CS 213, Fall 1999
Lab Assignment L4
Assigned: October 28
Part I: Due Fri., November 5, 11:59PM (Note change!)
Part II: Due Wed., November 10, 11:59PM

Logistics

Randy Bryant (Randy.Bryant@cs.cmu.edu) is the lead person for this assignment. The files for this assignment can be retrieved from

/afs/cs.cmu.edu/academic/class/15213-f99/L4/L4.tar

Once you’ve copied this file into a (private) directory, run the command tar xvf L4.tar. There are two parts to this lab. Both are to be handed in electronically. Each part counts 50%. In both cases you will hand in a single file called trans.c. You should fill in your team information in the structure at the beginning of this file. For the first part, you will be graded on the code and comments for the routine good_cache_transpose. For the second you will be graded on the code and comments for the routine good_throughput_transpose.

Introduction

The purpose of this assignment is to gain experience in optimizing code for a memory intensive application. Consider a procedure to copy and transpose the elements of an $N \times N$ matrix of type int. That is, for source matrix $S$ and destination matrix $D$, we want to copy each element $s_{ij}$ to $d_{ji}$. This code can be written with a simple loop:

```c
void row_transpose(int *dest, int *src, int dim)
{
    int i, j;
    for (i = 0; i < dim; i++)
        for (j = 0; j < dim; j++)
            dest[RIDX(j,i,dim)] = src[RIDX(i,j,dim)];
}
```

where the arguments to the procedure are pointers to the destination (dest) and source (src) matrices, as well as the matrix size $N$ (dim). The macro RIDX is defined as follows:
Table 1 summarizes the measured performance of these two transposition routines on matrices for six different values of $N$. We express the performance in terms of throughput, measured in megabytes per second on one of our Pentium III Xeon machines. That is, if transposing an $N \times N$ matrix requires $T$ seconds, then the throughput is $(N \times N \times 4 \times 10^{-6})/T$. As can be seen for both of the routines (shown in the rows labeled “Row Scan” and “Column Scan”), the throughput decreases as $N$ increases. This can be attributed to the memory system performance—larger matrices make poorer use of the different levels of cache memory.

In our measurements, we see the effects of both spatial and temporal locality. There is spatial locality in the two matrices—adjacent elements in a given row will be adjacent in memory. In addition, our measurements were performed by repeatedly calling the routines on the same source and destination matrices. Thus, some portions of the source and destination matrices may be cache-resident in successive calls, giving temporal locality.

Making this code run fast requires two types of optimizations. First, we can see that we are not exploiting the cache structure very well. Although the row scan routine does a good job exploiting the spatial locality of the source matrix, it does a poor job for large values of $N$ with the destination matrix. The column scan routine has the opposite problem. Second, the code generated by GCC is not very efficient. Looking at the assembly code, one sees that the inner loop requires 10 instructions, 5 of which reference memory—one for the source, one for the destination, and three to read local variables from the stack.

```c
#define RIDX(i, j, n) ((i)* (n)+ (j))

void col_transpose(int *dest, int *src, int dim)
{
    int i, j;
    for (j = 0; j < dim; j++)
        for (i = 0; i < dim; i++)
            dest[RIDX(j,i,dim)] = src[RIDX(i,j,dim)];
}
```
Your job is to devise a transpose routine that runs as fast as possible. To make life easier, you can assume that \( N \) is a multiple of 32. Your code must run correctly for all such values of \( N \), but we will measure its performance only for the 6 values shown in Table 1. We will measure the improvement of your procedure relative to the row scan procedure, computing the ratio of the throughputs. That is, if procedure \texttt{row\_scan} has throughput \( P_N \) and your version of \texttt{good\_throughput\_transpose} has throughput \( P_N' \), then the performance ratio \( R_N \) is computed as \( R_N = \frac{P_N'}{P_N} \). To summarize the overall effect of your optimizations, we will compute the geometric mean of these ratios:

\[
R = \sqrt[6]{R_{32} 	imes R_{64} 	imes R_{128} 	imes R_{256} 	imes R_{512} 	imes R_{1024}}
\]

The geometric mean is the preferred method for summarizing a set of ratios. For example, the column scan code is slower in some cases than the row scan code and faster in others. The geometric mean is 1.43, indicating that, on the whole, it is slightly faster.

**Part I: Optimizing Simulated Cache Performance**

The best way to improve the performance of the transpose routine is to first focus on its cache behavior. To more precisely measure the cache behavior we will run the routines using a simulated cache that has 16KB capacity, is direct-mapped, and has a 32 byte block size.

To capture the memory accesses of the transpose routines, we write them using a special macro \texttt{COPY} as follows:

```c
/* Matrix transpose based on row-wise scan of source matrix */
void row_transpose(int *dest, int *src, int dim)
{
    int i, j;
    for (i = 0; i < dim; i++)
        for (j = 0; j < dim; j++)
            COPY(&dest[RIDX(j,i,dim)], &src[RIDX(i,j,dim)]);
}
```
This macro is defined to simply perform the copy when compiled for the normal benchmarking. When compiled for the cache simulation mode, however, it will also call routines to simulate the cache indexing and tag matching of a direct-mapped cache. The simulation routines determine whether each of the two accesses (one read and one write) hits or misses in the cache. By writing your array accesses in this fashion and compiling the code with the command make trace, you will generate a program trace that will compute the simulated cache performance for the transpose routines.

We impose the restriction that you may only perform $N^2$ copy operations in your transposition code. You could artificially improve the miss rate of your code by performing redundant copies, but this would not be helpful in your ultimate goal of achieving maximum throughput. In addition, this rule prevents you from copying multiple source values to local variables and then writing them to the destination. Such techniques can be useful on machines with many registers, but this is not the case for IA32 machines.

You should fill in the code for the routine named good_cache_transpose in the file trans.c. Table 2 shows the improvement our solution was able to obtain. The overall performance is computed as the average miss rate for the 5 values of $N$, since the miss rate for $N = 32$ is 0. If you can get an average miss rate less than or equal to 0.140, you will get full credit for performance for this part of the assignment.

### Part II: Optimizing Measured Throughput

As can be seen in the third entry of Table 1, our cache-optimized code does not run much faster than the original code. There are two reasons for this. First, the L1 cache of the Pentium III Xeon is 4-way associative, rather than direct-mapped. Thus, our cache simulator is not a reliable indicator of the true cache performance. Second, the code we wrote to minimize cache misses has a very complex control structure. It compiles into code that is not very efficient. Some of the basic principles in optimizing the cache performance can be utilized to optimize throughput, but anything that is too exotic simply does not compile into fast code.

Your task for this part of the assignment is to write a routine good_throughput that has as high a throughput as you can achieve across the 6 values of $N$. Your grade for this part will depend on the geometric mean of the throughput improvement you achieve. Improving the actual performance of your routine requires a fair amount of low-level code optimization. The best method is to repeatedly transform the code, check the generated assembly (with the command make trans.s to see how much simplification you’ve obtained, and benchmark the code.

Use the copy macro in this routine so that you can also evaluate the simulated cache performance, even though this is not an accurate predictor of throughput. You will find that you can simplify the loop by rewriting the array indexing code with pointer code. You will also want to unroll loops so that the inner loop performs the copy operation for a number of matrix elements—say between 16 and 64. This will allow you to simplify the address computation and to reduce the loop overhead. In the inner loop of our code, each copy requires just three instructions: one to read the source value into a register, one to write the register to the destination, and one to update a pointer.

To check your progress you can compile and run the code on your own machine using the command make benchmark, and then running the program benchmark. You can also submit it to the class server using the command make update NAME=yournname. The measured results will be returned to you by electronic mail. In addition, a class WWW page will show the results for the entire class. Note that you can measure the performance of a number of transpose routines in one batch. See how to “register” a routine in procedure register_transposers in the file trans.c. This is helpful when you want to try out a number of variant routines. The emailed results will report the performance for all of your routines. The WWW page, as well as our grading routines, will only show the performance for the routine
Rules and Evaluation

You may write any code you want, as long as it satisfies the following:

- It must be in ANSI C. You may not use any embedded assembly language statements.
- It must be correct. You will get no credit for code that does not compile or does not correctly transpose a matrix for any value of \( N \) that is a multiple of 32. You will also be penalized if your code prints any extraneous information.
- It must access the source and destination arrays only with the \texttt{COPY} macro.
- It must not interfere with the time measurement mechanism or the cache simulation.

You are allowed to define macros, additional global variables, and other procedures in the file \texttt{trans.c}. If you modify any other files, make sure your \texttt{trans.c} still runs with the original versions. Your grade in each part will be based on the following:

- Correctness. You will get no credit for buggy code!
- Performance (80%). Based on the simulated average miss rate (Part I), or the geometric mean of the throughput improvement (Part II). You will get full credit on Part I if your average total miss rate is less than or equal to 0.140. You will get full credit on Part II if your performance index is greater than or equal to 3.00.
- Documentation (20%). Using comments in your code, you should describe how your code optimizes cache performance (Part I) or throughput (Part II). For Part I, you should include an analysis of both the read and the write miss rates.

Hand In

For both parts, you will hand in your code using the command \texttt{make handin NAME=yourname}, where \texttt{yourname} is the Andrew ID of one of the team members. You can submit updated versions with the command \texttt{make handin NAME=yourname VERSION=XX}, where \texttt{XX} is the version number, i.e., 2, 3, \ldots. Only your highest numbered version submitted before the deadline will be graded. Note that we will use two different handin directories for Parts I and II, so you can restart your version numbering for Part II.