M3: Computer Simulation.

John N. Carter

April 2015

Abstract

This exercise looks at the operation of real programs on real computers. Three topics are addressed: Variation in Instruction mix; Pipeline hazards and Cache Design.

1 Introduction

The aim of this experiment is to introduce you to the concepts of simulating and measuring computer performance, using real programs, before committing to a hardware design

Supervisors should have a look at the annotated note, available from me.

Marks on understanding and progress will be based on the numbered questions.

Supervisor notes: Gage progress by the answers to the numbered questions i.e. has answers for 7 questions and notes to show how they were obtained give them 3.5 for progress.

This experiment is in three parts, spaced over a single session. Specifically

1. In the first part you will investigate the differing instruction mixes used to solve different problems. Table 1 lists the different tasks and work loads..
2. You will investigate the prevalence of Data and Branch hazards.
3. You will see how programs make use of the cache..

gcc-1K.trace.csv is available on line, the rest are too big and will be preloaded on to the workstations in the Electronics Lab.

As you already know simulation is an important tool for the electronic engineer. In CPU design this is doubly so, not only must one simulate at a hardware level one must simulate at a software level to test that the architecture is appropriate for the task(s) in mind.

Preparation: *You should read this document carefully.*

Program Name	**File Name**	**Comments**
art	art.trace.csv	Artificial Intelligence
gcc	gcc.trace.csv gcc-1K.trace.csv	gcc compiler, note the gcc-1K version is shorter and for testing.
hmmer	hmmer.trace.csv	Hidden Markov Model (speech signal processing)
libquantum	libquantum.trace.csv	Quantum Mechanics Simulation.
mcf	mcf.trace.csv	Not sure what this does.

Table 1: Different Program Types

Preparation: *Revise your knowledge of C/C++ programming and Computer Architecture.*

Supervisor notes: Check what they know when you talk with them, make sure both students can code.

The following resources exist to support this experiment.

- This document.
- An M3 web site where you will find down loadable code, links to documentation (and where appropriate, local copies). Notification of errors and corrections will be published here.
- The simulation trace files loaded on the ECS workstations.

If code is written in advance it should be printed out, with line numbers before you come into the laboratory (the SciTE text editor will do this for you). Software development should be incremental and evolutionary as you are learning about new concepts. Follow the methodology of this experiment and write simple programs.

Supervisor notes: As students if they understand how to drive the C++ compiler, either from am IDE such as Visual Studio, Eclipse or Code::Blocks. Lab development was done using the latter and the 32 bit version of MinGW.

1.1 Background Reading

The following are good sources of information on ideas related to this experiment.

Preparation: *You should look at all the following as part of your preparation.*

- C++ Programming.
- Module text for ELEC2204.

1.2 Analysing simulator output.

In this exercise you will be using the output of a simulator which simulates an Intel 'X86 class processor. This was developed at the University of Pennsylvania. The simulator breaks native instructions down into RISC-like micro-operations (AMD call these ROPS). It is these we will be considering. See the experiment web site for the exact format and a description of the trace output. The trace files have been preprocessed into csv format for easier decoding. A generic C++ program (**generic.cpp**) showing how these can be processed is explained in the following text.

```
1  /*
2   *  Generic Program to load, parse, process and report trace files in csv format.
3   */
4  #include <iostream>
5  #include <fstream>
6  #include <sstream>
7  #include <string>
8  #include <vector>
9  #include <stdlib.h>
10 int main() {
```

```
11      std::ifstream traceFile;
12      std::string row;
13      int counter = 0;
14      int limit = 10000000;
15      traceFile.open("gcc.trace.csv");
16      while(counter < limit) {
17          std::stringstream ls;
18          std::string val;
19          std::vector<std::string> all;
20          /*
21           *  Initalise for processing here.
22           */
23          if(!traceFile.good())
24              break;
25          traceFile >> row;
26          ls.str(row);
27          while(std::getline(ls, val, ',')) {
28              all.push_back(val);
29              }
30          if(all.size() != 14) {
31              std::cout << all.size()
32                        << " Bad trace size, ignore and continue.\n";
33              continue;
34              }
35          /*
36           *  Procesing here
37           */
38          counter++;
39          if(counter % 100000 == 0) {
40              if(counter > 999999) {
41                  std::cout << (float) counter / 1000000.0
42                            << " Million Lines Processed\n";
43                      }
44              else {
45                  std::cout << (float) counter / 1000.0
46                            << " Thousand Lines processed\n";
47                  }
48              }
49          }
50      /*
51       *  Report results here.
52       */
53      std::cout << "Finished\n";
54      return 0;
55      }
```

A line by line description follows.

4-9 Include files for file streams and basic data structures.

10 Main program starts here.

11 Declare *traceFile* to manage the input trace file.

12 *row* defined to hold each line from the trace file as it is read.

13 *counter* to keep a count of the number of files read.

14 *limit* to restrict the number of lines to be processed. Make it very big to process the whole file, greater than 100 Million should be enough.

15 Open the trace file

16 Loop over the lines in the trace files until count exceed limit or the end of file is reached.

17 *ls* is declared as a *stringstream*. This is a string that can be read like a file.

18 *val* is declared as a string to hold lines from the trace file as they are read.

19 *all* is declared as a vector of strings to hold the component parts of each line of tracing.

23-24 checks that we have not reached the end of file.

25 A row is read

26 Converted to a *streamstring*.

27-29 Looping over *ls*, reading and returning chunks ending in ',' and adding these to the vector *val*. Building up a vector or array of the component sub strings.

30-34 Checking that there are exactly 14 fields in the trace line (for the version with physical memory addresses attached the number of elements is 15). The error is reported, ignored and processing continues.

38 The line counter.

39-49 Very pretty progress reporting.

53-55 The end.

During the lab you will customise this program to make measurements from the instruction traces.

20-22 Add your code to initialise counters and other things.

35-37 Your code to process trace lines goes here.

50-52 Replace these lines with your code to report your results.

1.3 Preparation

1. Study the file **generic.cpp** and ensure that you understand how and what it does.
2. Compile and run **generic.cpp**, use the file *gcc-1K.trace.csv* as input. Ensure that it behaves as you expect and just reads its input, count lines and exits.
3. Study the file **countcode.cpp** (Section 2.1.1) and understand how the occurrence of the same text strings are counted as used in Section 2.1.
4. Prepare code to measure the pipeline properties Section 2.2
5. Make plans for measuring the effects of a directly mapped instruction cache, in Section 2.3.

Instruction Types
Logical
Integer Arithmetic
Floating point
Load
Store
Control: jump, Branch, Call, Conditional branch.

Table 2: Classification of Instructions.

6. If you feel that you will achieve the mandatory part of the lab with time to spare, make plans to investigate dynamic branch prediction, see section 3.1.

As part of your preparation you will write a number of small program to measure different things about the instruction traces. Remember to save these with different names.

2 Experimental Work.

2.1 Different Usage of Instructions for Different Tasks.

As you might expect different computing tasks require the use of different instruction. For example a word processor or editor might have lots of character and integer type instructions, while a MP3 media player might have lots of floating point instructions.

Question 1: Why is this?

Supervisor notes: MP3 uses a DCT algorithm, mostly implemented in floating point, often using specific instructions to operate on small arrays and blocks of data.

What mix of instructions would you expect for a game like Quake?

Supervisor notes: A good mix of FPU for graphics and physics with integer for rendering and control.

Run a program (section 2.1.1) to count micro-opcodes on the art, hmmer, mcf, gcc and libquantume trace files. Inspect the output and classify the results in the following broad classes, see Table 2.

Question 2: Now comparing the distributions obtained for the different traces, what deductions can you make about the different programs?

Supervisor notes: The major difference is between those programs that are floating point rich and those that are not.

2.1.1 Counting Instructions.

The following program (**countcode.cpp**) will count instructions.

```
1 #include <iostream>
2 #include <fstream>
3 #include <sstream>
4 #include <string>
```

```
 5 #include <vector>
 6 #include <map>
 7 #include <stdlib.h>
 8 int main() {
 9     std::ifstream traceFile;
10     std::map<std::string, int> frequency;
11     std::map<std::string, int>::iterator it;
12     std::string row;
13     int counter = 0;
14     int limit = 10000000;
15     traceFile.open("gcc.trace.csv");
16     while(counter < limit) {
17         std::stringstream ls;
18         std::string val;
19         std::vector<std::string> all;
20         if(!traceFile.good())
21             break;
22         traceFile >> row;
23         ls.str(row);
24         while(std::getline(ls, val, ',')) {
25             all.push_back(val);
26             }
27         if(all.size() != 14) {
28             std::cout << all.size()
29                       << " Bad trace size, ignore and continue.\n";
30             continue;
31             }
32         val = all[13];
33         it = frequency.find(val);
34         if(it == frequency.end()) {
35             frequency[val] = 1;
36             }
37         else {
38             frequency[val] ++;
39             }
40         counter++;
41         }
42     std::cout << frequency.size()
43               << " different instructions used\n";
44     for(it = frequency.begin(); it != frequency.end(); ++it) {
45         std::cout << it->first
46                   << " => "
47                   << it->second
48                   << '\n';
49         }
50     return 0;
51     }
```

This program uses a new data structure, the *map*. This is often known as an associative array or dictionary. The best way to think of it is as an array with arbitrary indexes. These could be integers, floats, strings or other data structures. The best way to understand them is to see them in action. In the following annotations the differences from the generic program are emphasised.

6 Include the *map* class.

10 Define a map *frequency* which has string indexes and integer values.

11 Define an iterator over the map, *it*. We use this like integers iterating over a regular array.

32 We reuse the variable *val* to take a copy of the 13th element of the trace. This is the text string giving the micro-opcode.

33 We search the map *frequency* for an index the same as *val*, and store the location in the variable *it*. If the index is not found in the map, then *it* points or refers to the end of the map.

34-36 If the micro-opcode is not found, then we start a new map entry *frequency[val]* and set it equal to 1.

37-39 It is found so we just add 1 to *frequency[val]*. This is counting the occurrences of the micro-opcodes.

44-49 Finally we reuse the variable *it* to loop over the map from *frequency.begin()* to *frequency.end()* incrementing the iterator as we go. *it->first* refers to the index and *it->second* refers to the value.

Preparation: *Run this program on the file* gcc-1k.trace.csv *before you come into the lab. Make sure you understand how it works and what the output means.*

Finally to make life easier the file *helper.cpp* on the lab page has functions to count things into a map and a compact function to report progress in millions of lines processed. Make sure you understand *helper.cpp* and *countcode.cpp* before you start to use them.

2.2 The Pipeline.

In this section we will make some measurements and see how data (2.2.1) and branch (2.2.2) hazards will affect the operation of a pipeline. We will assume that this processor executes micro-opcodes in a 4 stage pipeline with FETCH, DECODE, EXECUTE and WRITE/READ. There will be slight differences from the MIPS processor as FETCH will be from the micro-opcode generator, not main memory, while the WRITE/READ phase will consist of register writes **and** memory writes and reads. In this and subsequent sections you need only measure the properties of the file **gcc.trace.csv**.

Preparation: *If an instruction uses register 3 as destination. How may instructions must be executed before the register can safely be used as a source.*

Supervisor notes: One can argue for both two and three.

Supervisor notes: I guess most people will count stalls per micro instruction, I did in the example. There is an argument for counting per instruction.

2.2.1 Data hazards.

This processor has 16 registers i.e. 0 to 15 and two special registers 44 and 45. A data hazard exists if the source for a register instruction should have been updated by an instruction one or more cycles previously. If there was no pipeline this would not be a problem however there is a pipeline so hazards potentially exist. Determine the severity of the problem by counting the number of such occurrences.

This should be simple, comparing both the source fields to the destination field of previous instructions. It is easy to determine the previous trace values by defining additional vectors to hold previous values

```
1  ...
2  int main() {
3      ...
4      std:vector<std::string> minus1, minus2;
5      /*
6       * previous two lines, read 1st two lines
7       * of trace file into these
8       */
9      ...
10     while(...){
11         ...
12          ...
13          minus2 = minus1; // This is two instructions previously.
14          minus1 = all; // This is one instruction previously.
15          ...
16      }
17      ...
18  }
```

Hint: Rather than trying to program up hazard detection. It will be easier to count the number of register pairs between source and destination in the same way you counted instructions, i.e. in a map. You can make suitable keys by simply concatenating the source and destination strings, i.e. `val = all[2] + minus1[4];`. Generate the map and print it out. Then count the number of pairs that would give rise to data hazards i.e. the number of times register 3 is both source and destination is an example.

Supervisor notes: Try and stop them using logic to count the occurrences. They are much better at spotting the patterns in the output than they are programmers. The compiler seems to be doing a good job with register allocation, the number of data hazards is not great.

Question 3: What percentage of register operations cause data hazards?

Supervisor notes: They should be clear about what they call a hazard, especially how many instructions back do they look and stall the pipeline.

2.2.2 Branch hazards.

Branch hazards occur when the pipeline is filled with the wrong instructions. For example the pipeline contains the following instructions but the branch jumps to a different portion of code.

Question 4: In our 4 stage pipeline how many cycles could be lost for a mistaken branch.

Tag	Offset	Block
10 bits	10 bits	8 bytes or 3 bits

Table 3: Suggested Cache structure assuming 23 bit Address bus

There are various strategies we could adopt:

- Always assume that the branch will be taken.
- Always assume the branch will not be taken
- Chose randomly every time. You can use the built in `rand()` function. The code `rand() % 2` will give you 0 and 1 randomly.

The simulator tells us whether each branch is taken or not. Chose a branch strategy and count the number of times this gives the incorrect result and leaves the pipeline filled with useless instructions.

Question 5: Which of the three is the best?

Supervisor notes: Random has 25,640, true has 51,034, false has 117,100. Random wins with gcc.

If you have time at the end implement Dynamic Branch Prediction (see 3.1).

2.3 The cache.

In this section we are going to consider the effects of a cache on the execution of the simulated programs. We are only considering a cache for program instructions, not data. Thus it is only necessary to consider memory accessed via the program counter.

If you look at the trace for all the different programs you will find that the addresses used are very similar. This is because the addresses are virtual addresses. However a cache works with physical memory not virtual memory.

The simplest way to map virtual memory to physical memory is to do it on a one to one basis but this is not a good idea in a real multi-tasking environment as each program will need its own share of memory. Assuming a virtual and physical page size of 20000_{16} the virtual memory addresses have been mapped to physical addresses for each trace file. Each program is manually allocated its own range of physical address in the 400000_{16} bytes of physical memory. The details of this are not relevant.

In these new trace files, prefixed with the *abs*, the physical address of the virtual memory specified by the program counter has been appended as a hexadecimal number. A small example file can be found on the experiment web page. In them selves these individual simulations are not interesting as the cache hit rate is very high. For a more realistic situation one or more parts of each trace file has been concatenated together to simulate the effects of a multi-tasking operating system. This is stored in the file *abs.mixed.trace.csv*, a short part of this is on-line.

400000_{16} bytes of physical memory is equivalent to an address bus that is 23 bits wide. Table 3 shows a suggested allocation of bit fields in the address bus for a directly mapped cache. Each cache element holds 8 bytes, enough for, two 32 bit instructions, a 64-bit integer or a double precision float. We are not considering the value that is stored only its original address. The tag and offset are each 10 bits wide, making a 1k Word cache for 4 Megabytes of memory.

Supervisor notes: Explain that this is a very artificial situation but it should demonstrate the behavior of a real cache.

Implement code to measure the effects of a cache as described in Table 3. That is, has a memory address been accessed before and has its value previously been stored in the cache.

We are not implementing a fully functional cache, only that part concerning locating a value in cache. This can be done with two arrays, both the same size, to hold the cache valid bit and the current tag. You should use a *bool* or logic array, for the valid flag and an integer array to hold the tag. For a 10-bit offset/line each array will be 1024 or 2^{10} elements in length.

The following code suggests how the generic program might be modified to measure the effects of a cache[1]:

```
1  /*
2   *  This code determines if an address is in the cache.
3   */
4      ...
5      std::string spad;
6      int ipad, line, tag;
7      ...
8      bool valid[1024];
9      int tag[1024];
10     bool cachehit;
11     ...
12     spad = all[14]; // absolute address of PC in hex textual representation.
13     ipad = (int) strtol(spad.c_str(), 0, 16); // Integer address after converting
14     ...
15     line = (ipad >> 3) & ((1 << 10) -1);    // line offset
16     tag = (ipad >> 13) & ((1 << 10) -1);    // tag value
17     ...
18     cachehit = false;
19     if(valid[line] == false) {
20             lines[line] = tag;
21             valid[line] = true;
22             }
23     else {
24             if(lines[line] == tag) {
25                     cachehit = true;
26                     }
27             else {
28                     lines[line] = tag;
29                     }
30     ...
```

Question 6: How do the constants used in computing the *line* and *tag* values relate to Table 3.

Supervisor notes: See fully worked example programs.

Use *abs.mixed.trace.csv* to measure the effects of your cache.

Given the total number of instructions (N) in *abs.mixed.trace.csv*, a short sample is on-line, print out the number of cache hits every $N/100000$ instructions. Cut and paste the numbers into Excel or MATLAB and draw a graph of *cache hit rate* vs *time*.

[1] ... indicates missing code to be filled in from the generic example.

Question 7: What can you tell from the graph? Can you spot the boundaries between the programs?

Is there an optimum cache configuration? Vary the number of bits for tag and offset to try and determine what the values are. Specifically vary the size of the size of the cache line from 8 to 12 to see if there is an optimum.

Question 8: What are the best values you can find?

Supervisor notes: Not sure there is a strong maximum, but the important thing is that they look for it.

Question 9: How would you use a *map* class to implement a fully associative cache.

Supervisor notes: First version just use tag + line field to compute an index for a map. Use this to define a map for the valid bit map. If they are switched on they may want to limit the size of the map. In this case they will need a replacement strategy.

Finally, remember that this is a very artificial situation. If you were evaluating a cache strategy for a real (Intel, ARM) processor you would have to do significantly more simulation with real work loads. Imagine the simulation that needs to be done for your laptop.

3 Additional work.

If you have some spare time you might like to try this

Supervisor notes: I've (jnc) not done this.

3.1 Dynamic Branch Prediction

Implement dynamic branch prediction, using a map addressed by the branch location in the memory code to hold the state variable for each branch location. Initialise each branch, on first occurrence to taken.

Question 10: Is this better than fixed/random strategies? Is it worth it, in terms of the extra complexity incurred?

Clearly there must be a limit to the number of branch states the can be remembered. How would this be implemented and how would you replace branches when you run out of capacity.