**332 Advanced Computer Architecture**

# Exercise 5 (ASSESSED): Instruction-Level-Parallelism in Computational Chemistry

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

**Background** This exercise is about "FullDiagOdd", a simplified quantum chemistry application, given to us by Mike Bearpark in Imperial's Chemistry department. The program was written in Fortran 77 and has been automatically translated into C (using "f2c") in order to be executable under simplescalar, which only has a C compiler [1].

The methods implemented by the program are described in more detail in:

> **Excited states of conjugated hydrocarbon radicals using the molecular mechanics - valence bond (MMVB) method**. Bearpark MJ, Boggio-Pasqua M. *THEORETICAL CHEMISTRY ACCOUNTS 110 (2): 105-114 SEP 2003*. (http://dx.doi.org/10.1007/s00214-003-0461-3)

It's a very stripped-down model quantum chemistry application, which does two things:

1. For the specified number of electrons, all possible spin configurations are generated (each electron can have up or down 'spin'), together with any non-zero interactions between them.
2. These interactions are assembled into a matrix (Hamiltonian), which is diagonalised to obtain energy levels of ground and excited states of the system being modeled, along with the corresponding weighting coefficients of the individual electron configurations.

Much of a standard computational chemistry code (e.g. www.gaussian.com) is missing[2].

## Compiling and running FullDiagOdd

Copy the source code directory tree to your own directory:

```
cd
cp -r /homes/awb01/Teaching/ACA06/FullDiagOdd ./
```

Now compile the program:

```
cd FullDiagOdd
make
```

[1] A little manual tweaking was required for the "loc" function).

[2] Rather than specifying a molecular geometry and working out any interactions between atoms from first principles (ab initio), we guess numbers between 0 and 1. In a slightly different context, this approach has a long history in quantum chemistry (as Huckel Molecular Orbital theory).

Now you can run the program:

```
./mat.x86 <5a.dat
```

This reads input from the file `5a.dat`, and writes its output to screen.

You have been provided with a selection of input files of various sizes: The biggest, "11a.dat", takes a few tens of seconds to run. Use small ones with the simulator!

## Running 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 ./mat.ss \
  < 5a.dat >/dev/null
```

This should take less than two minutes (15 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 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 FullDiagOdd) 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 \
     ./mat.ss < 5a.dat 2>&1 >/dev/null | grep sim_IPC
done
```

You can find this script in `scripts/vary_ruu`.

#### 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 2:4 with linespoints
```

To save the plot as a postscript file, try:

```
set term postscript eps
set output "psfile.ps"
plot [][] 'table' using 2:4 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 -g diasym.c
```

The assembler code generated by the compiler is delivered in "diasym.s" (the "-g" flag adds debugging information including ".loc" lines that relate assembly code back to C source line numbers).

## Experimenting with compiler options

The compiler accepts many parameters to control optimisation. Try "man gcc" to read about them[3]. For example, "-funroll-loops" encourages the compiler to unroll loops. Modify the Makefile accordingly. Does this improve the performance of the simulated machine? What happens to the IPC?

*Paul Kelly and Ashley Brown, Imperial College London, 2006*

[3] 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.