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. Working individually, do the exercise and write up a short report presenting and explaining your results. Submit your work in a pdf file electronically via CATE.[1] The CATE system will also indicate the deadline for this exercise.

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://users.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/ACA15/Susan ./
```

Now compile the susan.c program:

```
cd Susan
make
```

Now you can run the program:

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

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:

[1] `https://cate.doc.ic.ac.uk/`

```
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/ACA15/Susan_ExtraImages.

Although you should not need to, if you do want to clean previously compiled files type:

```
prompt> make clean
```

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 Outs/large.
```

This should take less a minute (50 seconds when I tried it). We will mostly be using an extension of simplescalar, called "wattch", that also computes the energy used by the simulated architecture:

```
prompt> /homes/phjk/simplesim-wattch/sim-outorder ./susan.ss Ins/input_large.pgm
```

Studying microarchitecture effects

Choose a Linux machine on the DoC network.[2]

Study the effect of various architectural features on the performance of the benchmark.

Vary the RUU size between 2 and 256 (only powers of two are valid). (Use a script `varyarch`.) Plot a graph showing your results (see below). Explain what you see. Now do the same but look at the total energy consumed — use the script `varyarch-energy`.

Vary the other microarchitecture parameters (leave the cache parameters unchanged). Where is the bottleneck (when running this application) in the default simulated architecture? Justify your answer.

Can you find the "sweet spot" architecture that runs this program with optimum energy efficiency? To be precise: find the simple-scalar configuration which finishes the computation in the minimum total energy. See the section below on how SimpleScalar's Wattch extension estimates the energy utilisation.

Write your results up in a short report (not more than four pages including graphs and discussion). The best solutions will be the ones which report a systematic strategy to find the optimum implementation, and which offer some insight and analysis of the results that you observe.

[2] See `https://www.doc.ic.ac.uk/csg/computers/`.

Tools and tips

The first output to look at 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.

How wattch reports energy utilisation

Wattch reports an estimate of the total energy required for the computation, tagged with "total_power_cycle_cc1" (the comment on this line is misleading - it *is* energy, not power).

Wattch is documented in this article,

`http://www.eecs.harvard.edu/~dbrooks/isca2000.pdf`

Figure 5 in the article shows the impact of Wattch's three different clock gating models - "cc1", "cc2" and "cc3". For this exercise you are invited to use "cc1" (this is what the script "varyarch-energy" reports). "cc2" and "cc3" are approximate models reflecting more optimistic/ambitious circuit-level implementations of power optimisation. "cc1" assumes that each unit is is fully on if any of its ports are accessed in that cycle. "cc2" assumes power scales linearly with port usage. "cc3" assumes ideal clock gating where the power scales linearly with port usage as in "cc2", but disabled units are entirely shut off.

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.

Paul Kelly, Imperial College London, 2015