IS2202 Lab 1: *Memory hierarchy, Coherence, Multithreading and Memory Models*

Professor: Mats Brorsson
TA: Ananya Muddukrishna, Georgios Varisteas

April 10, 2014

1 Goals

The goal of this lab is to allow you to conduct some memory hierarchy experiments using the Simics simulation environment. Using the Simics *g-cache* memory hierarchy simulator module and a Simics UltraSparc T1 processor model, you will collect execution statistics of some benchmarks and make architectural conclusions based on the results. You will also make experiments on the importance of memory consistency models and synchronization primitives.

The lab consists of three parts: (i) cache simulation with Simics, (ii) Multithreading simulation with Simics, and (iii) Experiments with memory ordering on x86-64-platforms. Each part consists of a directed section and an open ended section. Everyone will do the directed section 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. You can work alone or in a group of two. Submit one report per group which contains your solutions for the directed portion and any open-ended portion which you have explored. Both students need to be identified in the report if you have worked in a group.

2 Introduction to Simics

The first two parts of this laboratory exercise will be done using a full-system computer simulator called Simics. The interface presented by Simics to the OS and application programs make the target machine look like a normal machine.

2.1 Interaction with Simics

Host machine refers to the remote server on which you are running Simics. *Target* machine refers to the simulated machine running inside of Simics. *Workspace* refers to the directory in which you create as your Simics workspace.

In this lab exercise, what you type on the host's command line will be referred to like this:

```
host$ this is what you type
```

What you type on the Simics command line will be referred to like this:

```
simics> this is what you type
```

What you type on the target machine's command line will be referred to like this:

```
target# this is what you type
```

2.2 Simics first steps

For this lab exercise, you can use any of these servers as the host machine:

```
[avril|columbiana|atlantis|subway|shell|malavita]@it.kth.se
```

Login to these servers using your KTH.SE credentials using the following command:

```
ssh -X server_name -l your_KTH.SE_login_name
```

Tip: If you use a Windows OS, you will need an X server environment such as XMing. Another quick and easy way to setup an X environment is first install VirtualBox (http://www.virtualbox.org/) on Windows and then install Ubuntu linux on VirtualBox.

Once logged into the host, set up the Simics license environment by defining the environment variable `LM_LICENSE_FILE`.

```
host$ export LM_LICENSE_FILE=27005@lic03.ug.kth.se
```

Create a Simics workspace in your AFS home directory. To create a Simics workspace called `simics-workspace-lab1` in your home directory, run:

```
host$ /pkg/simics/3.0.31/simics-3.0.31/bin/workspace-setup
~/simics-workspace-lab1
```

Note: If you do not have enough space in your home directory, we suggest that you create your workspace in the /nobackup directory of the server. First create a directory having the same name as your login name in /nobackup and then

create the simics workspace within it. Remember that this directory is not backed up and is periodically cleaned up by system scripts. So move important data such as simulation results and modified simulation scripts into your AFS home.

Now that your workspace has been created, you are ready to begin simulation. Navigate to your workspace directory and load the Sun UltraSparc T1 machine called `niagara-simple`.

```
host$ cd ~/simics-workspace
host$ ./simics targets/niagara-simple/niagara-simple-solaris
-common.simics
```

Simics will start up and present you with a command line, and a new terminal window should also appear. The new window is the terminal of the target machine being simulated by Simics. The simulation begins paused. Type,

```
simics> continue
```

to begin the simulation and watch the machine boot up. After a short while you should be logged in to the target machine as root. This simulated machine is configured to act as an uniprocessor by default and is running the Solaris 9 OS. Type `bash` on the target and run some basic Solaris commands that you know. An interesting Solaris command is `psrinfo` which gives the number of active processors in the niagara-simple machine.

Create a directory called lab-1 on the target.

```
target# mkdir -p /home/lab-1
```

Pressing `Ctrl-C` in the terminal with the Simics command line interface will pause the simulation and make the `simics>` prompt reappear. While the simulation is paused, you will not be able to interact with the simulated machine via its terminal. Remember to restart the simulation with `continue` before attempting to execute commands on the simulated target.

Checkpointing

Once the machine has booted, it would be best to save the state of the simulation so that we can resume from this point the next time we start Simics, without having to sit and watch the machine boot up again. This process is called *checkpointing*, and it will be very useful to you at times when you wish to run from a certain single point multiple times to collect different types of data. To create a checkpoint called *after_boot.conf*, simply pause the simulation and run,

```
simics> write-configuration after_boot.conf
```

The checkpoint is now saved. To use the checkpoint in the future, simply start Simics with the `-c` flag, like this:

```
host$ ./simics -c after_boot.conf
```

One important thing to remember is that Simics checkpoints only save the differences between the current checkpoint and the previous checkpoint. This is good because it saves lots of space. The problem is that if you create one checkpoint, keep running, create a second checkpoint, and then go and delete the first checkpoint, then the second one will not work any more. Keep this in mind when creating and deleting checkpoints.

Mounting the host filesystem

The next thing to be done is to mount the file system of the host machine on the target machine. This will allow you to access any files you have stored on the host machine inside the simulation you are running. To mount the host file system, all you have to do is type

```
target# mount /host
```

This will mount the `/` directory of the host machine on the `/host` directory of the target machine. This command mounts the host machine's file system as read only. Mounting with write permissions is possible but experimental, and will not be required for this lab. Now you can just use `cp` to copy files from the `/host` directory tree into `/root` or `/home` or wherever you like inside the simulated machine. Remember that these files will be lost forever if you quit Simics without checkpointing (though of course you could re-add them in a new simulation if necessary).

2.2.1 Simulation commands

The Simics simulation can be controlled with a variety of simple commands. You have already learned how to run and suspend the simulation with `Ctrl-C` and `continue`. Other useful commands include instructions to step forward a certain number of instructions or cycles:

```
simics> step-instruction 100_000
simics> step-cycle 100_000
```

Long numbers can include _ separators for clarity. Many instructions also have abbreviated short forms, such as `si` for `step-instruction` or `c` for `continue`. `continue` can also be followed by an integer argument to advance the simulation by that many instructions without printing the intermediate ones.

Other useful commands print information about the current state of the simulated system. `pregs` and `pfregs` display the contents of all the machine registers or floating point registers.`read-reg` and `write-reg` can be used to view or modify the contents of specific registers, and `get` and `set` will do the same for memory locations. `ptime` will display how much simulated time has elapsed over the course of the simulation, and `pstats` will report statistics for the individual processor itself.

The command display can be used to cause one of the other commands to be

executed whenever the simulation is suspended. For example,

```
simics> display ptime
```

will cause the `ptime` command to be run every time the simulation is paused. Keep track of the display id reported when a display command is first executed, as this is how you will specify which one to turn off when you are ready to stop displaying a certain command:

```
simics> undisplay 1
```

Finally, adding a `!` character at the beginning of a command entered at the `simics>` prompt will cause that command to be executed by the outside shell, and not by Simics. However, it is often easiest to have two `ssh` sessions or `GNU-Screen / TMUX` windows open, one of which is running Simics and one of which is used to execute commands on the host machine.

Other useful Simics commands can be found in the Simics User Guide distributed with this lab. The Simics in-built help system is also useful for quick command references and can be accessed by typing,

```
simics> help command_name
```

2.2.2 Copy benchmark files into the target

The benchmark files of the lab are found in the benchmarks directory of the archive lab1-efiles.zip. Unzip this archive into your Simics workspace directory and copy all of the files within the benchmarks directory in to the target machine's `/home/lab-1` directory via the mounted host file system.

Checkpoint the simulation now.

```
simics> write-configuration benchmarks_copied.conf
```

Now you are done with the basic setup of the simulation environment for lab1.

You can use any of the indicated servers to run the lab experiments. Get started early with the rest of the lab because the experiments will take a significant amount of time to run, and this will only be exacerbated by contention for the servers.

3 Cache simulation with Simics

3.1 General methodology

At its core, Simics is an instruction set architecture simulator, not a machine performance simulator. While Simics presents an interface to the OS and programs running within it that makes the target machine appear to be a normal machine, in actuality many of the simulated machine's functions are idealized

far beyond the capabilities of a real machine. For example, in normal execution mode, main memory accesses in Simics appear to take zero cycles.

However, it is possible to attach a cache simulator module to Simics and use it to gauge a program's cache performance or the effectiveness of a particular cache hierarchy. The cache simulation module included with Simics is called *g-cache*. These modules can be linked into hierarchies, connected to multiple processors and have a variety of adjustable parameters. g-caches are configured and attached to the memory hierarchy by means of simple Simics scripts containing python commands. The `simics>` prompt can also accept Python commands prefixed by an `@` symbol.

One of these g-cache parameters is the delay accrued by a memory request as it passes down through each level of the hierarchy to the simulated memory interface. By running Simics in `-stall` mode, users can force the delayed execution of instructions according to whether or not the data the instructions require is contained in caches or memory. Simultaneously, the cache modules record statistics about the memory requests which have accessed them. `-stall` mode is a more detailed mode of simulation, and therefore noticeably slows down the operating speed of the target machine.

A further methodological detail is that when the caches are first attached to the simulation, they are empty. Any statistics recorded from them initially will only reflect the compulsory misses encountered as they fill up. For this reason, it is necessary to *warm* the caches by running the intended experiment application for millions of instructions before the true cache performance statistics can be collected.

Thus, the general methodology of this lab takes the following form:

1. Run the simulation at full speed to get the file system loaded, the experiment files set up, etc.
2. Checkpoint the simulation.
3. Restart simulation in `-stall` mode.
4. Load the caches using a .simics script.
5. Warm the caches by executing benchmark code.
6. Pause or breakpoint the simulation.
7. Reset all the caches' statistics.
8. Continue to run the benchmark and collect real cache data.

Data from the cache modules can take several forms. `<cache name>.info` reports the configuration of the cache, `<cache name>.status` displays the current value of every cache line and `<cache name>.statistics` reports cache performance statistics.

3.2 Collecting statistics from a simple cache

You will use three benchmarks namely *equake, vortex* and *parser* which can found in the benchmarks directory of the archive lab1-efiles.zip. Instructions for execution of these benchmarks are given in Table 1.

Benchmark	**Execution instructions, assuming you are in /home/lab-1 on the target**
vortex	`target# cd ./255.vortex/data/test/input` `target# ./../../../vortex bendian.raw`
equake	`target# cd ./183.equake/data/test/input` `target# ../../../quake < inp.in`

parser	`target# cd ./197.parser/`
	`target# ./parser data/all/input/2.1.dict <`
	`data/test/input/test.in`

Table 1: Benchmark execution instructions

For each benchmark, you will do the following:

Start Simics in stall mode and load the benchmarks_copied.conf checkpoint file:

```
host$ ./simics -stall -c benchmarks_copied.conf
```

The simulation begins paused. Execute the following commands to ensure proper operation of the benchmarks and to load the g-cache configuration:

```
simics> magic-break-enable
simics> istc-disable
simics> dstc-disable
```

Load the unified single cache module into the simulation. In this case, the cache module is just a .simics file with a sequence of Simics commands.

```
simics> run-command-file add-unified-cache-niagara.simics
```

Using `cache.info` look at the configuration of the cache you have loaded. Compare the output you see with the .simics file you just executed to see how the cache was configured using simple declarative Python statements.

```
simics> cache.info
```

Run one of the benchmark programs as per instructions in Table 1.

The benchmark you just ran will quickly reach a *magic* breakpoint and the simulation will pause. A magic breakpoint is a special simulation-pausing instruction that is programmed into the benchmark. Run for 100,000,000 instructions in order to warm the cache. This may take a few minutes as Simics is now running in a more detailed simulation mode.

```
simics> c 100_000_000
```

Reset the cache statistics to clear the data recorded for the warming period.

Then record data for the next million instructions.

```
simics> cache.reset-statistics
simics> c 1_000_000
```

Display the recorded statistics.

```
simics> cache.statistics
```

Continue the simulation, halt the benchmark, and run another benchmark, remembering to clear the cache statistics before you collect more data.

For each benchmark, record the hit ratio for all types of memory requests (instruction fetch, data read, data write). Which benchmark has the best cache performance? Which has the worst?

3.3 Determining benchmark working set size

Your task in this section is to determine the working set size of each of the benchmarks by varying the size of the simple unified cache (add-unified-cache-niagara.simics) used in Section 3.2. Record the measurements you make that support your claim. Which benchmark seems to have the largest working set, and how big is it?

Note that once a cache has been configured to be a certain size, changing its size parameters will have no effect on simulated performance. You must restart Simics and reattach the cache after modification.

3.4 Minimal instruction and data caches

For this section you will use the split instruction and data caches defined in add-split-cache-niagara.simics. Examine this file and note that each cache contains only one line! Also note that there is a new module present: an *id-splitter* that routes instructions to the appropriate L1 cache. Run the four benchmarks listed in Table 1 and record their performance on this minimal cache. What can you learn about the relative locality of data vs. instruction accesses?

Modify the cache so that it still only has one line, but make this line 4 times longer (128 bytes). Record how this changes performance. What does performance under the improved cache indicate about the spatial locality of each benchmark? What can you learn about the relative locality of data vs. instruction accesses?

3.5 Collecting statistics from a cache hierarchy

In this section we will attach a more complicated cache hierarchy to our simulation in order to conduct a more realistic study of the cache behavior of these

benchmarks. This new hierarchy is defined in the file add-complex-2level-cache-niagara.simics. The hierarchy has a split L1 data cache (dc) and instruction cache (ic), both of which are connected to a larger L2 cache (lev2c). Note that there is also a new module present: a *transaction-staller* that simulates the delay incurred by accesses to main memory. This configuration is similar to the one illustrated in the Simics User Guide page 200.

Note that memory accesses are now diverted from physical memory to the id-splitter, since it is now at the top of the hierarchy. It is important to note that if you decide to reconfigure to caches, then you must connect them back into the hierarchy such that you preserve the flow of timing delay down from the id-splitter all the way to the transaction-staller.

Yours tasks for this section are to:

- Collect L1 and L2 hit ratio statistics for the three benchmarks running on the complex 2-level cache hierarchy. How effective are the L2 caches at collecting misses from the L1 level?
- Modify the L2 cache (lev2c) so that it is only twice the size of each L1 cache (i.e. 8 KB). Pick one benchmark, and report how its L2 statistics change as a result of this size modification. Do the same for an L2 cache that is twice the size of the original (i.e. 1 MB). What can you conclude from these results?
- Look at the stall penalties assigned to each cache level (3 cycles in L1, 10 cycles in L2, 200 cycle to memory). Calculate the average memory access time of each of the three hierarchies that you have collected data for.

3.6 Open-ended exercise: Application tuning for a given cache hierarchy

Matrix multiplication is a task common to a wide variety of scientific and machine learning programs. Often, the performance of the computationally intensive core of these codes is based almost solely on the efficient execution of this common operation. Matrices used in such calculations can be quite large and so cache performance often has a direct impact on overall program performance.

To address the problem of multiplying large matrices, many scientific codes make use of a blocked matrix multiply algorithm. The algorithm divides the matrix multiplication task into smaller size chunks which only use a subset of the data contained in large matrix. By adjusting the block size, users can control the amount of data being operated on at any given time. More information about the specific mechanics of this algorithm can be found on the web. You have been given a GEneralized blocked Matrix Multiply (GEMM) source code file called gemm.c and a 2-level cache hierarchy called add-2level-cache-OE-niagara.simics.

Your task is to design a memory hierarchy for the GEMM benchmark. You will modify the cache to better suit a specific implementation of the blocked matrix

multiply code. This is analogous to creating a custom cache hierarchy for a special-purpose machine.

Use the source code gemm.c and the cache hierarchy file add-2level-cache-OE-niagara.simics as the baseline. You can modify the baseline cache configuration in any way you like. For example, you can add new caches, change the cache replacement and write policies. Consult the Simics User Guide chapter 18 for more information on configurable cache parameters. You can examine the gemm benchmark source code but you cannot modify it. The target machine does not have the *gcc* compiler. A precompiled version of GEMM for Sparc/Solaris can be downloaded from the course web and yoy can transfer the compiled binary to the Simics target by mounting the host file system.

You may also want to use the delay, power, and area statistics reported by CACTI (http://quid.hpl.hp.com:9081/cacti/) as further motivation for your chosen configuration. Use the normal interface with 1 bank and technology node of 65nm. Creating an enormous cache is useless if the access time, power, and area overheads are prohibitive. Report on the configuration you select and any performance statistics you recorded or calculations you made that prove it is well suited to the GEMM benchmark.

4 Multithreading simulation with Simics

In this part of the lab, we will run a parallel benchamrk on a multithreaded processor model backed by a complex memory hierarchy.

4.1 Target hardware

The Niagara Simics target machine actually features an 8-core UltraSPARC T1 processor which conforms to SPARCv9 architecture. Each of the 8 cores 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 (1 core x 1 context, default), 2 (2 cores x 1 context), or 32 (8 cores x 4 contexts each) thread contexts. Solaris treats each of the hardware thread contexts as a separate CPU.

We will use the 32-thread context configuration with a complex memory hierarchy private caches. Each of the eight cores will now have its own private L1 cache. The four threads on each core share their core's private L1 cache. All L1 caches are write-through and are backed by a shared L2 cache. The private caches are kept coherent by using a bus-based MESI protocol.

4.2 The multithreaded benchmark

The benchmark we will use in this lab is called Parallel Differential Equation Solver (PDES). This is a multithreaded implementation built using OpenMP. This program along with input files and associated runtime libraries is found in the benchmarks/rb.pdes/ directory in the archive lab1-efiles.zip. The PDES program is invoked with these piped command line parameters:

```
target# ./pdes nthreads < input_file
```

where `nthreads` is the number of threads that the program will use and `input_file` can be one of [inp25, inp100, inp200].

4.3 Setup

In this lab section you will boot up the Niagara machine and create new checkpoints with 32 thread contexts:

1. Start Simics.

   ```
   host$ ./simics
   ```

2. Set the number of CPUs to 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. Once the machine has booted, start bash, mount the host file system and copy the entire directory simics-workspace-lab1/benchmarks/rb.pdes to the target /home/lab-1 as explained in Section 2.2.

5. Create a checkpoint for this machine configuration.

6. Run the script add-breakpoint.simics. This will add a breakpoint at the appropriate point in the execution of PDES.

   ```
   simics> run-command-file add-breakpoint.simics
   ```

7. Setup the target machine's loader for the correct execution of PDES:

   ```
   target# export LD_LIBRARY_PATH=/home/lab-1/rb.pdes/lib:
   $LD_LIBRARY_PATH
   ```

8. Begin execution of PDES with the following command line parameters:

   ```
   target# ./pdes 32 < inp200
   ```

9. Simics should breakpoint after PDES starts the "Benchmarking ..." phase as seen on the target console.

10. Create a new checkpoint at this time. Setup is complete. Exit Simics.

4.4 Multithreaded performance study

In this section, you will run the multithreaded workload and study the performance of the private caches and the MESI protocol.

1. Start Simics in `-s` stall mode, continuing from the configuration you saved in the last setting. Run the add-2level-cache-mt-niagara.simics script to connect a 2-level cache hierarchy with eight private L1 caches (l1cache[0-7]), one shared L2 cache (l2cache) and one memory unit (staller). This script does something new: it configures the caches using a seperate Python file called configure-caches.py.

   ```
   simics> run-command-file add-2level-cache-mt-niagara.simics
   ```

2. Run the simulation for 1 million instructions to warm the caches.

   ```
   simics> c 1_000_000
   ```

3. Once the caches are warm, reset their statistics so that we can collect accurate data from them.

   ```
   simics> l2cache.reset-statistics
   simics> l1cache0.reset-statistics
   simics> l1cache1.reset-statistics
   simics> l1cache2.reset-statistics
   simics> l1cache3.reset-statistics
   simics> l1cache4.reset-statistics
   simics> l1cache5.reset-statistics
   simics> l1cache6.reset-statistics
   simics> l1cache7.reset-statistics
   ```

4. Run the simulation for at least 10_000_000 instructions to collect data.

   ```
   simics> c 10_000_000
   ```

5. Examine the statistics for the L1 and L2 caches. Note that the statistics command now reports some MESI statistics. Record the data read, data write, and instruction fetch hit rates for the L2 cache, and the average data read, data write and instruction fetch rates and average MESI statistics for the L1 caches.

6. The data you just collected was for 8 KB L1 caches and a 512 KB L2. Exit Simics and edit the configure-caches.py file to change the L1 cache sizes to 64 KB.

7. Restart from checkpoint you created for this parameterization of M, and repeat the experiment.

How did the cache miss rates change? How did the MESI statistics change? Explain your observations.

4.5 Open-ended exercise: Design space exploration of the shared memory hierarchy

You are the architect in charge of designing the memory hierarchy for a chip multiprocessor similar to the UltraSPARC T1. You are being given a budget of 1 MB of on-chip cache, which must be allocated in some manner between the private L1s and shared L2.

Edit the configure-caches.py file to change the size of the various caches. You can change the number of cache lines, size of each line, and associativity of each cache, but do not change its write policy (only the lowest level cache can be write-back). Make sure the total size of all your caches is less than 1 MB.

The baseline layout for the memory hierarchy is = 8 I/D combined (unified) private L1s + 1 shared L2. You may stick with this hierarchy, or if you are feeling creative you can create additional levels or distinctions. Note that the snoopers and higher_level_caches parameters are used to enable MESI coherence and will have to be modified too. See the Simics User Guide for more details or e-mail us if you have doubts.

The simplified cache access time rule is as follows: A cache of 1 KB or smaller has a 1 cycle latency. Every doubling in size results in an increase of 1 cycle, up to a flat, shared 1 MB cache which has a latency of 10 cycles. Make sure that as you change sizes you adjust your latencies appropriately by rounding up if required.

Evaluate by running PDES with 32 threads and recording the cache statistics. Make sure to state what parameters you are using. Remember that you have to run in `-stall` mode for the caches to work.

Report on your chosen configuration and provide a convincing argument as to why it is the best. Report also on any configurations you tested that performed poorly.

5 Multiprocessors and memory ordering

The purpose of this part of the lab is to show the need for synchronization and the effects of consistency in real multicore computers. To demonstrate this, we will use a very simple algorithm, Listing 1, shows pseudo-code for two threads increment and decrement a shared variable in parallel.

Listing 1: Test code for two threads sharing data. n is the number of iterations to exectue and *thread* is the thread number . We would intuitively expect *shared_data* to be 0 after both threads have executed, which is only the case if the implementation is properly synchronized.

```
1 shared_data = 0;
2 for  (i = 0; i < n; i++) {
3    if (thread == 0) {
4       shared_data = shared_data + 1;
```

```
5       } else {
6           shared_data = shared_data - 1;
7       }
8   }
```

A working implementation of Listing 1 needs to perform the increment and decrement of the shared data atomically, without interference from the other thread. You will experiment with different ways to achieve this.

In this assignment we will use some x86 assembler code. You will not write any of the actual assembler instructions yourself, but you might still find the Intel Architecture Manuals[1] handy as a reference. if (turnif (turn

5.1 Atomic instructions

For simple tasks, such as incrementing or decrementing a counter, atomic instructions are the right tools for the job. The x86 ISA specifies a `lock` prefix that can be used on *some* instructions to force them to execute atomically. Some other architectures take a different approach. For excample, the Alpha and PPC use a special load and a conditional store instruction that does not modify memory if the data the store depends on has changed after the load. Such conditional stores can be used to implement other atomic instructions, such as compare-and-swap.

Atomic instructions are usually used to implement concurrent algorithms, such as hash maps and linked lists. They are also used to implement various mutual exclusion algorithms and barriers.

5.2 Memory ordering

The x86, lika ll modern processors, aggresively optimizes memory accesses using various caching and buffering mechanisms. These optimizations affect the order of memory accesses on the memory bus. The hardware is designed so that single-threaded user space applications cannot detec such reorderings.

Things get more complicated when multiple threads are involved. In this case, some of the reordering can be detected. For example, consider a thread that executes a store followed by a load. Due to buffering of the store, the load may complete before the store is globally visible. To force memory accesses to be ordered, the x86 architecture provides a set of *fence instructions*. We will only use the `mfence` instruction that prevents both loads *and* stores from being reordered over the fence. In addition to this instruction, the x86 ISA specifies separate load and store fences that only prevent reordering of one of the access types.

For simplcity and compatibility reasons, Intel requires that memory accesses are

[1] `http://www.intel.com/products/processor/manuals/`

Listing 2: Code for thread i to run a critical section, thread j is the second thread that competes for the critical section. The *turn* variable should be initialized to 0 and both *flag* variables should be initialized to *False* prior to executing the algorithm.

```
1  flag_i = TRUE;
2  while (flag_j) do {
3      if (turn != i) {
4          flag_i = FALSE;
5          while (turn != i) do {
6              // Do nothing or sleep
7          }
8          flag_i = TRUE;
9      }
10 }

12 // Do critical work

14 turn = j;
15 flag_i = FALSE;
```

never reorderd past atomic instructions. This allows most synchronization algorithms that use atomic instructions to work correctly without memory fences.

Note: The x86 architecture manuals specify guarantees for memory ordering, there is nothing that prevents an implementation from being stricter than the specification. In fact, you might see a stricter behaviour on some machines, e.g., some multithreaded Atom CPUs. Obviously, such undocumented behavious cannot be relied on in applications.

5.3 What are critical sections, and who is this Dekker guy?

The simplest way to update shared data structures is to define critical sections, where only one thread is allowed to execute at a time. A mutual exclusion, *mutex*, algorithm ensures that only one thread can execute in the critical section at any given time. In addition to ensuring mutual exclusion, mutex implementations also ensure that memory accesses cannot be reordered to happen outside the critical section.

Several algorithms have been developed to solve the critical section problem, we sill use one called *Dekker's algorithm*. Dekker's algorithm was attributed to the Dutch mathematician *Theodorus J. Dekker* in a manuscript from 1965 by *Edgar W. Dijkstra*. See Listing 2 for a pseudo-code description of the algorithm.

5.4 The lab assignment

All the files related to this part of the laboratory exercise can be downloaded from the course web. Download them and extract them in a suitable working directory in your home directory on one of the available Linux login servers as specified in section 2.2. The provided skeleton code consists of a handful files described below.

Makefile Automates the compilation. You can simply type `make` to compile both the pthreads version and the version using your own synchronization primitives. You may use `make clean` to automatically remove generated files from the working directory.

lab1.5.c Common code for running the experiments. You do not need to make any changes in this file.

lab1.5.h Common data structures and declarations] No need to edit this file.

lab1.5_asm.h Inline-assembler implementations for all of the atomic instructions used in this assignment. No need to edit this file, but it is a good idea to read through this file to see what the atomic instructions look like.

cs_pthread.c Reference implementation of the synchronization code. You do not need to edit this file.

cs_dekker.c Implement Dekker's algorithm here.

test_critical.c Implement Listing 1 here using critical sections.

test_incdec.c Implement Listing 1 here using atomic increments and decrements.

test_cmpxchg.c Implement Listing 1 here using atomic compare and exchange instructions.

You do not have to modify the `Makefile`, but it might be useful to have a look inside. The file contains rules to build the `lab1.5` binary. You compile the application by executing the `make` command in the source directory. There are multiple targets in the `Makefile`, e.g. the *clean* target. To execute the *clean* target, which cleans up the working directory, you simply run `make clean`.

The test tyupe and critical section implementation can be selected using command line options. Run `./lab1.5 -h` for information about available options. The idea is that you should be able to test whether you have placed your critical sections correctly by using the *pthreads* critical sections implementations. Once the critical section has been placed correctly, you may start working on your implementation of Dekker's algorithm.

5.5 Directed portion

Perform the following tasks on a multicore x86 machine. You may use any of the login servers named in section 2.2. If you use your own computer, amek sure it has a high-end, i.e., not Atom based, processor. Document your experiments in your report.

1. Run the *critical section* tests with the *pthreads* critical sections implementation (`./lab1.5 -t critical -c pthreads`). Does the counter return to its initial value? Why?/Why not?
2. Insert calls to `enter_critical` and `exit_critical` into `test_critical.c` to enter and exit critical sections to allow for correct parallel execution of the test. Use the *pthreads* version to test this. Does the counter return to the initial value now?
3. Implement Dekker's algorithm for synchronization in `cs_dekker.c`.
   (a) Why do the flag and turn variables have to be volatile?
   (b) Why doesn't the straightforward implementation work?
4. Add suitable memory barriers to the code to make the synchronization work correctly. Use the `MFENCE` macro.
5. Implement Listing 1 in `test_incdec.c` using both atomic and non-atomic *inc* and *dec* instructions. You may (read *should*) use the functions defined in `lab1.2_asm.h`. What happens when you use the non-atomic instructions? Why? What happens when you use the atomic instructions? Why?
6. Implement Listing 1 in `test_cmpxchg.c` using both atomic and non-atomic compare-and-swap instructions. You may use the functions defined in `lab1.2_asm.h`. What happens when you use the non-atomic instructions? Why? What happens when you use the atomic instructions? Why?
7. Compare the runtime performance when using critical sections and atomic increments/decrements. Which one is faster and why?
8. Compare the runtime performance when using atomic and non-atomic increments/decrements. Is there any difference in performance? Why?

5.6 Open ended portion

Implement queue locks in `cs_queue.c` using atomic instructions. See the lecture notes for details about the algorithm. Explain your solution and test the performance with multiple threads.

6 Acknowledgments

A large part of this lab has been adapted with permission from the organizers of CS152 Spring 2010 course at University of California, Berkeley and of course Advanced Computer Architecture (AVDARK) at Uppsala University 2013.