332 Advanced Computer Architecture

Exercise 6 (Assessed): the Splay Tree Challenge

This is the second of two assessed coursework exercises.

You may work in groups of two or three if you wish, but your report must include an explicit statement of who did what.

Submit your work electronically via CATE.

Background This exercise is about the same splay tree benchmark program as we studied under simulation in Exercise 4. However, this time the challenge is to make it go as fast as you can.

You are *encouraged* to modify the source code – up to and including using a completely different data structure.

Although the main interest is the splay tree, you are also invited to investigate performance optimisations in the random workload generation if you wish.

All-out performance

Basically, your job is to figure out how to run this program as fast as you possibly can, and to write a brief report explaining how you did it.

Rules

1. The goal is to reduce the execution time for the entire run (not just the time per query), as follows:

   ```
   time ./splaytest.x86 3 $SIZE $SIZE $SIZE 5
   ```

   where `$SIZE` is a parameter chosen so that the runtime and memory requirement on the platform you choose is reasonable. On a PC I suggest 100000000 (100 million).

2. You can choose any hardware platform you wish. You are encouraged to find interesting and diverse machines to experiment with. The goal is high performance *on your chosen platform* so it is OK to choose an interesting machine even if it's not the fastest available. On linux type "cat /proc/cpuinfo".

   Try the Apple G5s, ICT supercomputer resources (Itanium, Opteron) possibly PDAs, DSP processors, graphics co-processor or FPGAs. Please ask if you would like a suggestion.

3. Make sure the machine is quiescent before doing timing experiments. Always repeat experiments for statistical significance.

4. Choose a problem size which suits the performance of the machine you choose - the runtime must be large enough for an improvements to be evident. The really interesting problems are, of course, the long-running ones.

5. The program prints "`number_of_matches`" when it finishes. If you modify the program, make sure that it still prints the same value as the unmodified program.

6. You can achieve full marks even if you do not achieve the maximum performance.

7. Marks are awarded for

   - Systematic analysis of the application's behaviour

- Systematic evaluation of performance improvement hypotheses
- Drawing conclusions from your experience
- A professional, well-presented report detailing the results of your work.

8. You should produce a compact report in the style of an academic paper for presentation at an international conference such as Supercomputing (www.sc2000.org). The report must not be more than 7 pages in length.

Changing the rules

If you want to bend any of these rules just ask.

For example:

- I am very interested in performance in a multithreaded context, with multiple threads querying, inserting and deleting from the same tree. If you wish to explore this we can discuss it.
- Some possible optimisations enhance query performance at the expense of insertions and deletions. If you do this, we should discuss how to adjust the workload model to balance queries against insertions and deletions.

Hints, tools and techniques

Performance analysis tools: You may find it useful to find out about:

- Cachegrind and cg_annotate
- kcachegrind - kcachegrind.sourceforge.net - graphical interface to cachegrind
- gprof - standard command-line profiling tool.
- kprof - kprof.sourceforge.net - graphical interface to gprof
- VTune - Intel's (Windows and Linux) tool for understanding CPU performance issues and mapping them back to source code
  (http://www.intel.com/software/products/vtune/). Free trial.
- AMD's CodeAnalyst (installed on CSG Athlon machines - Start→Programming→AMD) (if you have an AMD machine)[1]
- Apple's Shark/CHUD tools
  http://developer.apple.com/tools/shark_optimize.html
- Sun's Performance Analyzer
  http://docs.sun.com/source/806-3562/ (if you have a Sun Sparc machine)
- oprofile
  http://oprofile.sourceforge.net/news/ (requires kernel rebuild)
- OpenSpeedshop for Linux
  http://oss.sgi.com/openspeedshop/

[1] To do this you will need to build the code using a native Windows compiler.

Compilers You could investigate the potential benefits of more sophisticated compiler techniques:

- Intel's compilers
  (http://www.intel.com/software/products/compilers/)
- The Pathscale compiler
  (http://www.pathscale.com/ekopath.html)
- Codeplay's compilers (www.codeplay.com) (free demo download?)
- IBM's compilers, for Cell (Playstation 3) etc.

Source code modifications You are strongly invited to modify the source code to investigate performance optimisation opportunities.

How to finish The main criterion for assessment is this: you should have a reasonably sensible hypothesis for how to improve performance, and you should evaluate your hypothesis in a systematic way, using experiments together, if possible, with analysis.

What to hand in Hand in a concise report which

- Explains what hardware and software you used,
- What hypothesis (or hypotheses) you investigated,
- How you evaluated what the potential advantage could be,
- How you explored the effectiveness of the approach experimentally
- What conclusions can you draw from your work
- If you worked in a group, indicate who was responsible for what.

Please do not write more than seven pages.

Paul Kelly, Imperial College, 2008