CS/ECE 757 Course Wiki Main » Homework 3

CS/ECE 757 Parallel Computer Architecture (Spring 2016)

Homework 3

Due Monday 2/29/2016

Point-of-contact Swapnil Haria <swapnilh@cs.wisc.edu> (for questions and handin)

You should do this assignment groups of two. No late assignments.

NOTES:

- Get gem5 working early! We have tried to minimize the hassle of setting up gem5, but it is somewhat finicky and while your **point-of-contact** (see top of page) is happy to answer questions, he probably won't be checking email at 2 am the night before the assignment is due.

- **You can do this on the galapagos machines.** Keep in mind that compiling and running gem5 is resource-intensive and may make other programs on the machine slow. If you log in remotely, run "who" and try to pick a machine no user is locally logged in on, or you may make enemies. You could also simply ssh into best-galapagos or best-adelie to be automatically assigned the least loaded machine.

Instructions

Directory Structure

For ease of running these instructions, we assume that you will set up your hw3 directory as follows:

gem5/binaries/ gem5/disks/ gem5/ m5threads/ programs/

That is, we expect that your copy of gem5 will be in parallel with these other directories. We'll explain where to get each one. You will have to create the 'binaries' and 'disks' directories yourself inside the gem5 folder, as well as 'programs' inside the top hw3 directory.

gem5 setup

Check out a copy of gem5 from the repository, then update to the version used in this tutorial.

```
hg clone http://repo.gem5.org/gem5

cd gem5

hg update 4162427127e9
```

gem5 uses scons to manage compiling (it's similar to make). When you run this it will ask to add a line about the gem5 style hook to your .hg config file; you can just hit enter. We'll compile for x86 and with a MOESI directory protocol.

```
scons -j9 build/X86/gem5.fast PROTOCOL=MOESI_CMP_directory
```

This will take a while and print out a lot of messages.

You now have a binary in build/X86 called 'gem5.fast'. You can see the arguments you can pass it by running it.

SE Mode

Make sure that gem5 works in Syscall Emulation (SE) mode by running hello world with the se.py config script.

```
build/X86/gem5.fast configs/example/se.py --ruby -c tests/test-
progs/hello/bin/x86/linux/hello
```

This should finish quickly and print out "Hello world!" as well as some other information. It will also generate a directory called m5out, which contains the configuration and stats from running the test. stats.txt contains general statistics as well as more specific ones about the coherence protocol. If you want to change some settings (such as output directory) take a look at the options for the config script:

```
build/X86/gem5.fast configs/example/se.py -h
```

Now let's run a multithreaded program in SE mode. Because we don't have an operating system, we need to use a special version of pthreads, called m5threads. Check out the m5threads repository parallel to the gem5 repo, and then build the library.

```
cd ..

hg clone http://repo.gem5.org/m5threads

cd m5threads

make
```

We'll first run an example pthreads program. Copy `/p/course/cs757-markhill/public/hw3/eg_pthread.c` to a directory called programs, then compile it statically. If you take a look at eg_pthread.c, you'll notice it has some special calls in it to functions called m5_*. These are defined in gem5/util/m5/m5op.h.

a. m5_reset_stats: resets the stats of the system

b. m5_work_begin: This specifies where a work unit starts

c. m5_work_end: This specifies where the work unit ends

These instructions specify where to start recording the stats from and defining a work unit, in case you want to reduce the number of work units to execute on gem5 without changing the program.

Since eg_pthread.c uses m5threads and also these special simulator functions, you'll need to tell the compiler where to find them.

```
cd ../programs

gcc eg_pthread.c -I../gem5/util/m5/ ../gem5/util/m5/m5op_x86.S -static
../m5threads/libpthread.a -o eg_pthread
```

Now, run the program. Note that you need to specify the number of cores to gem5, and that it is one more than the number of threads. -c specifies the command to run, and -o specifies any arguments to send to the program.

```
cd ../gem5
```

```
build/X86/gem5.fast configs/example/se.py --ruby -n 3 -c ../programs/eg_pthread -o "2"
```

Now that you have this working for the example pthread program, you should be able to perform the same set of steps on Ocean, as required for Problem 2.

Full System Mode

You will be making your own simulation scripts for this part of the assignment, instead of using the example 'fs.py'. Go through the third part of this tutorial - http://pages.cs.wisc.edu/~markhill/cs757/Spring2016/learning_gem5/part3/

Note that this tutorial is still a work in progress. Please bring any errors/mistakes to our attention at the earliest, by sending a mail to powerjg@cs.wisc.edu and CC swapnilh@cs.wisc.edu

The default scripts provided by the tutorial are only configured to work with a single core. Modify run.py to add an option to specify number of simulated cores, and modify system.py to support a multi-core simulation. Hint: Look at how a single core is instantiated and hooked up in the default script, and replicate that in all the appropriate locations.

gem5 can also boot and run a full-fledged operating system -- for our purposes, Linux. To do this, it needs the operating system (a copy of the kernel) and disk images. A copy of these can be found in `/p/course/cs757-markhill/public/hw3`. Because the disk file is large, you may want to copy it to /tmp, and then add a symbolic link to it from your gem5 installation. You will need to decompress the images using gunzip.

You will need to tell gem5 where to find the disks and the kernel binaries. Modify the scripts from the tutorial to do this.

You will need to edit the disk file x86root.img so that it contains the binary of your programs. To do this, we will mount the img as a filesystem. As we don't have root access, we will use a program called guestmount.

```
guestmount -a x86root.img -i /tmp/my_mountpoint
```

You will then be able to 'ls /tmp/my_mountpoint' to see the contents of the file system, and interact with it as normal. Copy your compiled program binaries onto the disk, then unmount it:

```
fusermount -u /tmp/my_mountpoint
```

To start Linux, use run.py as explained in the tutorial.

This will run indefinitely, until you stop it. To interact with linux running in the simulator, you will need to connect to it. This can be done with telnet.

```
telnet localhost 3456
```

Note that the port number will be listed after you start gem5. Also note that since Linux is running on the simulator, it will be SLOW. Running 'ls' may take multiple seconds.

To run the programs for this assignment, you need to start them through an rcS script. An rcS script is simply a shell script containing the commands you would like to run. It can be passed to the simulator using the --script flag. In either case, please remember that you do not want to include the bootup time in your results. Use '/sbin/m5 resetstats' after you boot but before running the program, and '/sbin/m5 exit' to end the simulation. See gem5/configs/boot/ for example scripts.

Problem 1 (10 points)

Simulate the 'eg_pthread' workload provided with this assignment with SE mode. Plot the speedup of the workload with varying number of threads t = [1, 2, 4, 8] relative to the t=1 case. You can look at sim_ticks in the stats.txt file to determine execution time.

Problem 2 (15 points)

Modify your pthread Ocean program from homework 2 in order to simulate it with gem5 in SE mode. Specifically, you need to instrument your Ocean program to record stats of the parallel phase of simulation. You will only simulate and time the parallel phase of Ocean using Ruby. See how eg_pthread was instrumented. Note: The parallel phase of the simulation does not include the creation of threads, it only includes the time it takes the threads to process their part of the ocean. Please be sure to only record stats from that part. Since simulation takes a long time, you will simulate a smaller sized ocean for this problem. Run your program on a 258x258 ocean for 50 iterations (or smaller with fewer iterations). Do not let them run for more than an hour or two for 8 threads. Again, plot the speedup of the workload with varying number of threads t = [1, 2, 4, 8] normalized to the t=1 case.

Problem 3 (10 points)

Simulate the 'eg_pthread' workload provided to you in the disk image with FS mode. Plot the speedup of the workload with varying number of threads t = [1, 2, 4, 8] relative to the t=1 case. Remember that you need to run with one CPU per thread.

Problem 4 (15 points)

Now run your instrumented Ocean in FS mode and plot the results as above.

What to hand in

Please turn your answers to the discussion questions as a printed sheet at the beginning of the lecture.

- Plots of speedup for each problem, above
- Your modified Ocean code
- A patch file showing the changes made to the FS scripts to support multi-core simulation
- The rcs script used to run the ocean binary in FS simulation (last three files also as a tarball, by email with subject exactly 'CS/ECE757 HW3' to swapnilh@cs.wisc.edu)
- Answers to the following questions:

  1.) Do your speedup numbers for ocean obtained with the simulator match the speedup numbers obtained in homework 2? Why or why not? Does the speedup *trend* observed with the simulator match the speedup trend from homework 2? Why or why not?

  2.) What is the difference between trace-driven simulation and execution-driven simulation? What are the advantages and disadvantages of trace-driven simulation? What are the advantages of execution-driven simulation? What kind of simulation did you do when you simulated eg_phtreads and ocean?

  3.) Did you observe any difference in speedups between SE and FS mode? Why or why not?

Page last modified on February 19, 2016, at 03:36 PM, visited times