332 Advanced Computer Architecture

Exercise 4 (ASSESSED): microarchitecture issues in splay trees

This is the first of two equally-weighted assessed coursework exercises. Working individually, work through the exercise and write up a short report presenting and explaining your results. Submit the pdf electronically via CATE[1]. The CATE system will also indicate the deadline for the exercise.

Background

A splay tree is a self-balancing binary search tree with the additional unusual property that recently accessed elements are quick to access again. It performs basic operations such as insertion, look-up and removal in $O(\log n)$ amortized time. For many non-uniform sequences of operations, splay trees perform better than other search trees, even when the specific pattern of the sequence is unknown. The splay tree was invented by Daniel Sleator and Robert Tarjan (this description from Wikipedia, which provides further useful background).

The big point is that splay trees are asymptotically as efficient as balanced search trees, but every time an item is accessed it is moved to the root of the tree – so it's much faster if items are frequently re-accessed. They suffer slightly higher overheads than the major alternatives (red-black trees and skiplists) with more uniformly-random access patterns.[2]

This exercise concerns the splay tree implementation from the GNU C Compiler, GCC. Splay trees are used for several purposes in GCC; we're using the version used at runtime in the Mudflap and MIRO bounds checking code (for optional runtime pointer error checking). It is extremely performance-critical.

We're working with an artificial benchmark that exercises the GCC splay tree code, using random insertions and random lookups. Since splay trees optimise for skewed access distributions, we use a random walk instead of a uniform random distribution.

Quite a large part of the benchmark's execution time is spent in the random number generator; I have included the source code from the GNU C Library (glibc-1.09). Many simulations rely on very fast generation of random numbers - so this code is also of some interest from a performance point of view. There are, of course, many random number generators to choose from, some with better properties than this one.

Running the splaytree benchmark

You can find the source code in

`/homes/phjk/ToyPrograms/ACA08/SplayTrees`

[1] `https://cate.doc.ic.ac.uk/`

[2] The original source is `http://www.cs.cmu.edu/~sleator/papers/self-adjusting.pdf`.

Copy this file to your own directory[3]. You should compile the application for the x86 platform using the following commands:

```
make x86
```

Now run the application as follows:

```
time ./splaytest.x86 3 50000000 50000000 50000000 5
```

This takes 6-20 seconds depending on your machine. It reports the number of random queries that succeed; this is pretty meaningless but provides a check that it's working properly.

Compile the application for the SimpleScalar simulator using the following command:

```
make simplescalar
```

Now run the application as follows:

```
~phjk/simplesim/sim-outorder ./splaytest.ss 3 50000 50000 50000 5
```

This takes 20-60 seconds. This is a much smaller run, as the simulator is very slow.

Details of the command-line arguments are given in Appendix A.

What to do

Part 1: influence of tree size

Your first job is to study how the performance varies with tree size, when the key distribution is uniform (that is, when the last parameter ("delta") is large).

1. Choose a linux machine on the DoC network[4]

   Try to make sure no other processes are active as they will interfere with your results. How many times do you need to repeat your experiments to get a reliable result?

   How long must the program run for the timing to be reliable?

2. Plot a graph that shows how the performance, in microseconds/query, as the tree size (number of insertions) is varied.

   You might like to use a script like this:

   ```
   ./scripts/varysizeuniformkeys ./splaytest.x86 100 13107200 1000000
   ```

   This runs 20 experiments, with tree sizes from 100 to 13107200, with 1000000 queries each time.

   What conclusions can you draw from the results? What factors do you think are influencing the time per query?

[3] This text is available in html for copy-and-paste on the course web site.

[4] See `https://www.doc.ic.ac.uk/csg/computers/`.

Part 2: influence of microarchitecture

Your next job is to study the effect of various architectural features on the performance of the splaytest application running under the SimpleScalar simulator.

For this part of the exercise, we will use DELTA=5, so the distribution of key is highly non-uniform:

```
~phjk/simplesim/sim-outorder ./splaytest.ss 3 50000 50000 50000 5
```

What to do:

1. Use sim-outorder's "-ruu:size" flag to vary the RUU size between 2 and 256 (only powers of two are valid). Plot a graph showing your results. Explain what you see.

   You may wish to use this script:

   ```
   ./scripts/ss_varyarch
   ```

2. Examine the other sim-outorder parameters (leave the cache parameters unchanged). Where is the bottleneck (when running this application) in the default simulated architecture?

   Does the bottleneck change when DELTA is changed?

   Justify your answer.

Write your results up in a short report (less than four pages including graphs and discussion).

Tools and tips

Plotting a graph

Try using the gnuplot program. Run the script above, and save the output in a file "table". Type "gnuplot". Then, at its prompt type:

```
set logscale x 2
plot [][0:2] 'table2' using 1:3 with linespoints
```

To save the plot as a postscript file, try:

```
set term postscript eps
set output "psfile.ps"
plot [][0:2] 'table2' using 1:3 with linespoints
```

Try "help postscript", "help plot" etc for further details.

Examining the assembler code

Compile the application using the "-S" flag.

```
~phjk/simplescalar/bin/gcc -O3 -S bounds-splay-tree.c
```

The assembler code generated by the compiler is delivered in "bounds-splay-tree.s".

kcachegrind and callgrind

You might find it interesting to try kcachegrind with this program. For example,

```
~phjk/valgrind/bin/valgrind --tool=callgrind --simulate-cache=yes \
--dump-instr=yes ./splaytest.x86 3 500000 500000 500000 5
```

(this is one very long command line). This generates a statistics dump called `callgrind.out.#####`, where `#####` is the process id - suppose it's 31636. Now run kcachegrind:

```
kcachegrind callgrind.out.31636
```

Paul Kelly, Imperial College London, 2008

Appendix A: splaytest command-line arguments

The splaytest program's command-line parameters are as follows:

```
./splaytest.x86 3 <NINSERTIONS> <NQUERIES> <KEY_RANGE> <DELTA>
```

The "3" selects test number 3, which the focus for the exercise (1 and 2 are for debugging). NINSERTIONS is the number of random insertions into the tree. NQUERIES is the number of random queries after the insertions. KEY_RANGE is the maximum key value used. DELTA is the maximum random step size for the random walk.

In our experiments you are recommended to make NINSERTIONS=NQUERIES=KEY_RANGE. DELTA should be at least 2; larger values make the key distribution more uniformly random, leading to worse splay tree performance.