HY425

Machine Assignment 1

Assignment: 3/11/2008

Due Date: 14/11/2008

Instructions: For this assignment you will need to submit a write-up (in .pdf) with your experimental findings from the SimpleScalar simulator. Send your tarball to Stamatis Kavadias (kavadias@ics.forth.gr). Use the following subject in your e-mail (please do not change the subject!): **HY425: MA1 Submission**.

Start early!

Introduction

The purpose of the machine assignments is to train you in compiling and running programs with the SimpleScalar Simulator. You will also be trained in measuring quantitative properties of software and hardware, which guide processor design choices. Lastly, you will learn how to evaluate the impact of alternative design choices on performance.

Preparation

Read Appendix A thoroughly. Visiting www.simplescalar.com and reading the documentation of the simulator is also strongly recommended. Following this assignment, you should then be able to answer the questions:

i) What is the role of simulators in processor design?
ii) Why is it advantageous to have several simulators?

In the course of this project you will run a large number of simulations, and it may be difficult to keep track of the results, unless you maintain a detailed log of your experiments (also known as a "lab book"). This log should contain a description of all the simulation runs you performed, your simulation plans, comparison of results, graphs, if any, etc. In addition as you will start using more detailed simulators, simulation time will increase. A log, which documents all the simulation runs you performed already, will help you avoid repeating runs and save considerable time.

Part A: Program profiling (50 points)

Download and setup the SimpleScalar simulator, following the instructions given in the attachment to this project description. Download the benchmarks from the course web page (available at http://www.csd.uoc.gr/~hy425/homework/benchmarks.tar.gz). These benchmarks are briefly described in the Appendix of the textbook (section A.3). You are asked to run the profiling simulator on these benchmarks and answer questions about some of their quantitative properties.

Your mission in this assignment is to fill the following table with all the available benchmark programs and instruction class profiles. For each instruction class, list the percentage of the total dynamic instruction count corresponding to that class. Then, you need to analyze each benchmark and answer the following questions:

i) Is the benchmark memory-intensive or computation-intensive?
ii) Is the benchmark integer or floating-point intensive?
iii) What percentage of the executed instructions are conditional branches? Given this percentage, how many instructions on average, does the processor execute between each pair of conditional branch instructions, without including the conditional branch instructions?

Table: Benchmark programs versus instruction profiles

Benchmark	Load	Store	Uncond. Branch	Conditional Branch	Integer Computation	Floating point computation
anagram						
go						
compress						
gcc						

Part B (50 points)

i) In this part you will evaluate some branch prediction alternatives for the programs in Part A. For this part, you will use the out-of-order processor simulator of SimpleScalar, using default parameters, except from the parameters of the branch prediction scheme. Fill in the information in the following table, by measuring the performance of the processor with the alternative branch prediction schemes. Each entry in the table should be the CPI of the processor with the program and branch prediction scheme corresponding to the entry:

Benchmark	predict always taken	predict always not-taken	perfect predictor	Bimodal predictor, 1024-entry branch history table, 512-entry, 4-way associative BTB	Bimodal predictor, 2048-entry branch history table, 512-entry, 4-way associative BTB
anagram					
go					
compress					
gcc					

A bimodal predictor is a predictor with a fixed-size table. Each entry in the table stores the history of one branch. Branches are used to index the table using some of their least significant bits (how many depends on the size of the table). An n-bit saturating counter is used to track the history of each branch. Simplescalar provides a bimodal predictor with 2-bit saturating counters for tracking branch history.

ii) Comment on the performance of the branch prediction schemes you evaluated. In particular, compare realistic branch prediction schemes against the unrealistic perfect branch predictor.

Appendix A: Short Guide to the SimpleScalar Tool-Set

This text is based on the manual by Ewa Z. Bem, School of Computing and Information Technology, University of Western Sydney Nepean, which in turn was based on the manual by Todd M. Bezenek, University of Wisconsin. It contains background material about the SimpleScalar toolset of simulators. SimpleScalar itself is available for download together with various tools and utilities including detailed documentation from http://www.simplescalar.com/

A1: SimpleScalar and Simulation in Computer Architecture

When computer architecture researchers work to improve the performance of a computer system, they often use an existing system to simulate a proposed system. Although the intent is not always to measure raw performance (estimating power consumption is one alternative), performance estimation is one of the most important results obtained by simulation. The SimpleScalar tool set is designed to measure the performance of several parts of a superscalar processor and its memory hierarchy. This document describes the SimpleScalar simulators. Other simulation systems may be similar or very different.

Overview of SimpleScalar Simulation

The SimpleScalar tool set includes a compiler that creates binaries for a non-existent processor. The binaries can be executed on one of several simulators that are included in the tool set. This section describes the goals of processor simulation.

The execution of a processor can be modelled as a series of known states and the time (or other costs, ie., power) required to make the transition between each pair of states. The state information may include all or a subset of:

- The values stored in all memory locations.
- The values stored in and the status of all cache memories.
- The values stored in and the status of the translation-lookaside buffer (TLB).
- The values stored in and the status of the branch prediction table(s) or branch target buffer (BTB).
- All processor state (ie. the pipeline, execution units (integer ALU, load/store unit, etc.), register file, register update unit (RUU), etc.)

A good way to evaluate the performance of a program on a proposed processor architecture is to simulate the state of the architecture during the execution of the program. By simulating the states through which the processor will pass during the execution of a program and estimating the time (or other measurement) necessary for each state transition, the amount of time that the simulated processor will need to execute the program can be estimated.

The more state that is simulated, the longer a simulation will take. Complex simulations can execute 100s of times slower than a real processor. Therefore, simulating the execution of a program that would take an hour of CPU time on an existing processor can take a week on a complex simulator. For this reason, it is important to evaluate what measurements are desired and limit the simulation to only the state that is necessary to properly estimate those measurements. This is the reason for the inclusion of several different simulators in the SimpleScalar tool set.

Profiling

In addition to estimating the execution time of a program on the simulated processor, profile information may be of use to computer architects. Profile information is a count of the number or frequency of events that occur during the execution of a program. One common example of profile data is a count of how often each type of instruction (ie., branch, load, store, ALU operation, etc.) is executed during the running of a program.

Profile information can be used to gauge the relative importance of each part of a processor's implementation in determining its performance when executing the profiled program.

The SimpleScalar Base Processor

The SimpleScalar tool set is based on the MIPS R2000 processor's instruction set architecture (ISA). The processor is described in MIPS RISC Architecture by Gerry Kane, published by Prentice Hall, 1988. The ISA describes the instructions that the processor is capable of executing - and therefore the instructions that a compiler can generate - but it does not describe how the instructions are implemented. The implementation is what computer architects change in order to improve the performance of a processor.

An existing processor can be chosen as a base processor for several reasons. These may include:

- The architecture of the processor is well known and documented.
- The architecture of the processor is state-of-the-art and therefore it is likely to be useful as a base for the study of future processors.
- The architecture of the processor has been implemented as a real processor, allowing simulations to be compared to executions on a real, physical processor.

An important consideration in the choice of the MIPS architecture for the SimpleScalar tool set was the fact that the GNU GCC compiler was available in source-code form, and could compile to the MIPS architecture. This allowed the use of this public-domain software as part of the SimpleScalar tool set.

Description of the Simulators

The SimpleScalar tool set includes a number of simulators designed for various purposes. They are described below. For those simulators we are using there are also a description of the important profiling options available.

sim-bpred

This simulator implements a branch predictor analyser.

sim-cache

This simulator implements a functional cache simulator. Cache statistics are generated for a user-selected cache and TLB configuration, which may include up to two levels of instruction and data cache (with any levels unified), and one level of instruction and data TLBs. No timing information is generated.

sim-cheetah

This program implements a functional simulator driver for Cheetah. Cheetah is a cache simulation package written by Rabin Sugumar and Santosh Abraham which can efficiently simulate multiple cache configurations in a single run of a program. Specifically, Cheetah can simulate ranges of single level set-associative and fully-associative caches.

```
#-option <args>   # <default> # description
-refs    <string> #      data # reference stream to analyze, {none|inst|data|
unified}
-R       <string> #       lru # replacement policy, i.e., lru or opt
-C       <string> #        sa # cache configuration, i.e., fa, sa, or dm
-a       <int>    #         7 # min number of sets (log base 2, line size for DM)
-b       <int>    #        14 # max number of sets (log base 2, line size for DM)
-l       <int>    #         4 # line size of the caches (log base 2)
-n       <int>    #         1 # max degree of associativity to analyze (log base 2)
-in      <int>    #       512 # cache size intervals at which miss ratio is shown
-M       <int>    #    524288 # maximum cache size of interest
-c       <int>    #        16 # size of cache (log base 2) for DM analysis
```

Note that 'line size' above is the same as block size. Most of the parameters above are give as log base 2 of the number, ie a line size of 16 bytes is given as '`-l 4`.

sim-fast

This simulator implements a very fast functional simulator. This functional simulator implementation is much more difficult to digest than the simpler, cleaner sim-safe functional simulator. By default, this simulator performs no instruction error checking, as a result, any instruction errors will manifest as simulator execution errors, possibly causing sim-fast to execute incorrectly or dump core. Such is the price we pay for speed!!!!

sim-outorder

This simulator implements a very detailed out-of-order issue superscalar processor with a two-level memory system and speculative execution support. This simulator is a performance simulator, tracking the latency of all pipeline operations.

```
# -option         <args>       # <default> # description
-fetch:ifqsize    <int>        #         4 # instruction fetch queue size (in insts)
-fetch:mplat      <int>        #         3 # extra branch mis-prediction latency
-bpred            <string>     #     bimod # branch predictor type
                                           # {nottaken|taken|perfect|bimod|2lev|comb}
-bpred:bimod      <int>        #      2048 # bimodal predictor config (<table size>)
-decode:width     <int>        #         4 # instruction decode B/W (insts/cycle)
-issue:width      <int>        #         4 # instruction issue B/W (insts/cycle)
-issue:inorder    <true|false> #     false # run pipeline with in-order issue
-issue:wrongpath  <true|false> #      true # issue instructions down wrong execution
paths
-commit:width     <int>        #         4 # instruction commit B/W (insts/cycle)
-cache:dl1        <string>     # dl1:128:32:4:l # l1 data cache config
-cache:dl1lat     <int>        #         1 # l1 data cache hit latency (in cycles)
-cache:dl2        <string>     # ul2:1024:64:4:l # l2 data cache config
-cache:dl2lat     <int>        #         6 # l2 data cache hit latency (in cycles)
-cache:il1        <string>     # il1:512:32:1:l # l1 inst cache config
-cache:il1lat     <int>        #         1 # l1 instruction cache hit latency (in
cycles)
-cache:il2        <string>     #       dl2 # l2 instruction cache config
```

```
-cache:il2lat    <int>        #       6 # l2 instruction cache hit latency (in
cycles)
-mem:lat         <int list...># 18 2 # memory access latency (<first_chunk>
<inter_chunk>)
-mem:width       <int>        #       8 # memory access bus width (in bytes)
-tlb:itlb        <string>     # itlb:16:4096:4:l # instruction TLB config
-tlb:dtlb        <string>     # dtlb:32:4096:4:l # data TLB config
-tlb:lat         <int>        #      30 # inst/data TLB miss latency (in cycles)
-res:ialu        <int>        #       4 # total number of integer ALU's available
-res:imult       <int>        #       1 # total number of integer
multiplier/dividers available
-res:memport     <int>        #       2 # total number of memory system ports
available (to CPU)
-res:fpalu       <int>        #       4 # total number of floating point ALU's
available
-res:fpmult      <int>        #       1 # total number of floating point
multiplier/dividers available

The cache config parameter <config> has the following format:

  <name>:<nsets>:<bsize>:<assoc>:<repl>

  <name>   - name of the cache being defined
  <nsets>  - number of sets in the cache
  <bsize>  - block size of the cache
  <assoc>  - associativity of the cache
  <repl>   - block replacement strategy, 'l'-LRU, 'f'-FIFO, 'r'-random

  Examples:   -cache:dl1 dl1:4096:32:1:l
              -dtlb dtlb:128:4096:32:r

Cache levels can be unified by pointing a level of the instruction cache
hierarchy at the data cache hiearchy using the "dl1" and "dl2" cache
configuration arguments.  Most sensible combinations are supported, e.g.,

  A unified l2 cache (il2 is pointed at dl2):
    -cache:il1 il1:128:64:1:l -cache:il2 dl2
    -cache:dl1 dl1:256:32:1:l -cache:dl2 ul2:1024:64:2:l

  Or, a fully unified cache hierarchy (il1 pointed at dl1):
    -cache:il1 dl1
    -cache:dl1 ul1:256:32:1:l -cache:dl2 ul2:1024:64:2:l
```

sim-profile

This simulator implements a functional simulator with profiling support.

```
# -option  <args>       # <default> # description
-nice      <int>        #         0 # simulator scheduling priority
-max:inst  <uint>       #         0 # maximum number of inst's to execute
-all       <true|false> #     false # enable all profile options
-iclass    <true|false> #     false # enable instruction class profiling
-iprof     <true|false> #     false # enable instruction profiling
-brprof    <true|false> #     false # enable branch instruction profiling
-amprof    <true|false> #     false # enable address mode profiling
-segprof   <true|false> #     false # enable load/store address segment profiling
-tsymprof  <true|false> #     false # enable text symbol profiling
-taddrprof <true|false> #     false # enable text address profiling
-dsymprof  <true|false> #     false # enable data symbol profiling
-internal  <true|false> #     false # include compiler-internal symbols during
                                      symbol profiling
```

sim-safe

This simulator implements a functional simulator. This functional simulator is the simplest, most user-friendly simulator in the simplescalar tool set. Unlike sim-fast, this functional simulator checks for all instruction errors, and the implementation is crafted for clarity rather than speed.

The sim-cache and sim-cheetah simulators simulate only the state of the memory system-they do not keep track of the timings of events. The sim-outorder simulator does. In fact, it simulates everything that happens in a superscalar processor pipeline, including out-of-order instruction issue, the latency of the different execution units, the effects of using a branch predictor, etc. Because of this, sim-outorder runs more slowly, but it also generates much more information about what happens in a processor.

Because sim-outorder keeps track of timing, it can report the number of clock cycles that are needed to execute the given program for the simulated processor with the given configuration.

A.2: Installing and running simulation experiments with SimpleScalar

The instructions below show how you build and install development tools for SimpleScalar. For the exercises in this lab you, however, do not need this. The actual simulators are enough.

For Cygwin on Windows/PC platform

If you want to install SimpleScalar on a Windows/PC platform, you need to first install the Cygwin Unix emulation environment. Download Cygwin from http://www.cygwin.org and install it. Make sure you include the development tools for gcc in your installation.

Then get the SimpleScalar package from http://www.simplescalar.com. Go to Tools in the Downloads section to the left and download simplesim-3v0d.tgz. This is a gzipped tar package. To unpack it, place the file in a directory[1] of your choice and do the following command:

```
gunzip –c simplesim-3v0d.tgz | tar xvf –
```

This will create a subdirectory simplesim-3.0 with the source code for all simulators described above. To build the simulators read the README file. Here are the steps (% means the command prompt):

```
% make config-pisa
% make
% make sim-tests (to verify that the simulators built OK)
```

The first step will set up the files for building the PISA target. The other alternative is an Alpha target. For information about the PISA instruction set, please see the SimpleScalar documentation: http://www.simplescalar.com/docs/users_guide_v2.pdf. It applies to version 2, but works for version 3 as well.

[1] The path to the directory must not contain any white spaces.

In order to be able to compile programs to run on the simulator, you need a port of cross-compiler and libraries for Cygwin. It can be found here:

http://www.eecg.toronto.edu/~moshovos/ACA05/hw/ss-gcc.usrlocal.tar.bz

- Install with: “bunzip2 –c ss-gcc.usrlocal.tar.bz | tar xvf – “
- Include /usr/local/bin in your path
- Compile programs using “ss-gcc”

For Linux on PC platform

Please refer to the instructions found on the link below for a Linux installation that works:

http://www.comp.nus.edu.sg/~panyu/simplesim.htm

A.3: Available benchmarks

The benchmarks described here are precompiled for SimpleScalar/PISA and can be downloaded from the link below:

http://www.csd.uoc.gr/~hy425/fall_2006/project/benchmarks.tgz

anagram

A program for finding anagrams for a phrase, based on a dictionary.

compress

(SPEC) Compresses and decompresses a file in memory.

go

(SPEC) Artificial intelligence; plays the game of "Go" agianst itself

perl

Calculates popularity of nodes in a graph based on the PageRank algorithm from Google.

gcc

(SPEC) Limited version of GCC