

# On-Disk Sequence Cache(ODSC): Using Excess Disk Capacity to Increase Performance

A Thesis Proposal  
Presented to the  
Department of Computer Science  
Brigham Young University

In Partial Fulfillment  
of the Requirements for the Degree  
Master of Science

by  
Christopher R. Slade  
October 27, 2004

## 1 Introduction

While improving computer performance, designers must make many trade-offs. One common trade-off is trading space for speed. This trade-off is found in both software and hardware. In software, a programmer may decide to use a tree structure instead of an array to achieve better performance. Trees use more space in memory by storing parent/children pointers, but operations like searches and sorts are performed in less time. In hardware, carry-look-ahead adders use more circuitry to decrease the amount of time each operation takes. Since hard drive performance is considered a limiting factor in the performance of a computer system[1, 2], computer performance can also benefit by trading disk space for increased hard drive performance.

Hard drive performance is divided into four basic tasks:

- seek time - the time needed to move the head to the cylinder where the requested data resides.
- rotational delay -the time for the requested sector to rotate under the head.
- transfer time - the time it takes to transfer data from the disk into a buffer in the hard drive.
- disk controller overhead - the time it takes to issue the command and transfer the data across the bus.

The sum of these tasks is the access time, or the total time to complete a hard drive request [3].

In order to improve system performance, file systems reduce disk access time by reducing seek time. By organizing files sequentially on the disk, the amount of seeking that must be done in order to read a file is reduced. However, if files increase in size after they have been placed on disk, the file system may no longer be able to store the file sequentially. In order to assist file systems, disk defragmenters are periodically run to reorganize sectors so that files are sequentially stored. By reducing the seek time, file systems and disk defragmenters improve disk performance.

Modern operating systems seek to improve disk performance by reducing the number of times they access the disk. Demand paging and disk caching are some of the techniques operating systems have implemented. Demand paging only reads a section of a file, or page, as it is needed instead of reading the full file off the disk. Although demand paging causes the hard drive to seek more, it increases overall performance because pages that are not needed are never read off the disk [4]. Disk caching stores copies of disk sectors (called blocks after they have been read off the disk) in memory. Disk caching increases performance not only because it reduces the number of disk accesses, but also because it makes disk writes non-critical to system performance [5].

Although demand paging and disk caching have improved disk performance, they have also increased the average seek time per request. Demand paging increases seek time because files are not accessed sequentially, and accesses to multiple files are interleaved. Since some blocks in a file can be removed from the disk cache without removing all the blocks in the file, disk caching also creates greater seek times by only requesting sectors that are currently not cached. Because demand paging and disk caching have increased seek time, sectors can be rearranged on the disk to further increase performance [6].

This work proposes a new technique to rearrange sectors by using an on-disk sequence cache (ODSC). The ODSC is a technique that can be implemented in either the hard drive controller or the low-level driver, independent of the file and operating systems. The ODSC uses a designated cache partition on the disk to hold copies of disk sectors and store them in the order that the operating system requests them. This method has an advantage over reorganization because a sector that is used in two different sequences is copied into the cache twice, once for each sequence.

Hard drives have significantly increased in capacity over the past couple of decades; an ODSC uses the extra capacity to reduce the relatively long access time. Also, by reducing the time penalty for missing the disk cache in main memory, an ODSC allows systems with less memory to perform just as well as systems without an ODSC but with more memory.

## 2 Thesis Statement

Trading disk space for an ODSC that caches sectors in the order they are accessed decreases the average disk access time. The reduced disk access time increases system performance and allows a reduction of main memory size while maintaining the same performance.

## 3 Methods

In order to demonstrate that an ODSC improves disk performance, an ODSC will be implemented in the Linux operating system, using the 2.4.26 kernel (the most recent, stable kernel when this work was started). The ODSC will do the following operations:

- collect and analyze traces of disk activity.
- copy sequences into the cache partition.
- translate requests from their original position to their cached position.

After the ODSC is implemented, experiments will be run with and without the ODSC to verify that the ODSC reduces disk access time, increases system performance, and that main memory size can be reduced while maintaining the same performance.

### 3.1 ODSC Implementation

This work's implementation of the ODSC is designed to take advantage of the optimizations that the Linux 2.4.26 kernel has implemented to improve disk performance. The Linux 2.4.26 kernel has a request queue with an elevator function that reorders disk requests to reduce seeking as much as possible. To take advantage of the request queue the ODSC must translate requests before they have been placed in the queue. In order to cache sectors in the order they are read off the disk, the trace must be collected after the request queue optimizations. Since the request queue optimizations occur between the block driver and the IDE driver, requests will be translated in the block driver, and the trace will be collected in

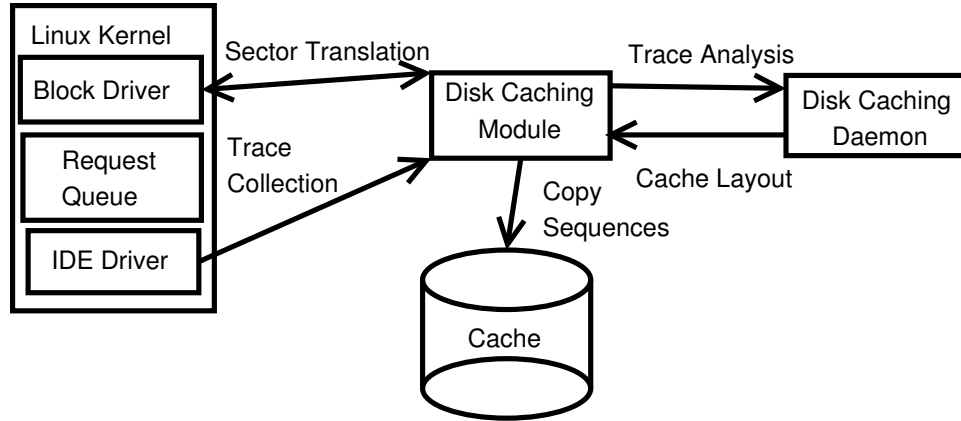


Figure 1: Design of the On-Disk Cache Sequence

the IDE driver. This organization will allow the ODSC to take advantage of the request queue optimizations while still caching sectors in the order they are read off the disk.

The ODSC is designed to facilitate changing caching strategies for future work. The three basic parts of the ODSC are the Linux kernel modifications, the disk caching module (DCM), and the disk caching daemon (DCD), as shown in Figure 1. The kernel modifications will interface with the DCM to collect the trace and translate requests. The DCD will read and analyze the trace from the DCM. From the trace analysis, the DCD will instruct the DCM to cache the sequences it found. The DCM will interface with the hard drive to physically copy the sequences on to the cache partition. Since the DCD decides what sequences should be cached, the caching strategy can be modified by just changing the daemon. The following is a more detailed description of each of these components.

### 3.1.1 Kernel Modifications

As mentioned above, the kernel will be modified by changing the block driver and the IDE driver. The block driver is the kernel module that handles requests for all block devices, including the hard drive, by putting the requests in the request queue for the low-level device drivers to handle. It identifies the low-level device driver by a major and a minor number that are associated with

the request. The ODSC will use the major and minor numbers to determine if the request belongs to a device that is currently being traced and cached. The following pseudo-code describes how the block driver will be modified to translate requests:

```

create_request(block_number, device){
    if dcm_device_cached(Major, Minor){
        temp = dcm_translate_block(block_number)
        if temp  $\neq$  block_number {
            if request = read {
                read(temp) and return data
            } else {
                write(temp) and continue
            }
        }
    }
    complete_request_normally()
}

```

Dcm\_device\_cached() and dcm\_translate\_block() are implemented in the DCM. Dcm\_device\_cache() returns true if the device is currently being cached. If the requested block is cached, dcm\_translate\_block() returns the cached position, otherwise it returns the original position. If the request is a write, the modifications will write the block to the cache and then continue to write the block to its original position. This write-through policy does not significantly impact system performance because, as mentioned earlier, writes are not critical to system performance due to the disk cache [5].

The IDE modifications occur in the do\_request function. This function sends the requests to the IDE controller. Do\_request() is modified such that, in addition to its original function, it calls dcm\_collect\_trace() with the sector, operation (read or write), and time as parameters.

### 3.1.2 Disk Caching Module

The DCM has three main functions. It must collect the trace from the IDE driver, translate the requests, and manage the cache on the cache partition. The trace collection is done using the same method that Peacock used to collect traces [7]. In order to collect the trace, the DCM allocates a buffer for the trace. When the buffer is filled, another buffer will be allocated. Once a buffer is read, it is deallocated. This creates a buffer that can grow as large as the available memory but also shrinks as it is read. If the trace is continuously being read, no more than two buffers should be allocated at one time. Because the DCM is a module, the trace can be read like a file through the Virtual Filesystem.

To translate requests, the DCM uses a hash table that is indexed by the real block number. If the real block number is not in the hash table (i.e. the block has not been cached), the DCM returns the real block number so the kernel will complete the request normally. If the real block number is in the hash table, it returns the corresponding cached block number.

The DCM is managed through IOCTL calls. Users level programs use IOCTL calls to communicate with device drivers. There are five IOCTL calls to control the DCM to:

- Turn the trace and cache on or off for a specific device.
- Retrieve the physical size of the cache on the hard drive.
- Clear the cache completely.
- Add a sector to the cache, with its corresponding location.
- Write the cache to the cache partition.

The DCD can interact with the DCM through these IOCTL calls. When the DCM is told to clear the cache or to add something to the cache, it invalidates the entire cache. After writing the cache to the cache partition, the cache is revalidated. To write the cache to the cache partition, the DCM copies sectors from the original position to the cache partition and then writes the hash table to the first few

sectors of the cache partition. When the DCM is initialized, it reads the hash table off the cache partition and validates the cache.

### 3.1.3 Disk Caching Daemon

The DCD reads the trace from the DCM, and then instructs the DCM to cache certain sectors. As stated above, the DCD can read the trace by reading the device file for the DCM. It will then analyze the trace to find sequences and determine which sequences should be cached. When the DCD detects that the system is not busy, it will tell the DCM to clear the cache, then add the new sequences to cache, and finally tell the DCM to write the cache to disk. The DCD will save enough space for the hash table at the beginning of the partition.

## 3.2 Experiments and Analysis

In order to demonstrate that the ODSC decreases disk access time and increases system performance, a bash script will be written to load several applications. Application loading is a good benchmark because while loading an application, 85% of the activity is consumed by disk I/O [6]. The script will be timed without the ODSC, and disk access times will be collected. The machine will then be rebooted to clear the disk cache. After the reboot, the script will then be run again with the ODSC activated to collect a trace. From the trace the DCD will find sequences and copy those sequences into the cache. After another reboot, the script will be timed with the ODSC installed, and new disk access time will be collected. If possible, the same procedure will be done with the actual booting of Linux instead of the bash script. By using different applications and running these experiments several times, enough data will be collected to statistically show that the ODSC reduces disk access time and increase the benchmark performance.

In order to show that main memory size can be reduced while maintaining the same performance, the same procedure above will be used after changing the memory size. Instead of physically removing the memory, the kernel can be modified to decrease the size of the disk cache. This will simulate a machine with less memory. The results should show that with less memory there is a slow down

without the ODSC that does not occur with the ODSC installed.

#### **4 Contribution to Computer Science**

This work will provide two main contributions to Computer Science. First, it will provide a tool that physically copies disk sectors after analyzing disk traces. This tool is modular enough to allow for different caching schemes to be implemented, which provides a basis for future research in the area of on-disk caches. Second, it will demonstrate that system performance can be increased by trading disk capacity for speed.

#### **5 Delimitations of the Thesis**

Although applicable to other operating systems and devices, this work will only use the Linux 2.4.26 kernel and IDE bus. This work will not compare different devices or operating systems. Analysis of different caching schemes, sizes of disk-cache space, and other variables will be left for future work. This work will only use one caching scheme with a fixed-size cache partition to show that ODSCs improve performance. Although it is assumed that this work will increase overall system performance, this work will only show that performance is increased during application loading.

#### **6 Thesis Outline**

- Chapter 1 - Introduction (2 Pages)
- Chapter 2 - Foundational Material and Related Work (13 Pages)
  - Foundational Material (10 pages)
    - \* Disk Drive Performance Fundamentals
    - \* Dynamic Sequence Detection
    - \* Caching Schemes
  - Related Work (3 pages)
    - \* Disk Defragmenters

- \* An Analytical Approach to File Prefetching[8]
  - \* Adaptive Block Rearrangement[2]
  - \* Dynamic Detection of Disk Access Patterns[7]
  - \* A System-Assisted Disk I/O Simulation Technique[4]
  - \* Reducing Application Load Time by Rearranging Data[6]
- Chapter 3 - The On Disk Sequence Cache (20 pages)
    - Overview
    - Trace Collection
    - Disk Request Translation
    - Kernel Modifications
    - Disk Caching Module
    - Disk Caching Daemon
  - Chapter - 4 Experiments and Results (15 pages)
    - Overview
    - Application Loading
    - Linux Booting
    - Reducing Memory
  - Chapter - 5 Conclusion and Future Work (2 pages)

## 7 Thesis Schedule

- Continue literature search (October 2004)
- Complete ODSC implementation (October - November 2004)
- Run experiments and collect data (November - December 2004)
- Analyze results and draw conclusions (December 2004 - January 2005)

- Write Thesis (January - February 2005)
- Defend Thesis (End of March 2005)

## 8 Artifacts

- An implemented ODSC using the Linux 2.4.26 kernel
- Disk traces of application loading
- Disk traces of Linux booting

## References

- [1] Sedat Akyurek and Kenneth Salem. Adaptive block rearrangement under UNIX. *Software - Practice and Experience*, 27(1):1–23, 1997.
- [2] S. Akyurek and K. Salem. Adaptive block rearrangement. *ACM Transactions on Computer Science*, 13:89–121, May 1995.
- [3] Spencer W. Ng. Advances in disk technology: performance issues. *IEEE Computer*, pages 75–81, May 1998.
- [4] Heng Zhou. A system-assisted disk I/O simulation technique. Master’s thesis, Brigham Young University, December 1998.
- [5] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O’Reilly, second edition, 2002.
- [6] Nianlong Yin. Reducing application load time by rearranging disk data. Master’s thesis, Brigham Young University, August 1998.
- [7] Alen Peacock. Dynamic detection of deterministic disk access patterns. Master’s thesis, Brigham Young University, April 2001.
- [8] Hui Lei and Dan Duchamp. An analytical approach to file prefetching. In *1997 USENIX Annual Technical Conference*, Anaheim, California, USA, 1997.

- [9] Alessandro Rubini and Jonathan Corbet. *Linux Device Drivers*. O'Reilly, second edition, 2001.
- [10] Chris Rummeler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, pages 17–29, March 1994.

This thesis proposal by Christopher R. Slade is accepted in its present form by the Department of Computer Science of Brigham Young University as satisfying the thesis proposal requirement for the degree of Master of Science.

---

J. Kelly Flanagan, Committee Chair

---

Eric G. Mercer, Committee Member

---

Christophe Giraud-Carrier, Committee Member

---

David W. Embley, Graduate Coordinator