Storage-Aware Caching: Revisiting Caching for Heterogeneous Storage Systems

Brian C. Forney Andrea C. Arpaci-Dusseau Remzi H. Arpaci-Dusseau

Computer Sciences Department
University of Wisconsin, Madison
{bforney,dusseau,remzi}@cs.wisc.edu

Abstract

Modern storage environments are composed of a variety of devices with different performance characteristics. In this paper, we explore storage-aware caching algorithms, in which the file buffer replacement algorithm explicitly accounts for differences in performance across devices. We introduce a new family of storage-aware caching algorithms that partition the cache, with one partition per device. The algorithms set the partition sizes dynamically to balance work across the devices. Through simulation, we show that our storage-aware policies perform similarly to LANDLORD, a cost-aware algorithm previously shown to perform well in Web caching environments. We also demonstrate that partitions can be easily incorporated into the Clock replacement algorithm, thus increasing the likelihood of deploying cost-aware algorithms in modern operating systems.

1 Introduction

Modern computer systems interact with a broad and diverse set of storage devices, including local disks, remote file servers such as NFS [30] and AFS [16], archival storage on tapes, read-only media such as compact discs and DVDs, and even storage sites that are accessible across the Internet [4, 19, 25, 36]. As new storage components are introduced [11, 31, 34], their behaviors and properties will likely become even more divergent than they are today.

Although this set of devices is disparate, one commonality pervades them all: the time to access them is high, especially as compared to CPU cache and memory latencies [23]. Due to the cost of fetching blocks from storage media, *caching* of blocks in main memory often reduces execution time of individual applications and increases overall system performance – often by orders of magnitude.

However, while storage technology has dramatically changed over the past few decades, important aspects of the caching architectures used by modern operating systems have remained unchanged. Though there have been innovations in mechanism, including the integration of the file cache and virtual memory page cache [21], copy-on-write techniques [26], and software emulation of reference bits [7], there has been little change in policy, with most operating systems employing LRU or LRU-like algorithms to decide which block to replace.

The problem with LRU and related caching algorithms is that they are *cost-oblivious*: all blocks are treated as if they were fetched from identically performing devices and can be re-fetched with the same replacement cost as all other blocks. Unfortunately, this assumption is increasingly problematic, as the manifold device types described above have a correspondingly rich set of performance characteristics. As a simple example, consider a block fetched from a local disk as compared to one fetched from a remote, highly contended file server; in this case, the operating system should most likely prefer the block from the file server.

Within such heterogeneous environments, file systems require caching algorithms that are *aware* of the different replacement costs across file blocks. Given that the slowest device roughly determines the throughput of the system, storage-aware caching seeks to balance work across devices by adjusting the stream of block requests. Hence, in a heterogeneous environment, a storage-aware cache considers workload behavior *and* device characteristics to filter requests.

Thus, in this paper, we explore the integration of cost-aware algorithms into an operating system page cache using simulation. Our simulation accounts for real-world factors such as an integrated page cache and simplicity of design. We build on previous work in cost-aware caching from the web-cache and theory communities, demonstrating that a separate set of partitioned algorithms are as effective, yet simpler, than proposals in those research areas.

Our study is set in the context of a network-attached disk system. Network-attached disks are an increasingly important storage paradigm, and present clients with both static and dynamic forms of performance heterogeneity [5, 6, 13]. However, the algorithms we develop are general and can be applied across a broader range of storage devices.

Our main results are as follows. We show storage-aware caching is significantly more performance robust than cost-oblivious caching and as robust as a leading web-caching algorithm. Since operating systems have specific implementation needs, we develop and evaluate a version of storage-aware caching that extends the commonly implemented Clock algorithm.

The rest of this paper is organized as follows. In Section 2 we give an overview of the algorithms that we investigate in this paper. We then describe our algorithm for selecting partition sizes in Section 3. Section 4 describes the assumptions of our environment in more detail and explains our simulation framework. Simulation results are in Section 5. We compare and contrast our work to existing work in Section 6, and Section 7 contains future work. Finally, we conclude in Section 8.

2 Algorithm Overview

This section provides an overview of the algorithmic space we explore. First, we describe existing cost-aware algorithms as a basis of comparison. We then present our caching algorithms, which are based upon partitioning the cache according to replacement cost.

2.1 Existing Cost-Aware Algorithms

The theoretical community has studied cost-aware algorithms as *k-server problems* [12, 20]. A restricted class of k-server problems, *weighted caching*, is closely related to cost-aware caching. *LANDLORD* [40] is a significant algorithm in the literature, which we use for comparison. LANDLORD is closely related to the leading web caching algorithm [10].

LANDLORD combines replacement cost, cache object size, and locality by extending both LRU and FIFO to include cost and variable cache object sizes within a cache. Since we configured LANDLORD to use LRU, we describe the LRU version. LANDLORD associates a cost with each object, which is called L. When an object enters the cache, LANDLORD sets L to H, which is the retrieval cost of the object divided by the size of the object. If object eviction is needed, LANDLORD finds the object with the lowest L value, removes it, and ages all of the remaining objects. LANDLORD ages pages by decrementing the L value of all remaining objects by the L value of the evicted object. Upon reference of an object in the cache, LANDLORD restores its L value to H. LANDLORD degenerates to strict LRU when all H values are the same.

LANDLORD has attractive theoretical and experimental properties. As shown by theoretical analysis, when the size of cache objects are the same, LANDLORD is k-competitive, where k is the size of the cache. Thus, in a fixed object size cache, LANDLORD performs within a factor of k of the optimal off-line algorithm over all possible request sequences [39].

2.2 Overview of Aggregate Partitioned Algorithms

All of the cost-aware algorithms in the literature are place-anywhere. A *place-anywhere algorithm* has two characteristics: blocks may occupy any logical location in the cache, independent of their original source or cost, and costs are recorded on a page granularity. The advantage of place-anywhere algorithms is they calculate, in a single value, the trade-off between locality and cost. Thus, at replacement time, these algorithms bias eviction toward pages with low retrieval cost.

In contrast to a place-anywhere algorithm, an *aggregate partitioned algorithm* divides the cache into logical partitions, where blocks within a logical partition are from the same device and thus share the same replacement cost. The algorithm aggregates replacement cost since it is a function of a device's performance. An aggregate partitioned algorithm benefits from the aggregation of blocks and cost metadata in two ways: the amount of metadata is reduced and the value of the metadata more closely reflects the current replacement cost of a block from a device. Thus, the space overhead is proportional to the number of devices currently used and blocks are more likely to be replaced when the replacement cost is low.

Conversely, place-anywhere algorithms only record the cost when the page is brought into the cache. Thus, when the cache has a reasonably large number of pages, as is common today, a place-anywhere algorithm is more susceptible to inconsistent cost values. Aggregate partitioned algorithms avoid this problem by aggregating cost metadata on a per-device basis. As the performance of the device changes, the cost metadata is rarely inconsistent for more than a brief period of time. While a place-anywhere algorithm could recognize the change in cost for a device, and propagate the new cost to all pages in the cache, the cost update requires a significant number of pages to be updated, increasing overhead and implementation complexity.

Aggregate partitioned algorithms strive to set the relative size of each partition to balance work across devices. We define work as balanced when the cumulative delay for each device within a period of time is equal. To balance work, the size of each partition reflects the relative

cost of those blocks in a simple and efficient manner. For example, in a storage system with one slow disk and one fast disk, the cache is divided into two partitions, with the slow disk likely receiving a larger partition. We describe precisely how the relative sizes should be configured in the next section.

To choose a victim block, a storage-aware algorithm first selects a victim partition and then a victim block within that partition. The victim partition is chosen such that its resulting size relative to other partitions maintains the desired proportions. The individual victim within that partition can be selected with any replacement algorithm, such as LRU, LFU, or FIFO.

A few distinctions to prior work in virtual memory systems should be noted.

Unified and partitioned virtual memory systems: In the traditional sense, partitioned virtual memory systems distinguish between file system pages and virtual memory pages. The two are managed separately. Our storage-aware algorithms do not explicitly distinguish between file system pages and virtual memory pages. Rather, in order to balance work, our algorithms distinguish between pages based on which device supplied the page. Additionally, storage-aware caching algorithms change the size of partitions dynamically. Most partitioned virtual memory systems do not change the size of the file system cache and virtual memory partitions.

Local and global page replacement: Local page replacement at eviction time considers processes in isolation, while global page replacement applies replacement across processes. Our storage-aware algorithms make per-partition replacement decisions, which is similar to the traditional notion of local page replacement. However, the decisions are based on cost *and* locality, not solely on locality as in local page replacement schemes.

2.3 A Taxonomy of Aggregate Partitioned Algorithms

In our work, we investigate a taxonomy of aggregate partitioned algorithms and show that dynamic aggregate partitioning is needed. The taxonomy is described in this subsection.

Two basic approaches are possible for aggregate partitioning: *static* and *dynamic*. In a static scheme, the ratio across partitions is selected once according to a one-time notion of the costs. However, with no knowledge of the workload and its resulting miss rates for a given cache size, one cannot *a priori* determine the relative sizes that lead to balanced work. Thus, dynamic partitioning is needed, in which the ratio of partition sizes adjusts as the requests are monitored.

Dynamic partitioning have the following three benefits. First, dynamic partitioning can adjust to the dynamic performance variations, or faults, common in modern devices [6]. Second, dynamic partitioning can react to contention at devices due to hotspots in workloads. Finally, dynamic partitioning can compensate for the fact that the performance ratios across devices can change as a function of the access patterns.

Dynamic partitioning can be divided into *eager partitioning* and *lazy partitioning*. With eager partitioning, when new partition sizes are desired, the algorithm immediately reallocates pages using new cost information. An algorithm with a lazy partitioning scheme gradually reallocates pages on demand to the desired size in response to the workload. Eager partitioning simplifies choosing a victim partition, since it is the same as the location of the new page, at the cost of removing pages which may be useful. Conversely, a lazy partitioning scheme only removes pages from partitions when they are truly needed by another partition.

With lazy partitioning, a block may replace any other block, as long as blocks are replaced at the proper frequency to maintain the desired partition size ratios. Thus, on replacement, one must explicitly choose a victim partition. We investigate a strategy based on an *inverse lottery*, as previously proposed for resource allocation [33, 37]. The idea is that each partition is given a number of tickets in inverse proportion to its desired size. When a replacement is needed, a lottery is held by selecting a random ticket; the partition holding that ticket is picked as the victim. The victim then gives up its least valuable page, and the lazy partitioning algorithm allocates the page to a new logical partition.

3 Selecting Partition Sizes

The main challenge with partitioned approaches is in determining how the relative sizes of the partitions should be configured. Storage-aware caching can be viewed as performing selective filtering of requests to devices. Assuming the slowest device limits the system throughput, the goal of storage-aware caching is to set the partition sizes such that an equal amount of work is sent to each device. More formally, for each device, the number of cache misses multiplied by the average cost of each miss should be equal.

3.1 Algorithmic Details

Our basic approach uses a dynamic repartitioning algorithm. In the algorithm, the storage-aware cache observes the amount of work performed by each device over a fixed interval in the past and predicts how the relative sizes of the partitions should be adjusted so that the work is equal. The algorithm's work metric is cumulative delay over a period of time. The delay is related

to the total number of requests but includes request service time variation within a device and between devices. In this algorithm, streaming accesses that do not fit in the cache are problematic as the algorithm does not currently detect this type of access.

Our algorithm measures the time spent waiting for each device over the past W device requests, where W is the window size, and records this as the *wait time* per device. If all of the wait times are approximately equal, then the current partition ratios are deemed adequate and they remain the same. If the wait times are not the same, then the size of those partitions with relatively large wait times should be increased, and the size of those partitions with relatively small wait times should be decreased. Of course, the ratio of the wait times across all devices should be considered simultaneously.

Selecting the appropriate amount by which to increment and decrement each partition is a non-trivial search problem, given that one does not know how a given change in partition size affects the future miss rate, especially in the presence of dynamically changing workloads. Thus, with our initial approach, we employ the simplest algorithm that we have found to meet our needs. The challenge is to find an algorithm that adjusts partition sizes quickly enough to find the right proportions, but not so quickly that the algorithm overshoots those correct proportions.

To meet both of these goals simultaneously, our approach aggressively increases the size of a partition when the wait time for the corresponding device is increasing and otherwise reacts in a conservative manner. As such, our algorithm makes observations about the wait time for each device during an epoch, and an action is taken based on the observation. A new epoch begins after W device requests complete and between epochs the cache is repartitioned.

Repartitioning occurs in four steps. First, the algorithm computes the per-device wait time and the mean wait time across devices during the epoch. Second, the algorithm computes a *relative wait time* for each partition by dividing the per-device wait time by the mean wait time. Next, the algorithm determines which partitions are *page consumers* and how many pages to give each consumer. Page consumers are partitions which have a relative wait time above a threshold T. A threshold is used to filter normal variations in the wait time not due to changes in workload or device characteristics. If no page consumers are found, repartitioning stops and the new epoch begins. Finally, the algorithm finds partitions with below-average wait times, called *page suppliers*, and reallocates pages from them to consumers until the consumers reach their desired size.

While repartitioning the cache, the algorithm classifies each partition into one of four states, and may take corrective action to change the partition size. The four states are shown in Figure 1 and are described as follows.

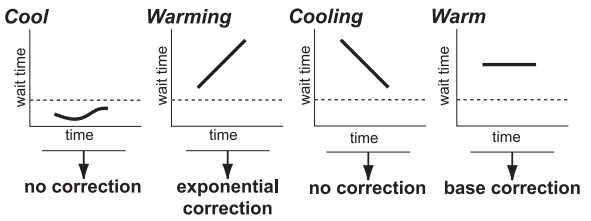

Figure 1: **Corrective actions taken by our repartitioning algorithm.** *This figure shows the four actions taken by our algorithm in response to four states. The graph shows the observation of the per-device wait time trend relative to the mean wait time as time progresses. A dotted line shows the mean wait time in each graph. Below the graphs are the actions taken in each state. While shown as fixed for overall clarity, the threshold is a constant value multiplied by the mean wait time.*

- **Cool: wait time below the threshold.** The wait time is within the normal operating regime. No corrective action is needed. Some cool partitions may become page suppliers, but none become page consumers.

- **Warming: wait time above threshold and increasing.** The algorithm infers an increasing wait time is due to changes in the workload or the device characteristics. Initially, the cache size is increased by I pages, where I is the *base correction amount*. If a partition continues to warm in subsequent epochs, the increase in cache size grows exponentially. A reclassification of a partition as warming from any other state restarts exponential correction.

- **Cooling: wait time above threshold and decreasing.** Corrective action during a set of epochs may have halted the increase in wait time for a partition and started a decline in wait time. The algorithm acts conservatively in the cooling state and does not change the partition size. A more aggressive approach that continues to increase the cache size for this state may over-correct and become unstable.

- **Warm: wait time above threshold and constant.** Based on experimental evidence, partitions are most often classified as cool, warming, or cooling. Thus, a constant wait time is unlikely to occur, and the partition moves to another state with a small change in the partition size. Thus, the algorithm increases the partition size by I.

The last step of the algorithm reallocates pages from page suppliers to page consumers. The algorithm biases collection of pages toward partitions that have the lowest relative wait times. To determine the number of pages removed from each page supplier, the algorithm first computes the *inverse relative wait time* ($IRWT$), which is just 1 - relative wait time, for each partition. Next, it sums the inverse relative wait times. Finally, the number of pages a partition j must supply is computed:

$$\frac{IRWT_j}{sum\ of\ IRWTs for suppliers} * \#\ of\ consumed\ pages$$

Note that there are three parameters to this algorithm: window size (W), threshold (T), and base increment amount (I). Each needs to be set with care. The value of W should be large to smooth out wait time variations and to sample a sufficient number of requests to determine accurately the effect of corrections. We have found $W = 1000$ provides sufficient smoothing and feedback. The value of I should be small since exponential correction is taken. We have found a value of 0.2% of the cache works well in practice. The threshold value, T, should be large enough to filter normal device performance fluctuation such as seek time. We have found $T = 5$ detects changes in wait time that warrant correction. Rather than use a fixed value of T, the algorithm could compute the threshold dynamically as the statistical variance of wait times and use the sum of the mean and the variance as the threshold. We do not discuss this adaptive approach here, but we plan to investigate it in the future.

3.2 Modifying Existing Replacement Algorithms

Not only do we desire to have a cost-aware cache that performs well, but also one that can be easily implemented in modern operating systems. Although attention has been paid to make it computationally efficient, LANDLORD needs a priority queue to efficiently find the lowest cost object and its use of L does not mesh well with common virtual memory hardware. Thus, it is not easy to combine LANDLORD with an existing code base.

Several modern operating systems, including Solaris [8], use a variant of the Clock page replacement policy in their unified page cache [7]. Thus, we desire an algorithm that can be incorporated easily into the Clock structure.

We introduce an extension of Clock that takes partitions into account, *Partitioned-Clock*. As in the base algorithm, Partitioned-Clock assumes that each page has a *use bit* which is set whenever the page is referenced; when a victim page is needed, the clock arm looks through successive pages for one that does not have its use bit set, clearing use bits as it sweeps. With Partitioned-Clock, each page also tracks the partition number to which it currently belongs; when a page is selected as a victim, not only must its use bit be cleared, but its partition number must match the partition number chosen for replacement (*e.g.*, as chosen by the lottery). We note that considering additional bits other than a single use bit is consistent with other variations of Clock that examine dirty bits or a history of multiple use bits.

There are a few optimizations that improve the performance of Partitioned-Clock. First, for the best approximation of lazy partitions, when searching for a replacement, only those pages belonging to the victim's partition should have their use bits cleared; clearing the use bits of all pages unnecessarily removes some of their usage history. Second, a separate clock hand for each partition also improves performance since it helps to further maintain the usage history of each partition.

As described previously, lazy partitions are simpler to implement than eager partitions when the ratio of their sizes is dynamic. Therefore, we focus on the Clock algorithm applied to lazy partitions. This version is termed *Lazy Clock*, and it uses inverse lottery scheduling to pick victims amongst partitions.

4 Evaluation Environment

This section describes our methodology for evaluating storage-aware caching. Specifically, it gives an overview of our simulator and describes our simulated storage environments.

4.1 Simulator

We have developed a trace-driven storage-system simulator to study the behavior of storage-aware caching. As configured, the simulated environment looks like a single client connected to sixteen network-attached storage devices. With our simulator, we are able to explore the performance impact of client workloads, data layout, caching algorithms, network characteristics, disk characteristics, and storage-system heterogeneity.

The simulator is driven by the workload of the client, which is specified in a trace file. The trace file represents data block requests that have been striped with RAID-0 across the full set of disks; each request specifies the starting offset and length of the data to read or write. To simulate a system under high demand, we consider a closed workload model, in which the completion of one disk request immediately triggers the next request.

The client has a local cache, with its replacement policies the focus of our investigation. We do not model the time for a cache hit, since it is small enough in a real system to be dwarfed by the cost of remote-block access. The time for a cache miss is the sum of network transit time plus the remote disk service time.

	Trace 1	Trace 2	Trace 3
Request distribution	uniform	exponential	exponential
Disk distribution	uniform	uniform	Gaussian
Locality	random	random	random
Mean request size	256 KB	34 KB	34 KB
Working set size	400 MB	425 MB	425 MB
# of requests	192,000	750,000	750,000

Table 1: **Characteristics of synthetic traces.** *This table summarizes the three synthetic traces used in the first set of experiments. The Gaussian distribution has a mean of disk 7 and a standard deviation of 3.*

Age (Years)	Bandwidth (MB/s)	Seek Avg (ms)	Rotation Avg (ms)
0	20.0	5.30	3.00
1	14.3	5.89	3.33
2	10.2	6.54	3.69
3	7.29	7.27	4.11
4	5.21	8.08	4.56
5	3.72	8.98	5.07
6	2.66	9.97	5.63
7	1.90	11.1	6.26
8	1.36	12.3	6.96
9	0.97	13.7	7.73
10	0.69	15.2	8.59

Table 2: **Aging an IBM 9LZX.** *We model the bandwidth, seek, and rotation time for a family of disks based on the IBM 9LZX manufactured in progressively older years. We assume bandwidth improves by a factor of 40% per year and seek and rotation time by 10% per year.*

Our storage device model roughly matches that described in Ruemmler and Wilkes [29]; we model cylinders and consider seek time, rotational delay, and bandwidth in calculating the transfer time for a given request. Specifically, if a disk request falls within the same cylinder as the previous request, we model it as sequential; *i.e.*, the seek and rotational delay are set to zero and the transfer time is determined by bandwidth. For non-sequential requests, the rotational delay is chosen uniformly at random from zero to a full rotation time; the seek time follows a non-linear model [29] and depends upon the cylinder distance between the current request and the previous request.

Our network model is based on LogGP [3] with endpoint contention. LogGP, which was designed to model communication within large parallel computers, depends on five parameters. L is the message latency through the network, o is the endpoint overhead, g is minimum time between message sends, G is the seconds per byte through the network, and P is the number of endpoints.

4.2 Workloads

To fully understand the impact of storage-aware caching algorithms, we study two sets of workloads: a variety of synthetic traces and a web server trace collected and analyzed by Roselli *et al.* [28]. We simply refer to the Roselli *et al.* web server trace as the Roselli trace.

We use synthetic workloads to control different request size distributions, working set size, locality distributions, and distribution of request across disks. The synthetic traces are summarized in Table 1. These traces are read only. Traces 2 and 3 have a variety of request sizes to stress small and large read requests, and Trace 3 adds a request imbalance across disks.

The Roselli trace is of an image server at the University of California, Berkeley from January 25, 1997. The image server ran a web server and a Postgres database, which stored the images. The trace alternates between large reads of several files, which are most likely the database tables, and small reads and writes.

4.3 Storage-System Characteristics

We have two goals in configuring the set of disks in our simulated environment. The first goal is to understand the full sensitivity of storage-aware caching algorithms to device heterogeneity; this requires a diverse range of configurations. The second goal is to understand how these algorithms perform in realistic scenarios; this requires a more focused set of tests.

To meet both of these goals simultaneously, we employ *device aging* and *performance-fault injection*. The idea behind device aging is to choose a base device (in our case, the IBM 9LZX) and age its performance over a range of years and use different collections of these disks to create configurations. The key, however, is that the performance of the base disk should not be scaled by a fixed amount; instead, each component (bandwidth, seek, and rotation time) should be scaled by its expected yearly improvement. Historical data suggests that a 40%/year improvement in bandwidth and roughly a 10%/year reduction in seek time and rotational latency is realistic (although perhaps on the aggressive side) [15]. Table 2 shows the performance characteristics of the aged devices used in our experiments. Note that although we consider progressively older disks (backwards aging), one could consider newer disks based upon the current year in a similar manner (forward aging).

Performance-fault injection allows us to dynamically

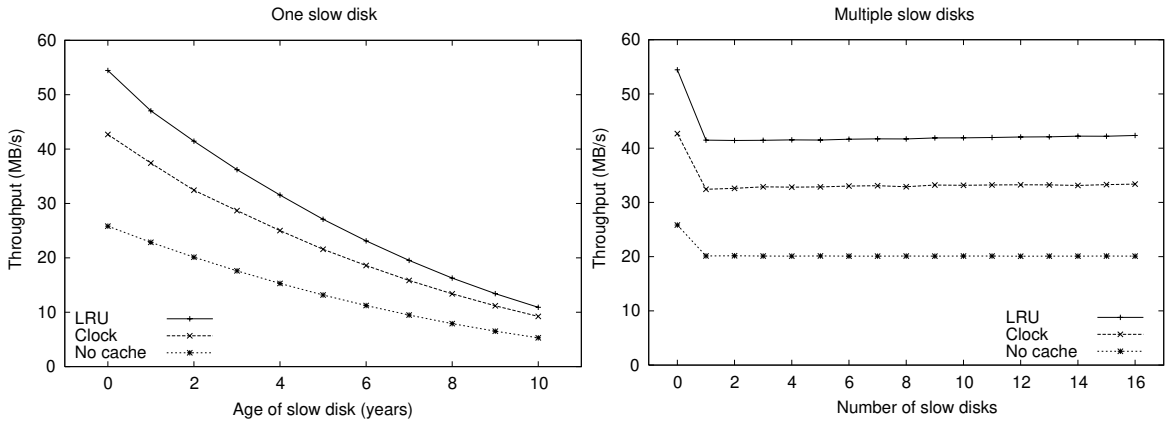

Figure 2: **Performance of LRU, Clock, and No Caching.** *The figures show the throughput of the storage system when Trace 1 is used. The figure on the left varies the age of a single disk along the x-axis. The figure on the right increases the number of two year old disks in the system.*

change the performance of a drive during an experiment. As described earlier, this could represent a disk stuttering before absolute failure, unexpected network traffic between the client and the drive, or a sudden workload imbalance.

4.4 Environment Configuration

This section describes the details of the simulator configuration. We configure the network so that it is not the bottleneck of the system and choose parameters that are similar to a 10 Gb/s Ethernet; thus, we set the bandwidth (*i.e.*, $1/G$) to 10 Gb/s, L to $1\mu s$, o to $0.4\mu s$, and g to 76 ns. In the future, we hope to investigate how network performance and caching interact in distributed storage systems.

We configured the simulator separately for the synthetic and Roselli traces. For the synthetic traces, we choose a sufficient number of requests to mitigate the effects of cold-start misses, and set the client cache size to 200 MB. For the Roselli trace, we set the cache size to 10 MB so that the hit rate is near 50%. If the hit rate is too high, few requests are sent to the disks, and thus, the heterogeneity of device performance is less of an issue. Since these traces did not include disk layout and file path information, we created a simple layout policy. The layout policy assumes RAID-0 striping. The policy lays out blocks in the order of first access.

We aged the disks in two scenarios. Both scenarios represent cases where the storage-system has been incrementally updated; that is, newer, faster devices have been added over time. In the first scenario, there is a single heterogeneous disk whose performance is aged across the entire range of years. In the second scenario, there are two groups of heterogeneous disks, one group with an age of zero, the other with an age of two years, and the relative size of the two groups is varied.

While these scenarios do not cover all real-world situations, they provide insight for common configurations. The first scenario mimics stuttering disks and increased workloads from other clients. The second scenario closely follows incremental upgrades of a disk array. Incremental upgrades often occur due to cost constraints that prohibit the replacement of an entire array when a small number of the disks fail.

5 Experiments

This section presents a progression of experiments demonstrating the effectiveness of storage-aware caching. We begin by motivating the need for cost-aware caching algorithms given heterogeneous devices. We then show that partitioned approaches can mask performance differences, but that configuring fixed partition ratios correctly is difficult even in a static environment. Next, we demonstrate that we can mask performance heterogeneity by adjusting the ratio of partition sizes according to on-line observations of the amount of work performed by each device. Finally, we show that partitioned approaches can be easily incorporated into operating system replacement policies and still perform quite well, and explore their performance robustness on a trace of a web server.

5.1 A Motivating Example

Our first set of experiments motivates the need for storage-aware caching algorithms given a storage system containing heterogeneous devices. Figure 2 shows the throughput obtained for Trace 2 using two common replacement policies that are not cost-aware, LRU and

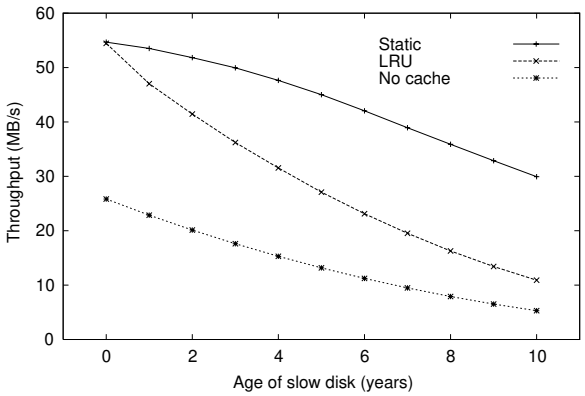

Figure 3: **Potential of partitioned approaches.** *One slow disk was aged as shown by the x-axis for three approaches: no cache, LRU caching, and a static partitioning of the cache according to disk performance. Trace 1 was used for this figure.*

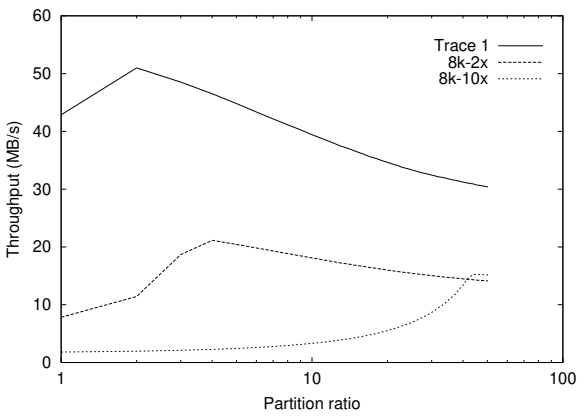

Figure 4: **Sensitivity of static partitioning on partition ratios and workload.** *The graph shows the performance of three different workloads run on 16 disks with one two-year old disk. The experiment varies the ratio of the slow disk partition to each of the fast disk partitions on the x-axis. The lower two lines both use 8 KB requests, but vary the ratio of requests sent to the slow disk versus the others, using either a ratio of 2:1 or 10:1.*

Clock, as well as with no caching. The graph on the left illustrates that as one of the disks is aged (and its seek, rotation time, and bandwidth decline) the throughput of the system drops dramatically, and the performance benefit of having a file cache decreases. For example, with LRU replacement, throughput drops from nearly 55 MB/s when all of the disks are equally fast down to only 11 MB/s when just a single disk has the performance of a 10-year old disk. Similarly, the graph on the right shows that the entire storage system runs at the rate of the slowest disk in the system; that is, the throughput with one slow disk and 15 fast disks is as poor as with all slow disks.

In contrast, a storage-aware caching algorithm should mask the performance of slow disks by allocating more of the cache to the slow disks; the slow disk thus has fewer requests to handle and does not harm the performance of the system as dramatically.

5.2 Configuring Partition Sizes

In our next set of experiments, we show that partitioned caching algorithms have the potential to mask heterogeneous performance, but that care must be taken in selecting the ratio of partition sizes. We begin by examining a static partitioning algorithm simply named *Static*.

In Figure 3, we show the performance of Static for Trace 1. In these experiments, the ratio of partition sizes is statically set and is directly proportional to the ratio of the expected service time of each disk. Since we use Trace 1, we know the mean request size is 256 KB. As such, we directly compute the expected transfer time from each disk as a function of its seek time, rotation delay, and peak bandwidth. The graph indicates a static partition strategy significantly improves performance relative to cost-oblivious algorithms such as LRU. However, Static performs well only when a priori mean request size and per-disk miss rate as a function of cache size is known. In the real world, this information is not known in advance.

The difficulty of correctly configuring a static partition strategy is illustrated in Figure 4. In this experiment, we examine a single storage configuration in which the one slow disk is two years older than the other disks. Along the x-axis, the graph varies the ratio of the partition sizes between the slow disk and each of the other disks in the system; for example, when this value is greater than one, the slow disk is given a correspondingly larger partition.

The three lines in the graph correspond to three different workloads, each of which has a different optimal value for the partition size ratios. The top line is the same workload as examined above; we verify that the highest throughput for this workload is approximately 50 MB/s, which matches that shown in Figure 3 with a two-year old disk. In the second and third lines, the distribution of requests across disks is changed such that the slow disk receives either twice or ten times as many requests as the other disks. The graph shows that each of these workloads has a different optimal partition ratio (*e.g.*, the best ratio for the top workload is 2:1 whereas the best ratio for the bottom workload is nearly 40:1). Further, the performance of each workload varies greatly with the partition ratio (*e.g.*, the performance of the 256 KB workload varies from 50 MB/s to 30 MB/s). This indicates we need

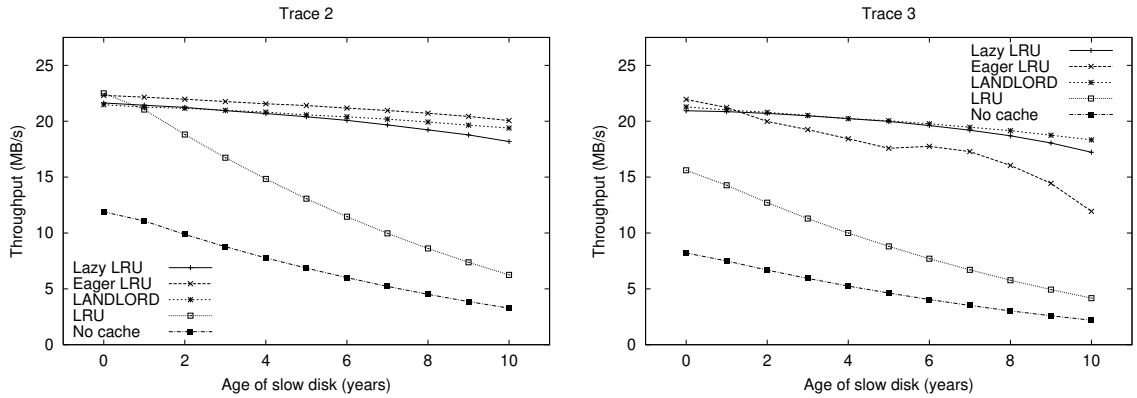

Figure 5: **Dynamic partitioning algorithms.** *The figure shows the performance of Eager LRU, Lazy LRU, and LANDLORD as one disk is aged. The left graph uses Trace 2 while the right graph uses Trace 3.*

an approach to select the partition ratios dynamically as a function of both the workload and the disk performance.

5.3 Partitioning to Balance Work

Our next set of experiments shows that by dynamically adjusting the size of each partition, our algorithms balance the amount of work performed by each disk and thus effectively hide heterogeneity. In doing so, we use the two classes of dynamic partitioning: eager partitioning and lazy partitioning. Lazy partitioning uses inverse lottery scheduling to pick victim partitions at replacement time. Both eager and lazy use LRU within their partitions. For simplicity, we refer to the first approach as *Eager LRU* and the second approach as *Lazy LRU*. In these experiments, we investigate Trace 2 and Trace 3 for a more realistic evaluation while continuing with well understood workload parameters.

Figure 5 compares the performance of the Eager LRU and Lazy LRU storage-aware algorithms to LRU and LANDLORD. In the left-most graph of the figure, we examine the workload with a uniform number of requests across disks. With this setup, the throughput with LRU degrades dramatically as the performance of the one slow disk is aged; specifically, throughput drops from approximately 23 MB/s to only 6 MB/s. Eager LRU and Lazy LRU are able to maintain the throughput of the system as the slow disk is aged; specifically, the performance of these algorithms is similar to that of LRU when all of the disks are the same speed, but with a ten-year-old disk, they are able to mask the impact of the slow disk and keep throughput between 16 and 20 MB/s.

The right graph of Figure 5 shows the challenges of a non-uniform number of requests across disks. Interestingly, even when all of the disks are identical, all of the cost-aware algorithms perform better than LRU. With this workload, the popular disks suffer from contention, and thus queueing delays make blocks from those disks more costly to fetch. By monitoring replacement cost, the cost-aware algorithms devote more of the cache to the popular disks and thus better balance the load across all of the disks. As with the previous workload, the performance benefits of cost-aware caching improve as one of the disks is aged. For a 10-year old disk, Eager LRU and LRU differ by over a factor of three.

Comparing the performance of Eager LRU, Lazy LRU, and LANDLORD, one sees that the performance of the three algorithms is similar, but not identical. The right graph most clearly shows the difference. Eager LRU is not as performance robust as Lazy LRU and LANDLORD. While Lazy LRU devotes the entire cache to the slow disk, Eager LRU continues to allocate a small amount of the cache to the fast disks. The immediate repartitioning of Eager LRU aggravates efforts to find a good partition size on Trace 3.

5.4 Clock-Based Replacement

As noted in Section 3.2, several operating systems use the Clock algorithm. However, Clock is not cost-aware. Thus, in this section, we evaluate the use of a lazy partitioned algorithm called Lazy Clock as a practical virtual memory page replacement algorithm. Again, we use Traces 2 and 3 and compare Lazy Clock to Clock. Experimental results are found in Figures 6 and 7.

In Figure 6, Lazy Clock performs well. As desired, Lazy Clock gives a greater proportion of the cache to slower devices and devices with more requests. Thus, Lazy Clock is able to mask performance differences even as the speed of the one slow disk degrades significantly. For example, with the imbalanced workload, Lazy Clock begins with a throughput of approximately 21 MB/s when all disks are identical and degrades to only about 17 MB/s when the slow disk is a full 10-years older

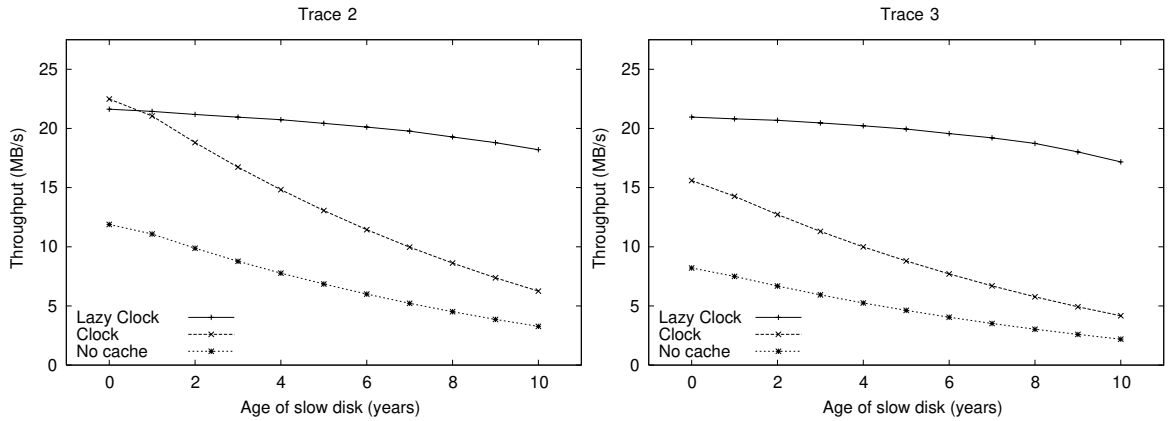

Figure 6: **Clock-based replacement algorithms.** *The figure shows the performance of Lazy Clock and Clock as one disk is aged. The same workloads are investigated as in Figure 5.*

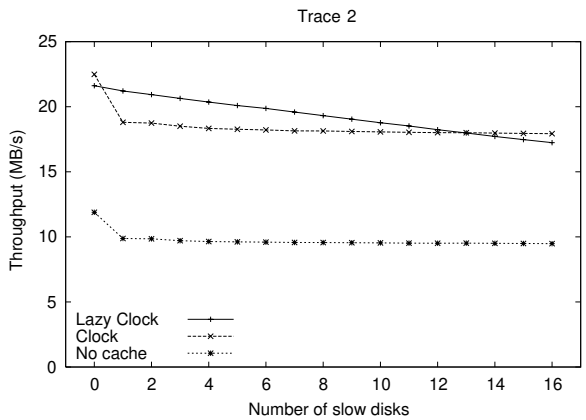

Figure 7: **Clock-based with multiple old disks.** *The figure shows the performance of Lazy Clock and Clock as the number of disks of age two years is increased. Trace 2 is used, which is the same workload as in the left graph of Figure 5.*

than the others. This throughput compares favorably to LANDLORD and Lazy LRU in Figure 5.

Figure 7 shows how Lazy Clock gracefully masks an increasing number of two-year old disks. While Clock is affected by any heterogeneity, the performance of Lazy Clock only slowly degrades. While the performance does not match Clock when the system is homogeneous, such as when there are zero or 16 two-year old disks, the performance is fairly close. Our experience has shown that a smaller base correction size when the devices are homogeneous can remove the discrepancy.

5.5 Dynamic Changes in Performance

To evaluate tolerance to performance faults, we show that our partitioned caching algorithms are able to react to changes in the relative performance of storage devices, in some cases more effectively than LANDLORD is able. In these experiments, we begin with a cluster of homogeneous disks and Trace 1. We inject a performance fault on one of the disks (disk 6) at a simulated time of 500 seconds (approximately half way through the simulation). The performance fault has the effect of slowing down that disk by a factor of two.

In the first graph of Figure 8 we show how Eager LRU adjusts the partition ratios for this change in performance. As one can see, the 16 partitions are all initially equal. After the performance fault and a window of W = 1000 disk requests have passed for observation, the algorithm observes that the waiting time of disk 6 is significantly higher than the average waiting time. The partition for disk 6 is then increased by a small amount and the partitions for all of the other disks are decreased by the necessary number of cache entries. The algorithm continues measuring the wait time of each disk and increasing the partition size of disk 6 until all of the wait times are approximately equal. The time-line shows that the correct partition ratio is found quickly.

In the second graph of Figure 8 we summarize our results by comparing the performance of LANDLORD, Eager LRU, Clock, and Lazy Clock. We plot throughput for the same workload as above when a performance fault occurs on one disk, four disks, or eight disks simultaneously. When the number of affected disks is small relative to the total number of disks in the system, aggregating replacement-cost information is beneficial. Specifically, Eager LRU achieves a throughput of approximately 54 MB/s with one performance fault, while LANDLORD maintains only 48 MB/s. Lazy Clock performs nearly as well as the others, and finally the cost-oblivious Clock algorithm performs the least well.

When the number of faulty disks increases to eight,

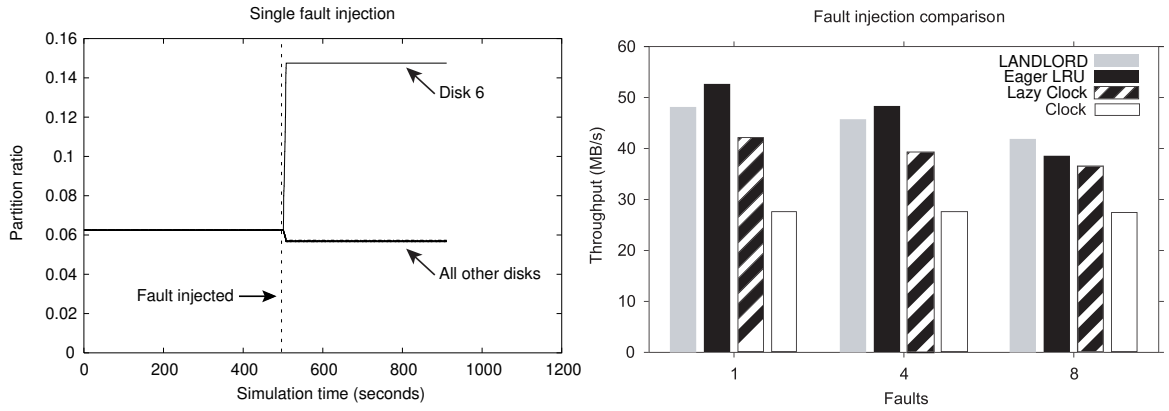

Figure 8: **Performance with dynamic faults.** *We consider a cluster of initially homogeneous disks and a performance fault at time 500 seconds that slows a disk by a factor of two. In the graph on the left, the performance fault occurs on a single disk and we show the partition sizes chosen by the Eager LRU algorithm. In the graph on the right, the performance fault occurs on either one, two, or four disks. In all cases, Trace 1 was used.*

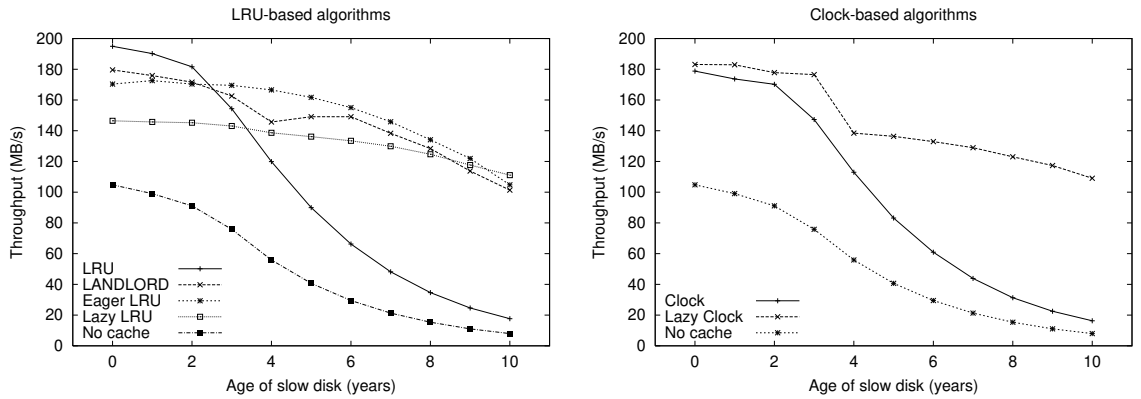

Figure 9: **Web server workload.** *This figure shows the performance of LRU- and Clock-based algorithms when run on a file system trace of a web server providing art images. The results of LRU-based algorithms are in the graph on the left side. Clock-based algorithms are in the right graph.*

the results change. Eager LRU does not mask the performance faults as well as LANDLORD. When there are two groups of disks with similar performance characteristics, our correction algorithm does not detect the severity of heterogeneity, making smaller corrections than are needed. An adaptive threshold value may help.

5.6 Real-World Performance

We conclude our experiments with an examination of our partitioned algorithms on a web workload. As the web server received a modest number of requests, this trace is shorter than our synthetic traces. Our partitioned algorithms are partially penalized by the shortness of the trace as they first need to move the partition sizes from an initial state. Similar to other experiments, we only investigate the aging of a single disk.

The results are shown in Figure 9. As expected, the cost-oblivious algorithms show a sharp drop off in performance as the age of the slow disk increases. LRU's performance falls to 10% of its peak performance over the age range and, for a more realistic range, falls nearly 30% when aged from zero to four years old. Clock shows a similar, though slightly less dramatic decline.

As expected, the adaptive algorithms show more robust performance. The performance degrades by 15% for Lazy LRU, 38% for Eager LRU, and 45% for LANDLORD. However, Lazy LRU performs more poorly than Eager LRU or LANDLORD. Lazy LRU has a poor interaction with the repartitioning algorithm and devotes the entire cache to the slow disk even when the disks are homogeneous. Eager LRU, however, distributes pages more evenly between pages and gradually changes the

distribution as the disk ages.

Lazy Clock also shows a relatively small decrease in performance of 40%. However, there is a sharp drop as the slow disk changes from three to four years old. This drop is due to a significant change in the partition ratios. Lazy Clock strongly favors the slow disk at age four, but has a weak preference at age three. The performance of LANDLORD also decreases between age three and four, but then increases. The trace's bimodal distribution of request sizes due to Postgres table reads interleaved with small web server reads and writes may introduce anomalous behavior.

Based on experiments not shown here due to space constraints, the performance of dynamic partitioning algorithms is sensitive to the base correction amount. The experiment in Figure 9 uses a fixed base correction amount across the range of slow disk ages. If the base correction amount starts small and increases as the disk ages, performance improves and nearly matches the base cost-oblivious algorithm for a homogeneous system. Thus, an adaptive base correction amount is needed for better performance.

6 Related Work

Work on cost-aware caching has occurred in the web cache and database communities. The web cache community has extensively studied cost-aware caching [10, 17, 18, 27, 38], with the addition of document size included in many of their algorithms. The web caching work differs from storage-aware caching in several ways. First, performance in the wide area varies much more than is common for storage systems. Second, web caching often uses whole document caching, which differs from fixed-size blocks used in storage systems. Finally, in web caching, the replacement cost of one web page is not strongly correlated with the replacement costs of other pages.

Broadcast disks [1, 2] continuously deliver data to clients through an asymmetric link following a broadcast schedule that is best able to meet the client's needs. When a client's needs are not met by the broadcast schedule, the client cache strives to manage the cache contents to mask the non-ideal broadcast schedule. Using knowledge of the broadcast schedule and probability of access, the cache manages its contents using an algorithm that generalizes LRU. Storage-aware caching differs in two ways. First, while it partitions cache pages by device, broadcast disks use a page rather than a device granularity to track replacement costs. Second, broadcast disks assume an infrequently changing broadcast schedule, whereas storage-aware caching must react to frequent changes in workload and device performance.

Recently, researchers have studied allocation of pages between different classes in prefetching [9, 24, 35], compiler-controlled memory management [14], and resizeable file buffer caches [22]. In prefetching, page allocation occurs between applications [9] or hinted and unhinted I/O references [24, 35]. For compiler-controlled memory management, the compiler provides application memory usage information to operating system global replacement policies using hints, reintegrating elements of local page replacement into global page replacement. Finally, Nelson's work [22] on resizeable file buffer caches evaluates the tradeoff between file buffer caches and virtual memory when a system is loaded. The work in these three areas is closely related to our work but does not directly address storage device heterogeneity.

7 Future Work

While storage-aware caching increases the storage system performance robustness by adapting to performance differences of devices, we see further areas of improvement and one new application domain, in addition to a study of a real implementation.

First, the partitioning algorithm presented has two significant limitations that more sophisticated and informed cost-benefit algorithms do not have. The limitations are a linear relationship assumption between cache size and hit rate and a reliance on proper values for window size, base increment amount, and threshold. The first limitation is evident if one considers access patterns with little locality or a working set larger than the cache. Intuitively, the algorithm should recognize instances where increasing the cache size does not decrease wait time.

Second, we believe a general framework for storage-aware caching where existing cost-oblivious policies can manage individual partitions should be studied. A framework approach has modularity as its primary strength; existing non-cache aware policies such as LRU, Clock, MRU, and EELRU [32] can be used with minimal effort and changes.

Third, our work has concentrated on non-cooperative client caching. We believe the combination of cooperative caching and cost-aware caching will lead to better performance robustness, especially for disk arrays where individual cache sizes are small.

Fourth, storage-aware caching can be applied to low-power environments. Storage-aware caching can be extended to include power as a retrieval cost. Thus, devices that are higher in power will likely be accessed less frequently and stay in low-power mode longer and more frequently.

Finally, since caching algorithms are affected by prefetching and layout decisions, we would like to explore the advantages and tradeoffs of integrated prefetching, layout, and caching decisions, in light of device het-

erogeneity. Previous caching and prefetching work [9, 24] in homogeneous environments has shown the benefits of integration. We believe this benefit extends to heterogenous environments.

8 Conclusions

Given the diverse characteristics of modern storage devices, we believe the time is ripe to re-investigate caching algorithms. To optimize performance, the task of a cost-aware cache is to control which blocks are cached, such that the amount of work performed by each storage device is roughly equal. In this paper, we have presented a family of cost-aware caching algorithms that are based on the notion of explicitly partitioning the cache; the size of each partition is configured such that it directly corresponds to the relative cost and usefulness of the data in that partition. These approaches have two advantages. First, partitions are able to aggregate replacement-cost information across many entries in the cache, reducing the amount of information that must be tracked and allowing the most recent cost information to be used for all blocks from the same device. Second, and most important, a virtual partition approach can be easily implemented within the Clock replacement policy, increasing the likelihood of adoption in real systems.

Acknowledgements

We thank members of the Wisconsin Network Disks research group for helpful discussions, contributions, and paper comments. In particular, Florentina I. Popovici provided the IBM 9LZX disk profile information used in our simulations, and Leslie Cheung wrote the layout code used in the Berkeley trace experiments. Omer Zaki contributed to early versions of this work. Comments from anonymous reviewers and particularly the tireless shepherding of Jeff Chase greatly improved the quality of the paper. The Condor distributed execution system was used to run simulations. Members of the Condor project, especially Todd Tannenbaum, supported our use of Condor. This work is sponsored by NSF CCR-0092840, CCR-0098274, ITR-0086044, and the Wisconsin Alumni Research Foundation.

References

[1] S. Acharya, R. Alonso, M. J. Franklin, and S. B. Zdonik. Broadcast Disks: Data Management for Asymmetric Communications Environments. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, CA, May 22-25, 1995*, pages 199–210. ACM Press, 1995.

[2] S. Acharya, M. J. Franklin, and S. B. Zdonik. Dissemination-based Data Delivery Using Broadcast Disks. *IEEE Personal Communications*, 2(6), December 1995.

[3] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model - One Step Closer Towards a Realistic Model for Parallel Computation. In *Papers of the 7th Annual ACM symposium on Parallel Algorithms and Architectures, Santa Barbara, CA, June 24 - 26, 1995*, pages 95–105. ACM Press, 1995.

[4] Apple Computer Corporation. iDisk. http://itools.mac.com.

[5] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, J. Bent, B. Forney, S. Muthukrishnan, F. Popovici, and O. Zaki. Manageable Storage via Adaptation in WiND. In *Proceedings of IEEE Int'l Symposium on Cluster Computing and the Grid (CCGrid' 2001)*, pages 169–177, May 15-18 2001.

[6] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. Fail-Stutter Fault Tolerance. In *Workshop on Hot Topics in Operating Systems (HotOS 8)*, Schloss Elmau, Germany, May 2001.

[7] O. Babaoglu and W. Joy. Converting a Swap-Based System to do Paging in an Architecture lacking Page-Referenced Bits. In *Proceedings of the 8th ACM Symposium on Operating System Principles*, pages 78–86, Pacific Grove, CA, December 1981.

[8] J. L. Bertoni. Understanding Solaris Filesystems and Paging. Technical Report TR-98-55, Sun Microsystems, 1998.

[9] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. Implementation and Performance of Integrated Application-Controlled File Caching, Prefetching, and Disk Scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343, November 1996.

[10] P. Cao and S. Irani. Cost-Aware WWW Proxy Caching Algorithms. In *USENIX Symposium on Internet Technologies and Systems Proceedings, Monterey, California, December 8–11, 1997*, pages 193–206, Berkeley, CA, USA, 1997.

[11] L. R. Carley, G. R. Ganger, and D. F. Nagle. MEMS-based Integrated-Circuit Mass-Storage Systems. *Communications of the ACM*, 43(11):72–80, November 2000.

[12] M. Chrobak, H. Karloff, T. Payne, and S. Vishwanathan. New Results on Server Problems. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '90)*, pages 291–300, San Francisco, CA, USA, Jan. 1990.

[13] T. Cortes and J. Labarta. Extending Heterogeneity to RAID level 5. In *Usenix 2001 Annual Technical Conference*, June 2001.

[14] A. Demke Brown and T. C. Mowry. Taming the Memory Hogs: Using Compiler-Inserted Releases to Manage Physical Memory Intelligently. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI-00)*, pages 31–44, Berkeley, CA, October 23–25 2000.

[15] E. Grochowski. IBM Leadership in Disk Stroage Technology. IBM Corporation, 2000.

[16] 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 of Computer Systems*, 6(1):51–81, February 1988.

[17] S. Jin and A. Bestavros. Popularity-Aware GreedyDual-Size Algorithms for Web Access. In *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS)*, April 2000.

[18] T. Kelly, Y. M. Chan, S. Jamin, and J. K. MacKie-Mason. Biased Replacement Policies for Web Caches: Differential Quality-of-Service and Aggregate User Value. In *Fourth International Web Caching Workshop, San Diego, California, 31 March - 2 April 1999*, 1999.

[19] J. Kubiatowicz, D. Bindel, P. Eaton, Y. Chen, D. Geels, R. Gummadi, S. Rhea, W. Weimer, C. Wells, H. Weatherspoon, and B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, November 2000.

[20] M. Manasse, L. McGeoch, and D. Sleator. Competitive Algorithms for On-line Problems. In *Proceedings of the twentieth annual ACM Symposium on Theory of Computing, Chicago, Illinois, May 2–4, 1988*, pages 322–333, New York, NY, USA, 1988.

[21] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[22] M. N. Nelson. Virtual Memory vs. The File System. Technical Report 90.4, Compaq Computer Corporation, 2000.

[23] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. AlphaSort: A RISC Machine Sort. In *1994 ACM SIGMOD Conference*, May 1994.

[24] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 79–95, Copper Mountain, CO, December 1995. ACM Press.

[25] Pro Softnet Corporation. iBackup. http://www.ibackup.com/.

[26] R. Rashid, A. Tevanian, M. Young, D. Golub, D. Baron, D. Black, W. Bolosky, and J. Chew. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pages 31–39, Palo Alto, CA, October 1987. Association for Computing Machinery, IEEE.

[27] L. Rizzo and L. Vicisano. Replacement Policies for a Proxy Cache. *IEEE/ACM Transactions on Networking*, 8(2):158–170, 2000.

[28] D. Roselli, J. R. Lorch, and T. E. Anderson. A Comparison of File System Workloads. In *Proceedings of the 2000 USENIX Annual Technical Conference (USENIX-00)*, pages 41–54, Berkeley, CA, June 18–23 2000.

[29] C. Ruemmler and J. Wilkes. An Introduction to Disk Drive Modeling. *IEEE Computer*, 27(3):17–28, March 1994.

[30] R. Sandberg. The Design and Implementation of the Sun Network File System. In *Proceedings of the USENIX Summer Conference*, pages 119–130, Berkeley, CA, USA, June 1985. USENIX Association.

[31] J. L. Sanford, P. F. Greier, K. H. Yang, M. Lu, R. S. Olyha, Jr., C. Narayan, J. A. Hoffnagle, P. M. Alt, and R. L. Melcher. A One-megapixel Reflective Spatial Light Modulator System For Holographic Storage. *IBM Journal of Research and Development*, 42(3/4):411–426, 1998.

[32] Y. Smaragdakis, S. Kaplan, and P. Wilson. EELRU: Simple and Effective Adaptive Page Replacement. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computing Systems (SIGMETRICS-99)*, volume 27,1 of *SIGMETRICS Performance Evaluation Review*, pages 122–133, New York, May 1–4 1999. ACM Press.

[33] D. G. Sullivan and M. I. Seltzer. Isolation with Flexibility: A Resource Management Framework for Central Servers. In *Proceedings of the 2000 USENIX Annual Technical Conference (San Diego, California), pp. 337-350*, 2000.

[34] The Infiniband Trade Association. http://www.infinibandta.org, June 2001.

[35] A. Tomkins, R. H. Patterson, and G. Gibson. Informed Multi-Process Prefetching and Caching. In *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 100–114. ACM Press, June 1997.

[36] A. Vahdat, T. Anderson, M. Dahlin, E. Belani, D. Culler, P. Eastham, and C. Yoshikawa. WebOS: Operating System Services For Wide Area Applications. In *Proceedings of the Seventh Symposium on High Performance Distributed Computing*, July 1998.

[37] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, Nov 1994.

[38] R. P. Wooster and M. Abrams. Proxy Caching that Estimates Page Load Delays. In *Proceedings of the 6th International WWW Conference*, April 1997.

[39] N. E. Young. The K-server Dual and Loose Competitiveness for Paging. *Algorithmica*, 11(6):525–541, June 1994.

[40] N. E. Young. On-line File Caching. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, Balitmore, MD, January 17-19, 1999*, pages 82–86. ACM Press, 1999.