Configuring and Scheduling an Eager-Writing Disk Array for a Transaction Processing Workload

Chi Zhang* Xiang Yu* Arvind Krishnamurthy† Randolph Y. Wang*

Abstract

Transaction processing applications such as those exemplified by the TPC-C benchmark are among the most demanding I/O applications for conventional storage systems. Two complementary techniques exist to improve the performance of these systems. *Eager-writing* allows the free block that is closest to a disk head to be selected for servicing a write request, and *mirroring* allows the closest replica to be selected for servicing a read request. Applied individually, the effectiveness of each of these techniques is limited. An *eager-writing disk array* (EW-Array) combines these two complementary techniques. In such a system, eager-writing enables low-cost replica propagation so that the system can provide excellent performance for both read and write operations while maintaining a high degree of reliability. To fully realize the potential of an EW-Array, we must answer at least two key questions. First, since both eager-writing and mirroring rely on extra capacity to deliver performance improvements, how do we satisfy competing resource demands given a fixed amount of total disk space? Second, since eager-writing allows data to be dynamically located, how do we exploit this high degree of location independence in an intelligent disk scheduler? In this paper, we address these two key questions and compare the resulting EW-Array prototype performance against that of conventional approaches. The experimental results demonstrate that the eager-writing disk array is an effective approach to providing scalable performance for an important class of transaction processing applications.

1 Introduction

Transaction processing applications such as those exemplified by TPC-C [27] tend to pose more difficult challenges to storage systems than office workloads. These applications exhibit little locality or sequentiality; a large percentage of the I/O requests are writes, many of which are synchronous; and there may be little idle time.

Traditional techniques that work well for office workloads tend to be less effective for these transaction processing applications. Memory caching provides little relief in the presence of poor locality and a small read-to-write ratio. As disk areal density improves at 60-100% annually [9], as memory density improves at only 40% per year, and as the amount of data in transaction processing systems continues to grow, one can expect little improvement in cache hit rates in the near future. Delayed write techniques become less applicable in the presence of a large number of synchronous writes that must satisfy strict reliability requirements, requirements that are sometimes not met by even expensive NVRAM-based solutions [13]. Even when it is possible to buffer delayed writes in faster storage levels, such as an NVRAM, the poor write locality implies that there are very few overwrites before the buffered data reaches disks. Furthermore, high throughput requirements in conjunction with scarce idle time make it difficult to schedule background activities, such as de-staging from NVRAM [13, 20], garbage collection in log-structured solutions [13, 21, 22, 24], and data relocation [19], without impacting foreground I/O activities. The net effect of these challenges is that transaction processing applications tend to be more closely limited by disk latency, a performance characteristic that has seen an annual improvement of only about 10% [9].

Although the traditional caching and asynchronous I/O techniques have not been very successful, a number of other techniques have proven promising. One is mirroring: a mirrored system can improve read latency by sending a read request to the disk whose head is closest to a target replica [2, 5], and it can improve throughput by intelligently scheduling the requests in a load-balanced manner. Mirroring, unfortunately, is not without its challenges. Chief among them is the cost

*Department of Computer Science, Princeton University, {chizhang,xyu,rywang}@cs.princeton.edu.

†Department of Computer Science, Yale University, arvind@cs.yale.edu.

This work is supported in part by IBM. Wang is supported by NSF Career Award CCR-9984790. Krishnamurthy is supported by NSF Career Award CCR-9985304.

of replica propagation—each write request is turned into multiple physical writes that compete for I/O bandwidth with normal I/O operations. High update rates, the lack of idle time for masking replica propagation, and poor locality only make matters worse.

While mirroring is more effective for improving a read-dominant workload, a technique called *eager-writing* is more effective for improving a write-dominant workload. Eager-writing refers to the technique of allocating a free block that is closest to the current disk head position to satisfy a write request [4, 6, 28]. Under the right conditions, by eliminating almost all of seek and rotational delay, eager-writing can deliver very fast write performance without compromising reliability guarantees, even for workloads that comprise of synchronous I/Os and have poor locality. What eager-writing does not address, however, is read performance.

Since data replication in a mirrored system improves read performance, and since eager-writing improves write performance, reduces the cost of replica propagation, and ensures a high degree of data reliability, it is only natural to integrate these two techniques so that we may harvest the best of both worlds. We call the result of this integration an *eager-writing array* or an *EW-Array*: in the simplest form, an EW-Array is just a mirrored system that supports eager-writing.

This integration, however, is not without its own tension. In order to achieve good write performance under eager-writing, one must reserve enough disk space to ensure that an empty block can be located close to the current disk head position. At the same time, to achieve good read performance under mirroring, the system needs to devote disk space to store a sufficient number of replicas so that it can choose a conveniently located replica to read. Given a fixed budget of disks, one must resolve this tension by carefully balancing the number of disks devoted to each of these two dimensions. To further complicate the matter, striping can improve both read and write performance, so one must also consider this third dimension of the number of disks devoted to striping. Although configuring a storage system based on the number of disk heads instead of capacity for TPC-C-like workloads is a common practice, and some previous studies such as the "Doubly Distorted Mirror" have incorporated "write-anywhere" elements in a disk array [19], what is not well understood is how to balance the number of disks devoted to each one of the mirroring, eager-writing, and striping dimensions to get the most out of a given number of disks.

While properly *configuring* an EW-Array along these three dimensions presents one challenge, request *scheduling* on such a disk array presents another. In the request queue of a traditional update-in-place storage system, the locations of all the queued requests are known. Although the scheduler can sometimes choose among several mirrored replicas to satisfy a request, the degree of freedom is limited. This is no longer the case for an EW-Array: while the locations of the read requests are known, the scheduler has the freedom of choosing *any* combination of free blocks to satisfy the write requests. Although disk scheduling is a well-studied problem in conventional systems, what is not well understood is how a good scheduler can exploit this large degree of freedom to optimize throughput.

The main contributions of this paper are:

- a disk array design that integrates eager-writing with mirroring in a balanced configuration to provide the best read and write performance for a transaction processing workload,
- a number of disk array scheduling algorithms that can effectively exploit the flexibility afforded by the location-independent nature of eager writing, and
- evaluation of a number of alternative strategies that share the common goal of improving performance by introducing extra disk capacity.

We have designed and implemented a prototype EW-Array. Our experimental results demonstrate that the EW-Array can significantly outperform conventional systems. For example, under the TPC-C workload, a properly configured EW-Array delivers 1.4 to 1.6 times lower latency than that achieved on highly optimized striping and mirroring systems. The same EW-Array achieves approximately 2 times better sustainable throughput.

The remainder of this paper is organized as follows. Section 2 motivates the integration of eager-writing with mirroring in an EW-Array. Section 3 explores different EW-Array configurations as we change the way the extra disk space is distributed. Section 4 analyzes a number of new disk scheduling algorithms that exploit the location independent nature of eager-writing. Section 5 describes the integrated simulator and prototype EW-Array. The experimental results of Section 6 evaluate a wide range of disk array configuration alternatives. Section 7 describes some of the related work. Section 8 concludes.

2 Eager-Writing Disk Arrays

In this section, we explain how eager-writing, mirroring, striping, and the combination of these techniques can effectively improve the performance of TPC-C-like applications.

2.1 Eager-writing

In a traditional update-in-place storage system, the addresses of the incoming I/O requests are mapped to fixed physical locations. In contrast, under eager-writing, to satisfy a write request, the system allocates a new free block that is closest to the current disk head position [4, 6, 28]; consequently, a logical address can be mapped to different physical addresses at different times.

A number of characteristics associated with eager-writing make it suitable for transaction processing applications. The chief advantages of eager-writing are excellent small write performance (in terms of both latency and throughput) and a high degree of reliability. The main component of the eager-writing latency is the time it takes for the closest free block to rotate under the disk head. Even at a relatively high disk utilization of 80% and a disk block size of 4 KB, this latency is well below 1 ms and can be made even lower with lower disk utilization. Furthermore, the improvement of this latency scales with that of platter bandwidth, which is improving much more quickly than seek and rotational delays experienced by update-in-place systems. This performance advantage of eager-writing is particularly appealing to a TPC-C-like workload, which has a large percentage of small writes. By committing the data synchronously to the disk platter, eager-writing also achieves a high degree of data reliability, a degree of reliability that is unmatched by even NVRAM-based solutions, which typically have far worse mean-time-to-failure characteristics [13].

Of course, no storage system can cater to all workloads equally successfully, and eager-writing is certainly no exception. One example is frequent sequential reads following random updates—eager-writing would destroy locality during the random updates, thus resulting in poor sequential read performance. One possible remedy is periodic data reorganization that restores physical data sequentiality. Fortunately, such complications do not arise in TPC-C-like workloads, which are characterized by small reads and writes with little locality. Another difficulty that may arise with eager-writing is caused by an uneven distribution of free blocks. For example, if free blocks are concentrated in one part of the disk but the disk head is forced by read requests into regions with few free blocks, then a subsequent write may suffer a long delay. Fortunately, such complications do not arise with TPC-C-like workloads either. Indeed, the random writes of TPC-C cause the free blocks to be evenly distributed throughout the disk under eager-writing; this is desirable because a free block is never very far from the current head position.

In short, transaction processing workloads like TPC-C can benefit a great deal from the performance and reliability advantages offered by eager-writing, while the very nature of the workload allows it to avoid the performance pitfalls of eager-writing.

2.2 Mirroring and Striping

A D_m-way mirror, in addition to ensuring a high degree of reliability, can improve small read performance in terms of both latency and throughput. It can improve latency because the system can schedule the disk head that is closest to a replica to satisfy a read request [2, 5]. It can improve throughput because any request can be satisfied by any disk, and an intelligent scheduler should be able to exploit the freedom in distributing the incoming requests to balance load.

Although cost per byte and capacity per drive remain the predominant concerns of the consumer market, due to the large cost and performance gaps between disk and memory, database vendors have long recognized the need for trading capacity to obtain higher performance while configuring storage systems. A D_m-way mirror is just one of the ways to improve performance by exploiting excess capacity. This approach, however, has an obvious limitation—as one increases the degree of replication, the cost of replica propagation becomes prohibitive. One possible way of addressing this high cost is to perform some of the propagations in the background during idle periods. Unfortunately, TPC-C-like workloads are characterized by a combination of high write ratio and scarce idle time, a combination that makes it difficult to realize the potential benefits of mirroring.

An alternative to mirroring is striping—by partitioning and distributing data across a D_s-way striped system, the system reduces the maximum seek distance by a factor of D_s as only a fraction of each disk is used. This is attractive compared to mirroring because there is no replica propagation cost. Unlike mirroring, unfortunately, striping cannot reduce rotational delay. As we raise D_s, only the seek time is lowered and that too at a diminishing rate. Furthermore, unlike mirroring, due to the partitioning of data, the choice of which disk to send a request to is limited, so it is more difficult to

perform load-balancing.

In practice, disk array designers have used a combination of mirroring and striping to form a *striped mirror* [3, 11, 26]. In a $D_m \times D_s$ striped mirror, data is partitioned into D_s sets, each of which is replicated D_m times. The configuration where $D_m = 2$ is commonly referred to as "RAID-10". The replica propagation cost remains an obstacle to achieving good performance on RAID-10; and one seldom chooses a replication factor D_m that is greater than two.

2.3 Eager-writing Disk Arrays

An EW-Array resembles a conventional striped mirror in how data is distributed and reads are performed. However, the two systems differ in how writes are satisfied: instead of performing a write to one of many fixed locations, a $D_m \times D_s$ EW-Array chooses a disk whose head is closest to a free block among D_m candidates to perform the foreground write. In cases where a higher degree of reliability is desired, the two disk heads that are closest to their free blocks are chosen to perform the foreground writes. The remaining $D_m - 1$ (or $D_m - 2$) writes are buffered in the *delayed write queues* of the remaining disks to be performed in the background, also in an eager-writing fashion.

In an EW-Array, reads enjoy good latency and throughput just as they do in a conventional striped mirror. Foreground write latency is improved greatly due to eager-writing. This latency can be even lower when there are more disk heads to choose from. Unlike a striped mirror, copy propagation is no longer the limiting problem because the writes are sufficiently efficient that they are easily masked even when idle time is scarce. As a result, an EW-Array can sustain higher I/O throughput. The low cost of replica propagation also makes it possible to raise the degree of replication D_m for even lower read latency or to increase the fraction of foreground writes for higher reliability.

3 Configuring an EW-Array

An EW-Array combines three techniques: eager-writing, mirroring, and striping. One commonality shared by all three of these techniques is that they all need extra disk capacity to be effective. We first examine individually how performance under each technique improves in response to increased capacity. We then analyze their combined effect. We use simple random workloads and simulation results in this section to study these techniques. More details about the simulation environment and results from more realistic workloads will be presented in later sections.

3.1 Impact of Extra Space on Eager-writing

In order for eager-writing to be effective, one needs to reserve enough extra space so that a free block can always be located near the current disk head position. Let disk utilization be U; we define *dilution* to be $D_d = 1/U$. For example, when $D_d = 2$, we use twice as much capacity as is necessary.

Figure 1(a) shows how the components of the average write cost respond to different dilution factors (D_d) under a simple random write workload running on a 10,000 RPM Seagate disk (ST39133LWV). (In this case, $D_m = D_s = 1$. The block size is 4 KB, and there is no queueing.) In this figure (and the rest of this paper), *overhead* is defined to include various processing times and transfer costs. Under eager writing, when the closest free block is located in the current track, only rotational latency is incurred. When the closest free block is located in a neighboring track, a track switch or a small seek is also needed, and this time is counted as seek time.

As we increase the amount of extra space, both the rotational delay and seek time decrease as the disk head travels a shorter distance to locate the nearest free block. This improvement reaches diminishing return as the overhead dominates.

3.2 Impact of Extra Space on Mirroring

Figure 1(b) shows how the components of the average read cost respond to different degrees of replication (D_m) in a mirrored system under a random read workload. (In this case, $D_d = D_s = 1$.) The read overhead is lower than the write overhead, because it takes longer for the disk head to settle when servicing a write request. Note that mirroring reduces both the seek and rotational delays of read requests. These components, however, remain significant if mirroring is the only technique employed.

3.3 Impact of Extra Space on Striping

Figure 1(c) shows how the components of the average read cost respond to different degree of striping (D_s) under a random read workload. (In this case, $D_d = D_m = 1$.) By restricting the disk head within a small seek distance, striping lowers seek delay. Unlike mirroring, it has no impact on rotational delay. As D_s increases, rotational delay dominates if striping is the only technique employed.

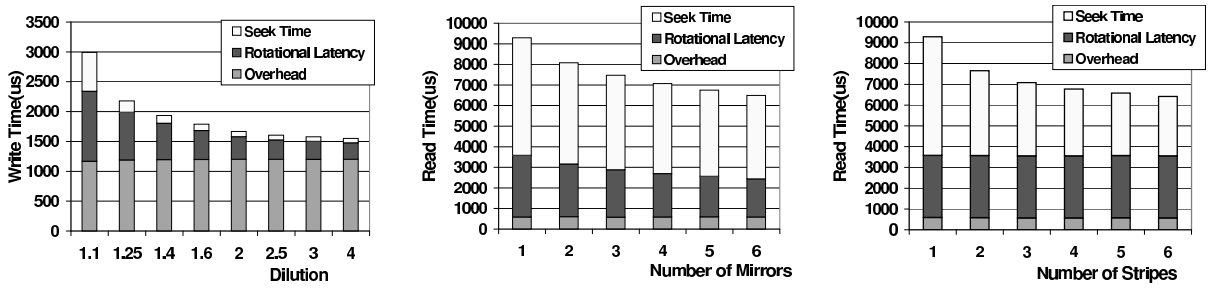

Figure 1: Components of average write response time as functions of (a) the degree of dilution D_d, (b) the degree of replication D_m, and (c) the degree of striping D_s.

3.4 Distributing Extra Space in an EW-Array

An EW-Array employs all three of the above techniques. A large D_d value allows for more efficient writes. A large D_s value more aggressively reduces the seek cost. A large D_m value more aggressively reduces the rotational cost of reads. Given a total budget of D disks and the constraint $D = D_d \times D_m \times D_s$, one must carefully balance these three dimensions to optimize the overall performance. The decision of how to configure these three dimensions is influenced by both the workload and disk characteristics. A workload that has a small read-to-write ratio and little idle time demands a large dilution factor D_d so that more resources are devoted to speeding up writes. Disks with large seek delays demand a large striping factor D_s, while disks with large rotational delay demand a large mirroring factor D_m.

In this section, we explore the impact of array configurations using a simple synthetic workload (that is part of the Intel "Iometer" benchmark [15]). More complex workloads are explored in Section 6. In each of the test runs, the length of the queue of the outstanding requests is kept at a constant. This is accomplished by adding a new request to the queue as soon as an old one is retired from it. Different queue lengths emulate different degree of idleness in the system. In all runs, the read/write ratio is 50/50.

Figure 2 compares the latency of alternative EW-Array configurations. In these experiments, the number of outstanding requests is one so there is no queueing. As a result, a relatively small dilution factor ($D_d = 1.25$) is generally sufficient for absorbing the writes while a relatively large $D_m \times D_s$ product improves read latency. A properly configured 4-disk EW-Array halves the latency achieved on a single-disk conventional system. Note that many of the configurations in Figure 2 have fractional values for D_s and D_d, yet $D_s \times D_d$ is always integral.

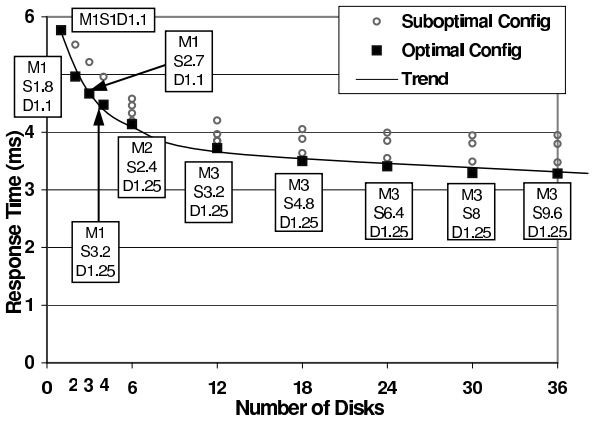

Figure 2: Comparison of response times of different EW-Array configurations. Each point symbol shows the performance of an alternative EW-Array configuration. A label "MaSbDc" denotes a $D_m \times D_s \times D_d = a \times b \times c$ configuration.

That means each replica stripes data across $D_s \times D_d$ disks. On each of those disks, only $1/D_s$ fraction of the tracks are actually used to store data, and utilization of those tracks is $1/D_d$.

Figure 3 shows how the throughput of optimally configured EW-Arrays scales with an increasing number of disks. We vary the number of outstanding requests per disk to emulate different load levels. For a fixed number of disks, as we raise the request arrival rate, a progressively larger dilution factor D_d is needed to absorb the disk writes that can no longer be masked by idle periods.

4 Scheduling on an EW-Array

When multiple outstanding requests are present in the I/O system, the order of servicing these requests has an important impact on the throughput of the system. Although disk scheduling is a well-studied problem, eager-writing presents a new challenge and a new opportunity. The challenge is that the physical locations of the write requests are un-

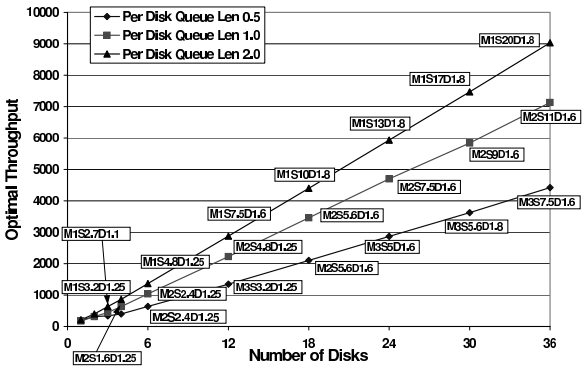

Figure 3: Throughput of optimal EW-Array configurations under different queueing conditions. Each point represents the performance of an EW-Array configuration.

known at the time the requests are queued. The opportunity is that the flexibility afforded by data location independence may enable an intelligent scheduler to achieve greater throughput. In this section, we first examine a number of eager-writing schedulers for a single disk; we then describe how we perform global scheduling across multiple disks in an EW-Array.

4.1 Naive Scheduling Algorithms

Given a mix of queued read and write requests, since write requests generally can be serviced quickly under eager-writing, one naive strategy is to simply schedule all the writes first. (To prevent starvation of read requests, one can augment this algorithm with simple heuristics such as imposing an upper-bound on the amount of time that a request can spend in the queue before it is forcibly scheduled.) We call this the *write-first* algorithm. One problem with this naive algorithm is that by greedily scheduling all the writes first, the scheduler may be missing opportunities of inserting some of these writes into naturally occurring latency gaps during the later read operations without adding much to the queueing time of the reads.

The opposite approach, an equally if not more naive algorithm, is to schedule all the reads first using an existing disk scheduling algorithm. We call this the *read-first* algorithm. Write requests that could have completed more quickly suffer long delays, and it is not hard to see why this algorithm is not optimal. We describe the read-first and write-first algorithms here not because of their practical utility, but because the problems encountered by these two extreme approaches may expose the pitfalls of eager-writing scheduling algorithms in general.

4.2 Eager-writing-based Shortest Access Time First Scheduling

The traditional shortest access time first (SATF) algorithm greedily schedules the request that is closest to the current disk head position [16, 23]. It takes both seek and rotational latency into consideration. We now extend this algorithm for eager-writing, and we call this extension the *SATF-EW* algorithm. The SATF-EW algorithm examines the queue and compares the location of the closest read request against the location of the closest free block. If the former is closer, we schedule the read request, else we schedule a write request into the closest free block. To avoid trapping the disk head in a small region and exhausting the free blocks, we always force the disk head to move in one seek direction until it can move no further and has to switch direction.

Unlike the naive algorithms described earlier, SATF-EW generally strikes a sound balance in scheduling read and write requests. When there are a large fraction of free blocks and there are many write requests, however, SATF-EW will tend to favor scheduling writes first; in the extreme case, it may degenerate to the write-first algorithm which, as we have explained earlier, may have its shortcomings.

4.3 Eager-writing-based Free Bandwidth Scheduling

"Free bandwidth" is different from bandwidth available during idle periods—the disk head may pass over locations that are of interest to some background operations even as it is "busy" serving foreground requests. Inserting some of these background requests into the foreground request stream should impose little penalty on foreground activities. Example applications that can benefit from free bandwidth are background activities such as data mining and storage reorganization [17].

Our next group of eager-writing scheduling algorithms are inspired by the approach of exploiting free bandwidth. Under this approach, we first schedule the reads using a known disk scheduling algorithm. Based on this schedule, we calculate a "deadline" by which the disk head must arrive at a target cylinder for each read operation so that the read operation can complete in time. Once the schedule and the deadlines of the reads are determined, we attempt to insert eager-writes into gaps among the reads if suitable free blocks can be located and the insertion of these eager-writes does not cause the disk head to miss the deadlines prescribed by the read schedule. In this case, the disk head also moves in one direction until it can move no further and has to reverse

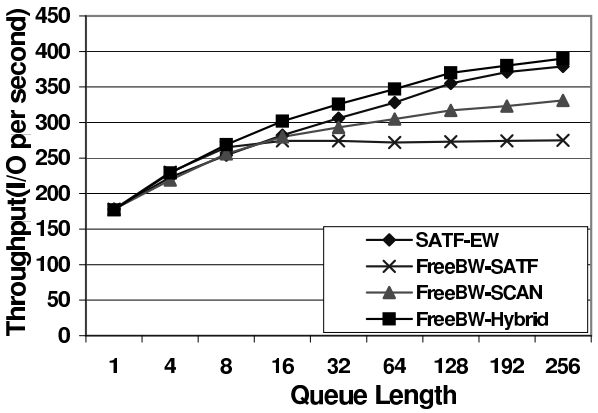

Figure 4: Throughput comparison of different eager-writing scheduling algorithms as we vary the number of queued requests.

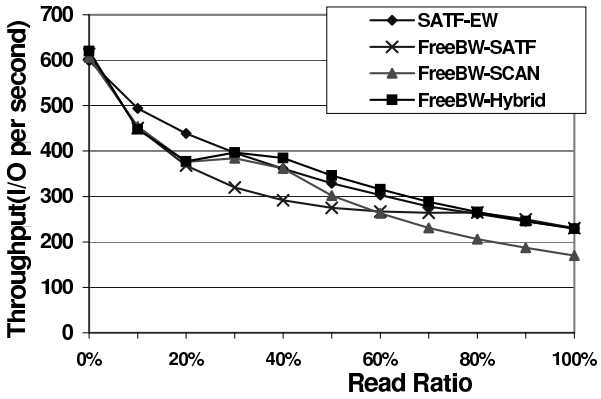

Figure 5: Throughput comparison of different eager-writing scheduling algorithms as we vary the fraction of queued operations that are reads.

its direction.

The above description is not sufficient for fully specifying the scheduling order—to complete the description, we must determine what scheduling algorithm to use to schedule the reads. We shall explore two possibilities: shortest access time first (SATF) as described above, and SCAN, which orders the read requests solely based on their seek distance. We call the resulting overall algorithms *FreeBW-SATF* and *FreeBW-SCAN* respectively.

These scheduling algorithms inspired by the exploitation of free bandwidth are a different way of balancing the scheduling of reads and eager-writes. When there are many read requests, however, these algorithms will tend to favor scheduling reads first; this happens because a large number of reads tend to reduce the gap among them and there is less room left for eager writes. In the extreme case, it may degenerate to the read-first algorithm which, as we have explained in Section 4.1, may have its shortcomings.

4.4 Comparison of Eager-writing Scheduling Algorithms

Figure 4 compares the throughput of different eager-writing algorithms as we vary the queue length. This simple simulated workload has a 50% write ratio and it runs on a disk with a dilution factor of 2.

SATF-EW works well for all queue lengths. In contrast, although it is known that SATF generally outperforms SCAN in a traditional storage system [16, 23], interestingly enough, FreeBW-SATF performs worse than FreeBW-SCAN, especially when the queue is large. This occurs because the aggressive scheduling of reads by SATF under high load leaves little "free bandwidth" for scheduling the eager-writes; consequently, FreeBW-SATF becomes similar to the read-first algorithm and unnecessarily penalizes writes. In contrast, by using SCAN to schedule the reads, FreeBW-SCAN causes the reads to be spaced further apart so more "free bandwidth" becomes available for eager-writes; consequently, the scheduling of reads and writes are better balanced and the overall performance of FreeBW-SCAN is better.

When there are a large number of reads but few writes, however, the performance of FreeBW-SCAN is not the best due to its failure to take rotational delay of reads into consideration. To address this shortcoming, we augment FreeBW-SCAN with a simple heuristics: when there is no write request in the queue, we replace SCAN with SATF. We call the modified algorithm *FreeBW-Hybrid*. Figure 4 shows that this hybrid algorithm performs the best for this workload—it even slightly outperforms SATF-EW due to its successful masking the eager-writes in the gaps of reads.

Figure 5 compares the throughput of these algorithms as we vary the ratio of reads. The queue length is 64 and the disk dilution factor is 2.

SATF-EW works well for all read ratios. When the read ratio is high, the disadvantage of FreeBW-SCAN is most apparent. In contrast, all other algorithms approach the performance of SATF when the read ratio approaches 100%. Interestingly, when the number of reads is small (but nonzero), the free bandwidth-based approaches also perform worse than SATF-EW. This is because the seek time among those small number of reads dominates and there is little rotational time left for scheduling the eager-writes. When there are a modest number of reads, due to both of its ability of successfully ex-

ploiting free bandwidth and its intelligent scheduling of reads, FreeBW-Hybrid is the best.

4.5 Scheduling an EW-Array

So far, we have described how to perform eager-writing-based scheduling on a single disk. The scheduling on an EW-Array is more complex because a read request can be serviced by any one of the disks that has a replica and a write request can return as soon as the first one or two copies are made persistent. We now incorporate the single-disk eager-writing algorithms described in the previous sections into the mirror scheduling algorithm employed by Yu et al [30].

A read request is sent to the idle disk head that is closest to a replica if at least one of the disks that contain the data is idle. If all the disks that contain the desired data are busy, a duplicate request is inserted into each of the relevant drive queues. Each disk employs an eager-writing scheduling algorithm as described in the previous sections. As soon as one of the duplicates completes, all remaining duplicate requests are canceled.

A write request can be sent to any one of the disks that are supposed to contain a replica. If any of these disks are idle, it is sent to the one that is closest to a free block. If all disks that should contain the desired data are busy, the request is inserted into the shortest queue. A second foreground write can be similarly scheduled for increased reliability. We set aside the remaining replica writes (if any) in a separate *delayed write queue* associated with each drive. Replica propagations from the delayed write queues are scheduled only when the foreground queues are empty.

5 Implementation

In this section, we describe a prototype EW-Array implementation that we use to experiment with the configuration and scheduling alternatives.

5.1 Architecture

The EW-Array prototype is implemented on the "MimdRAID" system developed by Yu et al [30]. Figure 6 shows how some of the MimdRAID modules have been replaced by EW-Array-specific components and how these components fit together. MimdRAID provides a framework and a number of useful features that enable one to conveniently experiment with novel disk array designs. We briefly highlight some of these useful features:

- MimdRAID exports a transparent logical disk interface on Windows 2000 so that the existing operating system and applications run unmodified on the underlying experimental disk array.
- Highly modularized components of MimdRAID, such as the "Array Configuration" and "Scheduling" components of Figure 6, can be replaced to allow for experimentation with new array designs.
- An accurate software-based disk head position prediction mechanism (in the "Calibration" layer) is crucial for realizing an EW-Array because the efficient scheduling of both eager-writes and reads depends on the precise knowledge of the rotational position of the disk head.
- At the lowest layer, the "SCSI Abstraction" module, which manages real Seagate disks, can be substituted with a disk simulator, so an EW-Array simulator and its implementation effectively share most of the code. The simulator can shorten simulation time of long traces by replacing physical I/O time with simulated time; it also allows us to experiment with a wide range of configurations without having to physically construct them.

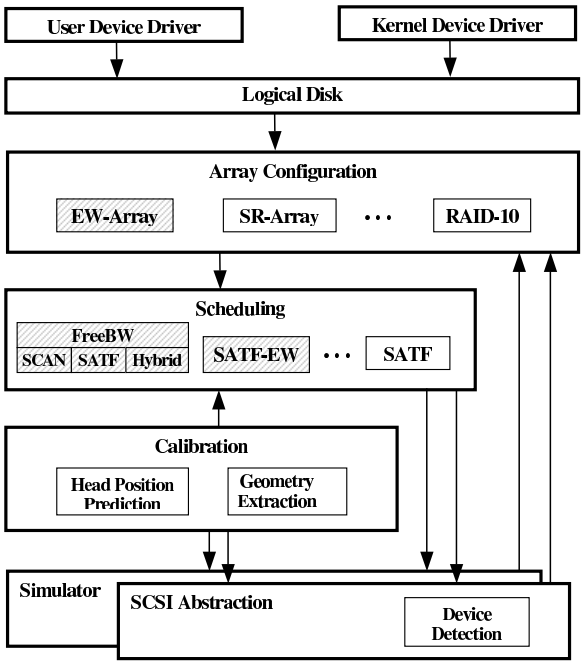

Figure 6: The MimdRAID architecture. Shaded parts are the modules added for an EW-Array.

Almost all the EW-Array-specific code in MimdRAID is concentrated in the Scheduling and Array Configuration layers. The Scheduling layer implements all the eager-writing scheduling algorithms described in Section 4. The Array Configuration layer is responsible for translating requests of logical I/O addresses to those of physical I/O ad-

dresses. This layer maintains a logical-to-physical address mapping and we describe this next.

5.2 Logical-to-physical Address Mapping

Under eager writing, the physical locations of a logically addressed block can frequently change. In this section, we detail how we query the logical-to-physical address mapping upon a read operation, how we maintain persistence of the mapping upon a write operation, and how we recover the mapping.

5.2.1 Querying the Mapping

A simple design is to store the entire logical-to-physical mapping in main memory. Reads are efficient and simple to implement: for each read operation, we simply use the logical address as an index to query a table to uncover the physical addresses. The price we pay is the cost of the map memory. A map entry in our system consumes four bytes per logical address per replica. The block size of our EW-Array implementation is 4 KB. With a $D_d \times D_m \times D_s$ EW-Array, the amount of map space is $D_m \cdot C/1000$, where C is the size of the logical disk which, in turn, is typically much smaller than the total amount of physical capacity in a TPC-C run.

We have chosen this simple design due to the nature of the transaction processing workload that we are targeting. First, the large number of spindles that are necessary for achieving good performance make the cost of the map memory insignificant. Second, the poor locality of the workload implies that the relatively small amount of memory consumed by the map would have delivered little improvement to read performance had the memory been used as a data cache instead. For a workload that exhibits more locality, we are currently researching the alternative approach of keeping only the most frequently accessed portion of the map in memory and "paging" the rest to disk.

5.2.2 Updating and Recovering the Mapping with Incremental Checkpointing

In designing a mechanism to keep the logical-to-physical address map persistent, we strive to accomplish two goals: one is low overhead imposed by the mechanism on "normal" I/O operations, and the other is fast recovery of the map.

Figure 7 shows the map-related data structures used in various storage levels. Updates to the logical-to-physical map are accumulated in a small amount of NVRAM. When the NVRAM is filled, its content is appended to a *map log region* on disk. (Both

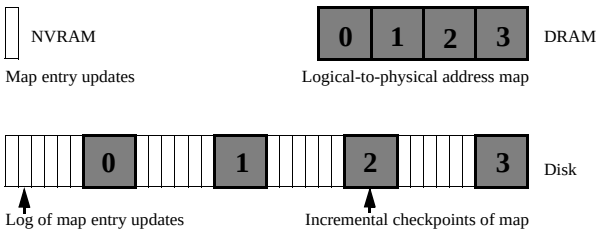

Figure 7: Logging of logical-to-physical map updates and incremental checkpointing of the map.

the NVRAM and the map log region on disk can be replicated for increased reliability.) We divide the logical-to-physical address map into M portions. ($M = 4$ in Figure 7.) Periodically, a portion of the map is checkpointed as it is appended to the log. After all M portions of the map are checkpointed, the map update log entries that are older than the Mth oldest checkpoint can be freed and this freed space can be used to log the new updates. The size of the map log region on disk is bound by the frequency and the size of the checkpoints. When we reach the end of the log region, the log can simply "wrap around" since we have the knowledge that the log entries at the beginning of the region must have been freed. The location of the youngest valid checkpoint is also stored in NVRAM.

During recovery, we first read the entire map log region on disk to reconstruct the in-memory logical-to-physical address map, starting with the youngest valid checkpoint. We then replay the log entries buffered in NVRAM. At the end of this process, the in-memory logical-to-physical address map is fully restored.

We note a number of desirable properties of the mechanism described above. The size and frequency of the checkpoint allows one to trade off the overhead during normal operation against the map recovery time. In particular, the checkpoints bound the size of the map log region, thus bounding the recovery time. Incremental checkpointing prevents undesirable prolonged delays associated with the checkpointing of the entire map. It also allows the space occupied by obsolete map update entries to be reclaimed without expensive garbage collection of the log.

It is interesting to compare the mechanism employed in managing the logical-to-physical address map of an EW-Array against those employed in managing data itself in a number of related systems such as an NVRAM-backed LFS [1, 21] and RAPID-Cache [13]. While we buffer and checkpoint *metadata*, these other systems buffer and reorganize *data*. The three orders of magnitude difference in

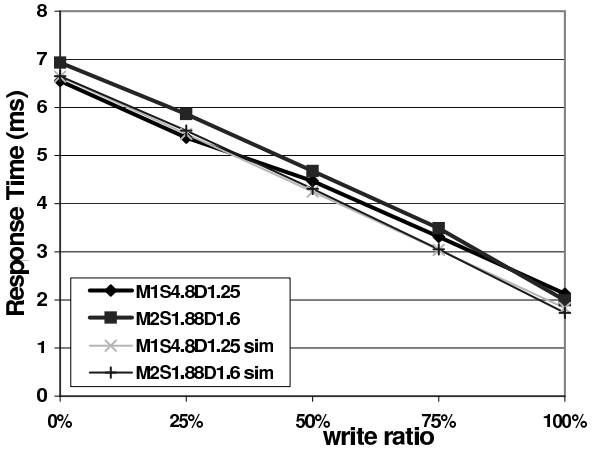

Figure 8: Comparison of response times on the EW-Array prototype and those predicted by the simulator as we vary the read/write ratio.

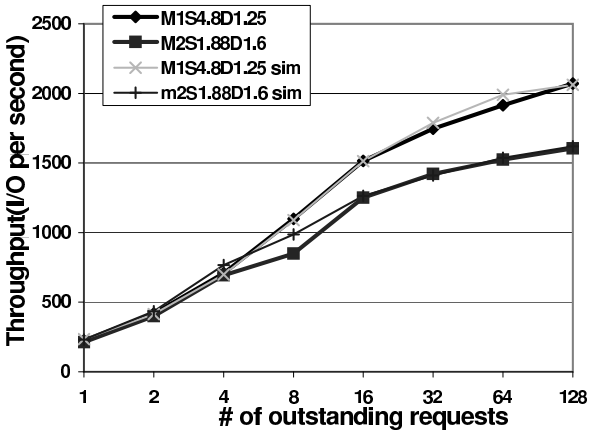

Figure 9: Comparison of throughput on the EW-Array prototype and that predicted by the simulator as we vary the per-disk queue length.

the number of bytes involved makes it easy to control the overhead of managing metadata in an EW-Array. Practically, in a $D_m \times D_s \times D_d = 2 \times 9 \times 2$ EW-Array prototype (whose disks are 9 GB each), with a 100KB NVRAM to buffer map entry updates, we have observed that the amount of overhead due to the maintenance of the map during normal operations of the TPC-C workload is below 1% and it takes 9.7 seconds to recover the map. The recovery performance can be further improved if we distribute the map log region across a number of disks so the map can be read in parallel.

5.3 Validating the Integrated Simulator

Operating system	Microsoft Windows 2000
CPU type	Intel Pentium III 733 MHz
Memory	128 MB
SCSI Interface	Adaptec 39160
SCSI bus speed	160 MB/s
Disk model	Seagate ST39133LWV 9.1 GB
RPM	10000
Average seek	5.2 m*s* read, 6.0 m*s* write

Table 1: Platform characteristics.

Due to the large number of configurations and the long traces that we must experiment with, the experimental results reported in Section 6 are based on those obtained on the simulator; therefore it is necessary to validate the EW-Array simulator using our EW-Array prototype. Table 1 lists some of the platform characteristics of the prototype.

We run a benchmark called "Iometer", a benchmark developed by the Intel Server Architecture Lab [15]. Iometer can generate workloads of various characteristics including read/write ratio, request size, and the maximum number of outstanding requests.

Figure 8 compares the response times measured on a number of six-disk EW-Array prototype configurations with those predicted by the simulator as we vary the read/write ratio. In these Iometer experiments, the number of outstanding requests is one, and the dilution factor of the EW-Array is two. The two EW-Array configurations ($D_m \times D_s \times D_d = 1 \times 3 \times 2$, and $D_m \times D_s \times D_d = 2 \times 1.5 \times 2$) have similar response times and they are closely matched by those predicted by the simulations. Since the eager-writes have much lower latency than reads, the response time decreases as the write ratio increases.

Figure 9 compares the throughput obtained on the same six-disk EW-Array configurations with that predicted by the simulator as we vary the queue length per disk. In these Iometer experiments, the write ratio is 50%. As the per-disk queue length increases, the $1 \times 4.8 \times 1.25$ EW-Array achieves greater throughput than the $2 \times 1.8 \times 1.6$ configuration because it becomes increasingly difficult for the latter configuration to mask the replica propagation even with a larger dilution factor. The throughput measured on the prototype matches closely the simulated result.

6 Experimental Results

In this section, we compare the performance of the EW-Array with that of a number of alternatives.

6.1 The TPC-C Trace

The eager-writing arrays are designed to target TPC-C-like transaction applications. We evaluate

the effectiveness of EW-Arrays with a trace supplied by HP Labs. It is a disk trace of an unaudited run of the Client/Server TPC-C benchmark running at approximately 1150 tpmC on a 100-warehouse database. It has 4.2 million I/O requests, 54% of which are reads. The I/O rate is about 700 I/Os per second in the steady state. Most of the requests are synchronous I/Os. The total data set is about 9 GB, distributed originally on 54 disks to achieve the desired throughput. The trace was collected on 5/03/94.

Two characteristics of the trace may be of concern due to its old age: the data rate and the size of the data set. With comparable number of disks and machines, the current technology can support a much higher data rate. To account for this development, in some of the following experiments, we raise the I/O rate by multiplying it with a "trace speedup" factor. For example, when the trace speedup is two, we halve the inter-arrival time of requests. The data set size factor is of less concern. In fact, only a small fraction of the space on the original traced disks was used to achieve the target I/O rate. Although a single modern disk can accommodate the entire traced data set today, it cannot support the data rate of the original trace. We shall vary the number of disks employed in a disk array onto which the traced data set is distributed. We study the effectiveness of various array configurations and the conclusions that we reach are independent of the size of the entire data set.

6.2 The Alternative Disk Array Configurations

In addition to the EW-Array configurations, we will experiment with the following alternatives:

- A *RAID-10* combines striping and mirroring: data is striped across a number of disks and each of the striped disks is also replicated once.
- A *Doubly Distorted Mirror* is a variant of a RAID-10. For each logical write request, two "write-anywhere" physical writes are performed to free locations near the disk heads. One of these two copies is later "moved" to a fixed location in the background [19].
- An *SR-Array* combines striping and rotational replication: data is striped across D_s disks, and each block is replicated D_r times within a track to reduce rotational delay, so a total of $D_r \times D_s$ disks are used [30].

For configurations that support replication on multiple disks, we shall experiment with two different reliability guarantee scenarios: in one scenario, a synchronous write request is allowed to return as soon as one physical write is committed to a disk platter; in a second scenario, a synchronous write request is not allowed to return until at least two physical writes are committed to two disks. When multiple requests are in the disk queue, the EW-Arrays employ the SATF-EW scheduler discussed in Section 4, and the other configurations employ variants of SATF.

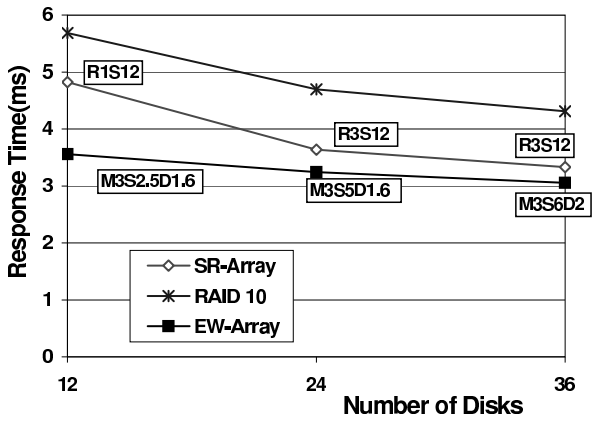

Figure 10: Comparison of TPC-C I/O response time on several disk array configuration alternatives as we vary the number of disks in the array. The SR-Arrays and EW-Arrays are labeled with their configuration parameters: an "RaSb" label denotes a $D_s \times D_r = b \times a$ SR-Array configuration, and an "MaSbDc" label denotes a $D_m \times D_s \times D_d = a \times b \times c$ EW-Array configuration.

6.3 Playing the Trace at Original Speed

We play the TPC-C trace at original speed in the experiments reflected in Figure 10. It compares I/O response times of the optimally configured EW-Arrays against those of the RAID-10s and the optimally configured SR-Arrays as we increase the number of disks. In these experiments, the second and subsequent replicas, if any, are propagated in the background.

An SR-Array generally outperforms RAID-10 because its combination of striping and rotational replication balances the reduction of seek and rotational delays better. An EW-Array outperforms both because of its substantially lower write latency, which also enables a higher degree of replication, which in turn lowers read latency. As we increase the number of disks, the performance benefit derived by an SR-Array from an increasing number of disks is larger than that of an EW-Array because the aggressive rotational delay reduction of the former benefits both reads and writes, while the write latency on an EW-Array is small to begin with and further improvements are marginal. In general, far fewer disks are necessary to achieve a specific latency goal on an

EW-Array.

6.4 Playing the Trace at Accelerated Rates

We play the TPC-C trace at accelerated rates in the experiments reflected in Figure 11. It compares I/O response times of various array configurations as we maintain a constant total of 36 disks for each configuration. (Note that the $D_s \times D_r = 36 \times 1$ SR-Array configuration is simply a conventional 36-way stripe.) In these experiments, we perform replica propagation in the background. (A maximum of 10,000 blocks can be buffered on an array for background propagation.)

The EW-Array has the best response time under all arrival rates and it generally delivers much higher sustainable throughput than conventional configurations. For example, the maximum sustainable throughput rates (expressed in terms of the speedup rate over the original trace speed) on a 36-way stripe, a RAID-10, a $2 \times 9 \times 2$ EW-Array, and a $1 \times 22 \times 1.6$ EW-Array are approximately 5×, 8×, 10×, and 14×, respectively.

As we raise the request arrival rate, idle time becomes more scarce and the replica propagation cost becomes felt by all configurations. We must successively reduce the degree of replication for both the SR-Array and the EW-Array. Thanks to the very low write latency of the EW-Array, however, the replica propagation burden on the EW-Array represents a much lighter load. The $2 \times 9 \times 2$ EW-Array remains a configuration of choice that offers sub-5 m*s* response times for an arrival rate that is as high as 9× the original, a rate that has rendered replica propagation a costly luxury that the other approaches cannot afford. The payoff of replication is better read response time than that on the other configurations.

In addition to eager-writing, two other factors contribute to the EW-Array's superior throughput. One is the greater flexibility afforded by its SATF-EW local disk scheduler. The other is the better load-balancing opportunities afforded by the array-wide scheduling heuristics as writes are scheduled to disks with shorter queues and reads are serviced by choosing among multiple candidate copies on different disks.

6.5 Effect of Double Synchronous Writes

In the previous experiments, all but the first replicas are propagated in the background. To raise the degree of reliability, one may desire to have two copies physically on disk before a write request is

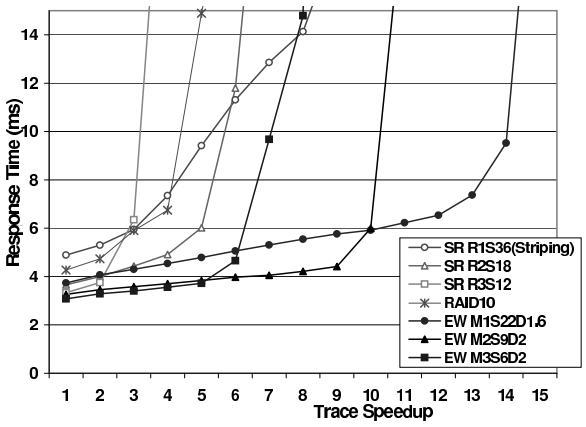

Figure 11: Comparison of TPC-C throughput on alternative disk array configurations as we vary the I/O rate. The total number of disks is a constant (36). The size of the delayed write buffer is 10,000 blocks.

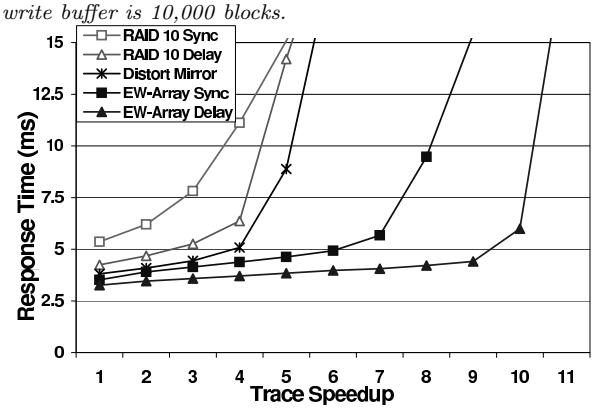

Figure 12: Effect on TPC-C throughput as we write to two disks synchronously. The total number of disks is a constant (36). Labels that include the word "Delay" denote experiments that propagate all but the first replicas in the background. Labels that include the word "Sync" denote experiments that write two replicas synchronously.

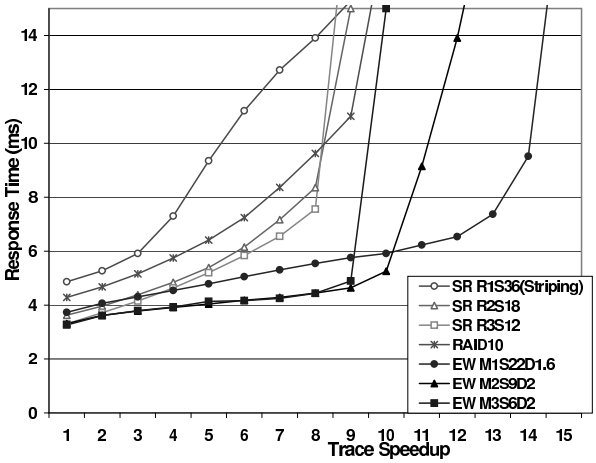

Figure 13: Effect on TPC-C throughput as we increase the size of the delayed write buffer to 100,000 blocks. The total number of disks is a constant (36).

allowed to return. We now study the effect of increasing foreground writes on three alternative configurations: a RAID-10, a Doubly Distorted Mirror (DDM), and a $2 \times 9 \times 2$ EW-Array. The DDM performs two synchronous eager-writes and moves one of the two copies to a fixed location in the background. We note that the system that we have called the DDM is in fact a highly optimized implementation that is based on the MimdRAID disk location prediction mechanism and the eager-writing scheduling algorithms, features not detailed in the words of "write anywhere" in the simple original simulation-based study [19]. (We do not consider SR-Arrays because the pure form of an SR-Array involves only intra-disk replication which does not increase the reliability of the system.)

As expected, extra foreground write slows down both the RAID-10 and the EW-Array. However, for a given request arrival rate that does not cause performance collapse, the response time degradation experienced by the RAID-10 is more pronounced than that seen on the EW-Array. This is because the cost of an extra update-in-place foreground write is relatively greater than that of an extra foreground eager-write. The performance of the DDM lies in between , because the two foreground writes enjoy some performance benefit of eager-writing but the extra update-in-place write becomes costly, especially when the request arrival rate is high. One of the purposes of this third update-in-place write is to restore data locality that might have been destroyed by the eager-writes. This is useful for workloads that exhibit both greater burstiness and locality. Unfortunately, the TPC-C workload is such that it does not benefit from this data reorganization.

6.6 Effect of the Delayed Write Buffer Size

We have seen that replica propagation imposes a significant cost on update-in-place-based disk arrays such as RAID-10 and SR-Array. One possible way of alleviating this burden to make these alternatives more attractive is to use a larger delayed write buffer. A larger delayed write buffer is useful in two ways. One is that it may allow larger batches of replica propagations to be scheduled and these larger batches can utilize the disk bandwidth more efficiently. The second source of efficiency is that a larger buffer can potentially more effectively smooth the burstiness so that replica propagation does not have to occur in the foreground due to lack of buffer space.

Figure 13 shows the results of repeating the throughput experiments shown in Figure 11 after

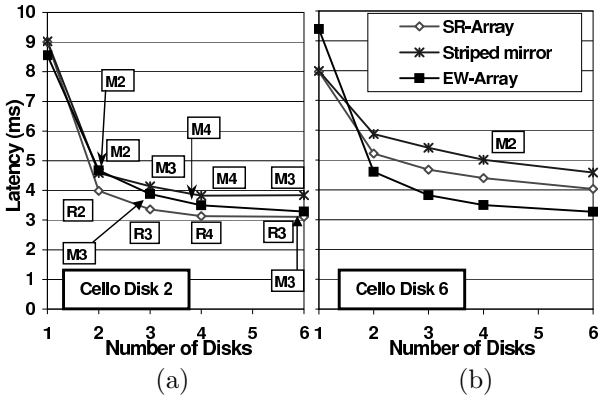

Figure 14: Comparison of the performance of different array alternatives under two file system workloads: (a) Cello disk 2, housing "/users", and (b) Cello disk 6, housing "/var/spool/news". Each data point represents the performance of the best configuration based on a given array alternative. The rectangular labels show the degree of mirroring (or replication) used in the configurations. The unlabeled configurations in the second figure have identical degrees of mirroring (or replication) as their counterparts in the first figure.

we have increased the delayed write buffer from 10,000 blocks to 100,000 blocks. As expected, the curves representing array configurations that require data replication have all shifted to the right, signaling higher maximum sustainable throughput rates. However, even with this aggressive delayed write buffering, the high cost of update-in-place is still apparent and the advantage of eager-writing is still significant.

6.7 Results of File System Workloads

Although the target workload of an EW-Array is TPCC-like transaction processing applications, it is natural to ask whether it works for other workloads. Figure 14 shows the performance results of two file system workloads that are selected from the HP "Cello" trace. Cello is a two month trace of a file server supporting simulations, compilations, editing, and reading mail and news. We use the traces of two disks during the week from 5/30/92 to 6/6/92. Disk 2 houses user home directories and disk 6 houses a news archive. To compensate for the relatively lower I/O rates of the trace, we speed up the trace playing rate by a factor of four.

The difference between Figure 14(a) and (b) is due to the different locality and the different read/write ratio of the two workloads. Cello disk 2 exhibits a higher degree of locality: the average seek distance on this disk is about half of that of disk 6. Cello disk 6, on the other hand, experiences a higher write ratio: 63.2% on disk 6 versus 45.2% on disk 2. Therefore, an SR-Array performs best for

disk 2 by preserving locality and balancing the reduction of seek and rotational delays; while an EW-Array excels for disk 6 by aggressively optimizing write latency.

7 Related Work

This paper combines four elements: (1) eager-writing, (2) data replication for performance improvement, (3) systematic configuring a disk array to trade capacity for performance, and (4) intelligent scheduling of queued disk requests. While each of these techniques can individually improve I/O performance, it is the integration and interaction of these techniques that allow one to economically achieve scalable performance gain on TPC-C-like transaction processing workloads. We briefly describe some related work in each of those areas.

The eager-writing technique dates back to the IBM IMS Write Ahead Data Set (WADS) system which writes write-ahead log entries in an eager-writing fashion on drums [7]. Hagmann employs eager-writing to improve logging performance on disks [10]. A similar technique is used in the Trail system [14]. These systems require the log entries to be rewritten to fixed locations. Mime [4], the extension of Loge [6], integrates eager-writing into the disk controller and it is not necessary to rewrite the data created by eager-writing. The Virtual Logging Disk and the Virtual Logging File Systems eliminate the reliance on NVRAM for maintaining the logical-to-physical address mapping and further explore the relationship between eager-writing and log-structured file systems [28]. All of these systems work on individual disks.

A more common approach to improving small write performance is to buffer data in NVRAM and periodically flush the full buffer to disk. The NVRAM data buffer provides two benefits: efficient scheduling of the buffered writes, and potential overwriting of data in the buffer before it reaches disk. For many transaction processing applications, poor locality tends to result in few overwrites in the buffer, and lack of idle time makes it difficult to mask the time consumed by buffer flushing. It is also difficult to build a large, reliable and inexpensive NVRAM data buffer. On the other hand, an inexpensive reliable small NVRAM buffer, as the one employed for the mapping information in the EW-Array, is quite feasible.

The systems that use the NVRAM data buffer differ in the way they flush the buffer to disk. The conventional approach is to flush data to an update-in-place disk. In the steady state, the throughput of such a system is limited by the average head movement distance between consecutive disk writes. An alternative to flushing to an update-in-place disk is to flush the buffered data in a log-structured manner [1, 21]. The disadvantage of such a system is its high cost of disk garbage collection, which is again exacerbated by the poor locality and the lack of idle time in transaction processing workloads [22, 24]. Some systems combine the use of a log-structured "caching disk" with an update-in-place data disk [12, 13], and they are not immune to the problems associated with each of these techniques, especially when faced with I/O access patterns such as those seen in TPC-C.

Several systems are designed to address the small write performance problem in disk arrays. The Doubly Distorted Mirror (DDM) [19] is closest in spirit to the EW-Array. The two studies have different emphasis, though. First, the emphasis of the EW-Array study is to explore how to balance the excess capacity devoted to eager-writing, mirroring, and striping, and how to perform disk request scheduling in the presence of eager-writes. Second, the EW-Array employs pure eager-writing without the background movement of data to fixed locations. While this is more suitable for TPC-C-like workloads, other applications may benefit from a data reorganizer. Third, the EW-Array study provides a real implementation.

While the DDM and the EW-Array are based on mirrored organizations, the techniques that may speed up small writes on individual disks may be applied to parity updates in a RAID-5. *Floating parity* employs eager-writing to speed up parity updates [18]. *Parity Logging* employs an NVRAM and a logging disk to accumulate a large amount of parity updates that can be used to recompute the parity using more efficient large I/Os [25]. The amount of performance improvement experienced by read requests in a RAID-5 is similar to that on a striped system, and as we have seen in the experimental results, a striped system may not be as effective as a mirrored system, particularly if the replica propagation cost of a mirrored system is reduced by eager writing.

Instead of forcing one to choose between a RAID-10 and a RAID-5, the AutoRAID combines both so that the former acts as a cache of the latter [29]. The RAID-5 lower level is log-structured and it employs a *hole-plugging* technique for efficiently garbage-collecting a nearly full disk: live data is "plugged" into free space of other segments. This is similar to eager-writing, except that eager-writing does not require garbage collection.

An SR-Array combines striping with rotational

data replication to reduce both seek and rotational delay [30]. A mirrored system may enjoy some similar benefits [2, 5]. Both approaches allow one to trade capacity for better performance. The difficulty in both cases is the replica propagation cost. The relative sizing of the two levels of storage in AutoRAID is a different way of trading capacity for performance. In fact, for locality-rich workloads, it is possible to employ an SR-Array or an EW-Array as an upper-level disk cache of an AutoRAID-like system.

Seltzer and Jacobson have independently examined disk scheduling algorithms that take rotational delay into consideration [16, 23]. Yu et al. have extended these algorithms to account for rotational replicas [30]. Polyzois et al. have proposed a delayed write scheduling technique for a mirrored system to maximize throughput [20]. Lumb et al. have exploited the use of "free bandwidth" for background I/O activities [17]. The EW-Array scheduling algorithms have incorporated elements of these previous approaches.

Finally, the goal of the MimdRAID project is to study how to configure a disk array system given certain cost/performance specifications. The "attribute-managed storage" project at HP [8] shares this goal.

8 Conclusion

Due to their poor locality, high update rates, lack of idle time, and high reliability requirements, transaction processing application such as those exemplified by TPC-C are among the most demanding I/O applications. In this paper, we have explored how to integrate eager-writing, mirroring, and striping in a eager-writing disk array design that effectively caters to the need of these applications. Mirroring and striping improves read performance, while eager-writing improves write performance and reduces the cost of data replication. The combination provides a high degree of reliability without imposing excessive performance penalty. To fully realize the potential of an EW-Array, we must address two issues. One is the careful balance of extra disk capacity that is devoted to each of the three dimensions: free space dilution for eager-writing, the degree of mirroring, and the degree of striping. The second is the intelligent scheduling of the queued requests so that the flexibility afforded by the high degree of location independence associated with eager-writing is fully exploited. Simulation and implementation results indicate that the prototype EW-Array can deliver latency and throughput results unmatched by conventional approaches for an important class of transaction processing applications.

Acknowledgement

We thank Ruoming Pang for proposing the incremental metadata checkpointing algorithm, Fengzhou Zheng for tirelessly running many of the experiments, Sumeet Sobti for editing an earlier draft, Walter Burkhard for careful shepherding of the paper, and the FAST PC members and reviewers for their excellent suggestions.

References

[1] Baker, M., Asami, S., Deprit, E., Ousterhout, J., and Seltzer, M. Non-Volatile Memory for Fast, Reliable File Systems. In *Proc. of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)* (Sept. 1992), pp. 10–22.

[2] Bitton, D., and Gray, J. Disk Shadowing. In *Proc. of the Fourteenth International Conference on Very Large Data Bases* (Los Angeles, CA, August 1988), Morgan Kaufmann, pp. 331–338.

[3] Borr, A. Transaction Monitoring in Encompass: Reliable Distributed Transaction Processing. In *Proc. of the Seventh International Conference on Very Large Data Bases* (Cannes, France, September 1981), IEEE Press, pp. 155–165.

[4] Chao, C., English, R., Jacobson, D., Stepanov, A., and Wilkes, J. Mime: a High Performance Parallel Storage Device with Strong Recovery Guarantees. Tech. Rep. HPL-CSP-92-9 rev 1, Hewlett-Packard Company, Palo Alto, CA, March 1992.

[5] Dishon, Y., and Lui, T. S. Disk Dual Copy Methods and Their Performance. In *Proc. of Eighteenth International Symposium on Fault-Tolerant Computing (FTCS-18)* (Tokyo, Japan, 1988), IEEE CS Press, pp. 314–318.

[6] English, R. M., and Stepanov, A. A. Loge: a Self-Organizing Disk Controller. In *Proc. of the 1992 Winter USENIX* (January 1992).

[7] Gawlick, D., Gray, J., Limura, W., and Obermarck, R. Method and Apparatus for Logging Journal Data Using a Log Write Ahead Data Set. U.S. Patent 4507751 issued to IBM, March 1985.

[8] Golding, R., Shriver, E., Sullivan, T., and Wilkes, J. Attribute-managed Storage. In *Workshop on Modeling and Specification of I/O* (San Antonio, TX, October 1995).

[9] Growchowski, E. Emerging Trends in Data Storage on Magnetic Hard Disk Drives. In *Datatech* (September 1998), ICG Publishing, pp. 11–16.

[10] Hagmann, R., and Garcia-Molina, H. Implementing Long Lived Transactions Using Log Record Forwarding. Tech. Rep. CSL-91-2, Xerox Corporation, Palo Alto, CA, February 1991.

[11] Hsiao, H.-I., and DeWitt, D. J. Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines. In *Proc. of the 1990 IEEE International Conference on Data Engineering* (February 1990), pp. 456–465.

[12] Hu, Y., and Yang, Q. DCD—Disk Caching Disk: A New Approach for Boosting I/O Performance. In *Proc. of the 23rd International Symposium on Computer Architecture* (1995), pp. 169–178.

[13] Hu, Y., Yang, Q., and Nightingale, T. RAPID-Cache—A Reliable and Inexpensive Write Cache for Disk I/O Systems. In *Proc. of the Fifth International Symposium on High Performance Computer Architecture (HPCA-5)* (Orlando, Florida, January 1999).

[14] Huang, L., and Chiueh, T. Trail: Write Optimized Disk Storage System. http://-www.ecsl.cs.sunysb.edu/trail.html.

[15] Intel Server Architecture Lab. Iometer: The I/O Performance Analysis Tool for Servers. http://developer.intel.com/design/servers-/devtools/iometer.

[16] Jacobson, D. M., and Wilkes, J. Disk Scheduling Algorithms Based on Rotational Position. Tech. Rep. HPL-CSP-91-7rev1, Hewlett-Packard Company, Palo Alto, CA, February 1991.

[17] Lumb, C., Schindler, J., Ganger, G. R., Riedel, E., and Nagle, D. F. Towards Higher Disk Head Utilization: Extracting "Free" Bandwidth from Busy Disk Drives. In *Proc. of the Fourth Symposium on Operating Systems Design and Implementation* (San Diego, CA, October 2000).

[18] Menon, J., Roche, J., and Kasson, J. Floating parity and data disk arrays. *Journal of Parallel and Distributed Computing 17*, 1 and 2 (January/February 1993), 129–139.

[19] Orji, C. U., and Solworth, J. A. Doubly Distorted Mirrors. In *Proc. of ACM SIGMOD Conference* (May 1993), pp. 307–316.

[20] Polyzois, C., Bhide, A., and Dias, D. Disk Mirroring with Alternating Deferred Updates. In *Proc. of the Nineteenth International Conference on Very Large Data Bases* (Dublin, Ireland, 1993), Morgan Kaufmann, pp. 604–617.

[21] Rosenblum, M., and Ousterhout, J. The Design and Implementation of a Log-Structured File System. In *Proc. of the 13th Symposium on Operating Systems Principles* (Oct. 1991), pp. 1–15.

[22] Seltzer, M., Bostic, K., McKusick, M., and Staelin, C. An Implementation of a Log-Structured File System for UNIX. In *Proc. of the 1993 Winter USENIX* (Jan. 1993), pp. 307–326.

[23] Seltzer, M., Chen, P., and Ousterhout, J. Disk Scheduling Revisited. In *Proc. of the 1990 Winter USENIX* (Washington, D.C., Jan. 1990), Usenix Association, pp. 313–323.

[24] Seltzer, M., Smith, K., Balakrishnan, H., Chang, J., McMains, S., and Padmanabhan, V. File System Logging Versus Clustering: A Performance Comparison. In *Proc. of the 1995 Winter USENIX* (Jan. 1995).

[25] Stodolsky, D., Holland, M., and Gibson, G. A. Parity Logging: Overcoming the Small Write Problem in Redundant Disk Arrays. In *Proc. of the 21st International Symposium on Computer Architecture* (1993), pp. 64–75.

[26] Teradata Corp. *DBC/1012 Database Computer System Manual Release 2.0*, November 1985.

[27] Transaction Processing Performance Council. *TPC Benchmark C Standard Specification*. Waterside Associates, Fremont, CA, August 1996.

[28] Wang, R. Y., Anderson, T. E., and Patterson, D. A. Virtual Log Based File Systems for a Programmable Disk. In *Proc. of the Third Symposium on Operating Systems Design and Implementation* (New Orleans, LA, February 1999), Operating Systems Review, Special Issue, pp. 29–43.

[29] Wilkes, J., Golding, R., Staelin, C., and Sullivan, T. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems 14*, 1 (February 1996).

[30] Yu, X., Gum, B., Chen, Y., Wang, R. Y., Li, K., Krishnamurthy, A., and Anderson, T. E. Trading Capacity for Performance in a Disk Array. In *Proc. of the Fourth Symposium on Operating Systems Design and Implementation* (San Diego, CA, October 2000).