Program 2
The Way the *Disk* turns

Matt McCarthy

Christopher Newport University
CPSC 621 Parallel Processing
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 scenario, I/O performance becomes a significant bottleneck. While the operating system has primary responsibility for optimizing disk reads, it only has a small number of read requests with which it can make decisions on how to optimally move the disk arm. 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

To create the test environment, we create a list of file ids listed in `trace.txt`, denoted I, and for each $k \in I$ we then 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 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[1] the list of file ids in sequential order, a mapping that

Source can be found at `https://github.com/matt-mccarthy/verdant-octo-kumquat`.

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

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, we produce 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[2]. This now takes a database file location, a map from each id to an offset in the database, instead of the directory map.

	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.

As shown in Figure 1, using the database file effectively doubles performance in our test environment. 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[3]. 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>`[4] to hold the cache. This map is initialized with all

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

[3]The test is run_experiment_ram in src/test_suite.hpp

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

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. Furthermore, our cache is serial and does not use prefetching.

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.

9.3 Results

We tested the cache[5] at three different sizes, 15000 entries, 20000 entries, and 25000 entries[6] with our results reported in Figures 2, 3, and 4.

Figure 2 describes the read performance of the cache. Our results show that the cache performs worse than a raw database file for sequential and random read, thus showing that our cache implementation has overhead. This overhead likely arises from garbage collection and the miss rate reporting. However, sequential and random read speeds do not test the cache's primary purpose, exploiting temporal locality.

If we inspect the trace test results we see a different story, that is, the cache performs better than the database file read, but worse than the RAM disk, as expected. In a way, the database file test uses a cache of size zero, while the RAM disk uses a cache of infinite size.

	DB	15k	20K	25k	RAM
Seq	213	368	369	404	184
Rand	267	425	422	461	237
Trace	4402	3500	3145	3137	2615

Figure 2: Read performance rounded to nearest ms.

While, the improved read performance of the trace indicates that our cache is effective, the best metric for cache effectiveness is the hit and miss rate. Figure 3 depicts the miss rate as we issued requests to the cache. For the most part, the miss rate decreased as the number of requests progressed towards around 2000000, but after this critical point miss rate began to slowly increase. This indicates that the trace exhibited more temporal locality as the number of requests approached 2000000, but afterwards users started requesting more varied entries.

Figure 4 contains the hit rates produced by the different cache sizes. As expected, the hit

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

[6]Approximately 200MB, 275MB, and 350MB respectively.

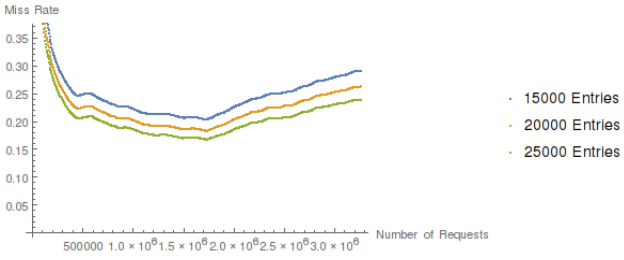

Figure 3: Number of Requests vs. Miss Rate

	15k	20k	25k
Hit Rate	71%	74.8%	76.1%

Figure 4: The overall hit rate at each cache size.

rate monotonically increases with the cache size.

10 Experimental Framework

The test was run on a machine running Arch Linux with the kernel `4.2-ck` with an Intel i7-4770k quad-core processor with eight threads at 4.2GHz and 16GB of DDR3 1333MHz RAM. 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.

The code we used to run our tests can be found at `https://github.com/matt-mccarthy/verdant-octo-kumquat`. We wrote all code for the tests in C++ with `g++-5.2.0` as the compiler with the `-O3` flag enabled. Furthermore, we utilized Boost to provide a C++ `mmap` wrapper. We wrote a shell script named `run.sh`that managed our test parameters and invoked our `db-gen` to create the directory and database. Afterwards, `run.sh` ran the `benchmark` program in its various modes in order to gather the results.

The usage of `run.sh` is:

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

For our tests, we ran,

```
run.sh trace.txt 250 14336 5 1 <Cache Size>
```

11 Future Work

Since our cache only exploited temporal locality, one could add prefetching to the cache in order to improve the hit rate by exploiting spacial locality. Furthermore, one could lower the overhead of the cache to improve read performance. Another interesting improvement would be to add multithreaded prefetching to see if any performance gains can be achieved.

12 Conclusion

As the data suggests, locality is a powerful tool at our disposal which can easily improve performance of disk I/O bound applications. By adding a simple cache with a least recently used replacement policy, we reduced the execution time of our toy database reader by 25% of the original exectution time. Furthermore, in order to get a hit rate within 20% of the maximal hit rate, we only need to store 350MB of our 2GB database in memory, thus leaving us more resources for other tasks.

References

[1] Ananth Grama et al. *Introduction to Parallel Computing*. 2nd ed. ISBN: 9780201648652.

[2] John Hennessy and David Patterson. *Computer Architecture, a Quantitative Approach*. 4th ed. Morgan Kaufmann Publishers: San Francisco, CA, 2008.