TDDD56 Multicore and GPU computing
Lab 2: Non-blocking data structures

Nicolas Melot
nicolas.melot@liu.se

1st November 2012

1 Introduction

The protection of shared data structures against inconsistent states, caused by several threads concurrently using the same structure is usually achieved through locks or semaphores. Semaphores and locks synchronize threads in order to prevent them from destroying a shared data structures by modifying it concurrently. Using locks, only one thread can perform an operation at a time and another thread has to wait for this operation to be finished before it can perform its own operation. This technique guarantees the data structures to be constantly consistent. However if many threads use the same structure at the same time, and if these threads use it frequently, there may be a long queue of threads waiting for getting access to the same resource, making these threads spending more time waiting access than performing computation. Non-blocking data structures help at reducing this effect, by eliminating as much as possible exclusive accesses to resources. Instead of locks, they rely on atomic instructions that check the operation to perform before committing them. If they can be performed safely, operations are done and a success is reported, otherwise the instruction reports a failure.

This laboratory work focuses on the case of a stack and how locks and non-blocking synchronization affect performance. It also pinpoints the ABA problem, an important issue from the use of non-blocking synchronization to protect data structures.

2 Lab rooms

During the laboratory sessions, you have priority in rooms “Southfork” and “Konrad Suze” in the B building. See below for information about these rooms:

2.1 Konrad Suze (IDA)

- Intel®Xeon™ X5660[1]
  - 6 cores
  - 2.80GHz
- 6 GiB RAM
- OS: Debian Squeeze

[1] http://ark.intel.com/products/47921/Intel-Xeon-Processor-X5660-(12M-Cache-2_80-GHz-6_40-GTs-Intel-QPI)

During lab sessions, you are guaranteed to be alone using one computer at a time. Note that Konrad Suze is not accessible outside the lab sessions; you can nevertheless connect to one computer through ssh at *ssh <ida_student_id>@li21-<1..8>.ida.liu.se* using your *IDA* student id. You can use Octave to process your measurements and generate result graphs (see section 3).

2.2 Southfork (ISY)

- Intel®Core™ 2 Quad CPU Q9550[2]
  - 4 cores
  - 2.80GHz
- 4 GiB RAM
- OS: CentOS 6 i386

Southfork is open and accessible from 8:00 to 17:00 every day, except when other courses are taught. You can also remotely connect to *ssh <isy_student_id>ixtab.edu.isy.liu.se*, using your *ISY* student id. Southfork does not provide Octave to generate graphs but Matlab is available to run alternative scripts (see section 3).

3 Installation

Fetch the lab 1 skeleton source files from the CPU lab page *http://www.ida.liu.se/~nicme26/tddd56.en.shtml* and extract them to your personal storage space. You also need performance measurement and processing scripts available at http://www.ida.liu.se/~nicme26/measuring.en.shtml. Fetch the script source files and copy the content of the directory "Octave" (or "Matlab" if you work in Southfork) to the lab source directory.

4 Before the lab

Before coming to the lab, we recommend you do the following preparatory work:

- Explain how CAS can be used to implement protection for concurrent use of data structures
- Sketch a scenario featuring several threads raising the ABA problem
- Implement in advance all the code required in sections 5 and 6 so you can measure their performance during lab sessions. Use the preprocessor symbol NON_BLOCKING to take decision to use either a lock-based stack, a software CAS-based stack or a hardware-CAS-based stack (respectively values 0, 1 and 2). This value is known at compile time and can be handled using preprocessor instructions such as *#if-#then-#else*; see skeleton for example.

5 During the lab session

Take profit of the exclusive access you have to the computer you use in the lab session to perform the following tasks

[2] http://ark.intel.com/products/33924/Intel-Core2-Quad-Processor-Q9550-(12M-Cache-2_83-GHz-1333-MHz-FSB)

- Measure the performance of a lock-based concurrent stack and generate a plot showing the time to perform concurrently a fixed amount of push, pop and both push and pop operations, as a function of number of threads involved
- Measure the performance of a CAS-based concurrent stack and build a graph featuring the same scenario as above. Make sure the performance test avoids the ABA problem (push or pop only, no garbage collection, etc).

6 Lab demo

Demonstrate to your lab assistant the following elements:

1. Explain how CAS can implement a safe data structure sharing between several threads
2. Explain the scenario raising the ABA problem you have implemented
3. Execute a multi-threaded program where several threads use CAS to push and/or pop concurrently to the same shared stack. Synchronize explicitly this program using pthread locks or semaphores in order to make the threads to synchronize each other until the ABA problem arises. Use a conditional statement (*if(...)*) to detect that the ABA problem indeed happened. Insert *printf(...)* along your synchronization routines in order to keep track of the important steps leading to the situation where the ABA problem rises.
4. Show compare and comment the performance of both lock-based and hardware CAS-based concurrent stacks, as a function of number of threads.

7 Helpers

stack_test.c provides a performance measurement skeleton. Write the missing code to run parallel work with the stack and capture the time it takes. Use the provided constants so you ca make use of the help scripts with reduced work. Run *bash start compile* to compile several suggested variants, then *bash start run <name for experiment>* to run them and measure their performance (note that you must name your experiment). Finally, run the script *plot_data.m* (*octave plot_data.m* or start Matlab and run the script plot_data.m) to generate graphs showing the behavior of your implementation. Modify the files *compile*, *run*, *variables* and *Makefile* at wish to fit further experiments you may need. You can read documentation about measuring performance and using the script set at the page *http://www.ida.liu.se/~nicme26/measuring.en.shtml*. Alternatively, feel free to use any other mean of measuring performance but in any case, make sure you can explain the measurement process and the numbers you show.

8 Investigating further further (optional)

You can investigate the problem further by implementing the following tests, measure the behavior they generate and elaborate an explanation and conclusion after your observations:

- Implement a software CAS (a compare-and-swap instruction implemented using locks). Measure its performance with various number of threads and compare the results with the classic lock-based stack and the Treiber stack. Use the preprocessor symbol *NON_BLOCKING* set to 1 to compile a software CAS-based stack.
- Implement a test that pushes only to a stack, pops only or both push and pop concurrently and observe the performance difference. Suggest reasons for performance differences.

- Relax your benchmark and simulate work of various length (possibly random) for your threads between two stack operations (push or pop). Measure and compare the performance between the lock-based stack, the Treiber stack using lock-based CAS and the hardware-based Treiber stack.