CSE 30321 – Computer Architecture I – Fall 2010

Lab 01: Architectural-level Performance Metrics
Total Points: 100 points
Assigned: September 7, 2010
Due: September 21, 2010

1. Goals

Thus far in lecture, we have discussed two important concepts – (1) Instruction Set Architectures (or ISAs) and (2) techniques for measuring architectural level performance. The primary focus of this lab is on measuring how hardware or software changes can impact performance. Upon completion of this lab, you should:

1. Have a better understanding as to what architectural level design features can help to improve performance – and when adding new features is *not* beneficial. For example, reductions in application execution time may saturate with the addition of more hardware – which suggests that the additions are *not* really that helpful.
2. Even more importantly, you'll learn how to use simulation tools that can be leveraged to evaluate *any* potential architecture – and for different application spaces.

2. Introduction and Overview

More specifically, you'll use an architectural-level simulator called *SimpleScalar.* SimpleScalar allows you to describe a microprocessor's datapath and memory hierarchy in a simple, text-based configuration file. For example, using the datapath for the 6-instruction processor discussed in Lectures 02 and 03 as an example, we might have entries in our configuration file that allow us to:

- Specify the number of ALUs available:
    - # total number of integer ALU's available
      -res:ialu 1

More advanced designs can also be described. For example, you can:

- Specify the time required to access on-chip memory
    - # l1 data cache hit latency (in cycles)
      -cache:il1lat 1
        - *(if data is requested from on-chip memory, it will take 1 CC to get it)*
    - # l1 instruction cache hit latency (in cycles)
      -cache:il1lat 1
        - *(if an instruction is fetched from on-chip memory, it will take 1 CC to get it)*
- Specify the time required to access off-chip memory
    - # memory access latency (<first_chunk> <inter_chunk>)
      -mem:lat 64 1
    - *(i.e. if data is requested from off-chip memory, it will take 64 CCs to get it)*
- Specify the bandwidth between your microprocessor and main memory
    - # memory access bus width (in bytes)
      -mem:width 4
    - *(i.e. there is a 32 bit path from the processor to off-chip memory)*
- …

(Recall from Lecture 1 that, generally, all of the data needed by a program will not be able to fit in the faster, on-chip memory. Instead, the most frequently used data is kept in faster, on-chip memory – and requests are made to slower, off-chip memory if data is not found on-chip.)

After describing the processor/memory architecture that you wish to evaluate, you can then use SimpleScalar to predict how a given *program* or *benchmark* will perform given that processor/memory

configuration. Detailed, architectural-level simulations allow a computer architect or compiler writer to see whether or not design changes to a processor/memory configuration are beneficial at the application-level.

With SimpleScalar, you can:

1. Analyze a given processor/memory configuration with pre-compiled benchmark suites
2. Study how your own code might benefit from a different processor/memory configuration. This is possible with the use of a special cross-compiler.

In this lab, we'll do both.

The version of SimpleScalar that we'll use is targeted toward the ISA associated with ARM microprocessors. There are several reasons for this choice:

1. The ARM ISA is very similar to the MIPS ISA discussed in class. For example, as seen below, a multiply instruction for an ARM ISA essentially has the same syntax as a multiply instruction for the MIPS ISA.

ARM and Thumb-2 Instruction Set
Quick Reference Card

Operation		§	Assembler	S updates	Action
Multiply	Multiply		`MUL{S} Rd, Rm, Rs`	N Z C*	Rd := (Rm * Rs)[31:0]

MIPS: Mul $9, $7, $8 # mul rd, rs, rt: RF[rd] = RF[rs]*RF[rt]

2. You should become comfortable with interpreting machine code for different microprocessor architectures.
   - The goal of the course is not to teach you MIPS or ARM assembly, but rather to (a) understand how the machine instructions for a RISC-like ISA accomplish some set of tasks specified by High-Level Language (HLL) code and (b) understand how changing code written in some HLL and/or a given processor-memory configuration can affect performance. In this regard, working with different ISAs is good.
3. ARM microprocessors are used *everywhere*. Thus, the microprocessor configurations and benchmarks that we will work with in lab are very representative of hardware you use, and programs that you run everyday.

3. Problem A

3.1 Benchmarks:

Different benchmark suites exist that allow a user to test a processor/memory configuration with a workload that is representative of how that processor/memory configuration might actually be used. As examples, a few popular benchmark suites are briefly discussed below in Table 1:

Table 1: Three example benchmark suites.

SPEC-Integer
A common benchmark suite from ***SPEC*** (Integer) is representative of what might be used for desktop-like workloads. Example programs in this suite include **gcc** (i.e. a compiler), **perl** (i.e. an interpreter), **vortex** (a database program), **gzip** (a compression tool), and **parser** (a grammar checker)
LINPACK
The ***LINPACK*** benchmark suite might be used if you were designing an architecture for a high-performance computing system (i.e. something that might be used for weather/climate modeling). "LINPACK is a software library for performing numerical linear algebra on digital computers. The LINPACK Benchmarks are a measure of a system's floating point computing power. Introduced by Jack Dongarra, they measure how fast a computer solves a dense *N* by *N* system of linear equations $Ax = b$, which is a common task in engineering."[1]
TPC
The Transaction Processing Performance Council (**TPC**) has organized a suite of benchmarks designed to test and stress database applications. For example, "The ***TPC Benchmark™H (TPC-H)*** is a decision support benchmark. It consists of a suite of business oriented ad-hoc queries and concurrent data modifications. The queries and the data populating the database have been chosen to have broad industry-wide relevance. This benchmark illustrates decision support systems that examine large volumes of data, execute queries with a high degree of complexity, and give answers to critical business questions."[2]

In this lab we'll use a benchmark suite called ***MiBench*.** MiBench is representative of functions that a microprocessor in an *embedded system* (a sensor, iPhone, iPod, e-Reader, etc.) might perform. The MiBench suite itself contains 35 benchmarks in 6 different classes – but we will run 1or 2 at most from a given class. Each benchmark will consist of a small dataset input and a larger dataset input (i.e. a small MP3 to decode and a large MP3 to decode). The specific benchmarks that we'll study, why they are important, and how to run them will be described in Section 3.3[3], which also will detail the specific tasks you will need to perform for Problem A.

3.2 Processor Configurations:

In this lab, you will use two Simplescalar configuration files that are representative of two ARM microprocessors: the ***StrongARM*** and the ***XScale***.

- "The ***StrongARM*** was a collaborative project between DEC and Advanced RISC Machines to create a faster ARM microprocessor. The StrongARM was designed to address the upper-end of the low-power embedded market, where users needed more performance than the ARM could deliver while being able to accept more external support. Targets were devices such as newer personal digital assistants and set-top boxes."[4]
- Versions of the next generation "***XScale*** microprocessors can be found in products such as the popular RIM BlackBerry handheld, the Dell Axim family of Pocket PCs, most of the Zire, Treo and Tungsten Handheld lines by Palm … The XScale is also used in devices such as PVPs (Portable Video Players), PMCs (Portable Media Centres), including … the Amazon Kindle E-Book reader, and industrial embedded systems."[5]

---

[1] text from: http://en.wikipedia.org/wiki/LINPACK
[2] text from: http://www.tpc.org/tpch/
[3] text from: "MiBench: A free, commercially representative embedded benchmark suite," M.R. Guthas, et. al.
[4] text from: http://en.wikipedia.org/wiki/StrongARM
[5] text from: http://en.wikipedia.org/wiki/XScale

A brief comparison of the StrongARM architecture to the XScale architecture is given in Table 2.

Table 2: Comparison of StrongARM to XScale processor.

Design Parameter	StrongARM	XScale	Comment
# of Integer ALUs	1	1	
# of Integer Multiplier/Dividers	1	1	
# of Floating Point ALUs	1	1	
# of Floating Point Multiplier/Dividers	1	1	
Amount of on-chip memory	16 KBytes	32 Kbytes	The XScale has 2X the amount of faster, on-chip memory
Time required to access on-chip memory	1 Clock Cycle	1 Clock Cycle	The time required to access faster, on-chip memory is the same on the StrongARM and the XScale
Time required to access off-chip memory	64 Clock Cycles	32 Clock Cycles	The time required to access off-chip memory is 50% lower on the XScale
Bandwidth between microprocessor and off-chip memory	32 Bits	64 Bits	The bandwidth between the microprocessor and off-chip memory is 2X higher with the XScale

Note that unless you attempt to answer an extra credit question or wish to change the name of the file that simulation output is written to, you will not need to edit the configuration files for this lab as they are included with the benchmark directory hierarchy that you will copy over from the course AFS space.

3.3 What to do:

1. The SimpleScalar executable will need to be run on a linux machine. For a list of machines that you should be able to log in to and use, see Appendix A. Thus, to get started:
   - Using a terminal program, **log on to a linux machine**.
   - Note that it should be possible to download and compile the simulation software on a personal machine but (a) we will not support this (as it would essentially require one-on-one installation sessions) and (b) you will probably want to run benchmarks in parallel (so using cluster machines is simply more efficient). That said, my former TA Aaron Dingler has prepared a document that provides some installation tips if you are interested in installing the software locally. (Please don't bombard Aaron with questions about this.)

2. For Problem A, you will need to **copy two directories** (from the course directory) into your home directory.
   - /afs/nd.edu/coursefa.10/cse/cse30321.01/Labs/01/lab_benchmarks
   - /afs/nd.edu/coursefa.10/cse/cse30321.01/Labs/01/mibench

   The directory "lab_benchmarks" contains processor configuration files and scripts that are needed to run each benchmark. While you will need to update the scripts and configuration files to reflect the path to where you copied the files to (many benchmarks require file writes), the scripts were included so that you do not have to spend an excessive amount of time determining what arguments a particular benchmark takes to run.

The directory “mibench” contains the pre-compiled benchmark executables for the entire suite – although we will only use 6 of them. The sub-directories in mibench also contain input files that are needed to run a given benchmark (i.e. a list of IP addresses to traverse, an MP3 file to decode, etc.)

3. Also, before getting started, it will be helpful if you **update your path** to point to the SimpleScalar executable. The executable can be found at:
   - /afs/nd.edu/coursefa.10/cse/cse30321.01/arm/simplesim-arm
     - It is called “sim-outorder”

   To update your path (i.e. so that you can just type “sim-outorder” from the command line) add the following line to the path in your .cshrc file:

   setenv PATH /afs/nd.edu/coursefa.10/cse/cse30321.01/arm/simplesim-arm:$PATH

   (Be sure to type: “source .cshrc”)

4. Now, let’s discuss benchmark specifics. Below, I will discuss:
   - What “class” a benchmark belongs to
   - What the benchmark does
   - **The syntax required to run the benchmark**
   - **What to look for when you are running the benchmark**

   In item 5, I will discuss what data you should collect (and how to find it in the file written as part of a simulation output).

   Benchmark #1: Ispell

Benchmark Class
The **Office** applications are primarily text manipulation algorithms to represent office machinery like printers, fax machines and word processors. The PDA market mentioned in the Consumer category also relies heavily on the manipulation of text for data organization.
Benchmark #1
Ispell is a fast spelling checker that is similar to the Unix spell, but faster. It supports contextual spell checking, correction suggestions, and languages other than English. The input consists of a small and large document from web pages.
How to run it…
The syntax for running Ispell in SimpleScalar is as follows: sim-outorder -config {config file} {ispell executable} -a -d {input dictionary} < {input file} > {output file} Thus, the Ispell benchmark takes a dictionary file, and a file to spell check as inputs. The output of the benchmark (not the SimpleScalar simulation) is written to a text file.
What to look for…
In the text file, you will see suggestions for words that are misspelled, mis-hyphenated, etc. For example, look for the word “instrment” when examining the benchmark output for the small input file.

Benchmark #2, #3, and #4: jpeg, lame, mad

Benchmark Class

The **Consumer Devices** benchmarks are intended to represent the many consumer devices that have grown in popularity during recent years like scanners, digital cameras and Personal Digital Assistants (PDAs). The category focuses primarily on multimedia applications with the representative algorithms being jpeg encoding/decoding, image color format conversion, image dithering, color palette reduction, MP3 encode/decoding, and HTML typesetting.

Benchmark #2

jpeg compress/decompress: JPEG is a standard, lossy compression image format. It is included in MiBench because it is a representative algorithm for image compression and decompression and is commonly used to view images embedded in documents. The input data are a large and small color image.

How to run it…

The syntax for running jpeg (compress) in SimpleScalar is as follows:
sim-outorder -config {config file} {compress jpeg executable} {input PPM file} > {output jpeg file}

The syntax for running jpeg (decompress) in SimpleScalar is as follows:
sim-outorder -config {config file} {decompress jpeg executable} {input jpeg file} > {output PPM file}

What to look for…

The PPM file should be larger (in file size) than the jpeg file. Also, you can use the output of compress as an input to decompress and vice versa. However, scripts are set up so that this is NOT required.

Benchmark #3

Lame is an MP3 encoder that supports constant, average and variable bit-rate encoding. It uses small and large wave files for its data inputs.

How to run it…

The syntax for running lame is as follows:
sim-outorder -config {config file} {lame executable} {wav file input} ./{name of MP3 output}

What to look for…

You should be able to play the output MP3 file in a standard MP3 player. You can also use the output of this benchmark as an input to Benchmark #4 (mad) – although this is not required. Note that this benchmark and the mad benchmrk will probably take the longest to run.

Benchmark #4

Mad is a high-quality MPEG audio decoder. It currently supports MPEG-1 and the MPEG-2 extension to Lower Sampling Frequencies, as well as the so called MPEG 2.5 format. All three audio layers (Layer I, Layer II, and Layer III a.k.a. MP3) are fully implemented. It uses small and large MP3s for its data inputs.

How to run it…

The syntax for running mad is as follows:
sim-outorder -config sa1core.cfg {mad executable} {MP3 file input} -o ./{wav file output}

What to look for…

You should be able to play the output wav file as well. You can also use the output of this benchmark as an input to Benchmark #3 (lame) – although this is not required. Note that this benchmark and the lame benchmark will probably take the longest to run.

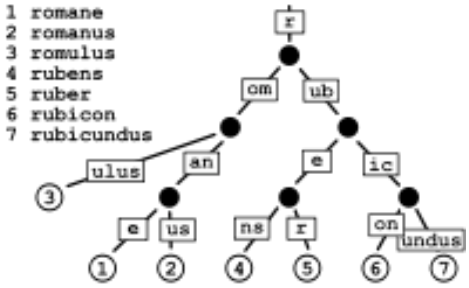
Benchmark #5: Patricia

Benchmark Class – Network

The **Network** category represents embedded processors in network devices like switches and routers. The work done by these embedded processors involves shortest path calculations, tree and table lookups and data input/output.

Benchmark #5

A *Patricia* trie is a data structure used in place of full trees with very sparse leaf nodes. Branches with only a single leaf are collapsed upwards in the trie to reduce traversal time at the expense of code complexity. Often, Patricia tries are used to represent routing tables in network applications. The input data for this benchmark is a list of IP traffic from a highly active web server for a 2 hour period. The IP numbers are disguised. See example of text Patricia trie[6] in this box.



How to run it…

The syntax for running Patricia is as follows:
sim-outorder -config {config file} {patricia executable} {input file of addresses to parse}

What to look for…

The acronym PATRICIA stands for: "**P**ractical **A**lgorithm **T**o **R**etrieve **I**nformation **C**oded **I**n **A**lphanumeric". When you run this benchmark, there will be a search for the IP addresses in the input file. When each is found, this will be noted as output.

Benchmark #6: Rijndael Encryption and Decryption

Benchmark Class – Data Security

Data Security is going to have increased importance as the Internet continues to gain popularity in e-commerce activities. Therefore, Security is given its own category in MiBench. The Security category includes several common algorithms for data encryption, decryption and hashing.

Benchmark #1

Rijndael encrypt/decrypt was selected as the National Institute of Standards and Technologies Advanced Encryption Standard (AES). It is a block cipher with the option of 128-, 192-, and 256-bit keys and blocks.

How to run it…

The syntax for running rijndael (encrypt) in SimpleScalar is as follows:
sim-outorder -config {config file} {rijndael executable} {file to encrypt} ./{encrypted file} e {128 bit key}

The syntax for running rijndael (decrypt) in SimpleScalar is as follows:
sim-outorder -config {config file} {rijndael executable} {file to decrypt} ./{decrypted file} d {128 bit key}

What to look for…

Note that the 128 bit key can be specified as a 32 digit, hexadecimal number.

While not required, you can use the output of the encryption benchmark as an input to the decryption benchmark (or vice versa).

Also, again, while not required, some other interesting things to try might be: (1) trying to read an encrypted file and (2) decrypting a file with a different key (and trying to read it).

[6] image from: http://upload.wikimedia.org/wikipedia/commons/thumb/a/ae/Patricia_trie.svg/200px-Patricia_trie.svg.png

5. Below, I discuss how you should analyze benchmark data/metrics (and also how to parse/find data)

   Writing simulation output to a file:
   When you run a script to execute a benchmark, the output will be written to a text file. The name of the text file is specified in the first entry in the configuration file:

   ```
   # redirect simulator output to file (non-interactive only)
   -redir:sim              sa1core_large.txt
   ```

   Thus, if you want the simulation output to go do a different file, just change this file name.

   Also, while some benchmarks write to a file during execution (i.e. as part of the execution process itself, not for the purposes of writing simulation output), the scripts that I have provided to run the benchmarks should ensure that multiple simulation instances do not write to the same file.

   Parsing the output file:
   When you look at the simulation results (i.e. sa1core_large.txt) you will see LOTS of data. Some data will be more understandable later in the semester, while other data would make more sense after you took a graduate-level computer architecture course. However, there should be several lines in the file that are quite familiar. For example:

   - sim_total_insn 1588563478 # total number of instructions executed
   - sim_total_loads 259741195 # total number of loads executed
   - sim_total_stores 184385892 # total number of stores executed
   - sim_total_branches 147474085 # total number of branches executed
   - sim_CPI 1.5260 # cycles per instruction

   You may want to pay attention to a few other lines too:
   - dl1.accesses 435160624 # total number of [data] accesses
   - dl1.hits 435071297 # total number of hits
   - dl1.misses 89327 # total number of misses
   - dl1.miss_rate 0.0002 # [data] miss rate (i.e., misses/ref)
   - il1.accesses 1265769872 # total number of [instruction] accesses
   - il1.hits 1265384028 # total number of [instruction] hits
   - il1.misses 385844 # total number of [instruction] misses
   - il1.miss_rate 0.0003 # [instruction] miss rate (i.e., misses/ref)

   The first 4 data items tell you how often data for a load or store instruction is found in “fast” memory. If there is a “miss”, then we must look for data in a “slower” level of memory. The “miss_rate” tells you how frequently a slower memory access is required. The second 4 items tell you how often an instruction encoding is found in fast memory (i.e. a “fetch”).

   Note that the number of “accesses” is essentially equal to the number of load and store instructions.

   Also, in the directory: /afs/nd.edu/coursefa.10/cse/cse30321.01/Labs/01/ there are simple scripts that you can use to parse an output text file to extract the above data from a simulation output file.
   - See: “parse_IC_count”, “parse_CPI”, and “parse_Memory”
   - (If you change the output file naming convention, and use the scripts, you may need to update them.)

Data to Collect and Questions to Answer:

Question A.1:
Run the benchmarks discussed above assuming a StrongARM configuration and an XScale ARM configuration. Using simulation output from each benchmark, complete Table 3 below. Assuming that processor clock rates are the same, how does the more sophisticated (XScale) design affect speedup? (Try to comment on the suite as a whole, not just benchmark by benchmark.)

Table 3: Performance of different benchmarks for small and large data sets given different processor configurations.

	StrongARM small input	XScale small input	Speedup	StrongARM large input	XScale large input	Speedup
Ispell	1.5807	1.4320	1.1038	1.5260	1.4175	1.0765
JPEG (compress)	1.2807	1.1895	1.0767	1.2968	1.1845	1.0948
JPEG (decompress)	1.3819	1.2582	1.0983	1.3829	1.2624	1.0955
Lame	2.1438	1.6872	1.2706	2.0747	1.6452	1.2611
Mad	1.7739	1.2422	1.4280	1.5835	1.2575	1.2592
Patricia	6.2542	1.3881	4.5056	6.0199	1.3866	4.3415
Rijndael (encrypt)	1.9230	1.4885	1.2919	1.2715	1.2291	1.0345
Rijndael (decrypt)	1.9516	1.5655	1.2466	1.2400	1.1940	1.0385

Answer
As an example, for the large data sets:
- Without Patricia, the average speedup is: 7.8601 / 7 = 1.123X
- With Patricia, the average speedup is 12.20 / 8 = 1.525X

Question A.2
What if the clock rates for the StrongARM configuration and XScale configuration are different – i.e. the clock rate for the StrongARM is X and the clock rate for the XScale is Y? Is speedup affected? If so, how? (Again, try to comment on the suite as a whole, not just benchmark-by-benchmark.)

Answer
If the clock rate of the XScale is greater, the speedup will increase.

Question A.3
Do any benchmarks seem to especially benefit from the more sophisticated XScale configuration? If so, why do you think this is the case? And if so, try to support your conclusion with simulation data. (You might reference the discussion that compares and contrasts the StrongARM and XScale configurations. You might also reference some of the other metric data that I discussed above.)

Answer
Data for Patricia- Small:

SA $CPI_{average}$	6.0199	**XScale CPI-average**	1.3866
SA $i1_{Miss\ Rate}$	4.5%	**XScale i1-Miss Rate**	0%
SA $CPI_{average\ (new)}$	1.4506		
SA $i1_{Miss\ Rate\ (new)}$	0%		

A.Extra Credit

For the benchmark where there is the biggest performance gap between the StrongARM design and the XScale design, create a hybrid configuration file and re-run the benchmark to see if there is a significant change if you alter the StrongARM architecture. How is performance affected?

4. Problem B

An understanding of the underlying processor-memory architecture can actually make you a better programmer. In this problem, you will see for the first time why this is the case.

4.1 Setup:

For Problem B, in addition to the sim-outorder SimpleScalar simulator, you will also use the SimpleScalar cross compiler – which allows you to compile C-code into ARM assembly that can be used within the SimpleScalar environment.

For this problem, you will study code that will multiply the same 2 matrices together – but in two different ways. (Recall from Problem A that matrix multiplies are an important part of the LINPACK suite.) To get started, **copy the "cross_compile" directory over to your AFS space** (see path below).

/afs/nd.edu/coursefa.10/cse/cse30321.01/Labs/01/cross_compile

In this directory, there are 2 sub-directories: "traditional_multiply" and "non_traditional_multiply". Each sub-directory contains the following:

1. A file with c-code for a matrix multiply
2. An ARM processor configuration file
3. A simple script to "run" your program on the ARM configuration.

Before you do any SimpleScalar simulations, you will first need to compile the C-code so that it can be run in the SimpleScalar environment. To do this, we'll use the SimpleScalar cross compiler. As in Problem A, it will be helpful if you **update your path to include the *arm-gcc* executable**.

To update your path (i.e. so that you can just type "arm-gcc" from the command line) add the following line to the path in your .cshrc file:

```
setenv PATH /afs/nd.edu/coursefa.10/cse/cse30321.01/arm/cross-compiler/gcc-4.1.1-glibc-2.3.2/arm-unknown-linux-gnu/bin:$PATH
```

Then, to compile the matrix multiply (or any) C-code so that we can study it within the SimpleScalar environment, you can simply type:

```
> arm-gcc traditional_matrix_multiply.c –o traditional -static
```

To run this program in the SimpleScalar environment (i.e. to see how efficiently the StrongARM configuration can perform this matrix multiply), you can simply use the script provided OR type:

```
> sim-outorder –config sa1core.cfg <cross-compiled executable name>
```

4.2 Data to Collect and Questions to Answer:

Question B.1

For each implementation of the matrix multiply, compile the C-code with the SimpleScalar cross compiler. Then, run the code (which performs a 1000x1000 matrix multiply) with the StrongARM processor configuration file. How does the execution time of the traditional matrix multiply compare to the performance of the non-traditional matrix multiply? You can assume that the clock rate is 233 MHz.

Warning: One of these simulations could take up to 1.5 days to finish, so be sure to plan ahead.

Answer:

	Clock Rate	CPI	IC	Execution Time	D1 Miss Rate	I1 Miss Rate
No Blocking	233 MHz	2.7982	58,126,096,440	698.1s	10.24%	0%
Blocking	233 MHz	1.1427	68,253,137,391	334.7s	0.12%	0%

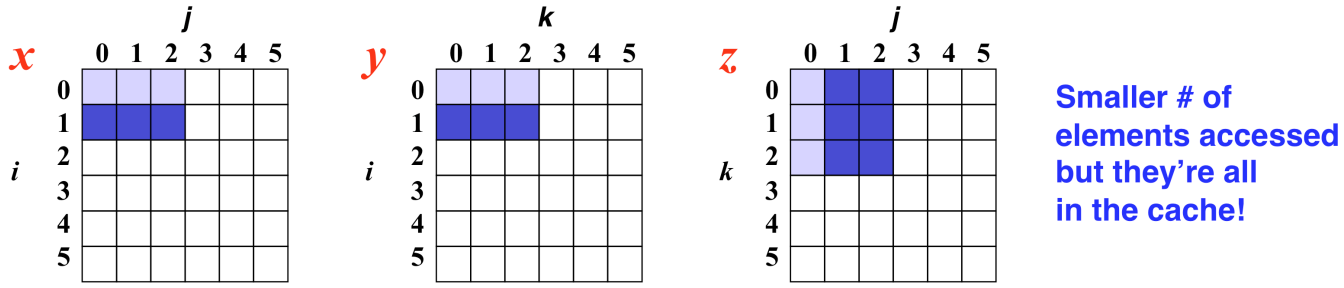
The blocking code is 2.08X faster than the blocking code.

Question B.2

Trace through the code for the alternative matrix multiply. What is this code doing differently? Why do you think it has a positive impact on performance? (Hint: Look at the metrics that you can extraact from the simulation output file as discussed in Problem A.)

Answer

Pictorially what happens is:



Smaller # of elements accessed but they're all in the cache!

B.Extra Credit

Think of another example that you believe might benefit from a code re-write. Re-write the code, test it, and describe your results. (Be sure that there is a "baseline" to compare to.) You don't necessarily have to demonstrate significant improvement. I am more interested in why you chose some particular code to try and improve.

5. Problem C

It is important to understand that better performance may not mean "faster execution time / lower latency" (i.e. one architecture may be considered advantageous over another if it can offer comparable – yet slower – execution time, and if battery life is significantly improved).

Setup:
We can use an extension to SimpleScalar called "sim-panalyzer" that estimates the average power required to execute a given benchmark. To use this executable, you should update your path one more time.

```
setenv PATH /afs/nd.edu/coursefa.10/cse/cse30321.01/arm/sim-panalyzer/PA2.0.3/Implementations/ARM/sim-panalyzer-2.0:$PATH
```

Data to Collect and Questions to Answer:
In the last part of this lab, you should rerun the *lame* and *mad* benchmarks (with small data sets) assuming the StrongARM and XScale ARM configurations. To run the benchmarks, you can use the same scripts / files that were used in Problem 1. However, **the "sim-panalyzer" executable should be used instead of "sim-outorder"**.

Record the average power.

- To find this in the output text file, type: "more <output.txt> l grep 'uarch.avgpdissipation'
- (Note that the units of the number provided are Watts)

Question C.1
Assume that for both configurations the microprocessor's clock rate is 233MHz. Calculate the execution time for the lame and mad benchmarks. Sim-panalyzer will call a slightly different version of sim-outorder, therefore the CPI values between Problems A and C may not be exactly the same.

Lame	Clock Rate	Instruction Count	CPI	Execution Time	Average Power
SA1	233 MHz	168040321	2.3393	1.687 s	0.5196 W
XScale	233 MHz	165078123	1.6866	1.195 s	1.0406 W

Mad	Clock Rate	Instruction Count	CPI	Execution Time	Average Power
SA1	233 MHz	47404636	1.9274	0.392 s	0.5357 W
XScale	233 MHz	46507175	1.2473	0.249 s	1.1067 W

Question C.2
Compare the differences in execution time to the average amount of power required for each benchmark given each configuration. From the standpoint of battery life, do you think that the more sophisticated design is worthwhile for these two benchmarks?

Answer

- For Lame, XScale is 57% faster than SA1. But the power is 2.07X higher.
- For XScale is 41% faster than SA1. But the power is 2X higher.

Answers will vary for justification.

6. What to Turn In:

You should complete a ***typed*** report with answers to:

- Questions A.1 – A.3
- Quesitons B.1 – B.2
- Questions C.1 – C.2

Appendix:

- student0{0,1,2,3}@cse.nd.edu
  - i.e. student 01@cse.nd.edu
- Also, see the list of machines at:
  - http://crcmedia.hpcc.nd.edu/wiki/index.php/Linux_Cluster_Host_List

Below is general feedback and my grading criteria for this lab.

Problem A1 (10 points)

- 2 points were subtracted if you had an outlier datapoint (i.e a speedup value that was inconsistent with the rest of your results. In these cases, you should re-run the simulation or re-check the simulation if the data point was far off from all other data points (most of the time, some error will have occurred).
- 5 points were **added** if you considered the performance of the suite as a whole and took into account average speedup with and without the Patricia outlier. This is nice analysis (as you tried to explain the performance advantages of a newer design with and without one particular benchmark). Generally, if you see an outlying data point like this, it's good to dig deeper.
- 1 point was subtracted if you did not discuss the performance of the suite as a whole

General comment: Most groups received full credit (or just lost a few points) for Problem A1

Problem A2 (10 points)

- 10 points were subtracted if you said that speedup is not affected
- Speedup is Old Execution Time divided by New Execution Time
    - Execution time is CPI x IC x Clock Cycle Time
    - Clock Cycle Time is 1/Clock Rate
- *Be sure to understand how speedup and execution time are defined*.

General comment: Only a few groups had trouble with this question, and most received full credit.

Problem A3 (25 points)

- The majority of groups (~60-70%) figured out that the Patricia benchmark performance had something to do with the amount of time data or instruction encodings were found in fast memory (instead of having to go to look for data or instruction encodings in slower, off-chip memory).
- If you mentioned memory and miss rates, you received full credit
- If you figured out that it had something to do with memory, but did not comment on miss rate, 5 points were subtracted
- If you just said that Patricia was the outlier, but did not explain, 15 points were subtracted as thinking/hypothesizing about the reason "why" was at the core of this problem.
- A few groups confused fast, on-chip memory with registers. This is not exactly the case. 2 points were subtracted for this case
- A few answers did not fall into any of the above buckets, and points were subtracted as appropriate.

General comment: I was fairly pleased with how people did on this question. It was designed to get you to delve into simulation data and think about what could be causing a data outlier. Many groups derived very thoughtful responses.

Problem B1 (10 points)

- I believe that several groups used "instructions committed" for instruction count rather than the number at line:

    sim_total_insn 58126096440 # total number of instructions executed

    - However, in most cases, groups that did this still understood the problem. As I can see the potential for confusion, and the answer was not adversely affected, no points were subtracted. However, if future labs, be sure to use the right number.
    - Note that I may have said "see me" if you did this (or marked your line with a star). If I included a star, this was your mistake and there is no need to see me.
- In a few other cases, groups reported times of hours or ~50,000 seconds. (This happened about 5-6 times.) If I did this, please see me to explain your logic – as you may get points back. (5 points were tentatively subtracted.)

Problem B2 (25 points)

- If you mentioned that the performance difference had something to do with memory, but did not elaborate, 5 points were subtracted.
- If there was no mention of putting smaller chunks of the matrices being multiplied into faster memory, 10 points were subtracted.

General comment: Again, ~60-70% of the class figured this out, and I was pleased with the thoughtful answers.

Problem C1 (10 points)

- Nearly everyone got this problem right.

Problem C2 (10 points)

- Nearly everyone got this problem right.