AWOL: An Adaptive Write Optimizations Layer

Alexandros Batsakis, Randal Burns
The Johns Hopkins University
Arkady Kanevsky, James Lentini, Thomas Talpey
Network Appliance™Inc.

Abstract

Operating system memory managers fail to consider the population of read versus write pages in the buffer pool or outstanding I/O requests when writing dirty pages to disk or network file systems. This leads to bursty I/O patterns, which stall processes reading data and reduce the efficiency of storage. We address these limitations by adaptively allocating memory between write buffering and read caching and by writing dirty pages to disk opportunistically before the operating system submits them for write-back. We implement and evaluate our methods within the Linux® operating system and show performance gains of more than 30% for mixed read/write workloads.

1 Introduction

Scaling trends in processor speed and disk performance (access time and throughput) have brought write performance into the critical path. Traditionally, reads have been considered more important than writes: appropriately given that reads are synchronous and writes are generally asynchronous. We refer to I/Os as synchronous and asynchronous to describe whether the issuing applications or the operating system is blocking awaiting their completion. Most cache-management algorithms focus on directing the population of read pages [11, 12, 17, 20]. However, as processors increase in speed, systems have the ability to create dirty pages at rates well beyond the disk's ability to clean them. Gill et al. [8] point out an annual growth rate of 60% for processors and 8% for disk access time. In such an environment, the synchronous reads depend upon the asynchronous writes, because (1) dirty pages consume memory that is unavailable for read caching and (2) write traffic to clean pages interferes with read requests. In a sense, there is a priority inversion [23] between reads and writes to which we need to apply priority scheduling, preferring reads to writes, and priority inheritance, performing writes that block high priority reads.

The static write and flush policies used by operating system memory managers are insensitive to processes that are actively reading data, the distribution of read versus write pages in the buffer pool, and outstanding I/O requests. This leads to bursty I/O patterns, which both stall other processes reading data and reduce the efficiency of storage. For different workloads, the operating system destages pages either too aggressively or not aggressively enough. We give several examples in Section 2.

Operating system caching is no longer a read-only problem. Operating system memory managers need to be enhanced to balance read and write workloads and to define adaptive destaging policies that are sensitive to workload and the population of read, write and free pages in memory. Recent research has identified the importance of improving write performance. Most of this work addresses write performance independent of reads, e.g. through improved scheduling [8] or separate non-volatile caches [4, 6, 22]. Works that consider reads and writes combined do so in the context of second-tier caches in order to determine what written data are likely to be read again [14].

We define an adaptive framework for destaging dirty pages to disk that reduces the interference of write traffic on read performance and increases the performance of the I/O subsystem. Depending on the workload, it controls the aggressiveness of the destaging policy in order to keep memory available in the page cache and increases disk throughput, while having a minimal impact on cache hit rates. The framework dynamically tunes the allocation of memory between read and write pages. To do so, it employs several techniques:

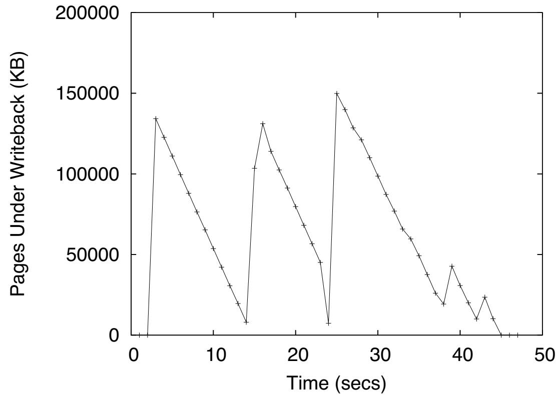

Figure 1: Number of pages under write-back when writing a 512MB file

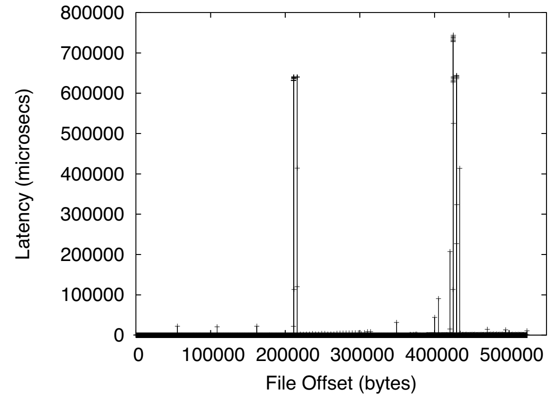

Figure 2: Write latency at each file offset when writing a 512MB file

(1) **smart scheduling of asynchronous writes** that adapts operating system destaging policies based on available memory and workload;
(2) **balancing the population of read and write pages** through multiple ghost caches that track the allocation of memory; and,
(3) **opportunistic write scheduling** using a unified memory/device scheduler I/O queue that allows the I/O system to actively flush dirty pages prior to the pages being submitted for write-back.

The framework does not define new algorithms for managing a read cache. Rather, it works with existing algorithms, e.g. ARC [17], LIRS [11], 2Q [12], adjusting the memory available for read caching. Also, these techniques do not affect the reliability or durability of data in that all operating systems policies, e.g. periodic update deadlines, are enforced.

To demonstrate the benefits of these techniques, we perform experiments based on microbenchmarks and macrobenchmarks that capture a wide range of workloads. Depending on the workload we are able to improve system throughput by more than 30% on average. Finally, the results show that our optimizations are valid not only for local file systems but also for network file systems, such as NFS.

2 Background: Deferred Writing

We give several examples in which the static policies used in operating systems result in poor system performance. We do so by demonstrating that different workloads require different parameterization of the same system variables. Our treatment focuses on Linux, but applies to other operating systems as well. It is also valid for both local and network file systems.

Modern operating systems defer the writing of dirty memory buffers to storage because this noticeably improves performance. In doing so, multiple writes to the same page may be aggregated and performed in a single update. Also, spatially related writes may be performed together, even when they occur at different times. Write operations are less critical than read operations, because a process is not suspended as a consequence of a write system call, whereas delayed reads block processes.

A dirty page might stay in memory until the last possible moment—sometimes system shutdown. However, keeping the page in volatile memory for a long period of time has two major drawbacks: first, in case of a failure all unstable updates will be lost and, second, dirty writes occupy memory pages that could be better used for read caching or other purposes, such as anonymous paging.

Traditional UNIX®[25] systems use a periodic update policy in which individual dirty blocks are flushed when their age reaches a predefined limit [18]. Modern systems use an additional criterion to decide when to destage dirty pages to storage. When the number of dirty pages in memory exceeds a certain percentage—the system-wide parameter *dirty_background_ratio* in Linux—a flushing daemon wakes up and starts writing dirty pages to disk until an adequate number of dirty pages have reached storage. By this operation, the flushing daemon ensures that there are always enough free pages available in order to allocate more memory to satisfy new reads and writes.

If applications dirty pages faster than the daemon flushes them to storage, the system will eventually reach the *memory pressure* state: the point at which the maximum allowed number of dirty pages in the system has been reached. Typically this limit is set close to half the size of the available RAM. Memory pressure hampers performance severely because it blocks all writing applications until there is free space for dirty buffers in RAM, which effectively makes all write operations synchronous. Memory pressure has an effect on reads as well. All pending writes that must be

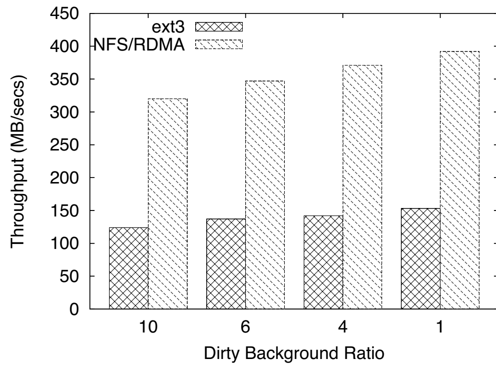

Figure 3: Throughput when writing a 2GB file sequentially

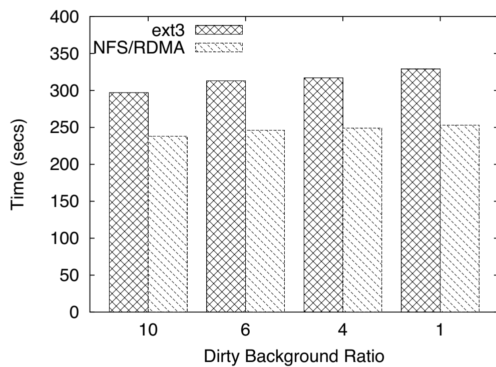

Figure 4: Time to compile Apache for ext3 and NFS over RDMA

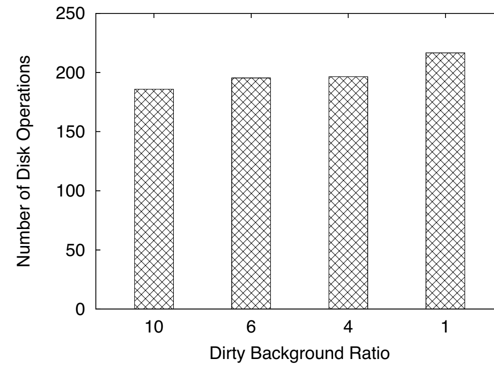

Figure 5: Average number of disk operations per second when compiling Apache

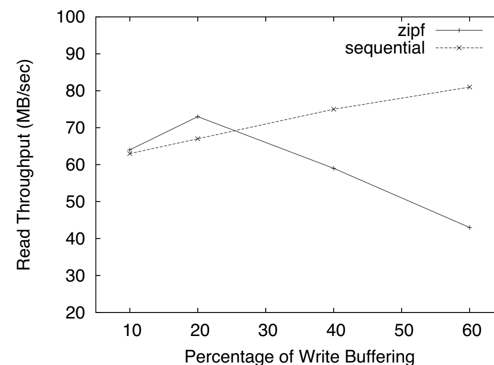

Figure 6: Read throughput for Zipf and sequential access patterns

written to storage interfere with concurrent reads, which results in queuing delays at the device level.

Figure 1 highlights the behavior of asynchronous writes. It shows the number of pages under write-back as a function of time. During a large file write, dirty pages are accumulated in memory until the number of dirty pages exceeds the *dirty_background_ratio* threshold. Then (after a few seconds in Figure 1), the flushing thread wakes up and starts writing pages to disk until their number is below the threshold. The remaining pages are written to the device when the period that dirty pages are allowed to stay in memory expires. Figure 2 shows that memory pressure occurs at two points in time while writing the file, producing significant increases in latency and, thus, a drop in the overall throughput.

The value of the *dirty_background_ratio* that triggers the flushing daemon to start the write-out is critical to system performance. Figure 3 shows the throughput of a process for writing a 2GB file sequentially under different values of the flushing threshold. The highest throughput occurs when the background updates start as soon as possible. The early start of the flushing operation ensures that there are always enough available memory pages. Figure 4 shows the time required to unzip and compile the Apache web server under various values of the same parameter. A less aggressive flushing policy yields the best performance. This is because the specific workload exhibits many short-lived files and block overwrites. More importantly, keeping the data in memory longer lowers the number of disk I/Os, a practice that makes buffering essential in the presence of concurrent I/O intensive processes (Figure 5).

The effect of the parameters on system throughput is also apparent in the case of NFS/RDMA over Infiniband. The high bandwidth of the device, significantly higher than the disk bandwidth especially in the case of an asynchronous NFS server, makes write buffering undesirable (Figures 3,4). Thus, the capability of the storage devices should be another factor in deciding when to start the write-out process.

An important, but over-looked, issue is the interference of writes and reads at the memory level. Asynchronous writes create dirty buffers that stay in memory for a period of time. This effectively reduces the number of blocks cached as a result of previous read operations. Most systems deal with the sharing of RAM

between write (dirty) and read (cached) pages by setting a hard limit on the maximum number of dirty blocks in memory in order to preserve space for frequently or recently accessed clean pages used to satisfy reads. This hard limit defines the memory pressure point. Thus, the decision of how much memory to use for buffering writes is again critical to the system throughput. Using more memory for reads allows for more cache. On the other hand, it causes the system to reach memory pressure faster.

Figure 6 demonstrates the sensitivity of throughput to the amount of write buffering allowed by the system. These IOzone [10] microbenchmarks show the throughput of a process that performs read and write I/O to two separate 512MB files. In one case, reads are sequential. In the other, the location of reads is drawn from a Zipf distribution. In both cases, writes are sequential. For sequential reads, more write buffering results in less competition for the disk bandwidth between read and background write requests, and, thus, improves performance. For Zipf reads, a very aggressive write buffering policy reduces the cache hit rate of read requests and hampers performance. However, too little write buffering does not yield optimal throughput either, because of the asynchronous nature, writes represent a background load on disk. The obtrusiveness of this load is a significant factor on the experienced throughput of all concurrent synchronous requests. The effect of the write traffic on the system performance is important even though, from the application perspective, writes are completed as soon as the kernel marks the modified buffers dirty.

Another performance concern—unrelated to system parameters and valid for local file systems only—is the selection of dirty pages to destage. The memory manager does not consider the I/O cost when selecting the dirty pages to write back to the device. Instead, it selects pages in the order they were accessed and submits them to the I/O scheduler which chooses the sequence of requests to be sent to disk. The memory manager is oblivious to the I/O cost because it has no information about which pages reside in the scheduler queue. On the other hand, the scheduler is able to reorganize only the buffers that reside on its queues; it has no knowledge of the memory contents. Dirty buffers remain in memory until the flushing policy selects the corresponding page to be written out to the device. During this period, the I/O scheduler may serve I/O requests that are physically close to the block locations of dirty buffered pages. Dirty memory pages, although proximal to the blocks that are about to be dispatched to the storage device, cannot be scheduled for writing. This behavior results in extra disk traffic and increases the number of disk seeks, because the dirty pages will eventually reach the disk at a later time when their flushing condition is met.

Our work modifies the destaging mechanisms of modern operating systems in order to improve the performance of asynchronous writes and, at the same time, to ameliorate the effects of the competition between synchronous and asynchronous requests at the device and at the memory level. We adaptively tune the write-back process to take into account current workload patterns and, as a result, improve write bandwidth and latency. Also, we develop an algorithm that shares memory between read and write pages, preserving the performance of read caching while improving write throughput. Finally, we implement an architectural change that integrates the memory manager and the I/O scheduler in order to make asynchronous writes less obtrusive, thus, reducing the interference with reads at the disk level.

3 Adaptive Write Scheduling

We propose a new, adaptive, destaging algorithm for volatile caches that manage pages in a read-write cache. Previous studies propose a variety of adaptive algorithms for write-only, non-volatile caches all of which attempt to maintain the cache occupancy as high as possible in order to optimize the order of writes to disk and minimize the I/O operations [3, 19, 27]. In non-volatile caches, pages are stable as soon as they reach memory. Thus, no deadline is assigned to a buffer and there are no starvation nor reliability issues. These methods are not directly applicable to our memory model.

3.1 A Write-Only Cache

At first, we will focus on the simple case where the memory is used only to accommodate writes. Subsequently, we enhance the algorithm to optimize performance in a unified read-write cache.

A destaging algorithm for a write-only memory cache should keep the cache occupancy as high as possible to enable ordering optimizations when dispatching pages. At the same time, it should avoid cache overflow so that no synchronous writes are forced (memory pressure). For a destaging mechanism to be effective, write requests should either be a cache hit or empty space should be available in the cache so that a new page allocation succeeds.

Previous work has compared a number of different destaging algorithms and has shown the performance benefits of a high-low watermark algorithm [27]. When the high threshold is crossed, the memory manager starts writing data out to disk until the percentage of dirty blocks is below the low threshold. The combination of two thresholds controls the start and stop of the

flushing of dirty buffered pages.

The most significant drawback of the high-low watermark scheme is that both thresholds are time-invariant. Deciding on the correct values is a hard problem and, more importantly, their optimal values depend heavily on the workload (Section 2). Under light I/O, the high threshold should have a large value in order to increase the write cache ratio. On the other hand, under heavy I/O, it is important for the system to maintain a sufficient number of available clean pages and a smaller value for the threshold is preferable.

We take a rate-based, adaptive approach to setting the high watermark, deriving its value from the rate at which the system dirties pages.

Let $h(t)$ be the value of the time-variant high watermark and $d(t)$ the rate that processes are dirtying new pages. This rate is measured by examining the number of "set dirty bit" operations at every time unit.

For an incoming I/O rate $d(t)$, the value for the high watermark at the end of the period is $h(t)$. If the data rate changes, we adjust the value of $h(t)$ based on the following intuition:

- If $d(t) \geq d(t-1)$ then the value of $h(t)$ should be reduced
- If $d(t) \leq d(t-1)$ then the value of $h(t)$ should be increased

Due to the computational requirements of performing complex arithmetic calculations in the kernel and the frequency of the operation, none of the advanced smoothing algorithms [2] are suitable for adoption. Control theory methods are also not practical because the watermark variance is not linear and depends heavily on the workload, making linear approximations hard [28].

We picked a simple smoothing function to adjust the high watermark value, based on the statistics we collect about the incoming data rate. Specifically, we use a formula for the relative change in the value of $h(t)$ so that the value of the threshold is inversely proportional to the change in the incoming data rate.

$$h(t) = h(t-1)\frac{d(t-1)}{d(t)}$$

.

We experimented with several different functions and algorithms to adjust the value of $h(t)$, such as moving average methods, step functions or exponential increase and backoff. This simple scheme provides competitive results along with very low computational requirements—an important factor given the frequency of this operation in the kernel. Nam and Park [19] provide some more mathematical insight to a similar scheme when describing a destaging algorithm for RAID-5 arrays. A detailed analytical study of the watermark variance problem, part of future work, may highlight areas of improvement and potentially lead to optimizations to our framework.

We also take a similar adaptive approach to setting the low watermark, deriving its value from both the process writing rate and on the I/O rate that the storage device can sustain. The difference between the low and high watermark determines the amount of data to destage upon activation of the write-out process. The value of the low watermark $l(t)$ cannot be higher than $h(t)$ and it depends on the the rate $c(t)$ the memory manager is able to clean pages, i.e. flush data to the device. If the device can sustain a high I/O rate compared to the incoming data rate then flushing can be less aggressive and $l(t)$ can have a higher value. On the other hand, if the incoming rate is higher than the outgoing rate, it is essential for the system to make clean pages available and $l(t)$ should be low. We define $l(t)$ as:

$$l(t) = l(t-1)\frac{d(t-1)}{d(t)}\frac{c(t)}{c(t-1)}$$

Again, we experimented with other scaling functions in adjusting the low watermark, but found that this simple scheme performed well in practice and has low computational requirements.

To avoid overtuning the watermarks, we rate limit the change of $h(t)$ and $l(t)$ so that $\frac{1}{2}l(t-1) \leq l(t) \leq 2l(t-1)$ and $\frac{1}{2}h(t-1) \leq h(t) \leq 2h(t-1)$.

The parameter t defines the time at which the system samples the I/O rates and sets the value of the watermarks. The value of the interval between two measurements of the incoming $d(t)$ and outgoing $c(t)$ rates is critical in how fast the system adapts to the workload patterns. Frequent measurements allow the system to adapt more quickly. On the other hand, sampling the page states too often increases computational overhead.We discuss the importance of the time unit selection in the experimental section.

3.2 A Read-Write Cache

The goal of a unified read-write caching scheme differs from that of a write-only cache. A write-only cache attempts to keep as many dirty buffered pages as possible without running out of available memory. A read-write cache must preserve a useful population of read-cache pages. Reserving more memory pages to buffer writes reduces cache hit rates, because it reduces the effective size of the read cache.

We define h_{max} as the maximum possible occupancy of memory with dirty pages before the write-back starts, so that the inequality $h(t) < h_{max}$ is always valid. We extend the destaging algorithm presented in the previous section by adaptively varying the

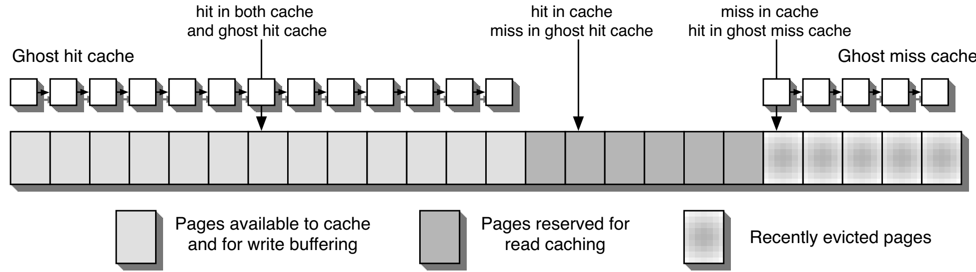

Figure 7: Using two ghost caches to identify the current working set

maximum occupancy ratio h_{max}. The intuition behind varying the h_{max} value is to provide an upper bound to the number of dirty pages in memory so that new dirty pages do not replace pages that are used for read caching by processes. Based on the fact that a read blocks the calling application whereas a write does not, our main goal is to maximize the read hit rate. To do so, we use a set of heuristics to identify what percentage of the RAM is actively used by processes to cache read data.

We use a ghost cache that holds meta-data information on blocks recently evicted from the cache (the ghost miss cache in Figure 7). In this way, we record the history of a larger set of blocks than can be accommodated in the actual cache. In the ghost miss cache, we keep an index of the blocks that were replaced as a result of write buffering only. If a clean block is replaced by another block, due to the regular eviction policy after a read, we do not add it to the ghost cache.

When a process issues a read, we look at the cache to determine whether it contains the requested block. If the block is not found, but it was recently replaced due to write buffering, its metadata information resides in the ghost cache. This indicates that if the system were buffering fewer write pages then this request would have resulted in a cache hit. An actual cache miss that hits in the ghost cache indicates that aggressive write buffering is interfering with read cache performance and that the value of h_{max} should be reduced. We note that our ghost cache does not rely on a specific eviction policy. It simply tracks recently evicted pages.

The ghost cache contains indexes of already evicted blocks, and, consequently, the effectiveness of the scheme is limited, because the signaling of over-buffering comes too late—the blocks must be fetched again from the storage device.

To make our scheme more proactive, we introduce a second ghost cache. For clarity reasons we call the first cache a ghost miss cache and the second a ghost hit cache. In contrast to the miss cache, the hit cache does not contain evicted blocks. Rather, the ghost hit cache contains the contents of a smaller virtual memory (Figure 7). The ghost hit cache contains *all* of the write buffered pages and the most recent/frequent read cache pages. The memory area outside of the ghost hit cache contains the least recent/frequent read cache pages. As an extreme example, if all the h_{max} fraction of the available memory contained dirty buffered pages then all read cache hits would occur in the reserved area.

The ghost hit cache allows us to detect the potential negative effects of aggressive write buffering prior to incurring penalties from cache misses. If all read requests hit in the ghost hit cache then a smaller amount of memory would capture the current working set and more write buffering can be allowed. Alternatively, read requests that are cache hits but are misses in the ghost hit cache indicate that further write buffering, shrinking the available read cache, will probably result in reduced cache hit rates, because the effective memory space for read caching is running out. Read cache hits both inside and outside the ghost hit cache scale the amount of memory available for write buffering and read caching based on the current memory usage and workload.

Our ghost hit cache works for all modern read caching algorithms that maintain recency and/or frequency queues. Our implementation derives the ghost hit cache and reserved regions from Linux's two queue approximate LRU implementation, which identifies the pages that would be evicted were the memory smaller. Many other recency and frequency based caching algorithms [11, 12, 17, 20] make similar information available.

We vary the value of $h_{max}(t)$ based on the hit rates of the two ghost caches as follows:

$$h_{max}(t) = h_{max}\left(1 - \frac{C(t) - GH(t) + GM(t)}{reads(t)}\right)$$

in which $C(t)$, $GH(t)$, and $GM(t)$ are the number of hits in the page cache, ghost hit cache, and ghost miss cache, $reads(t)$ is the number of total read requests during the last time interval respectively. The quantity $C(t) - GH(t) + GM(t)$ counts the read requests that

fall into the reserved area and in recently evicted pages. A large fraction of read requests falling in these regions indicates that aggressive write buffering is consuming memory in the ghost hit cache needed for read caching or that the current working set is larger than the available memory in the ghost hit cache.

We note that $h_{max}(t)$ is always lower than h_{max}. $h_{max}(t)$ reduces the high threshold based on the distribution of reads in the previous time period only. If there are no read requests, the value defaults to h_{max}. Thus, the initial value h_{max} should indicate the highest possible amount of RAM that the system administrator wants to devote to buffering. To avoid pathological cases that arise when over-tuning $h_{max}(t)$, we rate limit the change so that $\frac{1}{2}h_{max}(t-1) \leq h_{max}(t) \leq 2h_{max}(t-1)$.

Our implementation sets the ghost hit cache size to be an h_{max} fraction of the available memory. The ghost miss cache keeps an index of the blocks that would remain in the cache if the amount of available RAM were larger by 20%. The size of the ghost caches affects the responsiveness of the system in case of cache misses. In a small ghost cache, a few missed reads will force write buffering to be less aggressive. On the other hand, a larger cache requires a higher number of misses to limit write buffering.

The value of the static h_{max} parameter should depend on the relative cost of reads and writes in the system. In a RAID-5 system in which writes are expensive, more buffering space will improve performance. On the other hand, if a log-structured file system is used, more space should be devoted to read caching. In the results section we provide more information about the importance of parameter selection.

Finally, it is important to differentiate between the h_{max} threshold and the memory pressure point at which all writes become synchronous until enough pages are freed. Our framework does not stall writes even if they occupy more than h_{max} percent of the cache size, it just mandates more aggressive write-back. In the AWOL implementation, we define the memory pressure point $h_{pres} = \frac{Cache_{size} - h_{max}}{2}$. If the number of dirty pages reaches h_{pres} buffered writes become synchronous.

4 Opportunistic Queuing

Our adaptive high-low watermark algorithm attempts to optimize the write-out of dirty pages by deciding when to start the destaging process (high watermark) and how much data to flush (low watermark). Deciding what to destage is another important factor that affects the system performance. For example, it is preferable to flush a buffer that is physically close to a stream of read requests currently being serviced by the disk. This kind of optimization is not feasible in the memory, because the memory manager has no knowledge of file system and device characteristics, e.g. the device LBN corresponding to a dirty page. Our framework addresses this problem by delegating the responsibility of selecting which pages to be cleaned to the I/O scheduler. We achieve this by introducing a unified memory-scheduler queue. We have implemented our modifications, valid for local file systems only, in Linux. However they apply to most operating systems.

A typical I/O scheduler differentiates between synchronous (read) and asynchronous (write) requests by using two separate sets of queues for each type of request. Synchronous requests have a short deadline, on the order of microseconds, so that requests are dispatched quickly and the application does not block waiting for the operation to complete. On the other hand, asynchronous operations have less strict deadlines, on the order of a few milliseconds, depending on the scheduler's policy.

In both queues, the I/O scheduler keeps the list of pending I/O requests sorted by logical block number. When a new I/O request is issued, it is inserted into the LBN sorted list of pending requests. This prevents the drive head from seeking all around the disk to service I/O requests. Instead, by keeping the list sorted, the disk head moves in a straight line around the disk. If the hard drive is busy servicing a request at one part of the disk and a new request comes in to the same part of the disk, that request can be serviced before moving off to other parts of the disk.

We enhance the scheduler by adding a third queue for pages, namely the *opportunistic queue*. This queue maintains an LBN sorted list of pages that are in memory and have not yet been submitted to the device for write-back. When an application issues a write, the kernel marks the buffers dirty and, at the same time, it places a pointer to the buffers in the opportunistic queue. In contrast to the conventional scheduler queues, requests in the opportunistic queue do not have an assigned deadline. Requests may remain in the queue until the memory manager dispatches the corresponding page to the storage layer. Then, the buffer is moved to the asynchronous list and its priority is determined by the scheduling algorithm.

When the scheduler is ready to dispatch the next request from the pending queue, it searches both the asynchronous and the opportunistic list for requests that are close to the one that is about to be issued. Figure 8 illustrates this process—it represents a single queue at the device for simplicity. The spatial criterion for proximity follows the same policy that current schedulers use and is based on the logical block number (LBN). Head positioning information and knowledge about the physical layout on the disk are not available

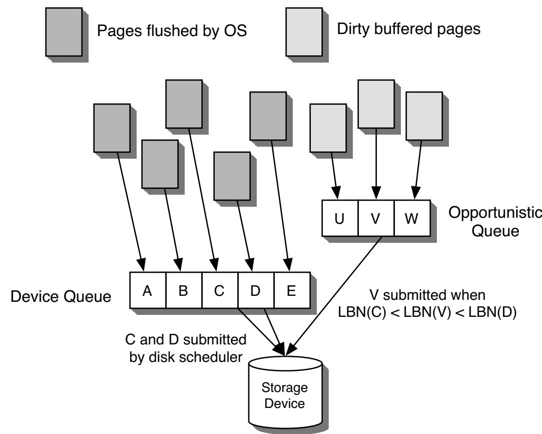

Figure 8: Opportunistic queuing

at this layer. Prior studies have proved the efficacy of a scheduling scheme based on LBNs [8, 15].

If one or more proximal requests are found, the I/O scheduler adds them into the dispatch list and removes them from the opportunistic queue. After the I/O is complete, it notifies the memory manager that the corresponding page is clean, so that the dirty bit is removed. If no appropriate request is found in the opportunistic list, the scheduler moves on to the next request in the synchronous queue. There is no performance penalty when no dirty buffered pages are dispatched; the opportunistic queuing scheme is an optimization, requiring only a few bytes of extra memory to keep the index of the buffer heads.

We implemented opportunistic queuing by modifying the deadline scheduler. Note that, this optimization requires modification to the data structures and logic of the scheduler but makes no assumption about the scheduling algorithm itself. Therefore, this mechanism is applicable to I/O schedulers in general and to the other Linux I/O schedulers in particular. In order to decide which requests to consider proximal, we select a simple criterion: its LBN must lie between those of the two next requests to be dispatched to the device.

The opportunistic queuing mechanism allows the system to commit pages to stable storage prior to them being submitted for write-back. This minimizes the interference of reads and writes at the disk level. The unified memory/scheduler queue reduces the average time for writes by ordering requests to the disk so that the service time for each request is minimized.

5 Evaluation

We have implemented our proposed changes in the 2.6.21 Linux kernel. We ran the experiments on a dual-core Xeon® machine with 2GB of RAM out of which about 1.5GB can be used for the page cache. The rest of RAM is reserved for the operating system itself and for running applications. For experiments on a local file system, we used a dedicated SATAII-300 hard drive. To evaluate our framework in a heterogeneous environment, we also performed measurements over the Network File System (NFS) using two different networks: gigabit Ethernet and 10-Gbps Infiniband. For Infiniband, we measured the performance of NFS/RDMA: a high-bandwidth, low-latency setup. The NFS server has 8GB of RAM and exports the filesystem asynchronously. By performing memory-to-memory operations (NFS client to NFS server), we measured the performance of the memory optimizations without the bottleneck of the disk device. Finally, throughout the experiments, we adjusted the value of the watermarks every one second. We experimented with different values of this parameter later in this section.

In order to evaluate our solution, we performed a series of microbenchmark and macrobenchmark experiments. The first set of experiments were based on IOzone: a popular benchmark suite that measures throughput and latency of I/O operations.

We use the following IOzone microbenchmarks:

- **No Reader, Sequential Writer (NRSW):** One process writing to a 1GB file sequentially.
- **No Reader, Zipf Writer (NRZW):** One process writing 1GB worth of data according to a Zipf distribution. Thus, certain popular blocks receive many accesses.
- **No Reader, Variable Writers (NRVW):** Several IOzone clients executing the NRSW or the NRZW workloads at random intervals. Each process uses a file between 100MB and 512MB in size. There are always between five and eight processes running in the system.
- **Sequential Reader, Sequential Writer (SRSW):** Two processes reading and writing different 1GB files sequentially.
- **Random Reader, Random Writer (RRRW):** Two processes reading and writing different 1GB files randomly.
- **Zipf Reader, Zipf Writer (ZRZW):** Two processes reading and writing different 1GB files according to a Zipf distribution.
- **Variable Readers, Variable Writers (VRVW):** Several IOzone clients executing the SRSW or the ZRZW workloads at random intervals. Each process uses files between 100MB and 512MB in size. There are always between five and eight processes running in the system.

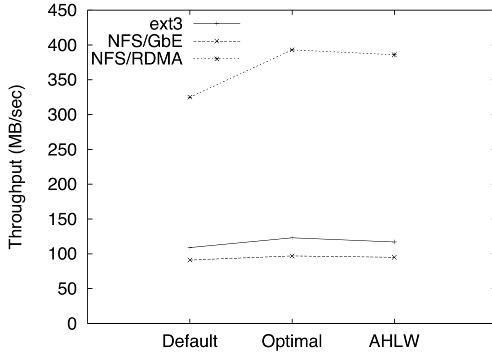

Figure 9: Throughput of a sequential writer (NRSW)

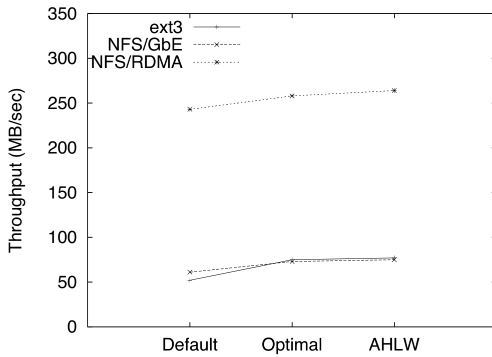

Figure 10: Throughput of a Zipf-distribution writer (NRZW)

5.1 The Adaptive High-Low Watermark Algorithm

First, we evaluate the effectiveness of the adaptive high-low watermark (AHLW) destaging algorithm using the IOzone workloads that perform writes only (NRZW and NRSW). These experiments do not use our read-write caching or scheduling optimizations. Figure 9 shows the throughput of Default, Optimal and AHLW under a sequential write (NRSW) workload. Default is the mainstream kernel with the default settings for deciding when to start the writeout (10% of available RAM). In Optimal, we have modified the value of the dirty_background_ratio to the optimal setting for this particular workload, which is 1%. We derived this number by manually altering the value of the parameter and rerunning the experiment until the highest throughput has been reached. Finally, AHLW is our modified version of the Linux kernel. AHLW performs almost as well as the optimal setup for the Linux kernel.

Figure 10 compares the throughput of Default, Optimal, and AHLW but this time under a workload that exhibits many rewrites (NRZW). AHLW throughput is slightly higher than Optimal—the optimal value of the dirty_background_ratio in this experiment is 41%. This is because the default Linux kernel uses a single threshold as opposed to the double watermark configuration. The double threshold decouples the decision of when to destage pages from how many pages to destage.

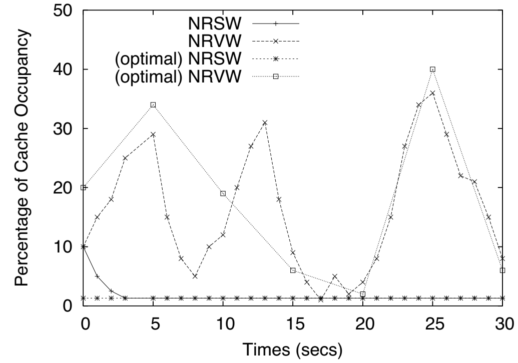

Figure 11: Variance of the high watermark for the NRSW, NRVW workloads and comparison with the optimal watermark values

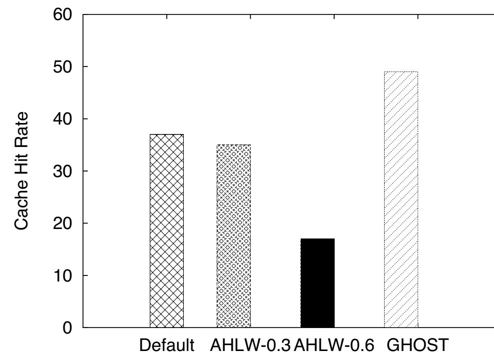

Figure 12: Comparison of cache hit rates for Default, AHLW and GHOST (ZRZW)

We now examine the performance of the weighting function that controls the high watermark. For the NRSW and NRVW benchmarks we plot the value of the watermark along with the optimal watermark value. We computed the optimal value by manually adjusting its value, rerunning the experiment from each time point and examining the number of destage I/Os, for each value. This optimal value is dynamic, whereas Figures 9, 10 use a static optimal. Figure 11 shows that in the case of a steady-rate workload (NRSW), the adaptive watermark quickly converges to its optimal value. For the variable rate workload (NRVW), the figure shows that there is room for improvement in the AWOL framework for a non-static workload.

5.2 The AHLW Algorithm with Ghost Caching

We now concentrate on the effect of write buffering on the cache hit rate. Specifically, we measure the system throughput and the cache hit rate under a workload that includes many re-reads and re-writes

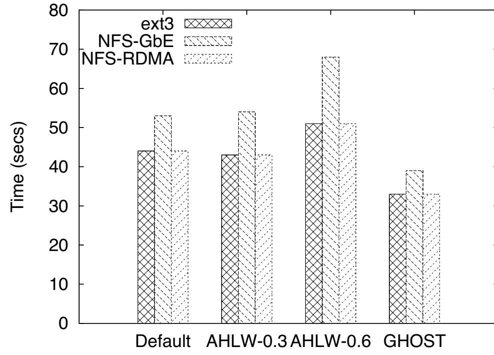

Figure 13: Execution times of Default, AHLW and GHOST under a read-write workload (ZRZW)

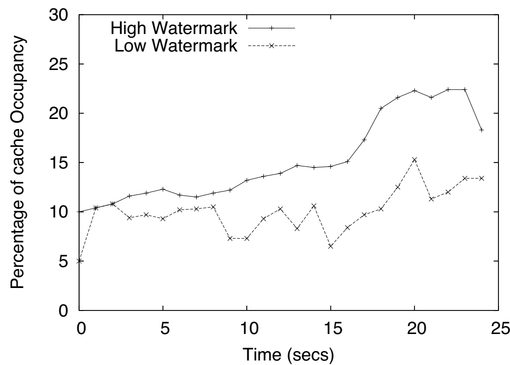

Figure 14: Variance of the high and low watermarks in GHOST (ZRZW)

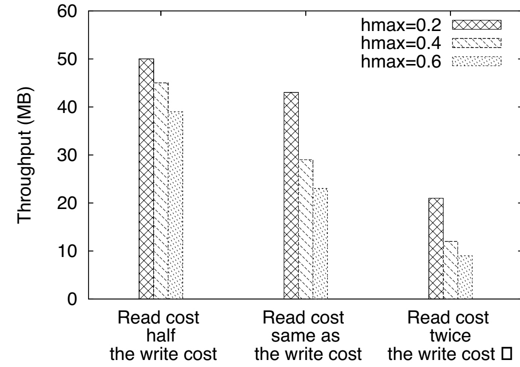

Figure 15: Comparison the read throughput as a function of the relative read-write cost (ZRZW)

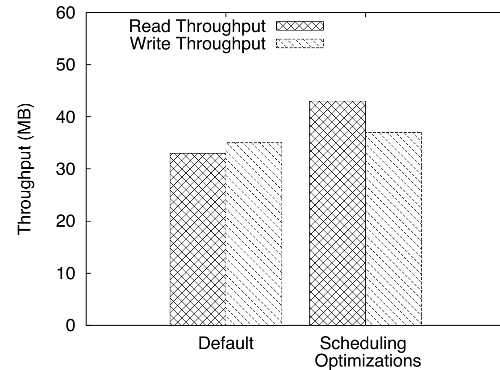

Figure 16: Comparison of the read and write throughput with and without opportunistic queuing (RRRW)

(ZRZW). We compare Default, AHLW with $h_{max} = 0.3$, AHLW with $h_{max} = 0.6$, and GHOST: the Linux kernel enhanced with both the adaptive watermarks and the ghost caching optimizations. For AHLW the h_{max} value is static.

In this scenario, the cache hit rate affects the performance dramatically. Figure 12 plots the cache hit rate for each of the configurations. GHOST achieves the highest hit rate. In contrast, AHLW with $h_{max} = 0.3$ yields the lowest hit rate but provides the fastest writes (not shown). Overall, GHOST provides the best performance. The slower the device the more evident are the advantages of ghost caching (Figure 13).

Figure 14 shows the variation of the high and low watermarks in GHOST as a function of time for the last experiment. Due to block reuse (cache hits), the rate of incoming I/O requests is not constant and the watermarks increase and decrease. Also, the sudden rises and drops in the value of $h(t)$ are due to the h_{max} constraint imposed by the ghost caching algorithm. The value of the low watermark shows more instability due to the non-constant rate that the device sustains for random writes and reads.

Lastly, Figure 15 shows the read throughput of GHOST in the ZRZW benchmark for different values of the h_{max} parameter as a function of the disk read/write cost. In systems where disk writes are more efficient than reads, *e.g.* log-structured designs, it is important to maintain a high cache hit ratio, and, hence, the value of h_{max} should be relatively low. In contrast, in systems where writes are expensive, *e.g.* RAID-5, the system should allow for more buffering. For this experiment, we artificially delay its type of request in the kernel to simulate the different environments. Our adaptive scheme adjusts the $h_{max}(t)$ value to the experienced workload. In general, the h_{max} parameter should be set to a high value (greater than 0.4 of the available memory).

5.3 I/O Scheduler Optimizations

We now assess the effectiveness of the I/O scheduler optimizations using the RRRW workload. The Linux kernel enhanced with the opportunistic queuing mechanism performs 20% fewer disk operations than the unmodified kernel. As a result, throughput is improved by more than 35% for reads (Figure 16). Random writes that are proximal to reads are scheduled immediately.

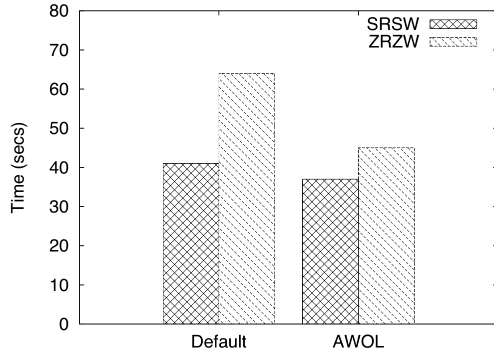

Figure 17: Performance of the ZRZW and SRSW benchmarks for Default and AWOL

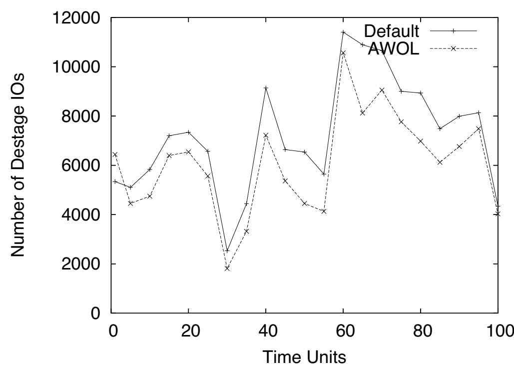

Figure 18: Number of destage I/Os per time unit as a function of time (VRVW)

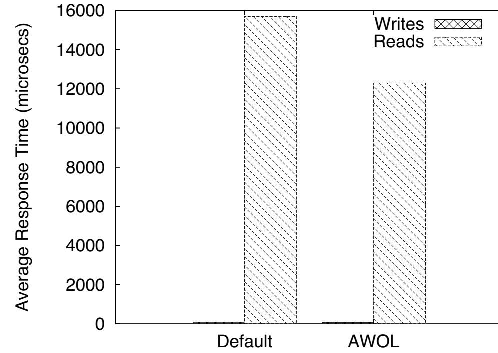

Figure 19: Average response times with a variable I/O rate (VRVW)

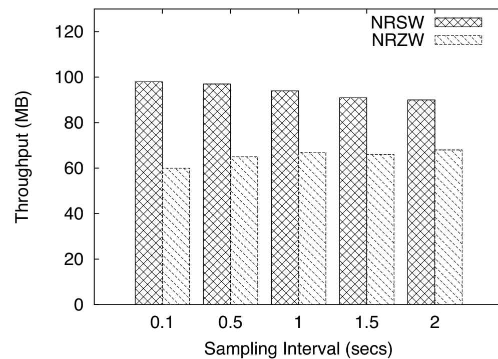

Figure 20: Throughput comparison under different values of the sampling interval (NRSW,NRZW)

This reduces the contention for disk bandwidth between reads and asynchronous write operations. Write throughput increases only modestly, because all write operations are asynchronous.

5.4 Putting it All together

We compare the performance of AWOL, our complete framework, with the unmodified Linux kernel. First, we compare the execution times of the SRSW and ZRZW workloads in ext3. Figure 17 shows that AWOL yields close to 35% improvement when compared with unmodified Linux for Zipf-distributed reading and writing (ZRZW). The performance improvement for sequential reading and writing (SRSW) is close to 10%. For SRSW, the superior performance arises from scheduler optimizations alone, because there are no potential benefits from read caching.

In the next experiment we examine AWOL's performance under a variable rate (VRVW) workload. Figure 18 plots the number of I/O destages (reads or writes) per time unit as a function of the time. The rises and drops in the graph show the changing data rate. A higher rate leads to more I/Os being dispatched to the device. AWOL performs fewer destage I/O operations than Linux on average. Figure 19 shows the user-perceived response times for the same experiment. All writes are buffered and complete in microseconds. Higher cache hit rates (ghost caching) and more efficient I/O scheduling (opportunistic queueing) result in much shorter average read response times for AWOL.

5.4.1 Adjusting the Sampling Frequency

Finally, we examine the importance of the sampling frequency. Figure 20 shows the throughput of the system under different values of this parameter for the NRSW and NRZW workloads. For the sequential writer example, frequent measurements allow the system to reach the optimal watermark value faster, at a price of higher CPU consumption (Figure 21). For a Zipf writer, too frequent measurements are prone to overtuning the watermarks, which results to lower throughput. For a read-write workload (ZRZW) with many re-reads, the cache hit rate is not affected by the sampling frequency (Figure 22). As with other system parameters, the optimal value of the sampling period depends on the experienced workload. In practice, a value of 1 second provides good throughput along with low computational requirements.

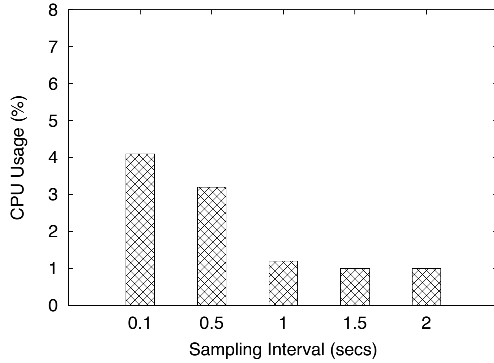

Figure 21: CPU consumption for AWOL under different values of the sampling interval (NRSW)

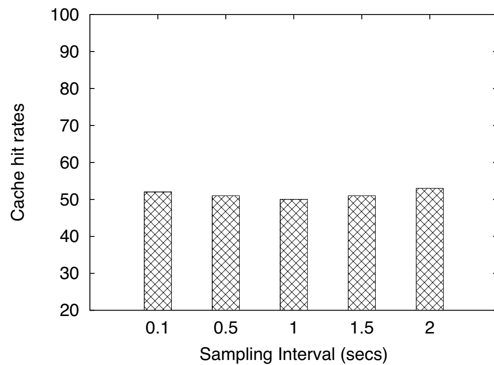

Figure 22: Cache hit rates for AWOL under different values of the sampling interval (ZRZW)

5.5 Macrobenchmarks

In the next experiment, we measure the time to untar and compile the Linux 2.6.20 kernel. This real-world workload exhibits lots of overwrites and intermediate, short-lived files. It accesses data sequentially (for the most part) with reads and writes interleaved. We compare the performance of Default and AWOL. The scheduler and destaging algorithm optimizations enable AWOL to reduce the execution time by almost 21% when compared with Default. Ghost caching also has some effect on this experiment. The cache hit rate (for both read and writes) is improved by 12%.

We also run TPC-C [26], a data-intensive, online transaction processing benchmark, for approximately one hour on a Postgres database. TPC-C issues small 4 KB random I/Os, two thirds of which are reads. The metric for evaluating TPC-C performance is the number of transactions completed per minute (tpmC). The amount of RAM available is critical to the reported performance. In our case, RAM covers only 20% to 30% of the working set. Figure 23 shows the throughput of unmodified Linux relative toAWOL for ext3 and NFS-RDMA. Our results are normalized because the

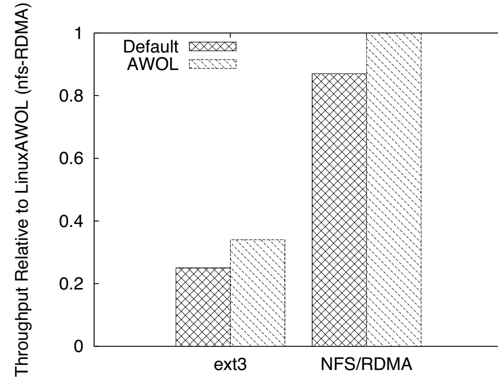

Figure 23: Performance of the TPC-C benchmark

test is unaudited. AWOL improves performance by more than 25%. The average value of $h(t)$ is 0.03 and its max value reaches 0.4. For the low threshold these values are 0.01 and 0.3 respectively. The performance difference of Default and AWOL comes as no surprise given the data-intensive nature of the benchmark.

6 Related Work

Early works that compare write-back caching observe that using cache memory for dirty pages may reduce read hit rates and identify that reads are more important than writes [13]. In fact, this was one of the fundamental arguments to continue using write-through caching.

Concerns about running out of available memory for dirty data, which results in slow write-back and stalls, arise first in processor caches, owing to their smaller sizes. Skadron [24] proposes the allocation of additional memory for dirty data and lazy retirement policies to mitigate these effects.

The periodic update policy used by most operating systems, e.g. every 30 seconds, leads to I/O bursts that can hamper system performance. Through analysis and simulation, Carson and Setia [5] showed that for many workloads periodic updates from a write-back cache perform worse than write-through caching. They suggest two alternate disciplines: (1) giving reads non-preemptive priority and (2) interval periodic writes in which each write gets its own fixed period in the cache. The first may starve writes indefinitely. The second requires complex queuing and timing mechanisms, but does stagger writes in time, assuming that pages are dirtied over time. Mogul [18] implements an approximate interval periodic write-back policy that staggers writes in time using a small (one second) timer. His evaluation shows that interval periodic writes reduce both response time and its variance.

Golding *et al* [9] propose to delay write-back until the system reaches an "idle" period. This reduces the

delays seen by reads by delaying competing writes until idle periods, possibly with the help of non-volatile memory. One aspect of AWOL defers write-back to the same effect. However, AWOL also writes more aggressively at times and adaptively chooses between deferring and aggressively writing pages.

The notion of write-performance dominating overall system performance has a long history in file and storage system design. We invoke the same scaling arguments as do other researchers. As processors increase in speed relative to disk or network throughput, they dirty pages faster than the storage subsystem can clean them. This makes write performance the overall system bottleneck. Ousterhout [21] invokes this argument in the original paper on log-structured file systems.

Non-volatile memory (NVRAM) offers one way to improve write-performance and tolerate write bursts by making data persistent without sending it to disk [4, 6, 22]. Because NV-RAM is more expensive than regular RAM and the read cache does not need to be persistent for correctness, NVRAM systems partition memory into a volatile read cache and a non-volatile write cache. Thus, they do not consider the balance of read and write pages in a shared memory. Again owing to its cost, NVRAM also tends to be deployed in server systems or in hybrid disk devices, whereas we focus on the adaptive allocation of memory within the operating system.

RAID controllers that use non-volatile memory for writes employ adaptive destaging policies that either vary the rate of writing [27] or the destage thresholds [19] based on memory occupancy and filling and draining rates. Such systems have quite different goals from ours, because cached writes are persistent. They wish to delay destaging data as long as possible. In contrast, operating system memories contain volatile writes and, thus, must destage data more aggressively consistent with operating system age thresholds.

Also for RAID controllers, Gill et al. [8] integrate recency into disk scheduling algorithms in order to aggregate multiple writes to data prior to destaging the data to disk. In the context of RAID controllers, writes are particularly expensive as they involve disk seeks among all disks in the RAID group. In our opportunistic queuing optimization (Section 4), we also employ information about page state into making write scheduling decisions. However, Gill et al. do so to delay writing pages that may be re-written and are in NVRAM. We use it to perform opportunistic writing when writes will be inexpensive because the disk head is already near the write location. Alonso and Santonja [3] also use recency of write access to defer writing data for pages written multiple times.

Free-block scheduling [16] describes a framework for enhancing disk head utilization and throughput by interposing background reading/writing tasks into the request stream. The authors include write-back as one of many possible uses. The disk write discipline of AWOL's opportunistic queuing mechanism is similar in concept to freeblock scheduling. In fact, we could incorporate freeblock scheduling to implement a more sophisticated version of our queuing optimization that uses more accurate information about the position and activity of the disk head. In contrast to freeblock scheduling, AWOL changes the fundamental write-scheduling framework. Pages are grabbed out of the page cache and queued immediately before the operating system submits them for write-back. In addition, our simple implementation, based on LBN alone, provides good empirical results without the complexity of implementing freeblock scheduling outside of disk firmware [15].

Recent caching work has explored the adaptive allocation of memory between recency and frequency for read pages. Two queue (e.g. 2Q [12]) versions of LRU split the LRU queue into a lower queue for pages accessed once (recency) and a higher queue (frequency) for pages accessed more than once. Several papers (ARC [17], LIRS [11]) size these queues adaptively in response to workload shifts. They do not consider the allocation of memory for write pages.

None of the described research balances the population of read and write pages in a shared memory and dynamically allocates memory across these classes of pages. EMC's Enginuity [7] is the exception. This storage controller manages a global memory for reads and writes, allowing the region used for non-volatile writes to grow and shrink over time in response to workload shifts. No algorithms or details are given.

Li et al. [14] classify writes by type in order to better manage a second-tier read cache. Writes that correspond to dirty pages that are evicted from a first level cache are good candidates for caching at the second tier, whereas writes that periodically clean dirty pages are poor candidates, because those pages are likely still cached at the first tier. Our system also implicitly classifies writes but in different dimensions. We are concerned with whether writes are synchronous, blocking an application, or asynchronous. We do not use explicit hints.

Finally, ghost caching has been used quite extensively to track pages beyond the size of available memory. Megiddo and Modha in ARC [17] provide an overview of the use of shadow (ghost) caches in the many read-caching algorithms that balance multiple queues. Wong and Wilkes [29] use the technique for exclusive caching in a hierarchy of caches. The major difference is that AWOL maintains two ghost caches;

a larger cache to detect misses that would be hits in a larger cache (similar to previous uses) and a smaller ghost cache to detect hits that would be misses in a smaller cache. The latter technique allows AWOL to detect when increased write-back caching would degrade cache hit rates and prevents AWOL from consuming too much memory for dirty data.

7 Conclusions

In this paper, we demonstrate how the static write policies used by the memory manager do not adapt well to the variable workloads modern operating systems experience. Overly aggressive write buffering eliminates the effective space for caching and hurts performance. On the other hand, if the memory manager starts the destaging process too early, the background write load interferes with foreground reads.

Our modifications to the memory manager and I/O scheduler enable the system to automatically tune the destaging process, depending on the workload. Our framework minimizes the interference of read and write traffic at the device level and also maximizes cache hit rates.

We implemented our changes to the memory manager and I/O scheduler in the 2.6.21 Linux kernel. We will make these changes available to the Linux community prior to the publication of these results under the GNU General Public License [1], which means that the source code will be freely-distributed and available.

References

[1] Gnu general public license. Version 2, Available at `http://www.gnu.org/licenses/gpl.html`. Accessed 12/4/2007.

[2] Smoothing data. Chapter on Data Smoothing (Wolfram Research) Available at `http://documents.wolfram.com/applications/`. Accessed 12/4/2007.

[3] M. Alonso and V. Santonja. A new destage algorithm for disk cache: DOME. In *EUROMICRO Conference*, 1999.

[4] P. Biswas, K. K. Ramakrishnan, and D. Towsley. Trace driven analysis of write caching policies for disks. In *ACM SIGMETRICS*, 1993.

[5] S. D. Carson and S. Setia. Analysis of the periodic update write policy for disk cache. *IEEE Transaction on Software Engineering*, 18(1), 1992.

[6] K. Chen, R. B. Bunt, and D. L. Eager. Write caching in distributed file systems. In *IEEE International Conference on Distributed Computing Systems*, 1995.

[7] Enginuity algorithms: Dynamically optimizing performance. Whitepaper. Available at `http://www.emc.com/pdf/techlib/c1033.pdf`. Accessed 9/4/2007.

[8] B. S. Gill and D. S. Modha. WOW: Wise odering for writes—combining spatial and temporal locality in non-volatile caches. In *File and Storage Technology Conference*, 2005.

[9] R. Golding, P. Bosch, C. Staelin, T. Sullivan, and J. Wilkes. Idleness Is Not Sloth. In *TCON'95: Proceedings of the USENIX 1995 Technical Conference Proceedings on USENIX 1995 Technical Conference Proceedings*.

[10] The IOzone Benchmark. Available at http://www.iozone.com.

[11] S. Jiang and X. Zhang. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *ACM SIGMETRICS*, 2002.

[12] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *Conference on Very Large Databases*, 1994.

[13] N. Jouppi. Cache write policies and performance. Technical Report WRL 91/12, Digital Western Research Lab, 1991.

[14] X. Li, A. Aboulnaga, K. Salem, A. Sachendina, and S. Gao. Second-tier cache management using write hints. In *File and Storage Technologies Conference*, 2005.

[15] C. Lumb, J. Schindler, and G. Ganger. Freeblock scheduling outside of disk firmware. In *Conference on File and Storage Technologies*, 2002.

[16] C. Lumb, J. Schindler, G. Ganger, and D. Nagle. Towards higher disk head utilization: Extracting free bandwidth from busy disk drives. In *Operating Systems Design and Implementation*, 2000.

[17] N. Megiddo and D. Modha. ARC: A self-tuning, low overhead replacement cache. In *File and Storage Technologies Conference*, 2003.

[18] J. Mogul. A better update policy. In *USENIX Summer Technical Conference*, 1994.

[19] Y. J. Nam and C. Park. An adaptive high-low water mark destage algorithm for cached RAID5. In *Pacific Rim Intenataional Symposium on Dependable Computing*, 2002.

[20] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *ACM SIGMOD*, 1993.

[21] J. Ousterhout and F. Douglis. Beating the I/O bottleneck: A case for log-structured file systems. *Operating Systems Review*, 23(1), 1989.

[22] J. F. Pâris, T. Haining, and D. Long. A stack model based replacement policy for a non-volatile write cache. In *Conference on Mass Storage Systems*, 2000.

[23] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(1), 1990.

[24] K. Skadron and D. W. Clark. Design issues and tradeoffs for write buffers. In *Symposium on High-Performance Computer Architecture*, 1997.

[25] The Open Group. The Single UNIX Specification. Available at http://unix.org.

[26] The TPC-C Benchmark. Available at http://www.tpc.org/tpcc.

[27] A. Varma and Q. Jacobsen. Destage algorithms for disk arrays with nonvolatile caches. *IEEE Transactions on Computers*, 47(2), 1995.

[28] M. Vidyasagar. *Nonlinear Systems Analysis, Second edition.* Prentice Hall., Englewood Cliffs, NJ, 2000.

[29] Theodore M. Wong and John Wilkes. My Cache or Yours? Making Storage More Exclusive. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 161–175, Berkeley, CA, USA, 2002. USENIX Association.

Trademark Notice:
NetApp and the Network Appliance logo are registered trademarks and Network Appliance is a trademark of Network Appliance, Inc. in the U.S. and other countries. Linux is a registered trademark of Linus Torvalds. Xeon is a registered trademark of Intel Corporation. UNIX is a registered trademark of The Open Group. All other brands or products are trademarks or registered trademarks of their respective holders and should be treated as such.