The Design and Implementation of a DCD Device Driver for Unix *

Tycho Nightingale†, Yiming Hu‡, and Qing Yang†

† *Dept. of Electrical & Computer Engineering*
University of Rhode Island
Kingston, RI 02881
{tycho,qyang}@ele.uri.edu

‡ *Dept. of Ele. & Comp. Eng. and Comp. Sci.*
University of Cincinnati
Cincinnati, OH 45221
yhu@ececs.uc.edu

Abstract

Recent research results [1, 2] using simulation have demonstrated that *Disk Caching Disk* (*DCD*), a new disk I/O architecture, has the potential for drastically improving disk write performance besides its higher reliability than traditional disk systems. To validate whether DCD can live up to its promise in the real world environment, we have designed and implemented a DCD device driver for the Sun's Solaris operating system. Measured performance results are very promising. For metadata intensive benchmarks, our DCD driver outperforms the traditional system by a factor of 2–6 in terms of program execution speeds. The driver also guarantees file system integrity in the events of system crashes or failures. Moreover, unlike other approaches such as Log-structured File Systems or Soft Updates, the DCD driver is completely transparent to the OS. It does not require any changes to the OS or the on-disk data layout. As a result, it can be used as a "drop-in" replacement for the traditional disk device driver in an existing system to obtain immediate performance improvement. Our *multi-layered device-driver* approach significantly reduces the implementation overhead and improves portability.

1 Introduction

Most Unix file systems use synchronous writes for metadata updates to ensure proper ordering of changes being written into disks [3, 4]. The orderly updating is necessary to ensure file system recoverability and data integrity after system failures. However, it is well known that using synchronous writes is very expensive because it forces the updates to be processed at disk speeds rather than processor/memory speeds [3]. As a result, synchronous writes significantly limit the overall system performance.

Many techniques have been proposed to speed up synchronous writes. Among them the *Log-structured File System* (LFS) [5, 4, 6] and *metadata logging* [7] are two well-known examples. In LFS, the only data structure on the disk is a log. All writes are first buffered in a RAM segment and logged to the disk when the segment is full. Since LFS converts many small writes into a few large log writes, the overhead associated with disk seeking and rotational latency is significantly reduced. In metadata logging, only the changes of metadata are logged, the on-disk data structure is left unchanged so the implementation is simpler than that of LFS. Both LFS and metadata logging provide higher performance and quicker crash-recovery ability than traditional file systems.

Soft Updates [3] converts the synchronous writes of metadata to delayed writes to achieve high performance. Soft Updates maintains dependency information of metadata in its kernel memory structures. The dependency information is used when an updated dirty block is flushed to a disk to make sure that data in the disk is in a consistent state. Ganger and Patt have shown that Soft Updates can significantly improve the performance of benchmarks that are metadata update intensive, when compared to the conventional approaches. Soft Updates requires only moderate changes to the OS kernel (Ganger and Patt's implementation on the SVR4 Unix MP system consists of 1500 lines of C code). In addition, it does not require changes to the on-disk data structures.

LFS, metadata logging, and Soft Updates all have good performance. However, they are not without limitations. For example, LFS and metadata logging require redesigning of a significant portion of the file system. Soft Updates also needs some modifications to the OS kernel source code which is not easily accessible to many researchers. Partly due to these limitations, LFS, metadata logging and Soft Updates

*This research is supported in part by National Science Foundation under Grants MIP-9505601 and MIP-9714370.

are not available on many commonly used Unix systems. Moreover, the implementations of LFS, metadata logging and Soft Updates are intrinsically tied to kernel data structures such as the buffer cache and the virtual memory system. They often become broken and require re-implementations when the kernel internal data-structures are changed with OS upgrading. For example, Seltzer et al. implemented an LFS for 4.4BSD [4]; McKusick implemented Soft Updates for FreeBSD, a descendant of the 4.4BSD system. Both the LFS and the Soft Updates code have been broken as a result of the evolution of FreeBSD. While the Soft Updates code is again working in FreeBSD 3.1, the latest stable version as of this writing, the LFS code is not. In fact, the FreeBSD team has decided to drop the effort to fix the LFS code because of the lack of human resources, and the LFS code has even been removed from the FreeBSD CVS repository.

In this research we used another approach that avoids many of the above limitations. Our approach is based on a new hierarchical disk I/O architecture called *DCD (Disk Caching Disk)* that we have recently invented [1, 2, 8]. DCD converts small requests into large ones before writing data to the disk. A similar approach has been successfully used in database systems [9]. Simulation results show that DCD has the potential for drastically improving write performance (for both synchronous and asynchronous writes) with very low additional cost. We have designed and implemented a DCD device driver for Sun's Solaris operating system. Measured performance results show that the DCD driver runs dramatically faster than the traditional system while ensuring file system integrity in the events of system crashes. For benchmarks that are metadata update intensive, our DCD driver outperforms the traditional system by a factor of 2–6 in terms of program execution speeds. Moreover, the DCD driver is completely transparent to the OS. It does not require any changes to the OS. It does not need accesses to the kernel source code. And it does not change the on-disk data layout. As a result, it can be used as a "drop-in" replacement of the traditional disk device driver in an existing system to obtain immediate performance improvement.

The paper is organized as follows. The next section presents the overview of the DCD concept and the system architecture, followed by the detailed descriptions of the design and operations of the DCD device driver in Section 3. Section 4 discusses our benchmark programs and the measured results. We conclude the paper in Section 5.

2 Disk Caching Disks

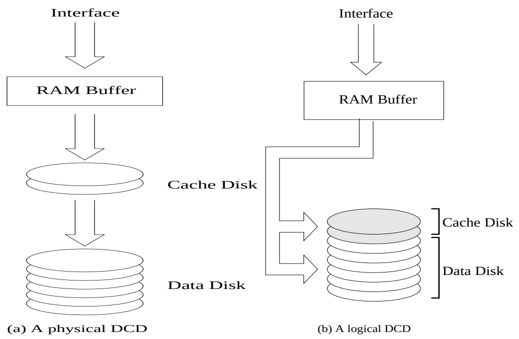

Figure 1: The structure of DCD

Figure 1 shows the general structures of DCD. The fundamental idea behind DCD is to use a small log disk, called *cache-disk*, as an extension of a small NVRAM (Non-Volatile RAM) buffer on top of a *data-disk* where a normal file system exists. The NVRAM buffer and the cache-disk together form a two-level cache hierarchy to cache write data. Small write requests are first collected in the small NVRAM buffer. When the NVRAM buffer becomes full, the DCD writes all the data blocks in the RAM buffer, *in one large data transfer*, into the cache-disk. This large write finishes quickly since it requires only one seek instead of tens of seeks. As a result, the NVRAM buffer is very quickly made available again to accept new incoming requests. The two-level cache appears to the host as a large virtual NVRAM cache with a size close to the size of the cache-disk. When the data-disk is idle or less busy, it performs *destaging* operations which transfer data from the cache-disk to the data-disk. The destaging overhead is quite low because most data in the cache-disk are short-lived and are quickly overwritten, therefore requiring no destaging at all. Moreover, many systems, such as those used in office/engineering environments, have sufficient long idle periods between bursts of requests. For example, Blackwell et al. [10] found that 97% of all LFS cleaning could be done in the background on the most heavily loaded system they studied. Similarly, DCD destaging can be performed in the idle periods, therefore will not interfere with normal user operations at all.

Since the cache in DCD is a disk with a capacity much larger than a normal NVRAM cache, it can achieve very high performance with much less cost. It is also non-volatile and thus highly reliable.

The cache-disk in DCD can be a separate physical disk drive to achieve high performance (a *physical*

DCD) as shown in Figure 1(a), or one logical disk partition physically residing on one disk drive for cost effectiveness (a *logical DCD*) as shown in Figure 1(b). In this paper we concentrate on implementing the logical DCD, which does not need a dedicated cache-disk. Rather, a partition of the existing data-disk is used as a cache-disk. In addition, the RAM buffer shown in Figure 1(b) is implemented using a small portion of the system RAM. Therefore this implementation is complete software without any extra hardware.

One very important fact is that the cache-disk of DCD is only a cache that is completely transparent to the file system. Hu and Yang pointed out that DCD can be implemented at the device or device driver level [1], which has many significant advantages. Since a device driver is well-isolated from the rest of the kernel, there is no need to modify the OS and no need to access the kernel source code. In addition, since the device driver does not access kernel data structures, it is less likely that any kernel changes will require a re-implementation of the driver. Furthermore, since DCD is transparent to the file system and does not require any changes to the on-disk data layout, it can easily replace a current driver without affecting the data already stored on the disk (as long as a free cache-disk partition somewhere in the system can be provided). This is very important to many users with large amount of installed data.

DCD can speed up both synchronous writes and asynchronous writes. Although asynchronous writes normally do not have a direct impact on the system performance, they often affect the system performance indirectly. For example, most Unix systems flush dirty data in the file system cache to disks every 30–60 seconds for reliability reasons, creating a large burst of writes [1]. This large burst will interfere with the normal read and write activities of users. DCD can avoid this problem by using a large and inexpensive cache-disk to quickly absorb the large write burst, destaging the data to the data-disk only when the system is idle.

Using results of trace-driven simulations, Hu and Yang have demonstrated that DCD has superior performance compared to traditional disk architectures in terms of I/O response times [1]. However, recent studies [12] have shown that I/O response times are not always a good performance indicator in I/O systems because not all I/O requests affect the system behavior in the same way. Trace-driven simulations also have limited accuracy because there is no feedback between the traces and the simulators. The main objective of this research is to examine whether DCD will have good performance in the real world. We believe that the best way to evaluate the real world performance is to actually implement the DCD architecture and measure its performance in terms of program execution times, using some realistic benchmarks.

[1] Some systems use a rolling update policy [11] to avoid this problem.

3 Design and Implementation

This section describes the data structures and algorithms used to implement our DCD device driver. Please note that while we have adopted the DCD architecture from [1], our implementation algorithms are considerably different from the ones reported in [1]. We choose these new algorithms mostly because they offer better performance and simplified designs. In some cases a new algorithm was chosen because of the different design goals of the DCD device driver and the original DCD. For example, the original DCD uses an NVRAM buffer and it reorders data in the NVRAM buffer to obtain better performance. However, when implementing the DCD driver, we have to use the system DRAM as the buffer. We can not reorder data in the DRAM buffer before the data is written to the disk, otherwise we can not maintain the order of updates to the disk. Keeping the order has some impact on the performance, but it is the price that has to be paid for maintaining the file system integrity.

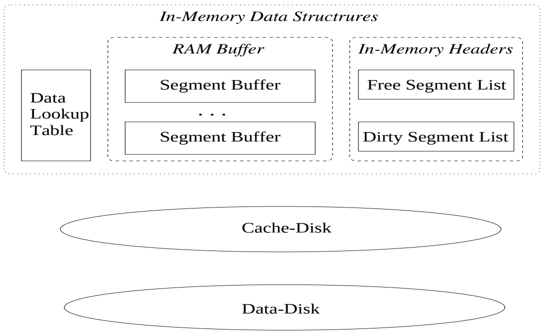

Figure 2: The Structure of the DCD Driver

Figure 2 shows the general structure of the DCD driver which consists of three main parts: the *in-memory data-structures*, a *cache-disk* and a *data-disk*. The In-memory data-structures include a *Data Lookup Table* which is used to locate data in the cache; a *RAM buffer* which comprises several *Segment Buffers* and a number of *In-memory Segment-Headers*. The *In-memory Segment-Headers* form two lists: the *Free Segment List* and the *Dirty Segment List*. The following subsections discuss these structures and the DCD operations in detail.

3.1 Cache-disk Organization

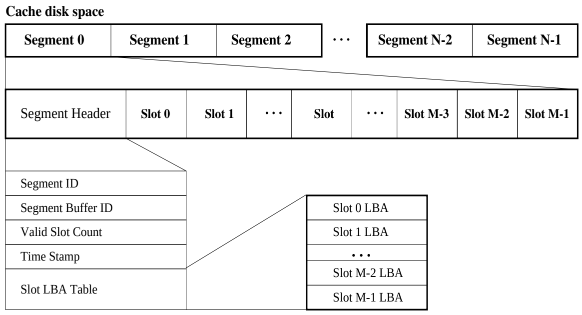

Figure 3: Cache-disk Organization

The original DCD in [1] writes variable-length logs into the cache-disk. However, when we tried to actually implement a DCD device driver, we found that variable-length logs make the destaging algorithm too complex, since it is difficult to locate a particular log in the cache disk. As a result, in the DCD device driver we use a fixed-length segment approach similar to the one used by LFS.

Figure 3 shows the data organization of the cache disk. The entire cache-disk is divided into a number of fixed-length *segments*. The data is written into the cache disk in fixed-size transfers equal to the segment-size, although the segment may be partially full. Each segment contains a *segment-header* followed by a fixed-number of data *slots*. A slot is the basic caching unit, and in our current implementation, can store 1 KB of data. The entire segment size is 128 KB plus two additional 512 byte blocks, used for the segment-header.

The segment-header describes the contents of a segment. It contains:

1. A *segment-ID* which is unique to each segment.
2. A *segment-buffer-ID* whose functions will be discussed later.
3. A *valid-slot-count* that indicates the number of slots in the segment containing valid data. This entry is used to speed up the destaging process.
4. A *time-stamp* which records the time when the segment is written into the cache-disk. During a crash recovery period, the time-stamps help the DCD driver to search for segments and to rebuild its in-memory data structures.
5. A *slot-LBA-table*. Each slot in the segment has a corresponding entry in the table. Each table entry is an integer that describes the *logical block address (LBA)* of the data stored in its slot. For example, if a data block written into block No. 12345 (i.e., its LBA) of the data-disk is currently cached in slot 1, then entry 1 of the table contains 12345 to indicate that slot 1 caches this particular data block. This information is needed since eventually data in the cache-disk will be destaged to their original addresses on the data-disk. If a slot in the segment is empty or contains invalid data, its entry in the slot-LBA-table is -1.

3.2 In-memory Segment-Headers

Once a segment is written into the cache-disk, its contents, including the segment-header (*on-disk segment-header*) are fixed. Furthermore, the on-disk segment-headers in the cache-disk are not accessed during the normal operations of the driver, but are used during crash-recovery. However, subsequent requests may overwrite data contained in the segment on the cache-disk, requiring updating of the segment-headers. For example, if a slot of an on-disk segment is overwritten, its corresponding *slot-LBA-entry* should be marked invalid, with a -1, to indicate that the slot no longer contains valid data. Additionally the *valid-slot-count* should be decreased by 1. It is prohibitively costly to update the on-disk segment-header for every such overwrite, therefore each segment-header has an in-memory copy (*in-memory segment-header*). Once a segment is written into the cache-disk, only the in-memory segment-header is updated upon overwriting. The on-disk segment-header remains untouched. If the system crashes, the in-memory segment-headers can be rebuilt by scanning the cache-disk and "replaying" the on-disk segment-headers in order of their time-stamps. Since the on-disk segment-headers have fixed locations on the disk, this scanning process can be done very quickly. We will discuss the crash recovery process in detail later in this section.

An in-memory segment-header is always in one of the following two lists: the *Free-Segment-List* and the *Dirty-Segment-List*. If a segment does not contain any valid data, its in-memory segment-header is in the Free-Segment-List. Otherwise it is put in the Dirty-Segment-List. When the system is idle, the destaging process picks up segments from the Dirty-Segment-List and moves valid data in these segments to the data disk. Once all data in a dirty segment is destaged, its header is moved to the Free-Segment-

List. Segments which are entirely overwritten during normal operation, i.e. the valid-slot-count becomes 0, are also put on the Free-Segment-List.

In our current prototype, the cache-disk is 64 MB and has about 512 segments (128 KB per segment). The size of each in-memory segment-header is 512 bytes. In total, 512 segment-headers require only 250 KB of RAM. Given today's low RAM cost, we put the entire 250 KB of headers in the system RAM so they can be accessed quickly.

3.3 Data Lookup Table

One of the most challenging tasks in this research is to design an efficient data structure and a search algorithm for the *Data Lookup Table*, used for locating data in the cache. In DCD, a previously written data block may exist in one of the following three places: the *RAM buffer*, the *cache-disk* and the *data-disk*. The Data Lookup Table is used to keep track of the location of data in these places. Since the driver must search the table for every incoming request, it is imperative to make the searching algorithm efficient.

In other I/O systems with a similar problem, a hash table with the LBA (Logical Block Address) of incoming requests as the search key, is usually used for this purpose. Unfortunately this simple approach does not work for the DCD driver. Since in Solaris and many other Unix systems, the requests sent to the disk device driver have variable lengths and may be overlapped. For example, suppose a data block B cached in the DCD has a starting LBA of S and a length of L. Another incoming request asks for a data block B' with a starting LBA of $S+1$ and a length of L'. If $L > 1$, then B and B' overlap, and the system should be able to detect the overlapping. Since B and B' have different LBAs, simply using the LBA of B' as the search key will not be able to find block B in the hash table.

Our profiling of I/O requests in Solaris demonstrated that the majority of requests are aligned to 4 KB or 8 KB boundaries and are not overlapped. Additionally, all requests were determined to be aligned to a 1 KB boundary. Therefore the Data Lookup Table was designed to efficiently handle the common cases (i.e., aligned un-overlapped 4 or 8 KB requests), while still able to handle other requests.

The minimum storage and addressing unit of the device driver was chosen to be a *cache fragment*, of size 1 KB and aligned to a 1 KB boundary. The fragments are designed to fit into the slots in the cache-disk segments. Since they are aligned and not overlapped they should be easy to find in the system. Although a fragment is the minimum unit used in the driver, most requests occupy several consecutive fragments. It is wasteful to break every incoming request into a series of fragments, and search the hash table for each fragment. Therefore a *cache line* was designed to hold 4 fragments or 4 KB of data aligned to a 4 KB boundary.

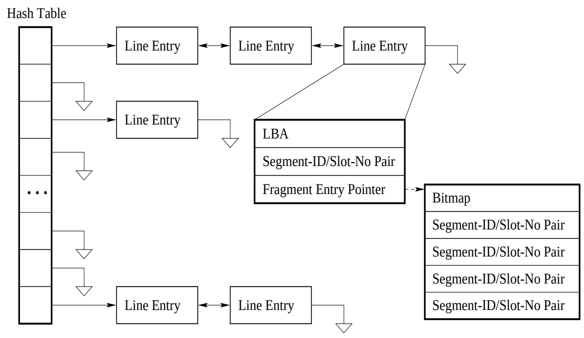

Figure 4: Data Lookup Table

The Data Lookup Table is in the form of a hash table chain, as is shown in Figure 4. Each entry in the Data Lookup Table, *line entry*, contains information associated with the blocks currently in the DCD. Each incoming request is mapped into one or more cache lines. The Data Lookup Table is then searched, using the LBA of the start of each cache line in the request as a search key. All entries in the hash table with the same hash value are linked together in a doubly-linked list.

A Data Lookup Table line entry consists of the following fields:

1. An *LBA* entry that is the LBA of the cache line.

2. A *segment-ID/slot-no* pair which uniquely determines the starting address of the data in the cache-disk. Although data cached in the DCD can be either in the RAM buffer or on the cache-disk, a unified addressing scheme is used. The segment-ID/slot-no pair are also used to address data in the RAM buffer. This unified scheme, discussed in more detail later, is used to reduce to bookkeeping overheard involved when data buffered in the RAM is written into the cache-disk.

3. An optional *fragment entry* pointer used to describe partial or fragmented cache entries.

While most I/O requests will map to one or two complete cache line entries, some requests are for *partial lines*. Also, overwriting can cause caches lines to be broken. For example, suppose a cache line has

4 consecutive fragments stored in segment S. If another request overwrites the second fragment in segment S and stores the new data in segment S', then the cache line becomes a *fragmented line* with data stored in several different places: the first fragment is in S; the second one in S'; and the remaining two fragments in S. To be able to handle both of these cases, the line entry, used in the Data Lookup Table, contains an optional pointer to a *fragment entry*, which describes both partial lines and fragmented lines.

Fragment entries contain the following fields:

1. A *bitmap* which marks the fragments of a cache line contain that valid data. Recall that a cache-line can contain 4 fragments, of 1 KB each, stored in 4 slots. If for example, a request contains only 2 KB, starting on a 4 KB boundary, then only the first two fragments of line are valid yielding a bitmap of 1100.

2. A table containing four possible *segment-ID/slot-no* pair entries which identify the locations of each fragment. The bitmap is used to determine which entries in the table are valid.

For every incoming request the Data Lookup Table is searched one or more times. If an entry is found in the Data Lookup Table (a cache hit), the requested block is contained somewhere in the DCD, either in the RAM buffer, or in the cache-disk. Otherwise if no corresponding entry is located (a cache miss), the requested block can be found on the data-disk.

Our I/O profiling results, of the prototype driver, show that most requests (50-90%) are contained in only one or two complete cache lines. The remaining requests need to access fragmented cache lines. Therefore the Data Lookup table is both space and time efficient.

3.4 Segment Buffers

Another important component of DCD is the RAM buffer. In our implementation the RAM buffer is divided into several (2–4) *Segment Buffers* which have the same structure as on-disk segments without segment-headers. Each Segment Buffer is assigned a unique *segment-buffer ID*.

Segment Buffers, much like their on-disk counterparts, are tracked using two lists: the *Free Segment-Buffer List* and the *Dirty Segment-Buffer List*, when not mapped to a segment. When a write request arrives, the driver picks a free segment buffer to become the *Current Segment-Buffer*. Meanwhile the driver also obtains a free disk segment from the Free Segment List to become the *Current Disk Segment*. The driver then "pairs" the RAM segment buffer and the disk segment together by writing the *segment-buffer ID* into the corresponding field into the header of the in-memory header of the disk segment.

From this point until the current Segment Buffer is written into the Current Disk Segment on the cache-disk, incoming write data is written into the slots of the segment buffer. However the Data Lookup Table will use the segment-ID/slot-no pairs of the Current Disk Segment as the addresses of these data (the unified addressing scheme). In addition, the RAM buffer does not have its own header. It shares the header of the disk segment. Imagine that the segment buffer "overlays" itself on top of the disk segment, as shown in Figure 5(a), therefore any data written to the slots of the Current Disk Segment is actually written into the slots of the Current Segment Buffer.

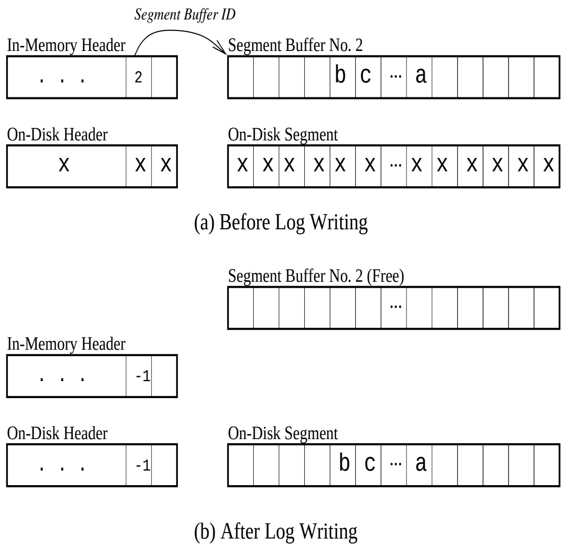

Figure 5: Segment Buffers and Disk Segments

If a read request tries to access a newly written data in the segment buffer before it is logged into the cache-disk, the Data Lookup Table will return a segment-ID/slot-no pair to indicate that the requested data is in a particular segment. However, the driver will know from the *segment-buffer ID* entry in the segment-header that the data are actually in the RAM buffer instead of in the on-disk segment.

When the current segment buffer becomes full, the driver schedules a log write so that the entire segment buffer is written into the Current Disk Segment on the cache-disk. Meanwhile another segment-buffer/disk-segment pair is chosen to accept new incoming requests. When the log write finishes, the driver then changes the segment-buffer ID field in the in-memory segment-header to -1 to indicate that all the data is now in the cache-disk, as shown in Figure 5(b). There is no need to update all of the Data Lookup Table entries for all the blocks in the segment. Finally the segment buffer is freed and placed on the Free Segment Buffer List, and the in-memory segment-header is placed on the Dirty Segment List waiting to be destaged.

3.5 Operations

3.5.1 Writes

After receiving a write request, the DCD driver first searches the Data Lookup Table and invalidates any entries that are overwritten by the incoming request. Next the driver checks to see if segment buffer/disk segment pair exists, if not, a Free Segment Buffer and Free Disk Segment are paired together. Then all of the data in the request is copied into the segment buffer, and their addresses are recorded as entries in the Data Lookup Table. Finally, the driver signals the kernel that the request is complete, even though the data has not been written into the disk. This *immediate-report* scheme does not cause any reliability or file system integrity problems, as will be discussed shortly in this section.

There are two cases where data written into the DCD is handled in a different manner. First, to reduce destage overhead, requests 64 KB or larger are written directly into the data-disk. Second, if a user request sets the "sync" flag in an I/O request, the request is also written directly into the data disk. Only after the requests finishes writing, does the driver signal completion to the kernel.

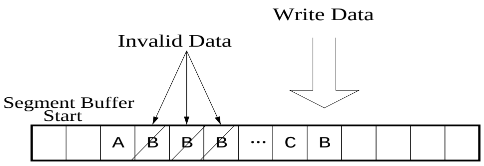

Figure 6: Overwriting in a Segment Buffer

When data is copied into the segment buffer, it is always "appended" to the previous data in the buffer, even when the new data overwrites previously written data. For example, Figure 6 shown a segment buffer containing blocks A, B and C. Block B has been overwritten several times. Instead of overwriting the old block B in the segment buffer, the new data of block B is simply appended to the segment buffer. The old data of block B is still in the buffer, although it becomes inaccessible (hence garbage). The *append-only* policy ensures that the ordering relationship between requests is maintained, and correctly recorded in the cache-disk. If the new copy of block B were to overwrite the old copy of the block B in the segment buffer, then order of updates between B and C would not have been preserved.

Although the append-only policy is necessary, it also has some drawbacks. Specifically, it increases the logging and destaging overhead since much garbage is logged into the cache-disk. One possible solution is to "compact" the segment buffer prior to writing it to the cache-disk. However the compacting process is a CPU-intensive task since it involves moving large amount of data in RAM. We have not tried this in our current implementation.

3.5.2 Reads

In DCD, data may be in one of three different places: the RAM buffer, the cache-disk and the data-disk. Conceptually read operations are simple; the driver just needs to find the locations of the requested data, get the data and return them to the host.

However, when we actually started the design and implementation, we found that reads are quite complex. Because of fragmentation, the contents of a requested data block may be scattered in all of the three places. The driver has to collect all data from all these places and assemble them together. This is not only complicated, but time-consuming.

Similar to writes, when a read request is received, the DCD driver searches the Data Lookup Table for entries. Most requests can not be found in the table, because newly written data is also cached in the file system. Therefore the system does not need to reread them. On the other hand the older written data has very likely been moved to the data-disk by destaging. Hu and Yang found in [1], that 99% of all read must be sent to the data disk. The simulations and experimental results confirm this; reads from other places seldom occur. Since virtually all reads are sent to the data-disk which has the same on-disk data layout as the traditional disk, DCD will at least have similar performance compared to a traditional disk [2].

For the majority of read requests, the DCD driver simply forwards them to the data-disk. For the remaining 1% of requests, if all the data are stored in the RAM buffer or consecutively on the cache-disk, the operation is also simple. However, for a

[2] In fact, compared to a traditional disk, DCD has better read performance because it removes the write traffic from the critical path therefore reducing the disk contention.

very small percentage of requests that access data scattered all over the system, the operation could be quite complex. After evaluating several alternatives, we adopted a simple solution: The driver first calls the *foreground destage* routine to move all the data related to the requests to the data-disk, then forwards the requests to the data-disk to fetch the data. While this approach may not be the most efficient one, it greatly simplifies our implementation. In addition, since such cases occur infrequently, the benefits of simple implementation override the low efficiency, and performance is not hurt noticeably.

3.5.3 Destages

In DCD, data written into the cache-disk must eventually be moved into the original location on the data-disk. This process is called *destaging.* The destage process starts when the DCD driver detects an idle period, a span of time during which there is no request. The threshold of the idle period (any idle period longer than the threshold will start the destage process) is a function of the cache-disk utilization. If the cache-disk is mostly empty, then the threshold can be as long as one second. However, if the cache-disk is mostly full, the threshold can be as short as 1 ms, to quickly empty the cache-disk, and prepare for the next burst of requests. Once the destage process starts, it will continue until the cache-disk becomes empty, or until a new request arrives. In the later case the destaging is suspended, until the driver detects another idle period.

The driver uses a "last-write-first-destage algorithm" [1], where the most recently written segment (the first one on the dirty segment list) is destaged first. While this algorithm is very straightforward, and works well, it may result in unnecessary destaging work since it always destages newly written data first. An optimization to this algorithm would be to use the segment-header's *valid-slot-count* to select a segment, for destage, which contains the least amount of valid data. Therefore reducing the number of writes needed to move the data from cache-disk to the data-disk. Also, when writing data blocks from the destage buffer to the data-disk, the driver does not try to reorder the writes. A possible optimization here is to reorder the blocks according their locations on the data disk to minimize the seek overhead. We will try these optimizations in the future version of the driver.

3.5.4 Crash Recovery

The crash recovery process of a system using DCD can be divided into two stages. In the first stage the DCD tries to recover itself to a stable state. In the second stage the file system performs normal crash-recovery activities.

The crash recovery of the DCD driver is relatively simple. Since cached-data is saved reliably to the cache-disk, only the in-memory data-structures, such as the *Data Lookup Table* and the *In-Memory Segment-Headers*, need to be reconstructed. We have designed the crash-recovery algorithm, but since it is not part of the critical path of the driver, this has not been been implemented in the current prototype.

To rebuild the in-memory data structures, the DCD driver can first read all of the *On-Disk Segment-Headers* from the cache-disk into RAM. For a cache-disk of 64 MB, only 512 headers need to be read. Since the cache-disk locations of the headers are known, this should not take too long. The headers in RAM are organized in a list ordered ascendingly according to their time-stamps. Starting with the first header on the list, the headers can be used in temporal order, to reconstruct the *Data Lookup Table* and the *In-Memory Segment-Headers.* Each time a new header is encountered, entries in the *Data Lookup Table* and *In-Memory Segment-Headers* may be overwritten, because data in the second segment may overwrite data in the earlier segment. This process continues until all the headers are processed. Essentially, all the cache-disk headers can be used to "replay" the entire history of requests captured on the cache-disk. At this point, the DCD driver returns to a clean and stable state, and the file system can start a normal crash-recovery process, such as running the "fsck" program.

3.6 Reliability and File System Integrity

DCD uses immediate report mode for most writes, except for those requests with the "sync" flags set by user applications. Write requests are returned from the driver once the data are written into the RAM buffer. This scheme does not cause a reliability problem for the following two reasons.

First, the delay caused by the DCD driver is limited to several hundreds of milliseconds at most. Once the previous log writing finishes (which may take up to several hundreds of milliseconds), the driver will write the current segment buffer to the cache-disk even before the segment buffer becomes full. Therefore the DCD driver guarantees that data be written in to the cache-disk within several hundreds of milliseconds. Since most file data is cached in RAM for 30–60 seconds before they are flushed into the disk anyway, we believe that the additional several hundreds of milliseconds should not result in any problem.

Second, for metadata updates, the "append-only" algorithm of the DCD driver guarantees the order of updates to the disk. So while metadata updates may be kept in RAM for up to several hundred milliseconds, they are guaranteed to be written into the disk in the proper order to ensure file system integrity. In this sense DCD is similar to other solutions such as LFS and Soft Updates, which also store metadata updates in RAM for even longer time but are able to maintain file system integrity.

3.7 Layered Device Driver Approach

One potential difficulty of the DCD device driver is that it may have to communicate with disk hardware directly, resulting in high complexity and poor portability. We solved this problem by using a *multi-layered device driver* approach. We implemented our DCD driver on top of a traditional disk device driver (End users specify which traditional disk driver will be used). The DCD driver calls the low-level disk driver, through the standard device driver interface, to perform the actual disk I/Os. This approach has three major advantages. First, it greatly simplifies the implementation efforts since the DCD driver avoids the complex task of direct communication with the hardware [3]. Second, the same DCD driver works with all kinds of disks in a system because all the low-level disk device drivers use the same standard interface. Third, it is easy to port our current implementation to other systems, since most Unix systems use similar device driver interfaces.

4 Performance Results

4.1 Benchmarks and Experimental Setup

To examine the performance of the experimental systems, we need some benchmark programs. Initially we have planed to use the Andrew benchmark [13]. However we found that the execution time of Andrew benchmark is very short (less than several seconds for most phases in the benchmark) on most modern machines, resulting in significant measurement errors. Also the benchmark does not exercise the disk system sufficiently in machines with reasonable amount of RAM, since most of its data can be cached in RAM.

Because of this, we have created a set of our own metadata intensive benchmark programs. Our benchmarks are similar to the ones used by Ganger and Patt in [3]. The benchmarks consist of 3 sets of programs: *N-user untar*, *N-user copy*, and *N-user remove*. *N-user untar* simulates one or more users concurrently and independently *un-taring* a tar file to create a directory tree. *N-user copy* and *N-user remove* simulate one or more users concurrently copying (using the "cp -r" command) or removing (using the "rm -r" command) the entire directory tree created by the *N-user untar* benchmark, respectively. We chose these benchmarks because they represent the typical and realistic I/O intensive tasks performed by normal users on a daily basis.

The particular tar file we used is the source-code distribution tar file of Apache 1.3.3. Apache is the most popular Web server as of this writing. The resulting source file tree contains 31 subdirectories and 508 files with an average file size of 9.9 KB. The total size of the directory tree is 4.9 MB.

To make our evaluation more conservative, we intentionally flush the contents of the kernel buffer cache by first unmounting then remounting the file system between each run. This creates a *cold read cache effect*, forcing the system to read data from the disk again for each benchmark run. Our measured results show that the performance improvement of the DCD driver over the traditional disk driver is much higher (up to 50% higher) when the kernel buffer cache is warm (that is, if we do not flush the contents of the buffer). The cold cache effect results in a large number of read requests that can not be improved significantly by DCD. Because of this artificially introduced cold cache effect, the performance results reported in the section should be viewed as a conservative estimate of DCD performance. Real world DCD performance should be even better, because the buffer cache will be warm most of the time.

All experiments were conducted on a Sun Ultra-1 Model 170 workstation running Solaris 2.6. The machine has 128 MB RAM and a 4 GB, high performance Seagate Barracuda hard disk drive. The system was configured as a Logical DCD with the first 64 MB of the disk used as the cache-disk partition of the DCD; the rest as the data-disk. The RAM buffer size of the DCD driver was 256 KB. The total RAM overhead, including the RAM buffer, the Data Lookup Table, the In-Memory Segment-Headers, etc., is about 750 KB, which is a small portion of the 128 MB system RAM.

The performance of the DCD device driver is com-

[3] Our prototype implementation consists of only 3200 lines of C code. Once we have validated our algorithms with an event-driven simulator, the first author took only two weeks to produce a working device driver.

pared to that of the default disk device driver of Solaris. We used the Unix *timex* command to measure the benchmark execution time, and the user and system CPU times. We also used the built-in kernel tracing facilities of Solaris to obtain detailed low-level performance numbers such as the average response times of read and write requests.

4.2 Measured Performance Results

In this subsection we compare the performance of our DCD device driver to that of the traditional disk device driver of Solaris. All numerical results shown here are the averages of three experimental runs.

4.2.1 Benchmark Execution Times

Figures 7–9 compare the benchmark execution times for the traditional disk device drivers with those of the DCD device driver. The total execution times are further broken down into the *CPU time* and the *waiting time*. The figures clearly show that these benchmarks run dramatically faster using the DCD driver than using the traditional driver. The DCD driver outperforms the traditional driver by a factor of 2–6 in all cases. The most impressive improvements come from the *N-user untar* and the *N-user remove* benchmarks that are very metadata intensive, where DCD runs more than 3–6 times faster than the traditional driver. The *N-user copy* benchmark spends much of its time on disk reads which can not be improved by DCD. Its performance improvement of 2–3 times is not that dramatic but still very impressive. Note that the performance difference is even bigger if we do not artificially flush the cache contents between each run. For example, our results show that the *N-user copy* benchmark also runs 4–5 times faster on the DCD driver than on the traditional driver if we do not flush the cache.

Figures 7–9 clearly indicate that the performance of the traditional driver is severely limited by the disk speed. Its CPU utilization ratio is only about 3–15%, meaning the system spends 85–97% of its time waiting for disk I/Os. While the performance of DCD is still limited by the disk speed, DCD significantly improves the CPU utilization ratio to 16–50%.

We can also see that the DCD driver and the traditional disk driver have similar CPU overheads. This demonstrates that our DCD driver algorithms are very efficient. Although the DCD driver is much more complex than the traditional disk driver, the former increases the CPU overhead only slightly compared to the later.

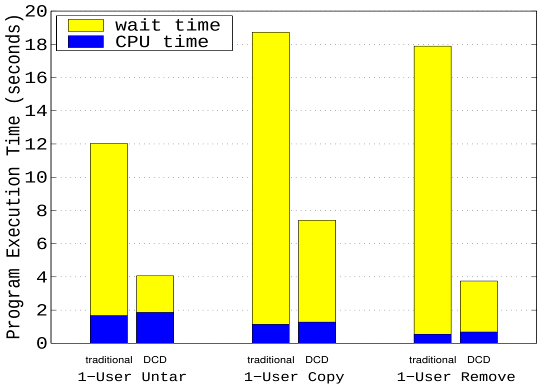

Figure 7: Benchmark execution times for *1-user untar, copy* and *remove*

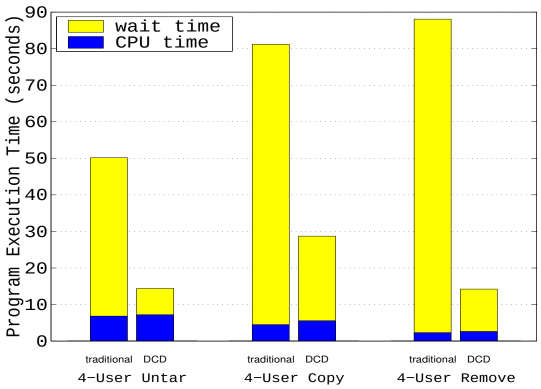

Figure 8: Benchmark execution times for *4-user untar, copy* and *remove*

4.2.2 Maximum Possible Speedup

We have shown that DCD can speed up the metadata intensive benchmarks by a factor of 2–6. To see if we can further improve the performance of DCD and to understand what is the performance limitation, we have measured the raw disk write bandwidth at various request sizes. The bandwidth is obtained by randomly writing 20 MB worth of data to the disk at different sizes, using the raw device driver interface.

Figure 10 shows the measurement results in terms of I/O bandwidth. The raw disk write performance at the 128 KB unit size is about 55 times faster than the 1 KB unit, 28 times faster than the 2 KB unit, 14.3 times faster than the 4 KB unit, and 7.4 times faster than the 8 KB unit sizes.

DCD always writes to the cache disk in the 128 KB segments. When running the *4-user remove* benchmark program on the traditional disk device driver, our profiling results show that about 50%

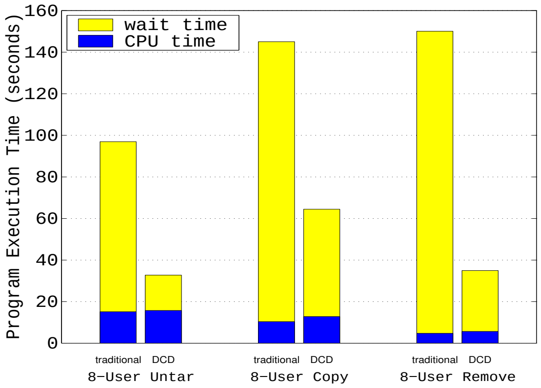

Figure 9: Benchmark execution times for *8-user untar, copy* and *remove*

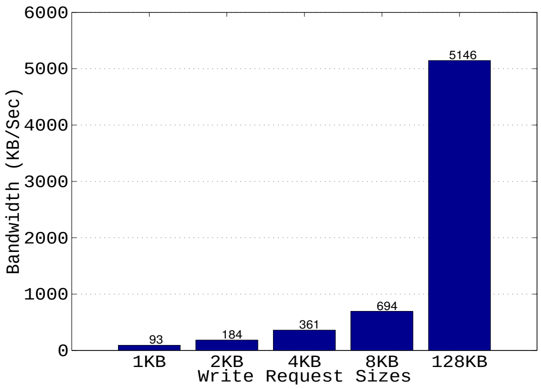

Figure 10: Raw disk write performance of the experimental system

of requests are in 8 KB, 15% in 2 KB and 35% in 1 KB. Therefore the maximum possible speedup of DCD over the traditional disk driver is approximately $7.4 * 50\% + 28 * 15\% + 55 * 35\% = 25.4$. The benchmarks also spend some time on reads and CPU overhead which can not be improved by DCD. For our *4-user remove* benchmark, the reads and CPU overhead account for about 7% of the total execution time on the traditional disk driver. As a result, we expect that the maximum achievable speedup be around $1/(7\% + 93\%/25.4) = 9.4$ without destage.

Our prototype DCD driver achieves a performance improvement of 6.2 times over the traditional device driver for the *4-user remove* benchmark. In other words, the prototype driver realizes two thirds of the potential performance gain. Although a small portion of this loss of gain is due to the overhead of destaging, our analysis indicates that there is clearly still room for further improvement in the implementation. One of the main performance limitations in the current implementation is our locking algorithm. Because of the time limitations, we used a very simple but ineffective locking scheme in the DCD driver for concurrency control. When a request arrives, we exclusively lock the entire cache for the request until the request is complete. This scheme significantly limits parallelism and has a very negative impact on the performance, especially for the multi-user environment. We plan to reduce the lock granularity to increase parallelism in the next version.

4.2.3 I/O Response Times

In order to further understand why the DCD device driver provides such superior performance compared to the traditional disk device driver, we measured the read and write response times of both the DCD and the traditional disk device driver, using the kernel tracing facilities of Solaris.

Figures 11 and 12 show the *histograms* of read and write response times of the DCD driver and the traditional driver. The data was collected while running the *4-user copy* benchmark. In these figures the X-axes are for response times and the Y-axes are for the percentage of requests. Note that the X-axes and Y-axes of the DCD driver and those of the traditional driver have a different scale. The figures for the traditional driver have an X-scale of 0–500 ms, while the figures of the DCD driver have an X-scale of 0–25 ms. Because the DCD driver has much lower response times, the histograms would have be crammed in the left part of the figures and become inscrutable had we used the same X-scale for all the figures.

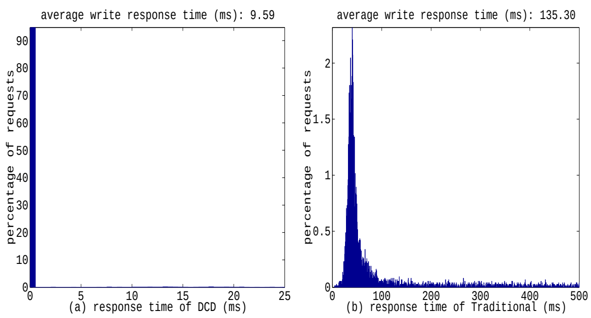

Figure 11: Histograms of Write Response Times of the DCD driver and Traditional Driver. *Virtually all requests here are synchronous.*

As shown in Figure 11, for the traditional disk driver, the response times for the majority of writes fall within the range of 25–150 ms. The average write

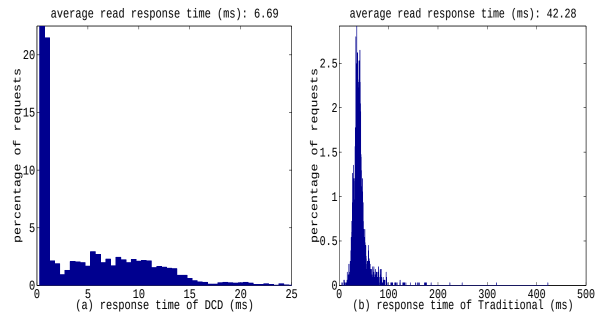

Figure 12: Histograms of Read Response Times of the DCD driver and Traditional Driver

response time is 135.30 ms. Such a long response time can mainly be attributed to the long queuing time caused by disk bandwidth contention among 4 processes.

The DCD driver, on the other hand, significantly reduces the response time of writes. About 95% of writes complete within 1ms, because they complete after being written into the RAM buffer. Another 3% of the writes, barely visible in Figure 11(a), take less than 25 ms. Finally, the remaining 2% of request takes about 25–1500 ms to finish. This happens when the cache-disk can not finish log writing quickly so the incoming requests have to wait. The average write response time of the DCD driver is 9.59 ms [4], which is is less than 1/14 of the response time of the traditional disk driver.

While the DCD driver can not speed up the read requests directly, it significantly reduces the average read response time in two ways. First, since DCD reduces write traffic at the data-disk, the data-disk has more available bandwidth for processing read requests. This eliminates the queuing time caused by disk bandwidth contention and decreases the average read response time. Second, the reduced write traffic at the data-disk allows a more effective use of the built-in hard disk cache. Since fewer writes 'bump' read data, the read-ahead cache hit ratio is significantly higher. This is shown in Figure 12 where approximately 40% of read requests complete within 1 ms because of the read-ahead cache hits. On the other hand, in the traditional driver very few read requests finish within 1 ms. This is because a large number of small writes constantly flush the read-ahead cache, rendering the cache virtually useless. Both the reduced bandwidth contention and the improved cache hit-ratio "shifts" the entire histogram to the far left. The average read response time of DCD is 6.69 ms, which is about 1/7 of the 42.28 ms response time of the traditional disk driver.

[4] This number is surprisingly high, considering the fact that 95% of requests finishes within 1ms. The bias is caused by the 2% of requests that finish within 25–1500 ms.

4.2.4 Destage Overhead

While DCD shows superb performance, for each data write, DCD has to do extra work compared to a traditional disk. It has to write the data into the cache-disk first and read it back from the cache-disk later for destage. This extra overhead may become a performance concern. However, we found that this extra work does not result in extra traffic in the disk system.

In DCD, all writes to and reads from the cache-disk are performed in large segment sizes — typically up to 32 requests can be written into or read from the cache-disk, both in one disk access. Therefore the overhead increases only slightly. Moreover, this small overhead can be compensated by the fact that data can stay in the cache-disk much longer than in RAM because the cache-disk is more reliable and larger than the RAM cache. As a result, the data in the cache-disk has a better chance of being overwritten, reducing the number of destaging operations. Hartman and Ousterhout demonstrated in [14] that between 36%–63% of newly written bytes are overwritten within 30 seconds and 60%–95% within 1000 seconds. Because of this, when compared to a normal disk driver, a DCD driver actually generates *fewer* total disk requests (including requests from the system and applications as well as the internal destaging requests).

5 Conclusion

In this paper we have designed and implemented a Disk Caching Disk (DCD) device driver for the Solaris operating system. Measured performance results are very promising, confirming the simulation results in [1, 2]. For metadata intensive benchmarks, our un-optimized DCD driver prototype outperforms the traditional system by a factor of 2–6 in terms of program execution speeds. The driver also guarantees file system integrity in the events of system crashes or failures. Moreover, unlike previous approaches, the DCD driver is completely transparent to the OS. It does not require any changes to the OS or the on-disk data layout. As a result, it can be used as a "drop-in" replacement for the traditional disk device driver in an existing system to obtain immediate performance improvement.

We plan to make the source code and binary code of our DCD device driver available to the public as soon as possible. We also plan to port the driver to other systems such as FreeBSD or Linux.

Acknowledgments

The authors would like to thank the anonymous referees for their advice on improving this paper, and Jordan Hubbard and Terry Lambert for providing information about the status of LFS and Soft Updates in FreeBSD.

References

[1] Y. Hu and Q. Yang, "DCD—disk caching disk: A new approach for boosting I/O performance," in *Proceedings of the 23rd International Symposium on Computer Architecture*, (Philadelphia, Pennsylvania), pp. 169–178, May 1996.

[2] Y. Hu and Q. Yang, "A new hierarchical disk architecture," *IEEE Micro*, vol. 18, pp. 64–76, November/December 1998.

[3] G. R. Ganger and Y. N. Patt, "Metadata update performance in file systems," in *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pp. 49–60, Nov. 1994.

[4] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin, "An implementation of a log-structured file system for UNIX," in *Proceedings of Winter 1993 USENIX*, (San Diego, CA), pp. 307–326, Jan. 1993.

[5] J. Ousterhout and F. Douglis, "Beating the I/O bottleneck: A case for log-structured file systems," tech. rep., Computer Science Division, Electrical Engineering and Computer Sciences, University of California at Berkeley, Oct. 1988.

[6] M. Rosenblum and J. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems*, pp. 26 – 52, Feb. 1992.

[7] U. Vahalia, *UNIX Internals — The New Frontiers*. Prentice Hall, 1996.

[8] Q. Yang and Y. Hu, "System for destaging data during idle time by transferring to destage buffer, marking segment blank, reordering data in buffer, and transferring to beginning of segment." U.S. Patent No. 5,754,888, May 1998.

[9] K. Elhardt and R. Bayer, "A database cache for high performance and fast restart in database systems," *ACM Transactions on Database Systems*, vol. 9, pp. 503–525, Dec. 1984.

[10] T. Blackwell, J. Harris, and M. Seltzer, "Heuristic cleaning algorithms in log-structured file systems," in *Proceedings of the 1995 USENIX Technical Conference: January 16–20, 1995, New Orleans, Louisiana, USA* (USENIX Association, ed.), (Berkeley, CA, USA), pp. 277–288, USENIX, Jan. 1995.

[11] J. C. Mogul, "A better update policy," in *Proceedings of the Summer 1994 USENIX Conference* (USENIX Association, ed.), (Berkeley, CA, USA), pp. 99–111, USENIX, June 1994.

[12] G. R. Ganger and Y. N. Patt, "Using system-level models to evaluate I/O subsystem designs," *IEEE Transactions on Computers*, pp. 667–678, June 1998.

[13] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, "Scale and performance in a distributed file system," *ACM Transactions on Computer Systems*, vol. 6, pp. 51–81, Feb. 1988.

[14] J. H. Hartman and J. K. Ousterhout, "letter to the editor," *Operating Systems Review*, vol. 27, no. 1, pp. 7–9, 1993.