Lab report for CA Lab2

Xiangzhe Xu

Nanjing University

April 16, 2018

This is the report of the CA Lab2. In this lab, students are required to analyze the performance of "cmove" instruction under different branch patterns and branch predictors.

1 CMOV

1.1 Brief introduction

The CMOVcc instructions check the state of one or more of the status flags in the EFLAGS egister (CF, OF, PF, SF, and ZF) and perform a move operation if the flags are in a specified state (or condition). A condition code (cc) is associated with each instruction to indicate the condition being tested for. If the condition is not satisfied, a move is not performed and execution continues with the instruction following the CMOVcc instruction. The CMOVcc instructions were introduced in P6 family processors; however, these instructions may not be supported by all IA-32 processors. [1]

1.2 Performance on modern processors

```
calab2 [master●●] % time ./ncmov
600000000
./ncmov  0.05s user 0.00s system 98% cpu 0.049 total
calab2 [master●●] % time ./cmov
600000000
./cmov  0.07s user 0.00s system 92% cpu 0.074 total
calab2 [master●●] %
```

As is shown in the figure, there is hardly any difference between the cmove version and the ncmov version, although I have changed the loop times to 10,000,000. It's reasonable to say that the cmove instruction would not affect the performance of programs running on a modern processor.

[1] Intel® 64 and IA-32 Architectures Software Developer's Manual

2 Preparation

2.1 Patch file

The patch file (also called a patch for short) is a text file that consists of a list of differences and is produced by running the related `diff` program with the original and updated file as arguments. Updating files with patch is often referred to as applying the patch or simply patching the files.

2.2 Source code analysis

```
1 #define choose(i, a, b) ({      \
2 unsigned long result;   \
3 asm("testl %1,%2 ; cmovne %3,%0" \
4     :"=r" (result)      \
5     :"i" (BIT), \
6         "g" (i),        \
7         "rm" (a),       \
8         "0" (b));       \
9 result; })
```

This is the CMOV version macro in cmov.c, which means that result is the output of this assemble sentence and is readonly. Also, the variable i should be placed in a general register and variable a should be placed in either memory or register. Finally, the variable b is the same with the 0th variable.

```
1 #define choose(i, a, b) ({
          \
2 unsigned long result;          \
3 asm("testl %1,%2 ; je 1f ; mov
    %3,%0\n1:"       \
4 :"=r" (result)          \
5 :"i" (BIT),             \
6     "g" (i),            \
7     "g" (a),            \
8     "0" (b));           \
```

```
9  result; })
10
11 #endif
```

The not-cmov version is similar with the above one.

3 WorkFlow

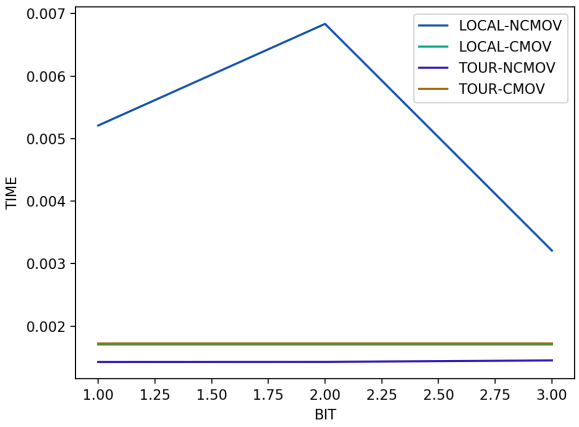
3.1 Bash script

```
1  CONFIG="--cmd=lab2/cmov1 --cpu-type=
       DerivO3CPU --caches --l2cache --
       bpred=LocalBP"
2
3  build/X86/gem5.opt configs/example/
       se.py ${CONFIG}
4
5  FILE_NAME=local_1_cmov
6
7
8  mkdir lab2/ret/${FILE_NAME}
9
10 echo ${CONFIG} > config.txt
11
12 cp m5out/stats.txt lab2/ret/${
       FILE_NAME}/stats.txt
```

This is a part of my bash script to generate the result.

3.2 Numbers

In this lab, I choose to use Numbers as my data processor for there's merely a dozen of result-sets. Writing a python script for them is time-consuming. Nevertheless, I use matplotlib to draw the diagrams.



4 Result

4.1 Time

4.1.1 Facts

In terms of running time, the branch version with local predictor consume the longest time. The performance of cmov is significantly higher than that of the branch one. In contrast, the branch version with tournament predictor even run faster than the cmov one. It's also interesting to find that the local-branch version is more sensitive to BITs than the other version.

4.1.2 Analysis