UNIVERSITY OF SOUTHERN CALIFORNIA

CS402 Assignment 01 – Performance

Michael Gerow

February 22, 2013

1 Tests

Getting the performance numbers of all the locking methods to line up with what one would "expect" them to be ended up being harder than I thought it would. I have two tests that I used to mark the performance of my locking implementations throughout development, but only one seems show fine being faster than rw being faster than coarse.
All of these tests make use of multiple processors to various degrees. This is especially important because the only way to really gain additional performance from a machine with multiple processors is to write code that effectively uses more than one when locking and unlocking resources.
I also wrote a little python script to run tests multiple times and compute the mean and standard deviation of the runs. These can be invoked with run_unbalanced_test.py and run_gen_test.py.
Also, all graphs that follow will have three vertical lines in them. These represent the left standard deviation, the mean, and the right standard deviation.

1.1 Unbalanced Test

In order to get a result where fine is faster than rw is faster than coarse I really had to tune the test quite a bit. The way this test works is that it first initialzes the database with seven nodes in a perfectly balanced arrangement, leaving four leaves on the bottom of the tree. I then had eight seperate threads make 50000 writes each that essentially made the tree incredibly imbalanced below those points. This alone was not enough to make sure that rw was faster than coarse, though, so I added eight extra threads that made essentially random reads throughout the database. This had the effect of making rw fast enough to outpace coarse but now fast enough to beat fine.

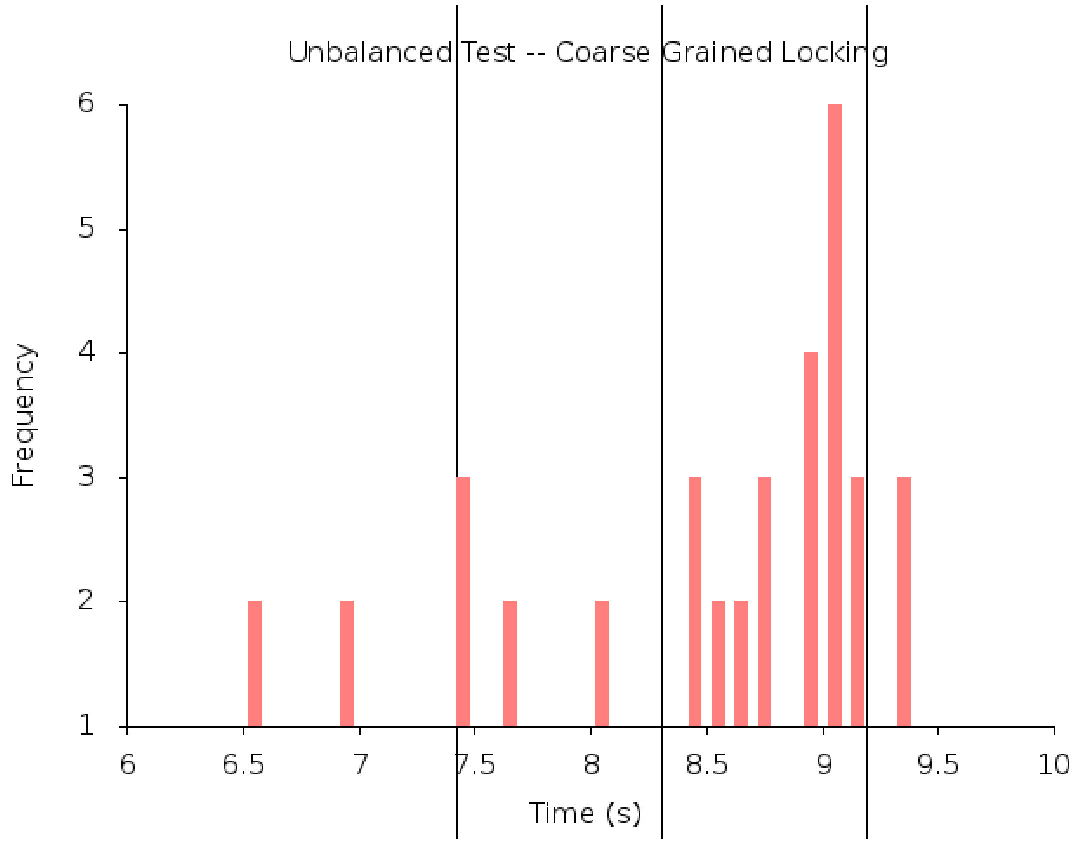
This test can be invoked by running run_unbalanced_test.py with the only argument being the number of trials you would like to try. This essentially pipes the file at test/unbalanced_test into each of the server types.

1.1.1 Data

```
server_coarse  mean: 8.3078  standard deviation: 0.886419742099
server_rw  mean: 7.2918  standard deviation: 0.495389724956
server_fine  mean: 5.2562  standard deviation: 0.306972643865
```

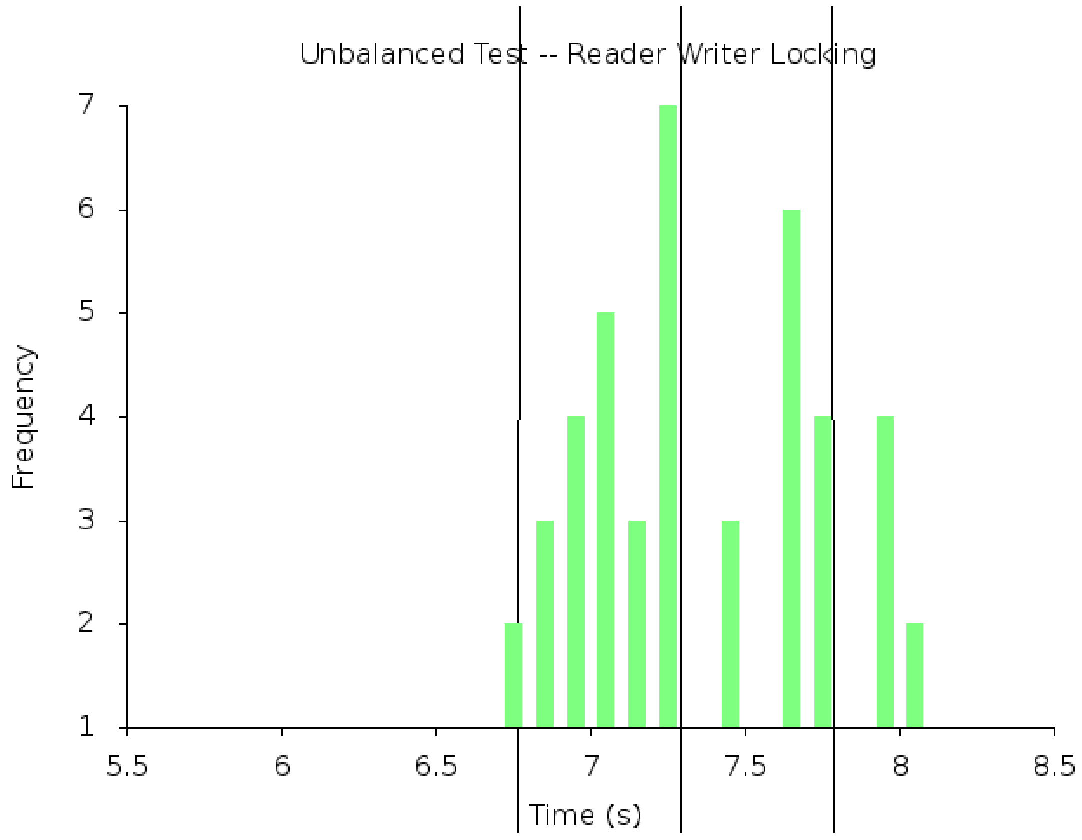
1.1.2 Graphs

Figure 1.1: A histogram of times taken by the coarse grained locking method for the unbalanced test



Please see *figures* 1.1, 1.2, and 1.3.

Figure 1.2: A histogram of times taken by the read write locking method for the unbalanced test



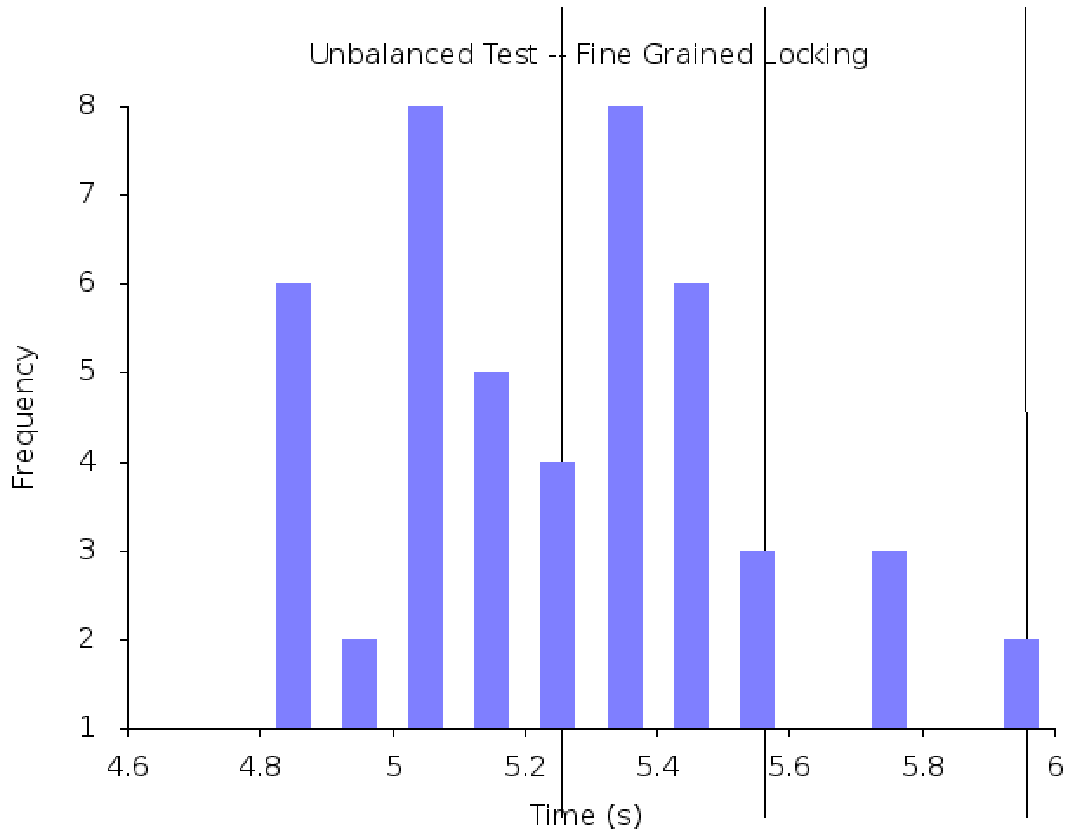
1.1.3 Analysis

We can clearly see that the performance ranks in expected order, that is that fine grained locking beats read write locking which beats coarse grained locking.

The reason fine grained locking seems to work so well in this scenario is because each of the eight writing threads essentially stick to their own part of the tree, making it easy for multiple threads to be writing to the tree at the same time. Read write, on the other hand, can only have one thread writing to the tree at once time. It is able to beat the coarse grained lock, though, because there is also a collection of reads in the test that the rw method is able to execute mostly in parallel while the coarse grained lock must serialize these accesses.

I would like to note, though, that this test took a fair amount of tweaking in order to get these results. For most random inputs it seems that the fine grained locking loses out to read write locking, as we will see in the next test.

Figure 1.3: A histogram of times taken by the fine grained locking method for the unbalanced test



1.2 Gen Test

This next test I called the "gen" test, mainly because it is generated randomly from the files in test/readergen.py and test/writergen.py. These scripts generate 50000 random reads between 0 and 16536 and 16536 random writes that go from 0 to 16536 (essentially hitting all values in the range). This test is read heavy, which I thought might somewhat closely model a database used in an application like a web server, so I thought it would be an interesting test case to look at.
As with the unbalanced test this test can be invoked by running run_gen_test.py with the only argument being the desired number of trials per server type. It essentially pipes the file at test/gen_test into each of the server types.

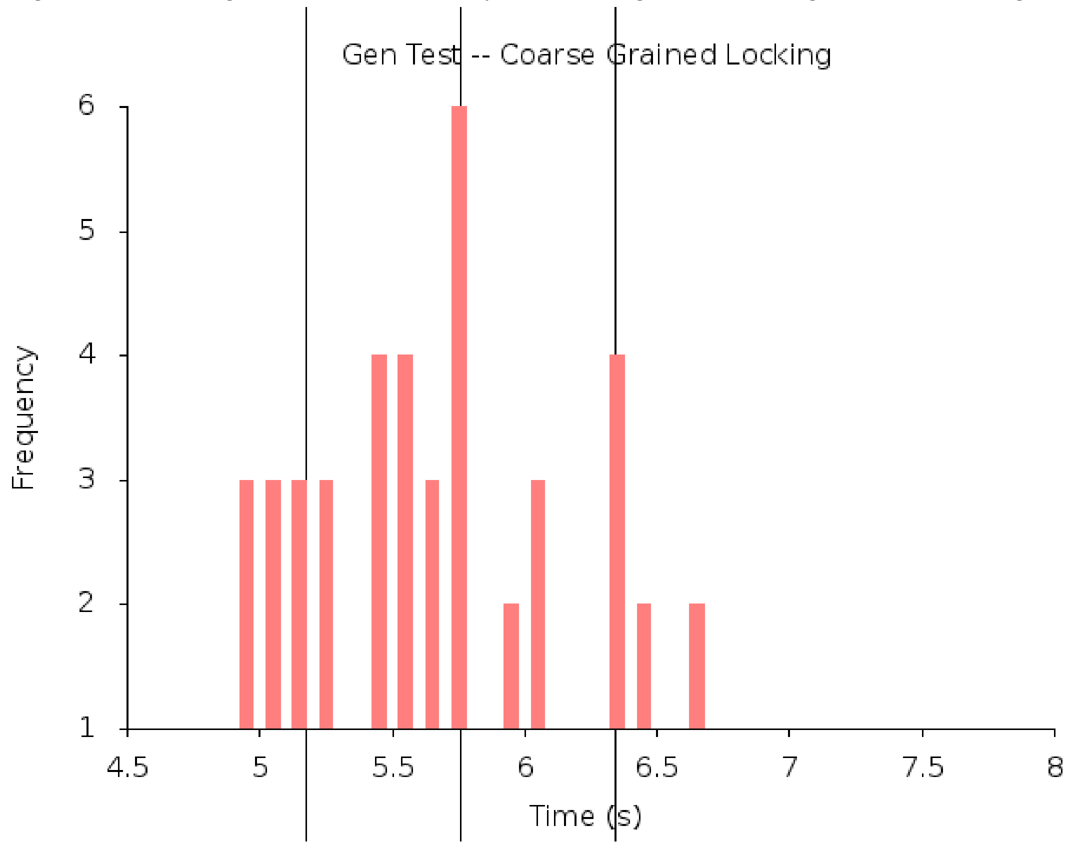
1.2.1 Data

```
server_coarse  mean: 5.759  standard deviation: 0.582357103433
server_rw  mean: 2.15  standard deviation: 0.311028576678
```

```
server_fine  mean: 3.0696  standard deviation: 0.116495773769
```

1.2.2 Graphs

Figure 1.4: A histogram of times taken by the coarse grained locking method for the gen test

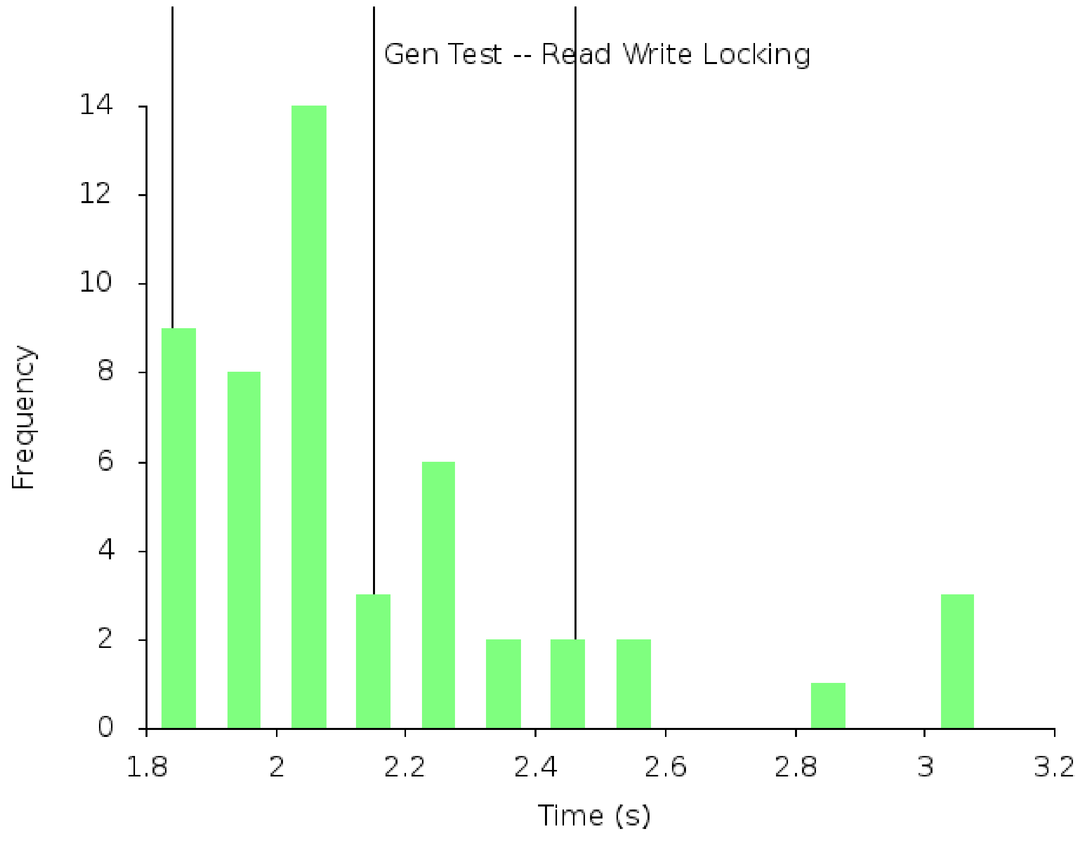


Please see *figures* 1.4, 1.5, and 1.6.

1.2.3 Analysis

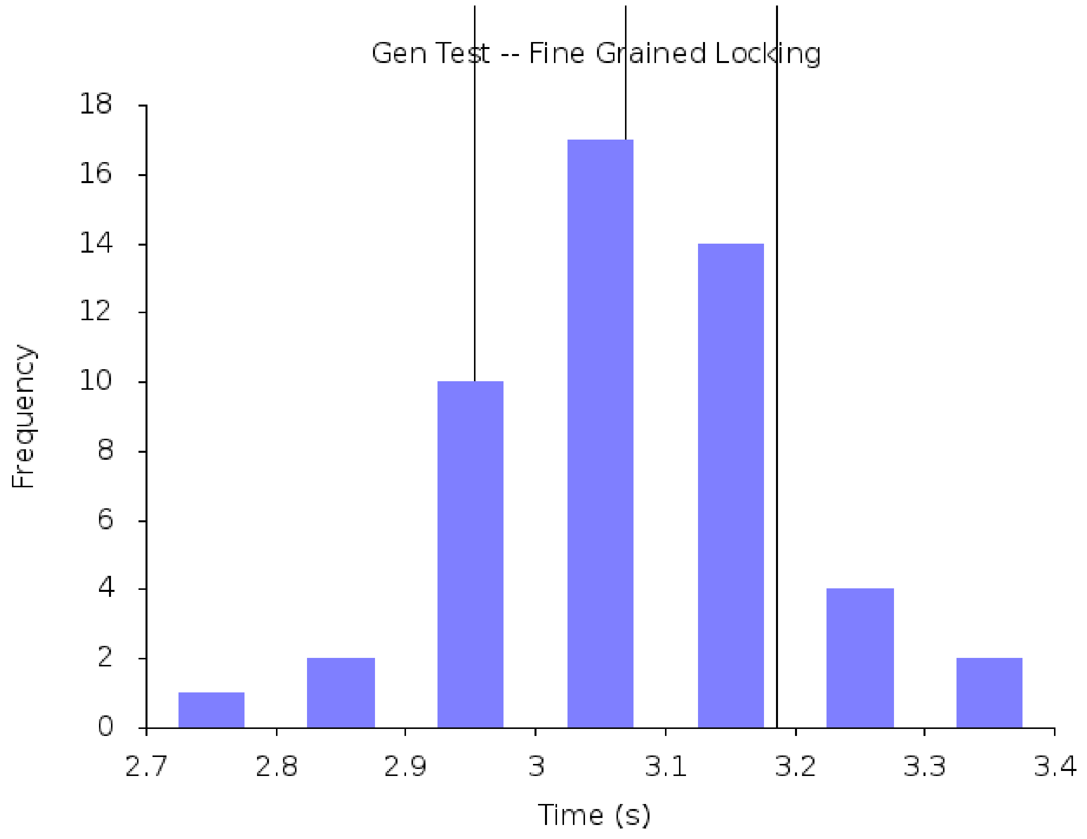
Somewhat unexpectedly, the fine grained locking seems to be losing out to the read write locking by a significant amount. One would expect a finer grain of lock would almost always result in an increase in performance, but this test seems to turn that on its head.
I have not done an extremely thorough exploration about why this might happen, but I can take a guess. It is likely that the fine grained lock happens to block a lot in the middle of reading the tree in order to wait on a lock to be freed. The overhead associated with constant blocking and unblocking may ultimately be too much to really beat the less fine grained read write locking.

Figure 1.5: A histogram of times taken by the read write locking method for the gen test



All that time the read write locking method is sending entire queries through the table without there even being a possibility of blocking midway through the table, eliminating the overhead that can bring.

Figure 1.6: A histogram of times taken by the fine grained locking method for the gen test


Gen Test -- Fine Grained Locking
Frequency
Time (s)