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	DB	RAM
Sequential	415.99	213.08	184.78
Random	463.32	266.93	236.494
Trace	8509.17	4401.58	2615.1

Figure 1: Read performance. All times reported in ms.

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

8 RAM Disk

For our next series of tests, we modified the database test to map the database file into memory using Boost's `mapped_file_source` and then read from that. As demonstrated in Figure 1, the RAM disk's read performance was better than the database's. More interestingly, the RAM disk's trace completed in nearly half the time of the database's trace.

While, mapping the whole database allows for exceptional I/O performance, mapping huge data sets into memory stops being viable after the size of the data set reaches a certain value.

9 Cache

Since pulling the entire file into memory can quickly become inviable, we implement a cache to store a set number of entries in memory as we progress through the trace.

We use an `unordered_map<int,entry_seq>`[6] to hold the cache. This map is initialized with all of the file ids and a `entry_seq` for each id. Each `entry_seq` object holds an offset in the database file, a pointer to memory (which is `nullptr` when it is not in cache), and an iterator to a position in the cache list.

When we access an entry in the cache, we check if the pointer in its corresponding `entry_seq` object is null. If it is, we add the entry into cache (read from the disk into a location in memory) and add a pointer to it in the back of the cache list, additionally we update the iterator to point to its location in the list. If it is not null, we move the pointer in the cache list to the end of the list and then update the iterator in the object to point to the new location of the pointer.

[6]Cache source is found in src/cache_seq.h cache_seq.cpp, entry_seq.h, and entry_seq.cpp

9.1 Perfect Cache

In order to determine the effectiveness of our cache, we caclulate the miss rate of a perfect cache of infinite size. In the trace, we counted the number of unique ids and then the total number of requests to determine that our perfect cache would have a hit rate of approximately .95, which counts only compulsory misses.

9.2 Replacement Policy

We implement a Least Recently Used replacement policy. We replace whenever we attempt to add an entry to the cache, but the cache is full. In order to replace an element, we simply make the `entry_seq` object at the front of the list delete its memory contents and then we remove the pointer from the list.

Once the cache is populated, our cache's asymptotic efficiency becomes $O(n)$ since on each add, we must also remove an entry.