0 seconds |
| 13 | 6 | 28 | 2.0 seconds |
| 14 | 6 | 28 | 2.0 seconds |
| 15 | 6 | 28 | 2.0 seconds |
| 16 | 6 | 28 | 9.0 seconds |
| 17 | 6 | 28 | 9.0 seconds |

Table 1: Run times for the cacheSize program

1. Caches (30 points plus 10 extra credit)
   Figures 1, 2, 3 and 4 show the source code for a program for measuring cache size. Table 1 shows the run times for this program when executing this code on a 233 MHz Pentium II. The sharp increase at log2(nbytes) = 16 indicates that an array of $2^{15} = 32768$ bytes fits in the L1 cache, but an array of $2^{16} = 65536$ bytes does not. Thus, the Pentium II appears to have a 32Kbyte L1 cache. The increased run time of 7 seconds with 268435456 references indicates that the cache miss penalty is rougly (7/268435456) seconds, which is about 26 nano-seconds.

   (a) (5 points) Use the cacheSize program to measure the L1 cache size and miss penalty on a machine of your choosing. Show your data and state the L1 cache size. State the CPU type and clock frequency for the machine that you used.

   (b) (5 points) Use the cacheSize program to measure the L2 cache size and miss penalty on the machine that you used in part (a). Show your data and state the L2 cache size.

   (c) (10 points) Modify the cacheSize program so you can use it to determine the cache block size for the machine that you used in part (a). Show your data and state the block size. Provide your code and include a brief explanation of how it works.

   (d) (10 points) Modify the cacheSize program so you can use it to determine the cache associativity size for the machine that you used in part (a). Show your data and state the block size. Provide your code and include a brief explanation of how it works.

   (e) (10 points, extra credit) Modify the cacheSize program so you can determine whether or not the machine that you used in part (a) has a victim cache.

```
/* cacheSize.c:
 *   A program for emperically determining the size of a cache.
 *   Usage:
 *      cacheSize log2(nbytes) log2(stride) log2(nrefs)
 *
 *   This program allocates an array of int's of size nbytes.
 *   It then executes a loop that reads elements in this array starting
 *   at the 0th element, and increasing the address by stride bytes for
 *   each successive reference until it reaches the end of the array.
 *   The program repeats this until its made a total of nrefs
 *   memory references.  If the size of the array is less than or equal
 *   to the L1 cache size, then all the references should hit in the
 *   cache and the program will run fairly fast.  On the other hand,
 *   if the size of the array is greater than or equal to
 *       (size of cache) * (associativity + 1) / (associativity)
 *   then all of the references will miss, and the program will run
 *   much slower.  I'm assuming that the cache size is a power of two;
 *   so, it's sufficient to check for nbytes a power of two.
 *
 *   Rather than forcing the user to figure out 2^k for various values
 *   of k, the command line arguments are the base 2 logs of the
 *   corresponding quantities.  For example:
 *      cacheSize 12 6 28
 *   allocates an array of 4096 (i.e. 2^12) bytes, takes strides of
 *   64 (i.e. 2^6) bytes, and makes a total of 268435456 (i.e. 2^28)
 *   references to the array.
 *
 *   I've declared all variables that are used in the inner loop as register
 *   and unrolled the inner loop 16 times to minimize the overhead of
 *   everything but the load instructions to get a clear contrast between
 *   the run times with cache hits and cache misses.  I compiled the code
 *      gcc cacheSize.c -O4 -o cacheSize
 *   to specify maximum compiler optimizations.  Finally, I print the value
 *   of "sum" at the end so the compiler can't discover an unused value and
 *   discard the loops.
 */
```

Figure 1: cacheSize.c: part I

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

int main(int argc, char **argv) {
    register int *a,      /* the array */
          *atop,          /* the next integer pointer beyond the array */
          *p;             /* the current pointer into the array */

    register int sum,     /* we add up all the values referenced */
          stride;         /* the separation between successive addresses */

    int i,                /* loop index for the outer loop */
          n_outer;        /* how many times to execute the outer loop */

    int l2nbytes,         /* log2(array size), a command line parameter */
          l2stride,       /* log2(stride),     a command line parameter */
          l2nrefs;        /* log2(total number of references to make),
                                             a command line paramater */

    /* make sure we have the right number of command line parameters */
    if(argc != 4) {
        fprintf(stderr, "usage: %s log2(nbytes) log2(stride) log2(nrefs)\n",
                  argv[0]);
        exit(1);
    }

    /* get our parameters from the command line */
    l2nbytes = atoi(argv[1]);
    l2stride = atoi(argv[2]);
    l2nrefs = atoi(argv[3]);
```

Figure 2: cacheSize.c: part II

```
/* make sure our parameter values are sensible */
if(((1 << l2stride) % sizeof(int)) != 0) {
    fprintf(stderr,
      "BAD PARAMETER for %s: stride must be a multiple of sizeof(int)\n",
      argv[0]);
    exit(1);
}
if(l2nbytes < (l2stride + 4)) {
    fprintf(stderr,
      "BAD PARAMETER for %s: array size must be greater than 16*stride.\n",
      argv[0]);
    exit(1);
}
if(l2nrefs < l2nbytes - l2stride) {
    fprintf(stderr,
      "BAD PARAMETER for %s: nrefs not a multiple of the number of", argv[0]);
    fprintf(stderr, " references per pass through the array\n");
    exit(1);
}

/* The parameters seem ok.  Allocate the array */
a = (int *)malloc(1 << l2nbytes);
if(a == NULL) {
    fprintf(stderr, "%s:  malloc failed.  Is nbytes too large?\n", argv[0]);
    exit(1);
}

/* set up the loops */
n_outer = 1 << (l2nrefs - l2nbytes + l2stride);
atop = a + (1 << l2nbytes)/sizeof(int);
stride = (1 << l2stride)/sizeof(int);
sum = 0;
```

Figure 3: cacheSize.c: part III

```
    /* the loops */
    for(i = 0; i < n_outer; i++) {
        for(p = a; p < atop; p++) {
            sum += *p; p += stride; sum += *p; p += stride;
            sum += *p; p += stride; sum += *p; p += stride;
            sum += *p; p += stride; sum += *p; p += stride;
            sum += *p; p += stride; sum += *p; p += stride;
            sum += *p; p += stride; sum += *p; p += stride;
            sum += *p; p += stride; sum += *p; p += stride;
            sum += *p; p += stride; sum += *p; p += stride;
            sum += *p; p += stride; sum += *p; p += stride;
        }
    }

    /* print sum (so the compiler can't optimize it out of existence) */
    printf("sum = %d\n", sum);

    /* we made it */
    exit(0);
}
```

Figure 4: cacheSize.c: part IV

```
double *matrixMultiply(double *a, double *b, double *c, int n) {
    int i, j, k;
    double sum;
    for(i = 0; i < n; i++) {
        for(j = 0; j < n; j++) {
            sum = 0.0;
            for(k = 0; k < n; k++) {
                sum += a[i*n + k] * b[k*n + j];
            }
            c[i*n + j] = sum;
        }
    }
    return(c);
}
```

Figure 5: A function for matrix multiplication

2. Code optimization (30 points)
   Consider the code for matrix multiply (based on that from class) shown in figure 5.

   (a) (10 points) Compile the code with the -S flag and no optimizations specified. Inspect the assembly code and identify which, if any, of the optimizations described in class were performed by the compiler.

   Turn in a marked up copy of the assembly code. State what compiler, what compiler flags, what CPU, and what operating system you used.

   HINT: You can add calls to printf (or any other function(s) that you like) to the code. The calls should be pretty obvious in the assembly code. This can help you figure out which assembly code statements correspond to which statements in the C program. Furthermore, you can pass one or more of the variables of matrixMultiply to the printf (or whatever). This can help you identify where variables are stored.

   (b) (10 points) Re-compile the code with the highest level of optimization. Identify what optimizations the compiler performed. As for part (a), turn in marked-up assembly code. specified, and state what compiler flags you used.

   (c) (10 points) Run the unoptimized and optimized versions. How much speed-up did the optimization achieve?

```
L1:     fld     $fa, 0($aa)        # $fa <- *aa /* aa = &(a[i,k]) */
        addiu   $aa, $aa, 8        # aa++
        fld     $fb, 0($bb)        # $fb <- *bb /* bb = &(b[k,j]) */
        addiu   $bb, $bb, $n8      # bb += n;
        fmult   $fx, $fa, $fb      # $fx <- a[i,k]*b[k,j]
        bne     $aa, $atop, L1     # for(..., k < n, ...)
        fadd    $sum, $sum, $fx    # sum += a[i,k] * b[k,j], in delay slot
```

Figure 6: MIPS assembly code for the inner loop of matrix multiply

3. Superscalar execution (20 points)
   Figure 6 shows the code that was presented in the Oct. 29 and Oct. 31 lectures for matrix multiplication. The Oct. 31 notes explain how this code can execute in two cycles per iteration (assuming no cache misses) on a MIPS R10000. The notes stated that the the two cycle limit is due to the two fld instructions and only one load/store unit.

   A more detailed look at the R10000 reveals that when it accesses its L1 cache, it provides data for *all* pending loads in the address queue that can be satisfied by the accessed cache set (both "ways" of the set are considered).

   (a) (10 points) Does the feature of satisfying multiple pending loads with a single cache access speed-up the execution of the matrix multiply inner loop? If so, by how much? If not, why not? Justify your answer by explaining how the various operations of the loop are scheduled on the R10000.

   (b) (10 points) Now, assume that the instruction fetch and decode bandwidths are each increased to eight instructions per cycle, and combine this with the cache optimization described above. Do these combined features speed up the matrix multiply loop? If so, by how much? If not, why not? Justify your answer.

```
b = 0;
for(k = 0; k < M; k++) {  /* relax the grid data M times */
    i_prev = N-1;
    for(i = 0; i < N; i++) { /* it's an N*N grid */
        i_next = i+1;
        if(i_next == N) i_next = 0;
        j_prev = N-1;
        for(j = 0; j < N; j++) {
            j_next = j+1;
            if(j_next == N) j_next = 0;
            /* relaxation step:                                   *
             *   Each grid value is updated to the average of its     *
             *   four neighbours.                                 */
            x[1-b, i, j] = (x[b, i_prev, j] + x[b, i_next, j] +
                              x[b, i, j_prev] + x[b, i, j_next])/4.0;
            j_prev = j;
        }
        i_prev = i;
    }
    b = 1-b;
}
```

Figure 7: Simplified relaxation code for a PDE solver

4. Snooping caches (20 points)

   Figure 7 shows the code for a simplified version of a relaxation method for solving two-dimensional partial differential equations (with periodic boundary conditions). This code can be sped-up by executing it on a shared memory memory multiprocessor (SMP). For example, assume `N` is a multiple of four and execution on a SMP with 16 processors. The execution consists of 16 threads, each responsible for updating 1/16 of the array. Assume that each thread has variables `i_base` and `j_base` with $\texttt{i\_base}, \texttt{j\_base} \in \{0, \texttt{N}/4, \texttt{N}/2, 3\texttt{N}/4\}$. In other words, `i_base` and `j_base` give the lower left corner of the `N`/4 × `N`/4 square updated by the thread. Figure 8 shows the code for each thread.

   Assume that each processor has a cache with sufficient capacity and associativity to avoid any capacity of conflict misses, and ignore cold-start misses (you can assume that `M` is large). Thus, all cache misses will be coherency misses.

   How many coherence misses does this code incur each time all the threads execute the body of the `k` loop? State any assumptions that you need to make to solve the problem (or choose to make to simplify the problem).

```
b = 0;
i_max = i_base + (N/4) - 1;
j_max = j_base + (N/4) - 1;
for(k = 0; k < M; k++) {  /* relax the grid data M times */
    i_prev = i_max;
    for(i = 0; i < N/4; i++) { /* it's an N*N grid */
        i_next = (i == i_max) ? i_base : (i+1);
        j_prev = j_max;
        for(j = 0; j < N/4; j++) {
            j_next = (j == j_max) ? j_base : (j+1);
            /* relaxation step:                                *
             *   Each grid value is updated to the average of its    *
             *   four neighbours.                              */
            x[1-b, i, j] = (x[b, i_prev, j] + x[b, i_next, j] +
                              x[b, i, j_prev] + x[b, i, j_next])/4.0;
            j_prev = j;
        }
        i_prev = i;
    }
    b = 1-b;

    barrier(); /* wait for all 16 threads to reach this point
                * before continuing to next update */
}
```

Figure 8: Parallel version of the relaxation code