Program 2
The Way the *Disk* turns

Matt McCarthy

Christopher Newport University
matthew.mccarthy.12@cnu.edu

November 2015

Abstract

We investigate the effect of spatial and temporal locality on disk I/O and cache performance.

1 Background

In order to parallelize code, a developer must first decide how to decompose the problem. A common and simple decomposition is data decomposition where the developer assigns different blocks of data to each processor. However, when the data is stored on a disk, like in a database, I/O performance becomes a significant bottleneck. While the operating system has primary responsibility for optimizing disk reads, it only has a small window of read requests with which it can make optimization decisions. Thus, when possible, the developer should endeavor to use spatial and temporal locality at the application level.

2 Set up

We consider a file system containing 2GB of text files, each of which is exactly 14KB. We then take a list of numbers, identified by an integer k such that

$$51 \leq k \leq 161811$$

which is stored in k.txt.

3 Test Environment

The test was run on a machine running Arch Linux with the kernel 4.2-ck with an Intel i7-4770k at 4.2GHz. The executables, the id list, and web server trace file were stored on a SSD and all directories

and databases were written to a 5400RPM mechanical harddrive mounted at ./ext. The mechanical harddrive was filled with a single EXT2 partition. We utilized a shell script named run.sh[1] that managed our test parameters.

The usage of run.sh is:

```
run.sh <trace> <modulus> <file size> <num trials>
<cache line> <number of lines>
```

We ran, run.sh trace.txt 250 14336 5 1 15000.

We generate the test environment with a program db-gen[2]. This script will invoke db-gen to create a list of file ids listed in trace.txt, denoted I, and for each $k \in I$ will compute $k \equiv_{250} k_h k_t k_o$ where k_h, k_t, and k_o are the hundreds, tens, and ones digits respectively. It will then produce a file /ext/db-files/k_h/k_t/k_o/k.txt, filled with 14336 null characters, which makes it exactly 14KB large. For example, with the id 361, it will create /ext/db-files/1/1/1/361.txt.

4 Sequential Read Time

To begin our test suite, we make a program, namely benchmark[3], read through each id in order from the disk. This linear traversal of the files will be the fastest the disk can possibly perform on our directory structure due to both, how the disk works and operating system optimizations.

We pass our sequential directory test[4] the list of file

---

[1] Located at https://github.com/matt-mccarthy/verdant-octo-kumquat

[2] Compiled from db-gen.cpp

[3] Source code for benchmark is benchmark.cpp and every file in the src directory in the GitHub repo.

[4] The source for this test is the run_experiment_dir function in src/test_suite.hpp

ids in sequential order, a mapping that takes each file id to a filename on the disk, and the file size (14336B). The test would then return a time in milliseconds, and was ran five times.

For the sequential read time our results were $t_{dir,s} = (415.99 \pm 50.77)ms$.

5 Randomized Read Time

In order to determine the seek time of the disk, we ran the test for sequential read time again but instead shuffled the file ids into a random order on each trial. The yielded a read time of $(463.32 \pm 2.18)ms$. This results in an average seek time of $3.38\mu s$.

6 Web Server Trace

Since in real use cases, data is accessed in neither a sequential nor a random manner. To demonstrate performance in the real life use case, we read file ids in the order an actual web server trace, namely `trace.txt`, provides.

By inspection, we saw that files were often accessed multiple times consecutively, which is strong temporal locality. Furthermore, many ids were numerically "close" suggesting some spatial locality.

Our results for this test, we passed the ordering of ids listed in order that the trace dictated to our directory test. This resulted in an average execution time of $(4401.58 \pm 11.5646)ms$.

7 Database

In addition to producing the directory structure, `db-gen` produces a database file that has the contents of each of the directory files pushed into one large file. The goal of this section is to determine the benefits of using a database file instead of a large directory structure. In theory, this should lower the amount of overhead incurred by repeatedly opening and closing files on the filesystem.

To test database performance, we modified the directory test to use a database file[5]. This now takes a database file location, a map from each id to an offset in the database, instead of the directory map. As shown in Figure 1, using the database file effectively doubles performance in our test environment.

---

[5]The test is run_experiment_db in src/test_suite.hpp

	Directory	Database
Sequential	415.99ms	213.08ms
Random	463.32ms	266.93ms
Trace	8509.17ms	4401.58ms

Figure 1: Directory vs. Database

By our inspection from Section 6, we could potentially reorder the files in the database in order to enhance the locality of the trace.