C152 Laboratory Exercise 4

Professor: Krste Asanovic
TA: Andrew Waterman
Department of Electrical Engineering & Computer Science
University of California, Berkeley

March 30, 2010

1 Introduction and goals

The goal of this laboratory assignment is to allow you to conduct some simple virtual experiments in the Simics simulation environment. Using Simics models of VLIW and multithreaded machines, you will collect statistics and make some architectural recommendations based on the results.

The lab has two sections, a directed portion and an open–ended portion. Everyone will do the directed portion the same way, and grades will be assigned based on correctness. The open–ended portion will allow you to pursue more creative investigations, and your grade will be based on the effort made to complete the task or the arguments you provide in support of your ideas.

Students are encouraged to discuss solutions to the lab assignments with other students, but must run through the directed portion of the lab by themselves and turn in their own lab report. For the open-ended portion of each lab, students can work individually or in groups of two or three. Any open-ended lab assignment completed as a group should be written up and handed in separately. Students are free to take part in different groups for different lab assignments.

You are only required to do one of the open ended assignments. These assignments are in general starting points or suggestions. Alternatively, you can propose and complete your own open ended project as long as it is sufficiently rigorous. If you feel uncertain about the rigor of a proposal, feel free to consult the TA or professor.

This lab assumes you have completed the earlier laboratory assignments. However, we will re-include all the relevant files from past labs in this lab's distribution bundle for your convenience. Furthermore, we will assume that you remember all the commands used in earlier labs for controlling Simics simulation. If you feel any confusion about these points, feel free to consult the first lab guide or the Simics User Guide.

1.1 Overview of the new machines

For this lab you will make use of two new target machines that Simics is capable of simulating. These machines are not as fully featured as the machines we have been using in previous sections, making complicated studies more difficult to conduct. However, this lab will serve as an introduction to the machines

The new first machine is an Intel Itanium processor called Vasa running Red Hat Linux. This processor uses the IA-64 architecture (VLIW). It is dependent on the compiler to statically exploit ILP by packaging instructions into 128-bit bundles.

The second new machine is a Sun Microsystems UltraSPARC T1 processor called Niagara-Simple running Solaris. This processor uses the SPARCv9 architecture. It has 8 cores, each of which has 4 hardware thread contexts that appear to the OS as separate cores. This machine employs fine–grained multithreading, meaning that each core switches between one of the available threads on every cycle. Available configurations of this machine in Simics have 1 (1x1), 2 (2x1), or 32 (8x4) thread contexts.

1.2 Graded Items

You will turn a hard copy of your results to the professor or TA. Please label each section of the results clearly. Make sure you name your open–ended section partners in your individual portion, and that the group results name you as a participating member. The following items need to be turned in for evaluation:

1. Problem 2.1: VLIW statistics for each benchmark and answers
2. Problem 2.2: Loop unrolling statistics and answers
3. Problem 2.3: Niagara statistics and answers
4. Problem 3.1/3.2 statistics and evaluations (include source code if required)

2 Directed Portion

2.1 Exploring VLIW assembly

The Intel Itanium processor target that we will use to study VLIW performance uses the IA-64 instruction set. The format of this bundle is displayed in Figure 1. Each bundle contains 3 41-bit instructions and a 5-bit template field that specifies the grouping of this bundle with adjacent bundles. Groups contain instructions that can execute in parallel. The first instruction in a bundle is said to be in slot 0 of the bundle, the second instruction in slot 1, and the last instruction in slot 2. The Simics `step-*` commands execute one slot at a time from a bundle.

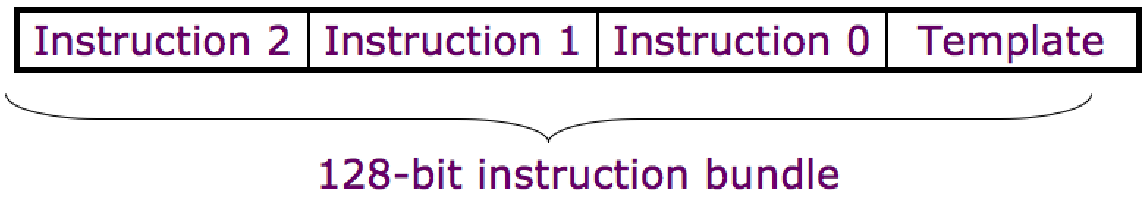

Figure 1: IA-64 instruction bundle

Simics is normally set up to process and report one instruction per processor at a time, so the way it disassembles instruction bundles is worth explaining. Since individual instructions do not have well-defined addresses, Simics uses the encoding scheme (bundle address + slot number) when dissassembling instructions. Since bundle addresses are always 16-byte aligned, the lower 4 bits in the bundle address are always zero and can be used to encode the slot number. For example, the following instructions are in the three slots of the bundle located at address `0xe000000004497f30`:

```
[cpu0] v:0xe000000004497f30 p:0x0000000004497f30        ld8.acq r14 = [r15] ;;
[cpu0] v:0xe000000004497f31 p:0x0000000004497f31        cmp.eq p6, p7 = 0, r14
[cpu0] v:0xe000000004497f32 p:0x0000000004497f32        nop.i 0x0
```

The Linux disk image used by Vasa is rather sparse in terms of available features. It lacks the software that normally allows us to mount the host machine's file system on the target machine's file system. This means the the only way to transfer files between the machines is to create ISO images and mount them on the target machine's simulated cdrom drive.

More importantly, there is no compiler installed on the simulated machine. Since CS152 students don't have access to the department's Itanium cluster, we could not compile IA-64 binaries from source even if we could copy over the source files. For these reasons, in this version of the lab we will only ask you to examine code from several programs already installed on the target machine.

First, boot up the machine: `host$ ./simics targets/ia64-460gx/vasa-common.simics`
The machine will take a few minutes to boot up. You may create a checkpoint if you wish, though you will probably not need it. Sometimes the X11 terminal of the simulated machine goes blank, but nothing is wrong and if you type the screen will refresh.

Run the following three commands one at a time on the target machine. For each command, once it begins executing quickly return to the Simics command line and pause execution with `control-C`. While this method is imprecise, it is acceptable for the analysis you will do in this section. The commands to run the mini-benchmarks are:

```
target# bzip2 /var/log/dmesg
target# yes
target# grep Swap /var/log/dmesg
```

Step through the code 30 'instructions' (i.e. 10 bundles) at a time with the command `simics> si 30`. Do this for at least 50 bundles. For each of the mini-benchmarks, report on:

1. The incidence of bundles that fill all/two/one/no slots with useful instructions.
2. The theoretical IPC of this machine assuming two bundles are fetched and executed per cycle (i.e. the way a real Itanium operates).
3. Whether there seems to be any correlation between the presence of noops in the bundles and the presence of any other instruction types. Give examples of code sequences you observe.

2.2 Investigating the effects of loop unrolling

In this section you will investigate the compiler's effectiveness at loop unrolling. We are unable to use the VLIW Itanium machine since it lacks a compiler, so instead you will compile and run code on the Bagle machine we have used in previous labs.

1. The code in the file `loop_unrolling.c` is a simple scalar–vector addition loop. After booting the machine, copy the source code into the target machine's file system. Compile the code:
   ```
   target#gcc -I/host/share/instsww/pkg/virtutech/simics-3.0.30/src/include
    -O3 loop_unrolling.c -o unroll
   ```
2. Enable magic break points and begin executing the loop benchmark.

3. When Simics breaks, investigate the assembly code as it runs using `si`. Is the loop unrolled? Provide the source code of a single loop iteration.

4. Turn on the loop-unrolling optimization and recompile.
```
target#gcc -I/host/share/instsww/pkg/virtutech/simics-3.0.30/src/include
 -funroll-all-loops -O3 loop_unrolling.c -o unroll_opt
```

5. Execute the new binary you made. When Simics breaks, investigate the assembly code as it runs using `si`. Is the loop unrolled? By how far? Provide the source code of a single loop iteration.

If you need help understanding what is going on with these codes you can use the command
```
target# gcc -c -g -Wa,-ahl,-L -funroll-all-loops -O3
-I/host/share/instsww/pkg/virtutech/simics-3.0.30/src/include loop_unrolling.c
```
to produce output which is the assembly code with comments that are the original source code. Redirect the command's output to a file to save it.

2.3 Running code on a Sun Niagara T1 processor

In this section you will run code on a simulated UltraSPARC T1 (Niagara) processor. As described in the introduction, this processor has 8 cores on a single chip. Each of these cores has 4 hardware thread contexts. On a given cycle, each core will select one of the available threads and allow it to issue instructions. Solaris treats each of the hardware thread contexts as a separate CPU.

In this lab section you will boot up the Niagara machine and create checkpoints that will be useful to you in both this lab and the final lab (6). You will examine some multiprogrammed workloads running on different configurations of the machine. Perform the following procedure for each of the machine configurations (1, 2, or 32 hardware thread contexts):

1. Start Simics.
```
host$ ./simics
```

2. Set the number of CPUs. This will be 1, 2 or 32.
```
simics> $num_cpus = 32
```

3. Run the configure script and let the machine boot. This will take additional time for the machine with 32 contexts.
```
simics> run-command-file targets/niagara-simple/
niagara-simple-solaris-common.simics
```

4. Load the host machine's file system (`mount /host`) and copy in the Solaris binaries of the benchmark programs (`bzip2_solaris`). Copy in the input files as well.

5. Create a checkpoint for this machine configuration. If you run out of disk space on your user account, inform the TA so instructional computing can raise the disk quotas.

6. Run multiple copies of a single benchmark. Remember you can run processes in the background with &.
```
target# bzip2_solaris input.jpg &
```
Does the OS run the different programs on the different cores? Check using `prstat`.

How does having more processes running affect execution time? How does this effect change when more cores are used? Check using `time` (pipe the output of multiple times running in the background to a file), e.g.:

```
target# time bzip2_solaris input.jpg >> output &
```

2.3.1 Pro Tip

By default the Niagara target machine does not use the `bash` shell and thus there is no tab-completion. In the target machine simply launch it and it should work fine: `target# bash`

3 Open-ended Portion

3.1 Design space exploration within the Niagara processor

In this section you will attach a memory hierarchy to the Niagara-Simple machine and explore the interactions between multiprogramming and the memory hierarchy. Your goal is to explore the relationship between the size and configuration of the cache, the latencies of the cache and memory, and the number of programs running on the machine. You can do this by measuring the number of instructions completed per CPU thread in a given number of cycles, and also by examining the statistics recorded by the cache.

Use the 32 context machine checkpoint. Start Simics in `-stall mode` to record cache statistics. Determine a workload of interest and move to an appropriate point in its execution. Use the `run-command-file` command to add the cache `add-niagara-cache.simics`. This script adds a single level cache shared between all the cores and threads, as well as a memory stall unit. The initial values are set to a 3 cycle cache hit time and 10 cycle memory latency. The cache is by default 512KB. You can change these values by modifying the `add-niagara-cache.simics` file.

Refer back to the Lab 2 guide if you need a reminder of how to attach, reset, and measure cache statistics.

This project is purposefully being left very open ended to allow you to demonstrate your ability to explore a design space an report on the performance tradeoffs inherent to it. State the conclusions you reach about chip multiprocessor cache performance and provide evidence in support of your claims.

3.2 Create optimized vector–vector addition code

The goal of this open-ended assignment is to create optimized vector-vector addition code. You are provided with a initial implementation in the file `vector_addition.c`. Your task is to optimize the code to improve its performance as much as possible. Use whatever optimizations occur to you (i.e. loop unrolling is where you should start, not where you should stop). You should use basic compiler optimizations (i.e. use the `gcc` flag `-O3`), but all further optimizations must be made in your C code (e.g. don't use `-funroll-all-loops`). You cannot make any assumptions about the length of the vector in advance; it will be passed in as a command line parameter.

The effectiveness of your code will be judged by the number of instructions it takes to execute on the following input vector sizes: 7, 33, 128, 225, and 1024 elements. Report the number of instructions executed between the two magic breakpoints for each of these sizes, using the bagle-common machine. Submit your C code and an explanation of the optimizations you have implemented.