LSU EE 7700-2 **Homework 2** Solution **Due: 27 February 2008**

To complete this assignment the PSE (pronounced see) visualization program will be used to analyze the execution of a SPARC v8plus program (our lighting demo). Use your class Linux accounts to complete this assignment and to avoid sluggishness, log in directly rather than remotely. (PSE is sluggish but still usable from a remote connection via Cox cable modem service, at least at my node on College Drive.)

For instructions on using PSE visit the EE 4720 procedures page,
`http://www.ece.lsu.edu/ee4720/proc.html`*, and look for the PSE instructions.*

For documentation on the SPARC instructions see
`http://www.ece.lsu.edu/ee4720/doc/JPS1-R1.0.4-Common-pub.pdf`.

Datasets, source code, and assembly code for the timing analysis versions of the demo-4-lighting.cc have been placed in `/home/classes/ee4720/com/gp/08/hw2` on the ECE Linux/Sun filesystem.

To view a dataset you could issue a command like.

```
pse /home/classes/ee4720/com/gp/08/hw2/demo-4-gcc43-4way.ds &
```

To save typing you might create a symbolic link to the dataset directory and use that:

```
cd ~                                          # Or place link elsewhere.
ln -s /home/classes/ee4720/com/gp/08/hw2      # Do this just once.
pse hw2/demo-4-gcc-4way.ds &                  # Run PSE in background.
```

All datasets are for simulations of an aggressive dynamically scheduled SPARC v8plus implementation.

`demo-4-gcc43.4way.ds`: Four-way superscalar, 64-entry reorder buffer, four floating-point units, and predict one block (branch) per cycle and fetch two non-contiguous regions per cycle.

`demo-4-gcc43.4way512rob.ds`: Four-way superscalar, 512-entry reorder, other parameters same as 4way.ds.

`demo-4-gcc43.16way.ds`: Sixteen-way superscalar, 1024-entry reorder buffer, 16 floating-point units, can predict three blocks (branches) per cycle and fetch up to four non-contiguous sets of instructions in a cycle.

The source code and assembler are in the hw2 directory.

Problem 1: Locate the loop that performs lighting calculations.

(*a*) Provide the address of an early instruction that does lighting calculations.

To find where lighting calculation execution is hover over segments in the overview plot and note the line numbers shown in the upper-right part of the window. These may show a line number in the lighting code itself, such as line `demo-4-lighting.cc:434`, or it may show a line number in a procedure called by the lighting code, such as `stl_dequeue.h:272` for the code implementing the vertex list. To tell that its lighting code click on the segment and look at the line numbers in the disassembly pane.

Segment 83 on gcc43-4way executes lighting code. The lighting loop starts with a dequeue of a vertex, the first instruction there is `0x1215c`. Note that the first lighting code instruction in file demo-4-lighting.cc is `0x129ec`.

(*b*) How many instructions are executed per vertex (for lighting, of course)?

A loop iteration handles one vertex, from `0x1215c` to `0x1215c` there are 85 instructions. (The number can be determined by looking at "Insn" in the status line at the start of two consecutive iterations.)

(*c*) What is the execution rate on the 4-way, 64-entry ROB system in vertices per cycle when there is one L1 cache miss (most iterations are like this)? (Since it will be less than one, this number is best given as a fraction, say $\frac{1}{123}$, where 123 is the time to light one vertex.)

Use decode of instruction `0x1215c` as the reference: $400,551 - 400,486 = 65$ cycles. Rate is then $\frac{1}{65}$ vertices per cycle.

(*d*) Estimate the minimum reorder buffer size needed to sustain a 4 IPC execution rate on a 4-way system when there is one cache miss. (To help answer the question look at the 512-entry ROB dataset.)

Notice that the 512 entry ROB system (demo-4-gcc43-4way-512rob.ds) executes at the maximum rate, so use that one for analysis. First, find the earliest lighting instruction `0x1215c` and then find the latest *completing* instruction; it is the `fmuls` at `0x12aec` and it completes $551 - 478 = 73$ cycles after being fetched. In that time $73 \times 4 = 292$ instructions could be fetched and so the ROB must be at least that large enough to avoid a ROB-full stall.

(*e*) Most (if not all) iterations suffer an L1 cache miss. Would prefetch help?

The 512-entry ROB system suffers no stalls, so it could not be sped up by prefetch. In fact, inserting prefetch instructions would slow down the 512-entry ROB system since there would be more instructions to fetch per vertex. However, the 64-entry ROB system suffers many cycles of stall per iteration, in part due to cache miss latency, and so prefetch instructions would help.

(*f*) Show why prefetch is easy for this loop (though not necessarily helpful). In your answer show the instructions that are currently present and how a prefetch instruction could be inserted.

To prefetch one needs to identify addresses that will be needed several iterations ahead and insert prefetch instructions using those addresses. The load at `0x1214c` misses the cache (see below). This load gets its address from the load at `0x12148`, the load at `0x12148` uses an addresses that's just incremented by 4 every cycle (see the instruction at `0x1215c`).

So to prefetch one might try the inserted instructions below, which prefetches the value needed 10 iterations ahead. For L1 misses just one iteration ahead would suffice.

```
.LLM1140  render_light+2135  demo-4-lighting.cc:474
.LLM932  render_light+1520  stl_deque.h:145
 0001215c  add  %g3, 4, %g3                                  <---- Address data
 00012160  cmp  %o5, %g3
 00012164  bpe,a,pn  %icc, +540i -> {0x129d4 .LLM1107  render_light+2062  stl_deque.h:234}
 0001216c  lduw  [ %fp - 32 ], %l4  {[0x7ffffd10]}  Dead by this insn at +85
 00012170  lduw  [ %fp - 20 ], %l1  {[0x7ffffd1c]}
 00012174  cmp  %l1, %g4
 00012178  bpne,pt  %icc, -16i -> {0x12138 .LLM924+4  render_light+1511  stl_deque.h:272}
 0001217c  mov  0, %g1  {0x0}
.LLM924+4  render_light+1511  stl_deque.h:272
 00012138  movgu  %icc, 1, %g1
 0001213c  cmp  %g1, 0
 00012140  bpe,a,pn  %icc, +23i -> {0x1219c .LLM942  render_light+1536  coord.h:176}
 00012148  lduw  [ %g3 ], %g2  {[0x1b82cc]}                  <---- Address data
 0001214c  lduw  [ %g2 + 28 ], %g1  {[0x1b875c]}             <---- Misses

 INSERTED: lduw [%g3 + 40 ], %g8    (Note: not really g8)   <---- Inserted
 INSERTED: prefetch  %g8 + 28                               <---- Inserted

 00012150  andcc  %g1, %o4, %g0
```

Problem 2: Locate the inner loop of the rasterization loop nest in demo-4-lighting and find a place in execution where that inner loop executes for a large number of iterations. Note that the inner loop body sometimes does and sometimes does not update the frame buffer.

A large part of this loop body is spent preparing the color value to be written, starting with the red, green, and blue components expressed as floating point numbers. Suggest (or search for) new

instructions that can perform the task faster. (This is a prime application of so-called multimedia instructions.)

For your answer you can make up credible instructions or choose some from a real ISA, such as SPARC VIS instructions.

The packed operand instructions would be useful for adding deltas to the RGB components:

```
! The instructions below interpolate red, green, and blue components
! (add delta_red, etc. to them).
!
00012918  fadds  %f19, %f21, %f19
00012878  fadds  %f14, %f20, %f14
0001287c  fadds  %f16, %f11, %f16

! They might be replaced by a SPARC Partitioned Add.
! To do that the red, green, and blue components would have to be
! packed into a single register, assume %f19.  The instruction below
! would do the work of the three above.

fpadd16 %f19, %f21, %f19
```

A large number of instructions are needed to take the FP red, green, and blue values and pack them into an integer. The single SPARC fppack16 instruction would replace them, this would have a larger impact than the fpadd16 instruction.

Problem 3: Continuing to look at the rasterization part of execution, locate the branch mispredictions.

(*a*) Estimate the impact of branch mispredictions on the three systems. Do so by examining the ROB plot. In your answer provide the segment numbers.

Look at segment 58 in demo-4-gcc43-4way.ds. Switch to the ROB plot (press the PED button) and examine the purple stripe along the top. The light-purple shows decodes of instructions that are later squashed due to branch misprediction. The relative area of these regions and the white areas that follow (part of the branch penalty), is roughly 15%.

(*b*) Some of these branches resolve early, some resolve late. What part of the code is responsible for the late-resolving branches?

One late resolving branch is for the Z test, `0x12898`.

(*c*) Could those late-resolving branches be avoided? (Look at the code from the `clampi` routine.)

The clampi routine uses conditional moves to avoid the need for a branch. Instead of using branch `0x12898` conditional moves could be used for updating the frame buffer: a register might hold the old pixel value, if the condition were true the new pixel value would overwrite it. Alot of code is executed to prepare a pixel for writing, so if a conditional move were used that code would always be executed. Looking at the execution though, time would still be saved because by the time the branch resolves code has advanced to the next iteration.