SimpleScalar exercise 1

CS425

2006-10-23

Due November 6th. Accounts for 1/6 of the total project score.

Inspired and modified with permissions from exercises at Lund University, Sweden and KTH, Sweden.

1 Introduction

After this exercise, you should be able to run programs with the SimpleScalar simulator. You should also be able to compile programs that can be run on the simulator. Finally, you should be able to comment on the impact of optimization on CPI.

You are expected to hand in a short written report, as a PDF document, in which you summarize your findings and experiences. The report must not be shorter than 2 pages. Include tables and thorough answers to all questions. Send the report by email to: s v e n k a @ i c s . f o r t h . g r

Preparation

Read Appendix A “Short guide to the SimpleScalar tool-set” thoroughly. You should be able then to answer the following questions:

- What is the role of simulators in processor design?
- Why is it advantageous to have several different simulators?

Note: In the course of these exercises you will run a large number of simulations, and it may be difficult to keep track of your results unless you maintain a lab book. This book should contain the 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, the simulation time will increase. A lab book, which documents all the simulation runs you performed already, will help you avoid repeat runs and will save you considerable time.

Assignment 1: Program behaviour (Instruction Profiling)

Download and set up the SimpleScalar simulator. Download the benchmarks from (they are briefly described in section A.3 of the “Short guide to the SimpleScalar tool-set”), and run the profiling simulator for them to find out the distribution of instruction classes.
Fill table 1 with all available benchmark programs versus instruction class profiles. Analyze each benchmark further:

1. Is the benchmark memory intensive or computation intensive?
2. Is the benchmark mainly using integer or floating point operations?
3. What percentage of the instructions executed are conditional branches? Given this percentage, how many instructions on average does the processor execute between each pair of conditional branch instructions (do not include the conditional branch instructions).

<table>
<thead>
<tr>
<th>benchmark</th>
<th>load</th>
<th>store</th>
<th>uncond</th>
<th>cond</th>
<th>integer</th>
<th>fp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>anagram</td>
<td></td>
<td></td>
<td>branch</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>go</td>
<td></td>
<td></td>
<td>branch</td>
<td></td>
<td>computation</td>
<td>computation</td>
</tr>
<tr>
<td>compress</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gcc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 1** Benchmark programs versus instruction class profiles.

**Assignment 2: Effects of compiler optimizations**

Compile the anagram program supplied with the SimpleScalar simulator and run it on the simulator. Observe that you can select more than one compiler optimization, but not all combinations make sense (like -O2 together with -O1 for example). Choose the 4 most relevant optimisations in order to make this program run faster. Motivate why you choose them! Use simulation to measure effects of each option separately and then all 4 together. Compare with executions of the program compiled with -O2 and -O0 (separately of course!). Fill in the table 2.

<table>
<thead>
<tr>
<th>Compiler optimization</th>
<th>no of instructions executed</th>
<th>no of cycles</th>
<th>CPI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-O2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-O0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Compare instruction profile for the program compiled with -O2 and -O0.

**Literature**

- Hennessy and Patterson: Chapters 1-2.
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
-ref s <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)
-i <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 given as log base 2 of the number, i.e. 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
-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> # dl1:128:32:4:1 # 11 data cache config
-cache:dl1lat   <int>   # 1 # 11 data cache hit latency (in cycles)
-cache:dl2     <string> # ul2:1024:64:4:1 # 12 data cache config
-cache:dl2lat   <int>   # 6 # 12 data cache hit latency (in cycles)
-cache:il1     <string> # il1:512:32:1:1 # 11 instr cache config
-cache:il1lat   <int>   # 1 # 11 instruction cache hit latency (in cycles)
-cache:il2     <string> # d12 # 12 instruction cache config
```
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 hierarchy 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
```

Or, a fully unified cache hierarchy (il1 pointed at dl1):

```
cache:il1 dl1
```

**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](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](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](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" against itself

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

- **gcc**
  (SPEC) Limited version of GCC
Appendix B: GCC Optimization options

Options of the form `-fflag` specify machine-independent flags. Most flags have both positive and negative forms; the negative form of `-ffoo` would be `-fno-foo`. The following list shows only one form—the one, which is not the default. You can figure out the other form by either removing `no-` or adding it.

- **-ffloat-store**
  Do not store floating point variables in registers. This prevents undesirable excess precision on machines such as the 68000 where the floating registers (of the 68881) keep more precision than a double is supposed to have.
  
  For most programs, the excess precision does only good, but a few programs rely on the precise definition of IEEE floating point. Use `-ffloat-store` for such programs.

- **-fmemoize-lookups**

- **-fsave-memoized**
  Use heuristics to compile faster (C++ only). These heuristics are not enabled by default, since they are only effective for certain input files. Other input files compile more slowly.
  
  The first time the compiler must build a call to a member function (or reference to a data member), it must (1) determine whether the class implements member functions of that name; (2) resolve which member function to call (which involves figuring out what sorts of type conversions need to be made); and (3) check the visibility of the member function to the caller. All of this adds up to slower compilation. Normally, the second time a call is made to that member function (or reference to that data member), it must go through the same lengthy process again. This means that code like this

  ```
  cout << "This " << p << " has " << n << " legs."
  ```

  makes six passes through all three steps. By using a software cache, a `"hit"` significantly reduces this cost. Unfortunately, using the cache introduces another layer of mechanisms, which must be implemented, and so incurs its own overhead. `-fmemoize-lookups` enables the software cache.

  Because access privileges (visibility) to members and member functions may differ from one function context to the next, g++ may need to flush the cache. With the `-fmemoize-lookups` flag, the cache is flushed after every function that is compiled. The `-fsave-memoized` flag enables the same software cache, but when the compiler determines that the context of the last function compiled would yield the same access privileges of the next function to compile, it preserves the cache. This is most helpful when defining many member functions for the same class: with the exception of member functions which are friends of other classes, each member function has exactly the same access privileges as every other, and the cache need not be flushed.

- **-fno-default-inline**
  Don't make member functions inline by default merely because they are defined inside the class scope (C++ only).
-fno-defer-pop
Always pop the arguments to each function call as soon as that function returns. For machines which must pop arguments after a function call, the compiler normally lets arguments accumulate on the stack for several function calls and pops them all at once.

-fforce-mem
Force memory operands to be copied into registers before doing arithmetic on them. This may produce better code by making all memory references potential common subexpressions. When they are not common subexpressions, instruction combination should eliminate the separate register-load. I am interested in hearing about the difference this makes.

-fforce-addr
Force memory address constants to be copied into registers before doing arithmetic on them. This may produce better code just as `-fforce-mem' may. I am interested in hearing about the difference this makes.

-fomit-frame-pointer
Don't keep the frame pointer in a register for functions that don't need one. This avoids the instructions to save, set up and restore frame pointers; it also makes an extra register available in many functions. It also makes debugging impossible on most machines.

On some machines, such as the Vax, this flag has no effect, because the standard calling sequence automatically handles the frame pointer and nothing is saved by pretending it does not exist. The machine-description macro FRAME_POINTER_REQUIRED controls whether a target machine supports this flag.

-finline-functions
Integrate all simple functions into their callers. The compiler heuristically decides which functions are simple enough to be worth integrating in this way. ‘

If all calls to a given function are integrated, and the function is declared static, then GCC normally does not output the function as assembler code in its own right.

-fcaller-saves
Enable values to be allocated in registers that will be clobbered by function calls, by emitting extra instructions to save and restore the registers around such calls. Such allocation is done only when it seems to result in better code than would otherwise be produced.

This option is enabled by default on certain machines, usually those, which have no call-preserved registers to use instead.

-fkeep-inline-functions
Even if all calls to a given function are integrated, and the function is declared static, nevertheless output a separate run-time callable version of the function.

-fno-function-cse
Do not put function addresses in registers; make each instruction that calls a constant function contain the function's address explicitly.

This option results in less efficient code, but some strange hacks that alter the assembler output may be confused by the optimizations performed when this option is not used.

**-fno-peephole**

Disable any machine-specific peephole optimizations.

**-ffast-math**

This option allows GCC to violate some ANSI or IEEE rules/specifications in the interest of optimizing code for speed. For example, it allows the compiler to assume arguments to the `sqrt` function are non-negative numbers.

This option should never be turned on by any `-O` option since it can result in incorrect output for programs, which depend on an exact implementation of IEEE or ANSI rules/specifications for math functions.

The following options control specific optimizations. The `-O2` option turns on all of these optimizations except `-funroll-loops` and `-funroll-all-loops`.

The `-O` option usually turns on the `-fthread-jumps` and `-fdelayed-branch` options, but specific machines may change the default optimizations.

You can use the following flags in the rare cases when 'fine-tuning' of optimizations to be performed is desired.

**-fstrength-reduce**

Perform the optimizations of loop strength reduction and elimination of iteration variables.

**-fthread-jumps**

Perform optimizations where we check to see if a jump branches to a location where another comparison subsumed by the first is found. If so, the first branch is redirected to either the destination of the second branch or a point immediately following it, depending on whether the condition is known to be true or false.

**-funroll-loops**

Perform the optimization of loop unrolling. This is only done for loops, whose number of iterations can be determined at compile time or run time.

**-funroll-all-loops**

Perform the optimization of loop unrolling. This is done for all loops. This usually makes programs run more slowly.

**-fcse-follow-jumps**

In common subexpression elimination, scan through jump instructions when the target of the jump is not reached over any other path. For example, when CSE encounters an if statement with an else clause, CSE will follow the jump when the condition tested is false.
-fcse-skip-blocks
This is similar to `-fcse-follow-jumps', but causes CSE to follow jumps, which conditionally skip over blocks. When CSE encounters a simple if statement with no else clause, `-fcse-skip-blocks' causes CSE to follow the jump around the body of the if.

-frerun-cse-after-loop
Re-run common subexpression elimination after loop optimizations has been performed.

-felide-constructors
Elide constructors when this seems plausible (C++ only). With this flag, GNU C++ initializes y directly from the call to foo without going through a temporary in the following code:

```c
A foo ();
A y = foo ();
```

Without this option, GNU C++ first initializes y by calling the appropriate constructor for type A; then assigns the result of foo to a temporary; and, finally, replaces the initial value of `y' with the temporary.

The default behavior (`-fno-elide-constructors') is specified by the draft ANSI C++ standard. If your program's constructors have side effects, using `-felide-constructors' can make your program act differently, since some constructor calls may be omitted.

-fexpensive-optimizations
Perform a number of minor optimizations that are relatively expensive.

-fdelayed-branch
If supported for the target machine, attempt to reorder instructions to exploit instruction slots available after delayed branch instructions.

-fschedule-insns
If supported for the target machine, attempt to reorder instructions to eliminate execution stalls due to required data being unavailable. This helps machines that have slow floating point or memory load instructions by allowing other instructions to be issued until the result of the load or floating point instruction is required.

-fschedule-insns2
Similar to `-fschedule-insns', but requests an additional pass of instruction scheduling after register allocation has been done. This is especially useful on machines with a relatively small number of registers and where memory load instructions take more than one cycle.

-O
Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function.
Without `-O', the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Statements are independent: if you stop the program with a breakpoint between
statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the results you would expect from the source code. Without `-O', only variables declared register are allocated in registers. The resulting compiled code is a little worse than produced by PCC without `-O'. With `-O', the compiler tries to reduce code size and execution time.

When you specify `-O', the two options `-fthread-jumps' and `-fdefer-pop' are turned on. On machines that have delay slots, the `-fdelayed-branch' option is turned on. For those machines that can support debugging even without a frame pointer, the `-fomit-frame-pointer' option is turned on. On some machines other flags may also be turned on.

-02
Optimize even more. Nearly all supported optimizations that do not involve a space-speed tradeoff are performed. Loop unrolling and function inlining are not done, for example. As compared to -O, this option increases both compilation time and the performance of the generated code.

-03
Optimize yet more. This turns on everything -O2 does, along with also turning on `-finline-functions'.

-00
Do not optimize.