CS 213, Fall 1998
Lab Assignment L3:
Understanding the Performance Impact of Memory Systems
Assigned: Oct. 29, Due: Wed., Nov. 11, 11:59PM

Dave O'Hallaron (`droh@cs.cmu.edu`) is the lead person for this lab.

1. Introduction

In this lab you will use a powerful analysis technique called *binary translation* to help you understand and optimize the performance of a memory intensive matrix transpose operation.

A binary translator is a program that inputs an executable binary, manipulates it in some way, and then outputs a new binary. The particular binary translator you will use in this lab is a very cool product from DEC called *Atom*. Atom allows you to write *tools* that navigate a binary file, procedure by procedure, and within each procedure, basic block by basic block, and within each basic block, statement by statement. As you navigate the binary, you can insert calls to arbitrary C functions.

An Atom tool consists of two C files: an *analysis file*, which consists of arbitrary C *analysis routines*, and an *instrumentation file*, which navigates the binary and indicates where to insert calls to the analysis routines. This is an extremely flexible model that is powerful enough to reproduce the functionality of powerful tools such as gprof, Pixie, and Purify.

Your first task is to write an Atom tool called *csim* that predicts the copy bandwidth of an $N \times N$ matrix transpose procedure running on a virtual machine with a simple direct-mapped cache. The tool works by tracing at run-time the effective addresses of the load and store operations in the transpose procedure, and then simulating the effect of these load and store operations on a simulated direct-mapped cache.

You will then use your tool to help you develop an optimized matrix transpose procedure that runs faster than the baseline versions because it exhibits better spatial locality.[1] Matrix transpose is a fundamental operation in a variety of graphics, signal processing, medical computing, and scientific computing applications, including synthetic aperture radar imaging, magnetic resonance imaging, narrowband tracking radar, and multi-dimensional fast Fourier transforms. So while it is small and looks simple, matrix transpose is subtle and important to understand.

When you are finished with the lab, you will have a better appreciation for this powerful notion of binary translation, and for the significant impact that the memory system can have on the performance of your application programs.

[1]We're having you simulate a memory system instead of running and measuring directly on the color Alphas because of the complexity of the Alpha memory system and the difficulty of getting tight repeatable measurements. Writing your own cache simulator has the additional benefit of clarifying your understanding of basic cache mechanics.

Step 0: Copy files

First create a (protected) directory to work in, and copy our starter code using the command `tar -xvf /afs/cs.cmu.edu/academic/class/15213-f98/L3/L3.tar`.

Step 1: Print and read the Atom documentation

Atom documentation and examples are in `/afs/cs.cmu.edu/academic/class/15213-f98/atom`. For starters, print and carefully read the Atom User Manual (`atom-user.ps`) and the Atom man page (`man1/atom.1`). The Atom User Manual is only 30 pages, but it is full of helpful examples, including an example of how to trace memory references.

Step 2: Write and debug the instrumentation file

The first step in developing your *csim* tool is to write and debug an instrumentation file. Augment the default instrumentation file (`csim.inst.c`) that we provided so that it instruments a binary in the following way:

- Instrument the procedure p (and only the procedure p) whose name is passed in *argv[1]*.
- At the beginning of procedure p, call the following analysis routine:

  ```
  void InitCache(char *procname)
  ```

- At the end of procedure p, call the following analysis routine:

  ```
  void PrintResults(char *procname)
  ```

- For each load or store instruction i in each basic block b in procedure p, call the following analysis routine:

  ```
  void DataRef(int type, long addr)
  ```

where *type* is either 0 for a load or 1 for a write, and *addr* is the effective address referenced by instruction i (not the address of instruction i itself!).

In this step, you can use the default analysis file (`csim.anal.c`) that we provided as is (except that you will need to fill in the team struct). The *InitCache* and *PrintResults* routines simply print out explanatory messages. Each invocation of *DataRef* prints a line containing the type of reference (load or store) and the address referenced by the load or store.

To compile your *csim* tool, modify the makefile for the size of the matrix to transpose (N) and the name of the function to trace (PROC). Then type "*make clean; make*" to build the instrumented binary `trans.csim`.

Running this instrumented binary will produce a trace file, which you can compare to the trace files `row_trans.4x4.trace` and `row_trans.64x64.trace`.

Step 3: Turn the analysis file into a cache simulator

Once *csim* is producing correct address traces, modify your analysis file so that it simulates an 8K-byte direct-mapped cache with 32-byte blocks and a write-through write policy. Assume that each load and store accesses 4 bytes of data, and that each address is 32 bits (even though Atom wants this declared as a long). Your simulator is special-purpose code, so there is no need to be general and support other block sizes or cache sizes. Your simulator should count the total number of reads, writes, read misses, and write misses.

The *InitCache* routine should initialize your cache simulator. The *PrintResults* routine should display the number of reads, writes, read misses, and write misses. The *DataRef* routine should simulate the effect of the reference on the cache[2].

Validate your results against *Dinero*, a public domain cache simulator that we've provided for you in the lab directory. See `dinero.doc` for directions on how to run Dinero and interpret its output.

Step 4: Add a performance model to the cache simulator

The ultimate goal of the *csim* tool is to predict the *copy bandwidth* of the matrix transpose operation on a virtual machine that uses your simulated cache. The copy bandwidth is the total number of bytes in the matrix divided by the running time in seconds of the matrix transpose. Copy bandwidth is expressed in units of MBytes/sec, where 1 MByte is 2^{20} bytes. To estimate the running time in seconds, use the following model:

- 200 MHz clock rate (1 MHz is 10^6 cycles/sec).
- Cache hits (reads or writes) take 1 cycle.
- Read misses take 20 cycles, which reflects the overhead of loading a word from memory into the cache. At 200 MHz, a cycle is 5 ns, which is roughly a factor of 20 slower than the memory cycle time of DRAMS, thus our estimate of 20 cycles for a read miss.
- Write misses take 40 cycles, because each miss requires the cache to read a block from memory, update the particular word, and then write the block back to memory.

From this point on, you must print your results by calling the *handin_output* routine that we have provided for you in the default analysis file.

Step 5: Analyze the base cases

Use *csim* to predict the copy bandwidth of a 64×64 matrix transpose using the *row_trans* routine and the *col_trans* routines.

If your simulator is working properly, you'll find that *col_trans* runs about twice as fast as *row_trans*. Why? You'll also find that with *row_trans*, every write misses, and with *col_trans*, every read misses. Why? Understanding these questions, will help you optimize matrix transpose in the next section.

[2]Notice that we are only dealing with addresses; we are not actually storing any data in the simulated cache

Step 6: Write an optimized version of matrix transpose

This step is the crux of the lab: Use *csim* to develop an optimized version of the 64×64 matrix transpose whose predicted copy bandwidth beats both *row_trans* and *col_trans*.

This optimized code should be in the *fast_transpose* routine in file `fast_transpose.c`. No assembly language is allowed; your optimized code must be written entirely in C. You should expect *fast_trans* to be about three times faster than *row_trans*.

Hand In

Your handin will consist of three files: `fast-trans.c`, `csim.inst.c` and `csim.anal.c`.

You will hand your files in with the command

```
make handin NAME=yourname
```

where `yourname` is replaced with the Andrew Id of the first person on your team. Your solution will be copied to the handin directory. If you need to submit another version to fix a bug, use the command

```
make handin NAME=yourname VERSION=versionnumber
```

where `versionnumber` starts at 2 and counts up with each submission *after* the initial handin.

IMPORTANT: To keep the grading task manageable and to allow the teaching staff to provide quick turnaround, your *csim* tool must generate its output by calling *handin_output* exactly once per invocation of the tool. Thus, when you execute an instrumented binary, it should produce exactly one line of output, which is the line of output generated by *handin_output*. Your handin code should contain no debug or other output. You'll lose points if it does, so be careful.

Grading

There are 100 points. We'll instrument your optimized matrix transpose procedure with our version of *csim* (which we've already validated against Dinero) and with your version.

The correctness of your *csim* tool is worth 70 points. The correctness and efficiency of your optimized matrix transpose routine is worth 30 points.

Remember the new policy: handins before the due day get 10% extra credit.

Logistics

As usual, you may work groups of up to 2 people.

Any clarifications and revisions to the assignment will be posted on the class bboard and Web page.