332 Advanced Computer Architecture

Exercise 5 (ASSESSED): instruction-level-parallelism in the SUSAN edge detector

This is the first of two equally-weighted assessed coursework exercises. Hand submit your solution via CATE.

Background

SUSAN is an acronym for Smallest Univalue Segment Assimilating Nucleus. The SUSAN algorithms cover image noise filtering, edge finding and corner finding. In this exercise we will study edge detection. For more information about SUSAN see

`http://www.fmrib.ox.ac.uk/~steve/susan/`

Simple edge detection algorithms work by differentiation in space, using a small local convolution filter. Unfortunately, this strategy works badly in the presence of noise, which is, of course, very common. SUSAN works on a different principle, "univalue segments": for each pixel in the image, find the contiguous, surrounding segment whose pixels that have the same value. The characteristics (eg size) of this "Smallest Univalue Segment Assimilating Nucleus" are strongly influenced by the presence of edges and corners.

Compiling and running Susan

Copy the source code directory tree to your own directory:

```
cd
cp -r /homes/phjk/ToyPrograms/ACA05/Susan ./
```

Now compile the susan.c program:

```
cd Susan
make
```

Now you can run the program:

```
./susan.x86 Ins/input_large.pgm -e > Outs/large.pgm
```

This reads input from the file `Ins/input_large.pgm`, and writes its output to the `Outs` directory. Both input and output files are in PGM (Portable Gray Map) format (see "`man pgm`".

You have been provided with a selection of input files of various sizes:

```
input_small.pgm  (76x95 pixels; from Susan distribution)
```

```
input_large.pgm  (384x288 pixels; from Susan distribution)
Cupboard.pgm     (1760x1168)
ToySoldier.pgm   (3072x2048)
```

If you need a larger one, see /homes/phjk/ToyPrograms/ACA05/Susan_ExtraImages.

Running Susan under the Simplescalar simulator

The makefile also builds a binary for execution using the SimpleScalar simulator. You can execute it by typing:

```
/homes/phjk/simplesim-3.0/sim-outorder ./susan.ss \
  Ins/input_large.pgm -e >/dev/null
```

This should take less than three minutes (80 seconds on a 3GHz Pentium 4).

What to do

Your job is to study the effect of various architectural features on the performance of the Susan application running under the SimpleScalar simulator:

1. Use sim-outorder's "-ruu:size" flag to vary the RUU size between 2 and 256 (only powers of two are valid). Plot a graph showing your results. Explain what you see. (see "Tools and Tips" below for how to automate this).
2. Examine the other sim-outorder parameters which determine the CPU's instruction issue rate (leave the cache and memory system parameters unchanged). Where is the bottleneck (when running Susan) in the default simulated architecture? Justify your answer.
3. What is the maximum number of instructions per cycle that can be achieved on this benchmark, assuming the cache and memory system is kept the same? Explore the benefits of increasing the RUU, Load-Store Queue, the number of arithmetic units and memory ports.

Write your results up in a short report (less than four pages including graphs and discussion).

Tools and tips

Use the fastest machine you can find. On a 3GHz Pentium 4 each simulation run (for problem size 1) takes less than 1.5 minutes. Use the "top" command to make sure you're not sharing it.

The most important output from the simulator is "sim_cycle" - the total number of cycles to complete the run. It's often also useful to look at "sim_IPC", the instructions per cycle - provided you always execute the same number of instructions. The time taken to perform the simulation "sim_elapsed_time" simply tells you how the simulator took.

Other outputs from the simulator can be helpful in guiding your search - eg "ruu_full", the proportion of cycles when the RUU is full.

Scripting the experiments

To do this you need to write a shell script. You might find the following Bash script useful:

```
#!/bin/sh -f
for ((x=2; x <= 128 ; x *= 2))
do
  echo -n "ruu-size" $x " "
  /homes/phjk/simplesim-3.0/sim-outorder -ruu:size $x \
     ./susan.ss Ins/input_large.pgm -e \
     2>&1 >/dev/null | grep sim_IPC
done
```

The output is delivered to the file "results". You can find this script in

```
scripts/vary_ruu
```

See also "vary_lsq".

Plotting a graph

Try using the gnuplot program. Run the script above, and save the output in a file "table". Type "gnuplot". Then, at its prompt type:

```
set logscale x 2
plot [][] 'table' using 1:4 with linespoints
```

To save the plot as a postscript file, try:

```
set term postscript eps
set output "psfile.ps"
plot [][] 'table' using 1:4 with linespoints
```

Try "help postscript", "help plot" etc for further details.

Examining the assembler code

Compile the application using the "-S" flag.

```
~phjk/simplescalar/bin/gcc -O3 -S susan.c
```

The assembler code generated by the compiler is delivered in "susan.s".

Experimenting with compiler options

The compiler accepts many parameters to control optimisation. Try "man gcc" to read about them[1]. For example, "-funroll-loops" encourages the compiler to unroll loops:

[1] Actually, this describes the gcc installed on our Linux systems, which is a more recent version. Many of the flags are the same as the somewhat older compiler we're using for SimpleScalar.

```
~phjk/simplescalar/bin/gcc -O3 -funroll-loops susan.c
```

Does this improve the performance of the simulated machine? What happens to the IPC?

Paul Kelly, Imperial College London, 2005