CS/ECE 347, Spring 1998

Lab Assignment 1:

Trace-Driven Cache Simulation

Assigned: Tuesday, February 3

Due: 3pm Thursday, February 12

1. Logistics

As always, you may work in a group of up to 3 people. Hardcopy handins are due at 3pm on the assigned date. All files and programs you will need are available in

```
/afs/cs/academic/class/15347-s98/public/labs/lab1/
```

Formatted text is preferred to hand written. Any clarifications and revisions to the assignment will be posted on the class bboard and Web page. Miroslav (`mvelev@ece`) is the primary point of contact for questions.

2. Introduction

The purpose of this assignment is to help you gain a deeper understanding of how caches work. The technique you will use is known as *trace-driven simulation*. A *trace* is a file that contains a sequence of memory accesses that occur during the execution of some program. The accesses are either instruction fetches, load operations, or store operations. A trace-driven cache simulator reads the address for each access in the trace and simulates the behavior of the cache in terms of the number of hits and misses.

Trace-driven cache simulators are useful tools for understanding the impact on miss rate of various design parameters such as cache size, block size, associativity, replacement policy, and write-allocation policy.

In this assignment, you will have to use Dinero, a public domain cache simulator available from the University of Wisconsin, in order to evaluate the performance of caches of different configurations. A dinero executable, compiled for Sparcs, is available from:

```
/afs/cs/academic/class/15347-s98/public/labs/lab1/dinero/bin/dinero.sun4_55
```

You can compile your own executable from the Dinero source files:

```
/afs/cs/academic/class/15347-s98/public/labs/lab1/dinero.tar
```

The dinero man page is saved as:

```
/afs/cs/academic/class/15347-s98/public/labs/lab1/dinero/man/dinero.cat
```

Trace file format

Each line in a Dinero trace file corresponds to a memory access, and consists of a two-tuple of the form *<label>* *<address>*, where *<label>* is an integer that gives the access type:

0: read data

1: write data

2: instruction fetch

and *<address>* is a 32-bit unsigned hexadecimal word-aligned byte address (without the usual '0x' prefix). Each access corresponds to a read or write of a 32-bit word.

Benchmark trace files

We have provided you with three benchmark trace files, each consisting of 100,000 memory accesses from a real program:

- `cc1.100k.din`: C compiler, integer code
- `tex.100k.din`: text processor, integer code
- `spice.100k.din`: circuit simulator, FP code (sparse-matrix processing, many loops)

The benchmark traces can be found in:

`/afs/cs/academic/class/15347-s98/public/labs/lab1/trace-files`

However, they are quite large (about 1MB each), so that you might want to use `ln -s` to create symbolic links to them rather than to copy them. Also, make a symbolic link to the Dinero executable with link name "`dinero`."

Command line arguments

Dinero recognizes the following command line arguments. Numeric arguments with a suffix of *K*, *M*, or *G* are multiplied by 1024, 1024^2, or 1024^3 respectively.

-u*unified_cache_size* Sets the unified cache size in bytes. Defaults to 0. If *unified_cache_size* is positive, both *instruction_cache_size* and *data_cache_size* must be zero. If zero, implying separate instruction and data caches will be simulated, both *instruction_cache_size* and *data_cache_size* must be set to positive values.

-i*instruction_cache_size* Sets the instruction cache size in bytes. Defaults to 0, indicating a unified cache simulation. If positive, the *data_cache_size* must be positive as well.

-d*data_cache_size* Sets the data cache size in bytes. Defaults to 0, indicating a unified cache simulation. If positive, the *instruction_cache_size* must be positive as well.

-b*size* Sets the block size in bytes.

-a*associativity* Sets the cache associativity. A direct-mapped cache has associativity 1. A *k*-way set-associative cache has associativity *k*. Defaults to direct-mapped placement (i.e., -a1).

-r*replacement_policy* Sets the block replacement policy. Valid replacement policies are LRU (i.e., -rl), FIFO (i.e., -rf), and RANDOM (i.e., -rr). Defaults to LRU.

-A*write_allocation_policy* Selects whether a block is fetched on a write miss. A write-allocate policy (i.e., -Aw) causes a block to be fetched and loaded into the cache on a write miss. A no-write-allocate policy (i.e., -An) causes blocks to be loaded only on non-write accesses that miss. Defaults to write-allocate.

-o*output_style* Selects the output style. If -o0, then output consists of a single terse line that summarizes the cache organization, total accesses, total misses, and miss rate. If -o1, then output is more detailed, with breakdowns on the miss rates for instruction fetches, loads, and stores. (See the man page for details.)

To help you avoid some tedious work, we have provided example script files in the directory:

`/afs/cs/academic/class/15347-s98/public/labs/lab1/scripts/`

The scripts `block.csh` and `block.pl` will run the experiments for Part I below. You will need to modify these to run experiments for parts II and III. Refer to the `README` file in the same directory for instructions on how to use these scripts.

Part I: Exploring associativity and block size

The design space for caches is enormous, and it is very easy for a designer to make wrong choices for the cache organization. Simulation based on traces of real programs is one important technique that system builders use to help them make more intelligent choices. In this problem you will play cache designer and use Dinero to identify the combination of cache parameters that minimizes miss rate.

1.1) Experiments (9 pts)

Suppose you are given a fixed unified cache size of 8K bytes, an LRU replacement policy, and a write-allocation policy of write-allocate (which translates to running Dinero with the following flags: -u8K -i0 -d0 -b? -a? -rl -Aw). Then for block sizes of 4, 8, 16, 32, 64, 128, and 256, and associativities of 1, 2, and 4, what combination of block size and associativity minimizes the miss rate for each of the `cc1.100k.din`, `tex.100k.din`, and `spice.100k.din` benchmarks? Fill in the table below for each benchmark.

assoc	block size (bytes)						
	4	8	16	32	64	128	256
1							
2							
4							

Plot the three curves, corresponding to the three associativity values for each benchmark, in a single graph. Place the miss rate on a linear Y axis and the block size on a logarithmic X axis. (Note: freehand-sketched graphs are not acceptable; use graphing software, e.g., `gnuplot` on unix and Excel on PCs).

The best way to summarize rates is to use the harmonic mean. Using the same table and graph format, present the aggregate miss rate:

$$\frac{\sum_{i=1}^{3} m_i}{\sum_{i=1}^{3} r_i}$$

where m_i is the number of misses for benchmark i, and r_i is the total number of accesses for benchmark i.

What combination of block size and associativity minimizes the aggregate miss rate for the three benchmarks?

1.2) Block size effect (2 pts)

You should have seen that the aggregate miss rate as a function of block size is non-monotonic. In other words, the largest block size does not induce the best miss rate. Explain this phenomenon.

1.3) Analyze the differences between the miss rates of the three benchmarks. (2 pts)

1.4) What is the gain that we obtain through associativity? (2 pts)

Part II: Exploring cache size and splitting

For this part of the assignment we will assume an associativity of 1, block size of 16 Bytes, an LRU replacement policy, and a write-allocation policy of write-allocate.

2.1) Relative balance between instructions and data (10 pts)

Suppose the size of the data cache can be 2K, 4K, 8K, 16K, or 32K. For each of these values, the instruction cache can also be 2K, 4K, 8K, 16K, or 32K. Find the miss rate in this design space for each of the benchmarks. Compute the aggregate miss rate. Explain the results. Why does an even or a non-even split make sense? Plot a 3-D graph, where the sizes of the instruction and data caches are on the horizontal axes (logarithmic scale), and the miss rate is on the vertical axis (linear scale).

2.2) Split vs. unified cache (3 pts)

Assuming that the data cache can vary in size as in 2.1) and that the instruction cache can be only equal in size to the data cache, compare the resulting miss rate to that due to a unified cache that is twice the size of the data cache. Explain the results for each of the three benchmarks and for the aggregate miss rate.

Part III: Cost based optimization

We will approximate the cost of a cache as the total number of bits, used for either storing data or bookkeeping. This measure roughly tracks the chip area required. In this part of the assignment, we will only consider 32 bit addresses, an LRU replacement policy, and a write-allocation policy of write-allocate.

3.1) Cost analysis (10 pts)

Derive a formula for the total number of bits in a cache. Assume that the number of Bytes per block is B, the number of sets in the cache is S, the number of blocks per set is E, the number of status bits per block is 2 (one Valid and one Dirty bit), the number of tag bits per block is t, and the number of state bits in the Finite State Machine to determine the LRU replacement policy for each set is $E \times (E - 1)$.

3.2) Show what the cost would have been for each of the configurations explored in Parts I and II. (10 pts)

Hint: Use a spreadsheet.

3.3) Find the optimum cache configuration for each of the benchmarks and for the combined workload (using the aggregate miss rate) under the constraint that the total number of bits in the cache be less than or equal to 250 Kbits. (30 pts)

You may choose a split vs. unified cache, and vary the cache sizes, block size, and asociativity.

Explain how you arrived at the optimum. This could include tables and graphs, documenting the experiments that you made.

Hand In

A hardcopy writeup.