Project Report for CS359

Part A: SimpleScalar-based Study Project

STUDENT ID: 5110309193

NAME: WANG Tianze

13.12.03

All of the files (installation package, build file, manual, problem sheet, intermediate results files and the report itself) referenced in the report can be downloaded via ftp://public.sjtu.edu.cn (user: wtze, password: public). I hope that they could be helpful to you if possible. And, as usual, if there's any problem, contact me via vendisky@gmail.com.

Project report for CS359

Part A: SimpleScalar-based Study Project

5110309193
Wang Tianze
CS.SJTU, China
vendisky@gmail.com

***Abstract*—This report is mainly about some experiments using Simplescalar. We start from zero, and end with concise conclusion of problems that will be discussed later. Basic settings, experiment description, results, discussion and conclusion will be covered gradually. I hope it could inspire some new thoughts into you, out of we've learnt from textbook.**

***Index Terms*—Simplescalar, miss ratio, associativity, IPC, prediction rate**

I. Basic settings

This part devoted to the aspects of basic settings. That is, the installation of Simplescalar, and the source of some files.

First, let's talk about installation. Dr. Zhu has provided us with some tutorials in course website, which led us to http://pages.cs.wisc.edu/~mscalar/simplescalar.html. Unfortunately, I found that the SimpleScalar tool set available there didn't work in my ubuntu 13.04 system. Compile error is reported while installing, for both 2.0 and 3.0 version.

As a result, I turned to google for help. In sina blog, I found a brilliant article on the problem. Here's the URL: http://blog.sina.com.cn/s/blog_55a4cddc01019usk.html. Now, the installation seemed to be a piece of cake...Until I found that everthing is OK in this version, except that sim-cheetah is missing. Oh god, how to decribe the feeling at that moment?

Give up is not my style, never! So, let's google again! At last, I downloaded a good version of simplesim-3v0d via http://www.ict.kth.se/courses/IS2202/software/. Next, I replaced simplesim-3v0d in the old installation package, uploading the new one to my ftp, and modify the build file(so downloading source is my ftp now), and mission accomplished.

If you are a rookie of Simplescalar, I will strongly recommend you to apply the follwing steps:

1) Login my ftp, download file ss.build.

2) Open terminal of your linux system, key in source ss.build, enter.

3) Waiting for the successful intallation.

Beautiful and convenient, isn't it?

Next, the source of some files. The manual and the problem sheet are available on Dr. Zhu's website. As for the 'go' benchmark, it's in ftp://ftp.cs.wisc.edu/sohi/Code/simplescalar, named simplebench.little.tar.

So that's what basic settings are all about. Let's move to the experiment description part.

II. Experiment description

In this part, I will cover the brief introduction of each experiment, and also, the operations we need to do in order to obtain the correct experimental data. Problem 1,2 and 8 will be discussed in the report.

Throughout the experiment we need to set the parameters of simulators. Reading manual carefully is enough to fulfill this task. However, I still encounter a few of difficulties beyond the setting works. There's some ambiguity in the problem sheet, and I queried the TA and other pals. At last, every detail seemed to be clear enough. Putting go.ss and go.in in folder Simplescalar/simplesim-3.0, and let's start.

A. Problem 1

This problem requires us to compare the benifits between increasing associativity and increasing the number of sets for caches of equivalent size, using simulator sim-cheetah. For details, see problem sheet pg. 1.

The command lines are as follows:

cd Simplescalar/simplesim-3.0

./sim-cheetah -b 11 -n 2 -refs unified go.ss 2 8 go.in

LRU replacement policy, min number of sets($128 = 2^7$), 16-byte(2^4) cache lines are default settings of sim-cheetah. Therefore, we don't need to write them in command lines. All we need to set is max number of sets(-b 11, $2048 = 2^{11}$), max degree of associativity(-n 2, $4 = 2^2$) and reference stream to analyze(-refs unified, reference stream is unified). 2 and 8 denote play quality and the board size respectively.

And here is the question: why use unified reference stream while it is not specially mentioned in the problem description? Well, you will know the answer in problem 2.

B. Problem 2

(a)

Problem 2(a) requires us to do some analysis which concern 6 group of given sets/associativity combinations, using simulator sim-cache. For details, see problem sheet pg. 2.

The command lines are as follows:

cd Simplescalar/simplesim-3.0

./sim-cache -cache:il1 dl1 -cache:dl1 ul1:128:16:1:l -cache:il2 none -cache:dl2 none -tlb:dtlb none -tlb:itlb none go.ss 2 8 go.in

This is a simple example configuring unified 128/1 set/associativity combination. For sim-cache, there are no default settings, you have to set every parameter on your own.

Since problem 2(b) requires seperate cache, we use unified cache in this case.

Let me explain the command lines for you. -cache:il1 dl1 points instruction L1 cache to data L1 cache. -cache:dl1 ul1:128:16:1:l obeys the format as <name>:<nsets>:<bsize>:<assoc>:<repl>, so the name is ul1, 128 sets, 16-byte block size, associativity is 1, LRU replacement policy. You see, in order to facilitate later analysis, block size and replacement policy are the same as problem 1. As for -cache:il2 none -cache:dl2 none -tlb:dtlb none -tlb:itlb, it is for acceleration.

(b)

Problem 2(b) requires us to compare miss ratios between unified and seperate cache. For details, see problem sheet pg. 2.

The command lines are as follows:

cd Simplescalar/simplesim-3.0

./sim-cache -cache:il1 il1:1024:16:1:l -cache:dl1 dl1:1024:16:1:l -cache:il2 none -cache:dl2 none -tlb:dtlb none -tlb:itlb none go.ss 2 8 go.in

This is a simple example configuring seperate 2048/1 set/associativity combination. Instruction cache and data cache halve the number of sets, so both of them are 1024. Nevertheless, the format remains the same. -cache:il1 il1:1024:16:1:l means that the name is il1, 1024 sets, 16-byte block size, associativity is 1, LRU replacement policy. I think it is needless to explain the rest of command lines.

C. *Problem 8*

Problem 8 requires us to compare the performance among 6 different branch predictors. For details, see problem sheet pg. 7.

The command lines are as follows:

cd Simplescalar/simplesim-3.0

./sim-profile -iclass go.ss 2 8 go.in

This is the pre-requisite of the problem. From it we can obtain some statistics of conditional branches. And now, let's see the main dish:

cd Simplescalar/simplesim-3.0

./sim-outorder -bpred nottaken go.ss 2 8 go.in

The parameter 'nottaken' represents a type of branch predictor. Totally there are 6 types, nottaken, taken, perfect，bimod, 2lev and comb. We need to try each of them, recording the data at the same time.

III. RESULTS

A. *Problem 1*

TABLE I. MISS RATIOS OF PROBLEM 1

No. of sets	Associativity			
	1	*2*	*3*	*4*
128	0.284249	0.184167	0.138383	0.112127
256	0.216331	0.126771	0.090319	0.073032
512	0.152656	0.081428	0.059816	0.050123
1024	0.112271	0.057275	0.04414	0.038888
2048	0.077602	0.04358	0.035345	0.030053

B. *Problem 2*

(a)

TABLE II. MISS RATIOS OF PROBLEM 2(A)

Set/associativity combination	Miss ratio
128/1	0.2842
128/2	0.1842
128/4	0.1121
2048/1	0.0776
2048/2	0.0436
2048/4	0.0301

TABLE III. SIM_ELAPSED_TIME OF PROBLEM 2(A)

Set/associativity combination	sim_elapsed_time(s)
128/1	2
128/2	2
128/4	2
2048/1	2
2048/2	2
2048/4	2

(b)

TABLE IV. MISS RATIOS(INSTRUCTION & DATA) OF PROBLEM 2(B)

Set/associativity combination	Miss ratio$_{instruction}$	Miss ratio$_{data}$
128/1	0.2842	0.2573
128/2	0.1842	0.1435
128/4	0.1121	0.0807
2048/1	0.0776	0.0579
2048/2	0.0436	0.0251
2048/4	0.0301	0.0129

C. *Problem 8*

TABLE V. STATISTICS OF INSTRUCTIONS OF PROBLEM 8

Type	Number	Percentage(%)
load	6175411	19.67
store	1979355	6.3

Type	Number	Percentage(%)
uncond branch	996935	3.18
cond branch	4001737	12.75
int computation	18241470	58.1
fp computation	0	0
trap	56	0

TABLE VI. BRANCH DIRECTION-PREDICTION RATE AND IPC OF PROB. 8

Predictor	Branch direction-prediction rate	IPC
nottaken	0.3214	0.6525
taken	0.3214	0.6599
perfect	1	1.15
bimod	0.8434	0.968
2lev	0.7514	0.9005
comb	0.8457	0.9737

IV. DISCUSSION

In this part, specific analysis of expermental data will be involved. And if you don't have much time, please jump to the conclusion part.

A. Problem 1

We define improvement as follows:

$$improvement = miss\ ratio_{old} / miss\ ratio_{new}, \quad (1)$$

Choose 4 groups to compare, compute them, and the sheet of improvement is as follows:

TABLE VII. IMPROVEMENT OF PROBLEM 1

Group of comparison	Improvement
128/1->128/4 vs. 128/1->512/1	2.53506 vs. 1.86202
512/2->512/4 vs. 1024/1->2048/1	1.62456 vs. 1.44675
128/4->512/4 vs. 512/1->2048/1	2.23704 vs. 1.96717
256/1 -> 256/4 vs. 128/2->512/2	2.96214 vs. 2.26172

We can see that for caches of equivalent size, increasing associativity(let's call it IA in order to facilitate later analysis) overwhelms increasing the number of sets(INoS) in cache, which is definitely a trend. And let's explore the issue in depth, what leads to the trend? Well, for caches of equivalent size, the essence of IA is for a single block in main memory, there are more ways in cache, while that of INoS is for a single set in cache, there are fewer mappings in main memory. And, the former is better than the latter.

We can find something explaining the phenomenon in our textbook. As is known to all:

$$misses_{overall} = misses_{compulsory} + misses_{capacity} + misses_{conflict}, \quad (2)$$

And in textbook pg. B-24, we can find the following sentence, "Compulsory misses are independent of cache size, while capacity misses decrease as capacity increases, and conflict misses decrease as associativity increases". Well, perfect answer, isn't it? The greater decrease of conflict misses is the reason why IA beats INoS. Since the other two misses level while changing, conflict misses is the decisive factor.

B. Problem 2

(a)

Table I and Table II bear a striking resemblance, which is easy to understand, in that the configuration of them are of the same.

For Table III, sim_elapsed_time is 2 seconds for all items, weird...And as I've discussed with TA in mail, the main reason contributing to it is the rapid development of processor. As a result, there's no chance for me to analyze the problem of relative speed with experimental data. But, maybe we can find the answer in textbook. From Figure B.13 in pg. B-29, we can obtain that for the same set number, average memory access time(which is closely related to sim_elapsed_time) decreases as associativity increases. However, the rule goes wrong meeting with larger set number, and in that case, there exists a maximum value of average memory access time somewhere in the middle of associativity.

Anyway, it's much easier to analyze the miss ratios. Clearly we can see that for the same set number, miss ratio decreases as the associativity increases.

(b)

The miss ratios of problem 1 is the same as that of problem 2(a), so, we choose problem 2(a) for reasons of decimal precision.

Here is the formula of miss $ratio_{seperate}$:

$$accesses_{overall} = accesses_{inst\ l1} + accesses_{data\ l1}, \quad (3)$$

$$miss\ ratio_{seperate} = accesses_{inst\ l1} / accesses_{overall} * miss\ ratio_{inst} + accesses_{data\ l1} / accesses_{overall} * miss\ ratio_{data}, \quad (4)$$

Using the formula as well as data from Table II and Table IV, we can obtain a sheet of comparison:

TABLE VIII. COMPARISON OF MISS $RATIO_{UNIFIED}$ AND MISS $RATIO_{SEPERATE}$ OF PROBLEM 2(B)

Set/associativity combination	Miss Ratio			
	unified	*instruction*	*data*	*seperate*
128/1	0.2842	0.2937	0.2573	0.2862
128/2	0.1842	0.2124	0.1435	0.1982
128/4	0.1121	0.1398	0.0807	0.1276
2048/1	0.0776	0.0867	0.0579	0.0808
2048/2	0.0436	0.0481	0.0251	0.0434

Set/associativity combination	Miss Ratio			
	unified	*instruction*	*data*	*seperate*
2048/4	0.0301	0.0377	0.0129	0.0326

I can't see any trend in the sheet...Well, unified beats separate every time, except for 2048/2 case. But what does it imply? That unified is better than seperate in miss ratios in most of cases?

This question is much weirder than the previous one. To resolve it, I asked some pals if there's any trend in their data. One of them said that unified wins everytime, and what about others? Well, the data differ from one to another, while the same point is that there's no clear trend.

And that calm me down. let the analyze begin, why unified is better than seperate in miss ratios in most of cases? Well, in my opinion, dividing the cache into two causes more capacity misses and conflict misses, and that increase the miss ratio.

Speaking of other factors which affect this choice, $accesses_{inst\ l1}$ and $accesses_{data\ l1}$ will be crucial. Think about a toy program like 'Hello World' which has few accesses, and unified/separate choice will be meaningless then.

C. Problem 8

We can easily read from Table V that 12.75 percentage of the instructions are conditional branches. And since:

$$instruction_{between} = instruction_{others} / instruction_{cond\ branch}\ , \quad (5)$$

After some simple calculation, we can obtain that the processor executes 6.845 instructions on average between each pair of conditional branch instructions.

All right, now let's describe the 6 predictors:

1) Precictor 'nottaken'

a) How the prediction is made

Always choose not to take the branch.

b) Information stored

None.

c) Amount of storage

None.

d) Relative accuracy

Rank #6 of 6.(same branch direction-prediction rate as 'taken', but lower IPC)

2) Precictor 'taken'

a) How the prediction is made

Always choose to take the branch.

b) Information stored

None.

c) Amount of storage

None.

d) Relative accuracy

Rank #5 of 6.

3) Precictor 'perfect'

a) How the prediction is made

Well, 100% of accuracy means nothing about high-class technique actually...Read the source code of sim-outorder.c, and you will know that each time the 'perfect' mistakes, it just turns back the clock. In other words, it cheats to a full score.

b) Information stored

None.

c) Amount of storage

None.

d) Relative accuracy

Rank #1 of 6.

4) Precictor 'bimod'

a) How the prediction is made

Use branch history table, which is a small memory indexed by the lower portion of the address of the branch instruction. And for bimod, 2-bit prediction schemes is applied in the table which obeys the following finite-state processor.

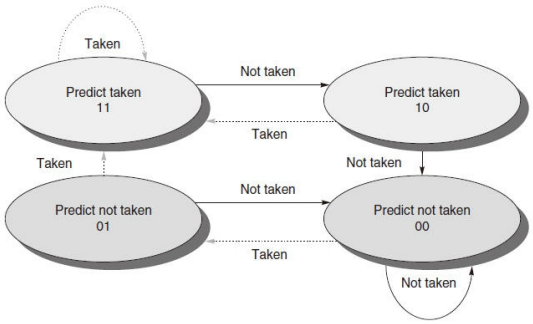

Fig. 1. The states in a 2-bit prediction scheme.

If you were confused at new terminologies, check textbook C-27 for details.

b) Information stored

2 bits for each item in table, representing the state.

c) Amount of storage

2 * <table size> bits, default value is 2048 for <table size>.

d) Relative accuracy

Rank #3 of 6.

5) Predictor '2lev'

a) How the prediction is made

Add information about the behavior of the most recent branches to decide how to predict a given branch. An (m,n) predictor uses the behavior of the last m branches to choose from 2^m branch predictors, each of which is an n-bit predictor for a single branch. As usual, check textbook Pg. 120 if you were confused.

b) Information stored

The global history of the most recent m branches can be recorded in an m-bit shift register, where each bit records whether the branch was taken or not taken. The branch-prediction buffer can then be indexed using a concatenation of the low order bits from the branch address with the m-bit global history.

c) Amount of storage

2^m * n * Number of prediction entries selected by the branch address bits. Default is 1, 1024 and 8 respectively.

d) Relative accuracy

Rank #4 of 6.

6) Predictor 'comb'

a) How the prediction is made

Named as 'comb', it is the combination of 'bimod' and '2lev', while adding a table called 'meta'. The meta table looks like bimod one. Nonetheless, it predicits whether we should apply bimod or 2lev using 2-bit predictor. The prediction method we applied is determined as the first direction, while another as the second one. When updating, if the first direction differs from the second one, we update the meta table. If not, we just need to update the table of each method.

b) Information stored

The corresponding part of 'bimod' and '2lev' + The meta table deciding bimod or 2lev.

c) Amount of storage

The corresponding part of 'bimod' and '2lev' + amount of the meta table(2 * <meta_table_size> bits, default value is 1024 for <meta_table_size>).

d) Relative accuracy

Rank #2 of 6.

It is simple to answer the question “How the prediction rate effects the processor IPC for the 'go' benchmark”: IPC grows as prediction rate grows.

Let's have a deeper look. Since the total number of instructions is of the same everytime, we can duduce that CPI, which is the reciprocal of IPC, should has a linear relation with prediction rate, which is not difficult to understand as long as you have learnt pipelining. So, let's draw a figure to prove it.

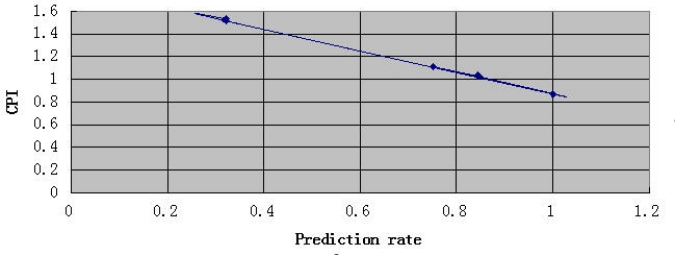

Fig. 2. Prediction rate - CPI figure.

Well, perfect proof this time.

And will the results for 'go' generalize to all programs? I don't think so, because we have only taken branch direction-prediction rate(BDpR) into the prediction-rate consideration, while ignoring the branch address-prediction rate(BApR). Therefore, if there's such a program whose BApR decreases rapidly as BDpR increases, we may find that IPC doesn't grow any more as BDpR grows.

V. Conclusion

This part is the core of the discussion part. It's for those who intend to get direct conculsions.

A. Problem 1

For caches of equivalent size, increasing associativity overwhelms increasing the number of sets in cache in miss ratio field.

B. Problem 2

(a)

For the same set number, average memory access time(which is closely related to sim_elapsed_time) decreases as associativity increases. However, the rule goes wrong meeting with larger set number, and in that case, there exists a maximum value of average memory access time somewhere in the middle of associativity.

For the same set number, miss ratio decreases as the associativity increases.

(b)

Unified cache is better than seperate one in miss ratios in most of cases.

C. Problem 8

12.75 percentage of the instructions are conditional branches.

The processor executes 6.845 instructions on average between each pair of conditional branch instructions.

For the description of 6 predictors, see previous part.

IPC grows as prediction rate grows, and its reciprocal, CPI has a linear relation with prediction rate.

The results for 'go' doesn't generalize to all programs.

Acknowledgment

Thanks to two of my pals, XU Zhuangdi and CAO Siyuan. I'm so fortunate to have such brilliant classmates to discuss academic problems with. Especially for XU, he has shed light on many subtle problems of the project for me.

Specially thanks to my dearest TA, SUN Yao, who always replies to the mail in a short time, and clearly states everything. Life is good, with so many wonderful persons around me, and it is worth fighting for.

References

John L. Hennessy, David A. Patterson: Computer Architecture, Fifth Edition: A Quantitative Approach.

Appendix

ss.build, the most rapid way to obtain Simplescalar

```
NAME=Simplescalar
PACKAGE=simplescalar
TOOL=simpletools-2v0
UTIL=simpleutils
SIM=simplesim
sudo apt-get install flex-old bison build-essential
cd
mkdir $NAME
cd $NAME
wget ftp://public.sjtu.edu.cn/simplescalar.tgz --user wtze --password public
tar xvfz $PACKAGE.tgz
export CC="gcc"
export HOST=i686-unknown-linux
export TARGET=sslittle-na-sstrix
export IDIR=~/$NAME
cd ~/$NAME
tar xvfz $TOOL.tgz
rm -rf gcc-2.6.3
cd ~/$NAME
tar xvfz $UTIL-990811.tar.gz
cd $UTIL-990811
./configure --host=$HOST --target=$TARGET --with-gnu-as --with-gnu-ld --prefix=$IDIR
make CC=gcc
sudo make install CC=gcc
cd ~/$NAME
tar xvfz $SIM-3v0d.tgz
cd $SIM-3.0
make config-pisa
make CC=gcc
cd ~/$NAME
tar xvfz gcc-2.7.2.3.ss.tar.gz
cd ~/$NAME/gcc-2.7.2.3
export PATH=$PATH:$IDIR/simpleutils-990811/sslittle-na-sstrix/bin
./configure --host=$HOST --target=$TARGET --with-gnu-as --with-gnu-ld --prefix=$IDIR
make LANGUAGES="c c++" CFLASS=-O3 CC="gcc"
sed -i 's/return \"FIXME\\n/return \"FIXME\\n\\/g' ~/$NAME/gcc-2.7.2.3/insn-output.c
make LANGUAGES="c c++" CFLASS=-O3 CC="gcc"
wget http://www.ict.kth.se/courses/IS2202/ar
wget http://www.ict.kth.se/courses/IS2202/ranlib
chmod 700 ar
chmod 700 ranlib
sudo cp ar $IDIR/sslittle-na-sstrix/bin/ar
sudo cp ranlib $IDIR/sslittle-na-sstrix/bin/ranlib
rm ar
rm ranlib
chmod +w ~/$NAME/gcc-2.7.2.3/obstack.h
sed -i 's/next_free)++/next_free++)/g' ~/$NAME/gcc-2.7.2.3/obstack.h
make LANGUAGES="c c++" CFLASS=-O3 CC="gcc"
sed -i '98i\
' ~/$NAME/gcc-2.7.2.3/libgcc2.c
cp $IDIR/gcc-2.7.2.3/patched/sys/cdefs.h $IDIR/sslittle-na-sstrix/include/sys/cdefs.h
make LANGUAGES="c c++" CFLASS=-O3 CC="gcc"
```

```
make enquire CC=gcc
sudo make install LANGUAGES="c c++" CFLASS=-O3 CC="gcc" PATH=$PATH:~/$NAME/bin
exit 0


// Ok, the follwing part are comments...Don’t care
// Authored by Feihong, sina blog
// Modified by WANG Tianze, CS.SJTU
// To use it, you need to connect to internet first
// You can find it in my ftp as mentioned previously
// And the reason why I enclose it here? Because it’s so beautiful and elegant. ^_^
```