

CS 6290: High-Performance Computer Architecture

Fall 2007

Class Project Report Due: December 6th before class

Part 1 (Everyone):

For this project you will need an updated version of the simulator. Download `sesc-20071016.tar.bz2` and then install and configure it like in Homework 3. Unlike Homework 3, where we used a single-threaded application, in this project we will use a multi-threaded application running on a multi-core system. The application you will need is `water-sp.mipseb` and the input files it uses are `random.in` and `input`. We will assume you have created a `water-sp` directory within the `~/sim` directory (see Homework 3), put the application and its input files there, and that you are running the simulations from within that `water-sp` directory.

To simulate execution of this application using only one processor, we can use the following command line:

```
../runsim/sesc.smp -iinput -c../runsim/sesc.conf \  
water-sp.mipseb -p 1 -n 512
```

The `-p 1` option tells the application to only use one processor, i.e. the application will only use one thread for all its work. To use more threads, simply specify a larger number with the “`-p`” option.

If you have tried to run the simulation using the command line specified above, you must have noticed that this simulation takes tens of minutes. This project requires you to change the simulator and you will likely need to do many simulation runs to debug your changes. Before we continue with this project description, it is time to learn a few very useful simulator options. The `-w <skip>` option tells the simulator to fast-forward the simulation to the point where the simulated machine has executed a number of instructions that is equal to `<skip>`. The `-y <insts>` option tells the simulator to only simulate `<insts>` instructions in full simulation and then end the simulation and generate the report file. Fast-forwarding is **much** faster than full simulation, but does not collect any statistics. The idea is to use the `-w` option to skip initialization and other “boring” parts of the application’s execution, then simulate a number of instructions in full simulation, then skip the rest of the execution. For the statistics to be useful, we need hundreds of millions of fully simulated instructions. So let’s skip the first 100 million instructions and then get statistics for the next 200 million:

```
../runsim/sesc.smp -iinput -c../runsim/sesc.conf \  
-w 100000000 -y 200000000 water-sp.mipseb -p 1 -n 512
```

Unfortunately, we can not use these two options to compare execution times of runs that use different parameters – we only know the execution time for the instructions that were fully simulated. As a result, execution times for runs that execute different numbers of instructions can only be fairly compared if the entire execution is fully simulated. In other words, you can use `-w` and `-y` options when debugging, but the results you report should be obtained using full simulation of the entire execution of the application.

In your project report, you should compare the execution time, of 1-thread, 2-thread, and 4-thread, 8-thread, 16-thread, and 32-thread execution of this application. You will notice that with more threads we have a shorter execution time, but we also have more instructions executed and a lower IPC. As a result, our **parallel efficiency** decreases as we add more threads (parallel efficiency is equal to the parallel speedup divided by the number of processors used). Before you run any parallel simulations, you should change the `NoMigration` parameter in the configuration file to be “false”. With `NoMigration` set to true, a thread that begins running on one processor can never use any other processor, so you can have several threads being multiplexed onto a processor although other processors in the system are idle.

One of the main reasons for the IPC becoming lower is that there are additional cache misses. The simulator is already counting cache misses for each cache. You should add code to the simulator to identify, for each cache miss in L2 caches, what kind of miss it is (compulsory, capacity, conflict, and coherence) and count each kind of misses separately. In your project report, you should provide the total number of L2 misses for 1-, 2-, 4-, 8-, 16-, and 32-threaded execution, as well as the breakdown of these misses into the four kinds (i.e. what percentage of all L2 misses are compulsory misses, what percentage of all misses are capacity misses, etc). Explain how and why the number of misses and their breakdown changes with the number of threads. The four kinds of misses are defined in the textbook, but here are a few helpful hints and simplifications:

- A miss is a compulsory miss if that block has never before been in the particular cache that is having a miss.
- A miss is a coherence miss if the block has been invalidated from the cache and not yet been replaced by another block. This means that there is still a matching tag in the cache, but the corresponding valid bit is not set. Note that the default SESC simulator does not keep a separate valid bit, it uses a tag of 0 to indicate an invalid block. Since you need the tag to remain intact after invalidations, you will have to find a way to either add a separate valid bit or do something else that would allow you to check the original (pre-invalidation) tag.
- A miss is a capacity miss if it is not a compulsory miss and it would be a miss even in a fully associative cache of the same size. You need to use a LRU stack or some other way to track which blocks would be in a fully associative cache at the time of the miss.
- A miss is a conflict miss if it does not belong to any of the above categories.
- You can choose to use a more precise definition of the four miss categories, but if you do so please explain in the report which definition you used and why is it better (more precise) than the one outlined above.

Part 2 (2-student teams):

You can do the project individually (no collaboration with other students), in which case you only need to do the Part 1 of the project. You may instead choose to do the project together with another student, in which case your 2-student team will also need to complete the second part of the project.

The second part of the project is to determine what percentage of overall execution time is attributable to cache misses, and what percentage of overall execution time is attributable to each of the 4 kinds of misses. For this, you will need to complete Part 1 of the project (for each cache miss identify what kind of miss it is), then determine how much stall time that miss is responsible for, and show the total stall time attributable to each kind of misses as a percentage of the overall execution time. Show and discuss your results in the report.

Hint: SESC uses fetch slots to attribute time to various stall causes. For example, if the processor is capable of fetching four instructions in a cycle, and in a particular cycle it fetches one instruction that ends up committing, then $\frac{1}{4}$ of a cycle (1 out of 4 fetch slots) is counted as “busy” (useful). If the ROB was full and that was what prevented the other three fetch slots from being used, then $\frac{3}{4}$ of a cycle is counted as a stall due to a “ROB full” cause, and other resources are treated the same way. When there is a stall for a given resource (e.g. ROB or load queue), you can check whether the first instruction that should leave the resource (e.g. the oldest instruction in the ROB, or the oldest load in the load queue) is still there because of a cache miss. If it is, the stall time can be attributed to that cache miss. As in Part 1, you can choose to use a better (more precise) way of “blaming” a cache miss for stalls, but if you do you should explain in your report how your “blaming” scheme works and why it is better than the one outlined above.

Format of the report:

The report should have 6 pages, using two-column IEEE/ACM conference proceedings format with 9-point font size. You can use an additional page to list the literature you used and cited in your report. When showing your results, charts are preferable to tables-of-numbers because 1) they make it easier for us to read and 2) they make it easier for you to interpret results and explain them. Pay particular attention to identifying and explaining any trends and unexpected results. For example, suppose that a particular kind of cache misses should occur more often with more processors. If for 4 processors we have fewer of that kind of misses than with 2 processors, you should point this out and briefly describe why it is happening (if you know) or why it might be happening (if you don't know the actual reason). Since you have a limited number of pages for your report, you should focus on presenting your results, interpretation of these results, and assumptions, rather than introductory and background material.