

GreenStor: Application-Aided Energy-Efficient Storage

Nagapramod Mandagere, Jim Diehl, David Du

DTC Intelligent Storage Consortium

University of Minnesota, Twin Cities

npramod@cs.umn.edu, jdiehl@ece.umn.edu, du@cs.umn.edu

Abstract

The volume of online data content has shown an unprecedented growth in recent years. Fueling this growth are new federal regulations which warrant longer data retention and a general increase in the richness of data content. To cope with this growth, high performance computing and enterprise environments are making use of large disk-based solutions that consume power all the time, unlike tape-based solutions. As a consequence, the energy consumption of the storage solutions has grown significantly. In this work we propose a storage solution called GreenStor, which makes use of application hinting on top of Massive Arrays of Idle Disks (MAID) to improve energy efficiency. GreenStor is centered on MAID, but with more efficient data movement to aid in energy conservation. Specifically, we propose an Extent-based Metadata Manager that achieves better space efficiency without sacrificing cache utilization and an Opportunistic Scheduling scheme that helps provide better use of application hints in a MAID system. Results show that our proposed opportunistic scheme for application hint scheduling consumes up to 40% less energy compared to traditional non-MAID storage solutions, whereas use of standard schemes for scheduling application hints on typical MAID systems is only able to achieve a smaller energy savings of about 25% versus non-MAID storage.

1. Introduction

Rapid digitization of content has led to extreme demands on storage systems. The nature of data access such as simulation data dumps, checkpointing, real-time data access queries, data warehousing queries, etc., warrant an online data management solution. Most online data management solutions make use of hierarchical storage management techniques to accommodate the large volume of digital data. In such solutions, a major portion of the data set is usually hosted by tape-based archival solutions, which offer cheaper storage at the cost of higher access latencies.

This loss in performance due to tape-based archive solutions limits the performance of the higher-level applications that make these different types of data accesses. This is particularly true since many queries may require access to older, archived data.

An attractive option for large distributed sites or data centers is to exploit large disk arrays to keep more data in low-latency storage. The decreasing cost and increasing capacity of commodity disks is rapidly changing the economics of online storage and making the use of these large disk arrays more practical. Large disk arrays also enable system scaling, an important property as the growth of online content is predicted to be enormous, both in at-rest storage (terabytes or more) and in delivered data (gigabytes or more per day). The enhanced performance offered by disk-based solutions comes at a price, however. Keeping huge arrays of spinning disks has a hidden cost, i.e., energy. Industry surveys suggest that the cost of powering the nation's data centers is growing at the rate of 25% every year [1]. Among various components of a data center, storage is one of the biggest energy consumers, consuming almost 27% of the total. To make matters worse, increasing performance demands have led to disks with higher power requirements; moreover, storage demands are continuously growing by 60% annually according to an industry report [2]. Given the well-known growth in Total Cost of Ownership, a solution which can mitigate the high cost of power, yet keep data online, is needed.

Massive Array of Idle Disks (MAID) is a design philosophy adopted by [3]. The central idea behind MAID is that all disks in a MAID storage array are not spinning all the time. Within a MAID subsystem, disks remain dormant (i.e., powered off) until the data they hold is requested. When a request arrives for data on a disk that is off, the controller turns on the disk, which takes around 10 seconds, and services the request. Additionally, a set of disks is designated as cache disks, which are always spinning (i.e., never turned off). This disk-based caching is necessary because the regular memory cache is usually not large enough to hold all of the frequently accessed data. MAID

controller tries to make sure that frequently accessed data is moved to the always-on cache disks. For this reason, the response time of the system is very tightly tied to the size of the cache disk set. By increasing the cache hit ratio, the controller tries to minimize the response time and also conserve energy. In addition to lowering power consumption, reducing the number of disks that are concurrently active significantly lowers overall storage subsystem costs by simplifying controller complexity. The savings increase as the storage environments get larger. A commercial product based on this idea, Copan MAID, has seen a great deal of success in the realm of archival systems. The main limitation is that, MAID concept works on the assumption that less than 5% of the stored data actually gets accessed on any given day and hence it tries to keep the most frequently accessed data in the cache disk set. However, this will not ensure good response time for non-cached data (10s of seconds). Data that is not cached could include data being accessed for the first time or data that cannot be cached due to its sheer volume or the access pattern, which is usually the case in most nearline systems.

Various studies of data access patterns in data centers suggest that on any given day the total amount of data accessed is less than 5% of the total stored [4]. Most energy conservation techniques make use of various optimizations to conserve energy, but this usually comes with a huge performance penalty. Access patterns for certain High Performance Computing (HPC) and enterprise applications have a lot of predictability, and this predictability could be more intelligently used to conserve power. The predictability of data access, or nature of future accesses, could be either learned by the system or be provided to the system by the applications that access the data hosted on the system. Learning techniques for this purpose have been met with limited success. This difficulty can be attributed to the dynamic nature and different purposes of various applications running on the same system. On the other hand, the idea of applications themselves supplying the hints about their future accesses does not suffer from these limitations. [5] and [6] have explored the idea of using application hints for the purpose of prefetching data ahead of time, thereby reducing the file system I/O latencies.

Our research is centered on the idea of using application hinting on top of MAID systems to build an energy efficient, near-online storage solution. The focus of our research is to identify and address the challenges in building a MAID-based storage solution that can exploit application hints efficiently. The primary challenges that we address in this paper are as follows.

Challenge 1: Storage Subsystem Architecture to Facilitate Energy Efficiency

Enterprise and high-end computing environments rely heavily on virtualization of storage resources. Virtualiza-

tion provides flexibility in resource allocation by decoupling the physical location of the resource from the logical view of the resource presented to the user/server. In other words, many physical disks can be viewed logically as a single large virtual disk. Though it drastically improves the usability of the system, the technique with which virtualization is implemented dictates the performance of the overall system. One of the main challenges in designing a virtualized system is determining the physical placement of the virtual cache/prefetch space. Since the system is virtualized, the cache could be either centrally located or distributed among the disk arrays. Though this choice of centralized or distributed location is not available for the large volume of regular data, both regular and cache data can still be laid out in different ways; i.e., on a collection of one or more single disks, striped across multiple disks within an array, or striped across multiple disk arrays. The choice of cache location and the cache and regular data layouts can have a drastic impact on the performance and energy efficiency of the overall system; hence, identifying the optimal architecture is very critical.

Challenge 2: Cache Metadata Management

In high performance computing and enterprise environments, the volume within data sets is huge; hence, any system designed based on caching or prefetching needs to be able to accommodate an entire working set in cache that is on the order of hundreds of gigabytes to a few terabytes. Since MAID arrays facilitate using disks for caching/prefetching, cache space is not the primary concern. The main problem is the amount of metadata that needs to be maintained in memory on the disk array or metadata server for mapping Logical Block Addresses (LBAs) to Cache Block Addresses. In the worst case, if we were to maintain a one-to-one mapping of LBAs and Cache Block Addresses, the metadata structure will be huge. For instance, consider a storage system containing 500TB of data (part of one or more virtualized LUNs/Address Spaces) with a block size of 32KB. If we decide to maintain a cache of 10%, or 50TB, using the one-to-one mapping would still require close to 15.6 billion mapping entries; that is, one entry for each logical block of the entire 500 TB data set.

A simple lookup table with these entries for the entire virtual address space would be the best option in terms of lookup times, but the size occupied by this structure would be close to 62GB, assuming each address entry takes 4 bytes. As described in Section 4, use of other, slightly more sophisticated, techniques would still require a large metadata structure. If one resorts to setting the granularity of data flow in and out of the cache to a fixed-size set of contiguous logical blocks instead of just one, the size of the mapping structure is reduced, but it results in poor utilization of the cache/prefetch space. Hence, we need a more

adaptive solution which exploits the nature of data access.

Challenge 3: Prefetch/Cache Space Management

Prefetch/cache space is a limited resource. Though MAID uses a large disk-based cache, the use of this cache space is manifold and hence leads to contention. This cache space is used for holding prefetched data blocks from the dormant disks and staged write data blocks heading to the dormant disks from higher-level applications. Since idle periods in servers are used for hint generation, the deadlines of hints could be spread out in time. When a storage subsystem receives I/O hints from multiple servers connected to it, the storage system has to make an informed decision to schedule these hinted requests based on their corresponding deadlines and the state of the cache. Making this informed scheduling choice is a complex task as the state of the system depends on various parameters such as incoming read I/O rate, incoming hint rate, incoming write I/O rate, outgoing write destage rate, the amount of used cache space, etc. Traditional schemes for prefetch scheduling [5] use cost-benefit analysis to try to maximize the utility of each buffer/cache slot while at the same time ensure fairness of scheduling (i.e., earlier deadlines first). However, in the proposed framework, the objective of the prefetch scheduler is to maximize the time a disk receives to service a hinted request. In doing so, each disk is given a greater chance to work in batch mode by executing requests in a collective manner, which leads to energy conservation.

In this work, we first propose the design of a storage subsystem called GreenStor, which is geared towards exploiting application hints to achieve better energy efficiency. Next, we propose a scheme which makes use of the semi-sequential nature of accesses without fixing the granularity of multi-block groups moved to/from cache. Specifically, we present an extent-based scheme for managing a large cache/prefetch space that keeps the amount of metadata needed at minimal levels without sacrificing the utilization of the cache. Finally, we propose a scheduling scheme that achieves the goals of energy conservation by facilitating opportunistic batch processing in an efficient manner and at the same time preserves fairness without adversely affecting the completion times of prefetch requests or the utilization of cache. The paper is organized as follows: Section 3 describes our proposed design of the GreenStor storage subsystem, Section 4 describes the extent-based metadata management scheme, Section 5 describes the deadline-based prefetch scheduling scheme, Section 6 discusses the simulation evaluation, and Section 7 provides the conclusions, and future work.

2. Related Work

[7] proposes a new energy conservation technique for disk array-based network servers called Popular Data Concentration (PDC). According to this scheme, frequently accessed data is migrated to a subset of the disks. The main

assumption here is that data exhibits heavily clustered popularities. PDC tries to lay data out across the disk array so that the first disk stores the most popular disk data, the second disk stores the next most popular disk data, and so on. The least popular disk data, and the data that are never accessed, will then be stored on the last few disks. [8] proposes a new solution called Hibernator, which is a disk array energy management system that provides improved energy savings while also meeting certain performance goals. The main idea here is the dynamic switching of disk speeds based on observed performance. This approach makes use of multi-speed disk drives which can run at different speeds but have to be shut down to make a transition between different speeds. Such disks have been demonstrated by Sony. However, such multi-speed disks have been proven to be costly to manufacture and hence have seen limited interest for large-scale production. A key point of interest here is the mapping of data blocks. Since data blocks are always moved around to different locations in the disk array, this mapping mechanism becomes very important. The scalability of the above mentioned approaches would heavily depend on this mapping mechanism, but the above works do not mention much about the mapping or its scalability.

[9] proposes a new type of hard disk drive which can operate at multiple speeds. Using Dynamic Rotations Per Minute (DRPM) speed control for power management in server disk arrays can provide large savings in power consumption with very little degradation in delivered performance. This technique dynamically modulates the hard disk rotation speed so that the disk can service requests at different RPMs. For such a scheme to work, a multi-speed disk must first be designed, for which there are several manufacturing/mechanical challenges. The RPM of the disk generally dictates the design of all the disk components. Hence, designing a disk which can operate at multiple speeds is very complex and almost infeasible; therefore, it has not been heavily pursued by disk vendors.

Application hinting has been studied in depth by several researchers. [5], [10], and [11] deal with the use of application hinting to minimize application response times by actively prefetching hinted data from disks into main memory ahead of time. Specifically, they explore tradeoffs of various scheduling schemes built using cost-benefit analysis. [12] explores the idea of cooperative processes to achieve energy conservation. Specifically, the authors propose a system where applications/processes provide hints (about IO operations) to the operating systems to aid in energy conservation of all types of IO devices. The main difference in our work, is that we focus on consolidated storage systems that are shared across multiple servers.

To the best of our knowledge, the idea of application hinting has not been explored before in the context of storage system energy conservation, specifically in the con-

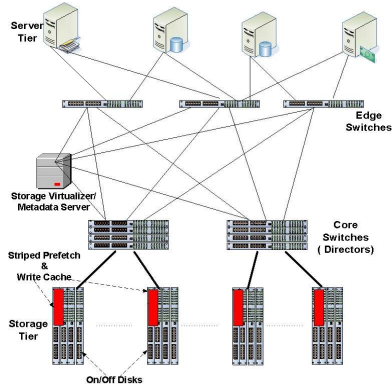


Figure 1. GreenStor Architecture

text of large shared disk farms. Our work primarily deals with taking an existing design (i.e., MAID) and optimizing the design for large-scale nearline storage using application hinting. We do not deal with multi-speed or DRPM disks. Instead, we assume standard disks in a MAID configuration with the ability to intelligently transition between Standby and ON states.

3. GreenStor Storage System Architecture

Enterprise and high performance computing environments typically make use of a set of disk arrays to satisfy their data storage demands. The storage subsystem in such scenarios is made up of several disk arrays that are connected to clusters of servers by means of a high speed Storage Area Network (SAN). These disk arrays can be connected to the servers in different configurations based on the requirements of the environment. In recent years, virtualization of the storage subsystem has gained importance as it facilitates better management of resources by allowing dynamic addition and removal of storage. Figure 1 shows our proposed GreenStor virtualized storage subsystem.

3.1. Virtualization Mechanism

Storage virtualization separates logical address space from the physical location of data. In essence, it adds a level of indirection between the file systems on servers or initiators and the storage subsystem. This type of indirection allows the storage subsystem to place, or lay out, the data blocks according to its own knowledge of the environment, and it also provides the flexibility to move data blocks around without having to inform the file system on the servers or initiators. This flexibility is very important for our design considering that locality of data has a strong impact on power conservation and the performance observed by end users or initiators. All requests made by the servers to the storage subsystem are directed to the Storage Virtualization device (SV), since this device presents virtual logical volumes, or VLUNS, to the servers. Typical storage virtualization devices can be categorized into two

groups; namely, in-band storage virtualization devices and out-of-band storage virtualization devices. With in-band storage virtualization devices, the SV is in the data path. Specifically, all read and write requests are made to the SV, and it services or redirects the requests accordingly. For read requests, the SV fetches the requested data and returns it to the requesting server. For write requests, the SV absorbs the writes and acknowledges back to the servers. The main issue with this approach is that the SV itself could become a bottleneck as it is in the data path. With out-of-band storage virtualization devices, the SV acts like a metadata server; that is, it is only in the control path and not in the data path. When a read request is made to the SV by a server, the SV converts the virtual addresses into physical addresses and returns this information to the server, and the server then uses this information to request the block from the original disk array. In the case of a write request, the same process is followed and the SV remains only in the control path. The main limitation of this approach is that for every read or write request, two exchanges take place, one between the server and the SV (metadata exchange), and the other between the server and the storage array (data exchange). If network latencies are high, this could severely affect the performance.

The choice of which form of storage virtualization to use is solely driven by performance considerations. This design choice does not have much impact on the energy efficiency of the storage subsystem. We make use of out-of-band virtualization as it is a more scalable solution, and the chances of the virtualization device becoming the bottleneck are reduced as only control information flows through it.

3.2. Application Hint Specification

Automatic hint generation is one of the key inputs to our scheme. Hint generation could be performed in several ways. [11] shows that several HPC applications can be modified to generate hints online, specifically, by modifying the binaries of these programs. [10] shows that certain classes of HPC applications can disclose their I/O requirements ahead of time. At the beginning of their execution, these applications could be made to disclose all future I/O accesses along with certain QoS requirements. Some researchers even suggest that application programmers, given the incentive of energy efficiency, could, by themselves, rewrite some of their application code to generate, either online or before execution, hints in advance. Both [11] and [10] show that compilers could be made to embed application hints by speculative execution of code. Our focus is on the efficient use of these hints and not the mechanism of hint generation; hence, we work independent of the mechanism of hint generation. However, we make one key assumption here: Instead of the applications just passing the hints about future access in a sorted manner, we assume that they can be programmed to also pass

an approximate time of future access. This assumption is reasonable considering that the applications know their execution rates and can predict their own progress rate better than the storage subsystem, unlike [13] where the disk subsystem infers the application execution rate by monitoring the incoming I/O rate. This assumption is needed in our design because we work from the level of remote consolidated storage that is shared by multiple servers and applications; thus, estimation of the application execution rate becomes very difficult given the fact that most I/O interfaces hide the initiating application information from the target storage devices.

3.3. Data/Cache Layout

In a virtualized storage subsystem, the virtual address space can be physically allocated in different ways (usually referred to as LUN Binding), namely:

1. Concatenating the address space of the individual Logical Units (LUNs) that constitute the virtual LUN to form a single virtual address space,
2. Striping the virtual address space across the LUNs that constitute the virtual LUN,
3. Using a mathematical mapping function for mapping between the virtual address space and physical address space, or
4. Using random allocation and storing the mapping information for each block in a lookup table.

Options 1&2 are the most popular approaches. Option 1 is the most flexible approach since it easily facilitates addition and deletion of constituent physical LUNs or growing and shrinking of virtual LUNs. Option 2 is very good in terms of avoiding hot spots and provides more parallel bandwidth because the disk accesses are evenly distributed across all constituent LUNs. However, addition or deletion of striped constituent LUNs requires reorganization of a large number of data blocks and hence limits its usefulness. Option 3 is very similar to Option 2 in that Option 2 uses a very simple mapping function (i.e., modulo). Option 3 also suffers from the same drawback of having to reorganize a lot of physical data blocks on expansion or contraction of the virtual LUN. Considering the size of data sets that we consider here, Option 4 is not a feasible option as the size of the lookup table needed to support such a scheme would be prohibitively large.

For our design we have decided to use Option 1, the concatenated address space model, due to the features it offers. It also aids in efficient destaging operation as shown in subsequent sections. The bandwidth is not a major cause of concern here; though we are concatenating address spaces, the constituent LUNs themselves may be internally striped

to provide sufficient parallel bandwidth. In our design, each disk array that is a part of this storage subsystem presents one or more LUNs to the SV. The SV concatenates these LUNs from multiple disk arrays, creates a single large concatenated address space, and presents this Virtual LUN to the servers. A SV may host multiple Virtual LUNs, each of different size.

The Prefetch/Write Cache LUN, highlighted in Figure 1, primarily controls the effective read/write bandwidth of the system. Hence, in our scheme, we have decided to use a combination of concatenation and striping across multiple disk arrays to exploit maximum parallel bandwidth. Each disk array that is part of this storage subsystem has a certain set of disks designated as Prefetch/Write Cache disks, which are always on. Each of these disk arrays creates an internally striped address space across its cache disks and presents this LUN to the SV. The SV concatenates these cache LUNs from different disk arrays to create a single large Virtual Prefetch/Write Cache LUN. The point to note here is that we force the striping of address space for constituent cache LUNs, but LUNs that constitute data LUNs may or may not be striped. It is important that the cache LUNs are striped to improve their performance because they control the effective bandwidth of read and write operations.

4. Extent-based Metadata Management

As in any virtualized storage subsystem that uses caching or dynamic movement of logical blocks, the mapping of a given logical block address to its actual location in the cache is a very key operation. The dynamic movement of logical blocks within the system necessitates the maintenance of additional metadata for this remapping operation. The amount of metadata is primarily dictated by the granularity of movement, which in its simplest case is a single logical block. File system block sizes range from 512 bytes to 256KB. Current systems commonly use a block size of about 16KB. However, due to coalescing of writes and read-ahead prefetching implemented by most file systems, I/O requests of about 32KB are very common.

Before going into the details of our mapping method, let us first describe some of the limitations of current techniques. As discussed in the introduction, a lookup table with a one-to-one mapping for the entire virtual address space is prohibitively expensive in terms of its size. A more viable option, in terms of space, is an inverted mapping table, which has an entry for each physical address in the cache. Even for this scheme, a cache of 50TB with a block size of 32KB would need close to 1.56 billion mapping entries to be stored. Considering the overhead of inverted tables, the space occupied by such a structure would be close to 12.5GB. Also, mapping operations on an inverted table are known to be relatively expensive in terms of computation time. A more commonly used approach for this map-

Logical Block Address(4B)
Cache Block Address(4B)
Pointer to next record(4B)
Parameters for Cache Replacement Algorithms

Figure 2. Hash Mapping Record

ping of logical blocks to their location in cache is to use a hash table. In this scheme, each mapping record would be of the format shown in Figure 2.

Collisions in a hash table could be solved using chaining, which adds the additional space overhead, resulting in mapping records of about 12 bytes each. The size of the overall metadata lookup structure would be about $1.56\text{billion} * 12\text{bytes}$, or close to 18.75GB of metadata for a cache size of 50TB. Note here that a record is stored for each block that is currently in cache, just like in the case of inverted tables. Though this amount is not that significant compared to the size of the overall dataset, this metadata structure needs to be kept in main memory that is generally reserved for holding cache/prefetch data blocks, and managing/mapping this record information becomes very cumbersome as the system scales to larger sizes. Also, this new metadata is to be maintained in addition to the metadata already maintained by the file system (using inodes). This new metadata is therefore an added overhead which is used for the sole purpose of caching or redirection. From an initiator or server’s point of view, the metadata maintained as part of inodes is inevitable, but this new metadata maintained in the storage subsystem is not easily justifiable.

One solution to this problem of the cache mapping structure being too big is to use a different granularity for movement of blocks from/to cache/prefetch space. [8] uses this approach with a relocation block size on the order of megabytes. Instead of moving one logical block at a time, in this approach, a set of contiguous logical blocks is classified as a relocation group and all data movement is at the level of these larger groups. This certainly reduces the size of the overall metadata structure, but the utilization of cache/prefetch space drops drastically. The problem here is that normal access patterns usually dictate the choice of the file system logical block size, and this is typically set at the creation of the file system. The values for this file system block size range from 512 bytes to 256KB. Hence, when data is moved into cache at higher granularity, there is high likelihood that only partial hits will be seen, leading to wasted cache space. For instance, if we were to use 1MB as our relocation group size, and if the file system uses a block size of 64KB, in the worst case (i.e., every 16^{th} block needs to be prefetched) we could end up with a cache utilization of $1/16$. On the other hand, if the data access pattern is semi-sequential or concentrated around certain regions of the address space, this idea of moving larger chunks of data at a time works really well.

The examples used in the above discussion of the limi-

tations of current techniques for metadata management assume that the average file system block size is 32KB or 64KB. However, this assumption might not always be the case. In systems with smaller block sizes, this problem of metadata management becomes an even bigger challenge as the volume of data grows. The following subsections describe how our extent-based mapping scheme is designed to overcome the limitations of current techniques.

4.1. Monitoring Access Patterns

At the heart of our design is the idea of using extents, or contiguous chunks of sequential blocks, to reduce the total metadata overhead. This idea has been used in file systems with a great degree of success. In our design the purpose and use is quite different, however. The main difference is that the selection of extent size at allocation time is not arbitrary, as is the case in file systems. We propose to use a heuristic called *Contiguity Quotient* to guide the selection of extent size. Contiguity Quotient (CQ) tries to quantify the contiguity, or sequentiality, of a cache group. Note that a cache group is nothing but a set of contiguous logical blocks of a predefined size. The number of cache groups is equal to the number of blocks in the entire virtual address space divided by the number of blocks in a cache group. We try to estimate the CQ heuristic for each cache group by observing the data access patterns. Specifically, we maintain one Access Bit per logical block in the cache group, amounting to a 2-byte record per cache group, assuming that each cache group contains 16 logical blocks. In addition, each record has a Mod Bit to indicate if any modification occurred or not (1 or 0, respectively). Algorithm 1 shows the process for updating the records in the monitoring structure. Monitor Reset Events could be trig-

Algorithm 1 Access Monitoring Algorithm

```

Initialize monitoring record: Set Mod Bit of all records to 0
while (1) do
  if (Monitor Reset Event) then
    Set Mod Bit of all cache groups to 0
  end if
  if (Data Access Event) then
    if (Mod Bit of cache group is 1) then
      Set corresponding Access Bit to 1
    end if
    if (Mod Bit of cache group is 0) then
      Set corresponding Access Bit to 1
      Reset Access Bits of all other blocks of this cache group to 0
      Set cache group Mod Bit to 1
    end if
  end if
end while

```

gered with a predefined periodicity. This algorithm tries to capture access patterns for a window of time equal to the period of the monitoring events. A count of the number of Access Bits set in each cache group’s record gives the value of the Contiguity Quotient for that group. A higher value of CQ indicates that if a block is accessed in that cache group,

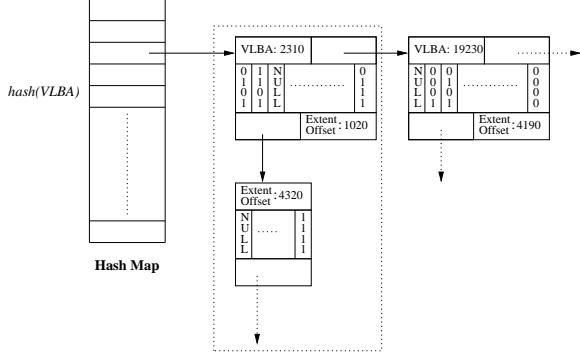


Figure 3. Global Mapping Structure

there is a very high probability that more blocks in the same cache group will be accessed.

4.2. Extent Allocation

If a block request is selected for scheduling, and if this request does not belong to a cache group which already has an allocated extent, the scheduler makes use of the Contiguity Quotient of its corresponding cache group to allocate an appropriate-sized extent for the purpose. Extents are allocated as a best effort service. If an extent big enough for a particular CQ is not available at the time of request, the Mapping Manager, which is a component of the SV, tries to find the best possible alternative for the request. The extent allocation mechanism is very similar to the mechanisms used by most extent-based file systems (ext4, XFS, VxFS, etc.). Free extents are maintained in a B+tree configuration to aid in fast lookup or search. Over time, the cache space could become fragmented, leading to a large number of small extents. In order to overcome this problem, one would have to defragment the space by moving data around in a way that increases the number of free contiguous blocks, or extents. This process of defragmentation could also involve combining multiple occupied extents with the goal of reducing metadata overhead. Specifically, if a cache group has more than one extent allocated to it (i.e., it is using a secondary record in the Global Mapping Structure), during the process of defragmentation, these occupied extents could be combined to form a smaller number of larger extents, thereby reducing metadata overhead.

4.3. Global Mapping

In our design we use hash tables for performing the lookups, but the structure of the record is unique. Figure 3 shows the hash table for a single Virtual LUN. Every request made by any server has a Virtual Logical Unit Number (VLUN) and a Virtual Logical Block Address (VLBA). The VLUN is used to select the corresponding hash table, and then the VLBA is hashed to get the index into this hash table. The hash function we use in our design is,

$$hash_key = \lfloor (VLBA / MAX_CacheGroup_Size) \rfloor \% \beta \quad (1)$$

where $MAX_CacheGroup_Size$ is the number of logical blocks per cache group, and β is a variable that controls the length of hash chains or the length of the hash table. Each record contains information about a certain set of contiguous logical blocks. Figure 3 shows two types of records; i.e., primary records and secondary records. The primary record holds the address of the virtual logical block (VLBA, used for resolving hash collisions), the address in cache of the start of the extent group (extent offset), an array of offsets representing VLBA's stored within the extent (extent offset array), a pointer to a secondary record, and a pointer to the next primary record with same hash value. The number of elements in the extent offset array is determined by the number of blocks that can be accommodated in the extent (which, in turn, is determined by the CQ). In essence, an array element gives the logical address of the block that is present in cache at that extent offset. To better illustrate, consider the first primary record in Figure 3. The Virtual Logical Block Address is 2310. The extent offset is 1020. The first element (offset of zero) in the array holds the value 0101, or 5. Adding the VLBA of the record to this number would give us the actual VLBA at cache location 1020, so the block of data with VLBA 2315 is stored in cache location 1020. Similarly, the third array element is null and hence the cache location 1022 is free or unoccupied. Considering that we would like to use variable-sized extents, the problem of assigning the correct-sized extent is challenging. Because the extent size is determined by heuristics (i.e., CQ), the result is sometimes imperfect. This leads to cases where two or more extents might need to be allocated for the same block of contiguous logical address space. To handle such cases, we propose using the secondary record structures. The design and function of these structures is very similar to that of primary records with the exception that they do not need to store VLBA's or chain pointers (since collisions would have already been resolved before coming to this stage).

Once a VLBA is checked against the Global Mapping Structure and it is determined if it is cached or not, the following transformations are applied to the VLBA to determine the DA# (Disk Array Number) and the LBA within that disk array:

$$DA\# = \lfloor VLBA / ConstituentLUNSize \rfloor \quad (2)$$

$$LBA = VLBA - (DA\# * ConstituentLUNSize) \quad (3)$$

where $ConstituentLUNSize$ is the size of the portion of the Virtual LUN that resides in each disk array. Note that the $ConstituentLUNSize$ is different for data LUNs and the cache LUNs. This translated information is returned to the requesting server by the metadata component.

5. Deadline-based Prefetch Scheduler

Our design heavily relies on the performance of the prefetch or cache space. The size of this cache space is proportional to the size of the entire data set and is accommodated on disk. However, the use or purpose of this cache space is manifold. First, this space is used to store data prefetched from dormant disks using hints from applications. Second, once prefetched into this space from the dormant disks, the data is eventually read from the space by a higher-level server. Third, this space is used for staging write requests destined for dormant disks. Fourth, this data is eventually destaged to the dormant disks. Obviously, these are competing/coordinating factors using the same resource. The optimal use of this resource, or space, is dependent on intelligent arbitration between these competing factors. Here, we model this problem as a classical producer-consumer problem and try to solve the problem with the goal of deep prefetching.

In this model of the system, the prefetch requests and the write requests coming into the system from the servers form the consumer classes. The write destage requests or dirty flushing requests/operations and the read requests are classified as producers because these operations free cache space and generate (produce) a free slot. Here, we make an assumption that once a read operation/request reads a previously prefetched data block from the prefetch space, the corresponding prefetch slot can be freed. This assumption is based on the fact that, in general, almost all initiating servers, or, specifically, the file system components on the servers, implement some form of caching. Hence, if a block is read over the network or SAN, it is usually cached locally by the file system, and any further request for the same block is served using this local copy that is stored in the local cache. Therefore, the assumption that a read request qualifies as a producer of a cache/buffer slot is a reasonable one to make. Further, this assumption can be easily adapted to other situations to implement an approximation of second chance, or even least recently used, behavior by just changing the weight given to this class of producer.

Out of these four types of requests, the prefetch request scheduling is the only one that is under our control. The other three types help in estimating the state of the system only. Specifically, the write requests and the scheduled prefetch requests, which are consumers of buffer space, along with the read requests for prefetched data and write destage requests, which produce buffer space, help us estimate the load on the system, or the state of the system. By strictly controlling the scheduling of the prefetch requests, one can control the load on the system. Scheduling of these prefetch requests can be performed in multiple ways, namely:

First Come First Served (FCFS): All prefetches that come

into the system are executed in a FCFS fashion without any consideration of their corresponding deadlines. This kind of scheduling continues as long as resources, in this case the empty buffer slots, are available. Clearly, this is not an optimal or fair approach because prefetch requests that have more stringent deadlines can be denied service due to their arrival order.

Earliest Deadline First: At any given point in time, all prefetch requests are ordered in increasing order based on their deadlines, and based on the number of empty slots in the system, a corresponding number of prefetch requests are executed in the order of earliest to farthest deadline. This approach is not optimal either. A problem arises because prefetches arrive at random times and there is not temporal ordering between prefetches. This is because multiple servers use the same consolidated storage systems and hence multiple applications generate prefetch hints asynchronously. For instance, consider a case where five buffer slots are available at time $t = i$; using this approach we could end up deciding to schedule the next five requests in the sorted deadline order, say $d_1 = 20, d_2 = 23, d_3 = 35, d_4 = 49, d_5 = 54$. Now, at time $t = i + 1$, if a new prefetch request arrives with $d = 22$, and if no new cache slots are free, we would end up holding back this prefetch request to a later point in time when a buffer slot becomes free. This is clearly not optimal or fair, and in a consolidated storage system, where multiple sources of application hints exist, the problem is quite severe.

Prefetch Horizon: The principle here is that there is no benefit in executing a prefetch request earlier than it is actually required. If the cost or execution time of a regular buffer/cache hit is T_{hit} , and the cost or execution time for servicing a prefetch is T_{disk} , then $p_h = \frac{T_{disk}}{T_{hit}}$ is called the Prefetch Horizon. Specifically, there is no benefit of executing any request that is more than p_h accesses away from the current request. This principle can also be expressed as follows: If a block retrieval time from disk is t_{disk} , and the prefetch deadline of request p is d_p , then there is no benefit of scheduling p until the requesting application progresses to the point $p_h = d_p - t_{disk}$, which is called the Prefetch Horizon. Prefetching a block after its prefetch horizon does not give any benefit. This kind of delayed scheduling helps achieve a more optimal scheduling and overcomes the shortcomings of the previous two approaches. The objective here is fairness based on deadline. Multiple schedulers like TIP2 [5], TIPTOE [13], and FORESTALL [14] have been designed based on this core idea. These techniques cannot be directly applied to our scenario for a few reasons. First, all the above schemes assume that data accesses can be segregated based on different initiators or application processes. In our scenario, from a storage controller's perspective, all block accesses are the same. It cannot differentiate between applications

that are using the storage. Hence, all data accesses need to be seen as a single stream of requests. Any inference of application speed made based on arrival rates of their corresponding I/O requests (as is the case in TIPTOE, FORESTALL, etc.) is not applicable in our setting. Second, in the previously mentioned schemes, application hints are assumed to be an ordered sequence of requests without any explicit deadline. Their deadline is inferred by estimating the application speed. In our scenario, we do not have any explicit knowledge of the speed of the applications that are supplying the hints because our decision making, or scheduling, is happening remotely (in the SAN or storage controller) and away from the servers hosting these applications. Hence, supplying hints without deadlines would not work in our scenario. Consequently, the corresponding scheduling mechanisms are not directly applicable, either. Third, the above schemes consider LRU cache in addition to prefetch cache. In our scheme, since we are at the remote storage system level, we do not have to consider LRU for caching recent accesses by assuming that such caches will be part of the local server cache or file system cache. Fourth, of all these schemes, a couple of them, TIPTOE and Aggressive/LRU, do perform deep prefetching. However, their objective is to minimize stalls caused due to hotspots by making use of idle disks. In our scenario, the objective is different.

Our first objective is to achieve fairness based on deadlines, and the second objective is to give the underlying disk subsystem as much time as possible to execute a prefetch request. The first goal is obvious based on the previous discussion, but the second goal is mainly due to concerns of energy conservation. Since MAID disks are used as underlying permanent storage, in order to maximize energy efficiency, the disks should be OFF as much as possible. If all prefetch requests are issued to a disk well ahead of time, the disk could potentially make an intelligent grouping of these requests such that it works in batch mode. For instance, if prefetch requests with deadlines, $d_1 = 40, d_2 = 43, d_3 = 60, d_4 = 110$ are queued at a dormant, or OFF, disk, the disk can decide when to turn on based on the nearest deadline, and when it is ON, it can finish servicing the other requests which have less stringent deadlines. By doing so, the disk does not have to stay ON or come back from the OFF state again to service the remaining requests. Conversely, if one uses the prefetch horizon technique, the disk will receive the request with $d = 40$ just before its deadline and will not receive the other requests until their deadlines approach current time. Therefore, this would lead to multiple ON/OFF transitions of the disk, which, in turn, leads to excess power consumption and increases the chances of failure. This goal of energy conservation by batch execution is not easily quantifiable as a benefit model.

With these goals in mind we design a scheduler to achieve optimal scheduling. We try to perform deep prefetching [13]; i.e., prefetching as far into the future as the system permits. In [13] deep prefetching was based on a cost-benefit tradeoff, but here we base it on the resource constraint (i.e., buffer free space availability). We propose a slight modification to the current application hinting paradigms in order to facilitate more optimal performance. Instead of supplying application hints in a sorted order, we propose that the applications also consider their I/O request rate, precalculate the deadlines of prefetch requests explicitly, and supply them to the storage system. By doing so, the storage system can make intelligent scheduling decisions based on deadlines without knowing much about the applications supplying the hints.

5.1. Opportunistic Deep Prefetcher

In our approach, when a new prefetch request arrives at the scheduler, it is checked against the Global Mapping Structure to determine if it has already been prefetched or write cached. If not, a check is made to see if any cache extent has already been allocated for the cache group to which the prefetch request belongs. If such an extent is found, and if free space is available within the extent, this request is dispatched to the corresponding disk array along with the address of the free block to be filled. Note that dispatching a request to the disk array or disk does not necessarily mean that the request will be immediately executed by the target. It just means that the target disk is free to service this request any time before its specified deadline. If the corresponding cache group, and hence the prefetch request block, does not have any mapping in the global structure, the deadline of this request is examined further in order to make a scheduling decision. The scheduling decision for this request is now determined by several factors like the state of the cache and the remaining time between the current time and the request deadline. The objective here is to ensure fairness in scheduling. If this request were to be dispatched, resulting in the use of a buffer slot, and a new request with an earlier deadline arrives soon after, then the earlier scheduling decision should not have any adverse impact on this new request. In essence, if this new request with an earlier deadline cannot be scheduled due to lack of buffer space, then the previous scheduling decision is considered unfair. In order to avoid such scenarios, our approach uses the current state, and estimates of system parameters, to predict the future state. Ideally, we would like to avoid all scheduling decisions that lead to unfair scheduling. This ideal case can only happen if we have perfect knowledge of the system and the system is not dynamic; i.e., properties do not vary with time. In such ideal cases, we could directly use the concept of maximum allocation, like in the Banker's Algorithm, to avoid deadlocks.

Safe & Unsafe States: A state is said to be safe if, and

only if, enough buffer slots are available to service all incoming prefetch requests that have a deadline earlier than the current request being serviced. A state is said to be unsafe if the number of buffer slots available is not sufficient to service all incoming prefetch requests that have a deadline earlier than the current request. Here, we assume that the prefetch requests are coming into the system at the rate PRR (Prefetch Request Rate). In the worst case, all the requests that can come between the current time and the deadline of the current request under consideration could have a deadline earlier than the deadline of the current request. In such a scenario, we check to see if we have enough free buffer slots in the system to service $PRR * (d - \text{current_time})$ requests, where d is the deadline of the current request. If the available free space is sufficient to service these requests, then the resultant state is a safe state; hence, the current request can be scheduled for servicing. If servicing of a given request results in an unsafe state, we do not schedule this request. It is held back until the state of the system changes or the deadline becomes closer, thereby increasing the possibility of entering a safe state.

In practice, the deadlines of requests between the current time and the deadline under consideration may not all lie before the deadline under consideration. Hence, we consider the average-case scenario by using an estimate of the deadline distribution. We obtain a distribution by maintaining a histogram of the deadlines of all incoming prefetch requests. This histogram of the number of requests per deadline can be translated to a probability value per deadline. The deadlines, and hence their probabilities, can fluctuate drastically over time due to variations in application behavior. Using a general averaging or cumulative summation scheme will not reflect the more recent behavior. Hence, for estimation purposes, we use a windowing estimation scheme by successively considering groups of m samples. By using this grouping, we try to ensure that the most recent m samples are considered for estimation of deadline probabilities. Equation 4 gives the deadline probability estimation equation,

$$P(d = x) = P_{t-1}(d = x) * \frac{m-n}{m} + P_{current}(d = x) * \frac{n}{m} \quad (4)$$

where $t - 1$ is the previous estimation window, m is the window size, and n is the number of samples collected in the current window. Now, if a request with deadline d arrives at time t , we estimate the expected state of the system based on the result of Equation 6,

$$E(U_t) = (U_t + 1) + d * (PRR * P(\text{deadline} < d)) \quad (5)$$

$$= (U_t + 1) + d * (PRR * \sum_{i=0}^d P(\text{deadline} = i)) \quad (6)$$

where U_t is the used cache size at time t . Note that we consider a probability that the deadlines of incoming requests

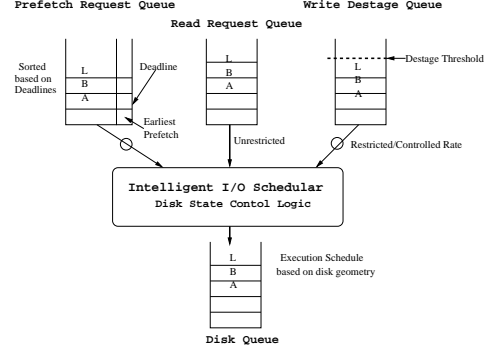


Figure 4. I/O Queuing Model

will be less than the current deadline. Using this probabilistic estimate gives us the average-case scenario. Now, if $E(U_t)$ is greater than the cache size, then the system is expected to transition into an unsafe state upon servicing of the current request; hence, this request is not scheduled and is held back for consideration at a future time instant. On the other hand, if $E(U_t)$ is less than the cache size, the system is expected to remain in a safe state and hence the current request is scheduled immediately (i.e., dispatched to disk for servicing).

5.2. Delayed/Lazy Execution

From an energy conservation point of view, a disk should be in its Standby or OFF state as much as possible, and the number of transitions between ON and OFF or Standby states should be minimized. If the disks were to service each and every prefetch request immediately, without considering the deadline, or service write destage requests when they arrive, our objectives of energy conservation would be violated. However, disks need to immediately service real-time read requests or requests which have not been hinted or prefetched. For the two non-immediate classes of requests, we try to develop a delayed, or lazy, scheduling mechanism based on intelligent queuing.

Disk Queuing: Standard disk queuing schemes use certain physical characteristics of the disks to arrive at an optimal access schedule. For our design the focus is not these disk scheduling schemes. We focus on adding more intelligence to the storage controllers for the purpose of energy management. Specifically, the storage controller maintains a set of logical queues for each disk drive. The properties, or dynamics, of these queues influence the power management decisions. Figure 4 shows the queuing model for a single, non-cache, MAID disk in our system.

The Write Destage Queue (WDQ) holds all destage requests sent by the Mapping Manager to the disk. These requests do not have any specific deadline or scheduling order. The Prefetch Request Queue (PRQ) holds all prefetch requests dispatched by the Mapping Manager to the disk. This queue is sorted based on the deadline of prefetch requests. The Read Request Queue (RRQ) holds all real-time

read requests for blocks on the disk. These requests do not have any specific order, but are to be considered the highest priority.

The final goal of this queuing system is to ensure that energy consumption of the disk drives is minimized without sacrificing real-time performance and that the deadlines of prefetch requests are met. Specifically, we develop a power management scheme which works according to the status of these queues. We develop a set of rules which govern the power management, or transitioning, of disks. A disk can make a transition to its Standby state (or OFF state) from its Idle (i.e., ON but not Busy) state if, and only if, all of the following conditions are met:

- RRQ is empty,
- Deadline of the prefetch request at the front of the PRQ is well beyond the disk transition time (ON to OFF plus OFF to ON again),
- Depth of WDQ is below a specified threshold, and
- Disk Idle Time, or time since the last real-time read access, is above a specified threshold.

A transition from the Standby or OFF state to an Active (Busy) or Idle state can be made if, and only if, one or more of the following conditions are met:

- A new Read Request arrives (RRQ is no longer empty),
- Deadline of the prefetch request at the front of the PRQ is close to the disk transition time,
- Depth of WDQ is above a specified threshold, or
- Time since the last transition into Standby or OFF state is greater than a specified threshold.

The last condition for a transition from a Standby or OFF state to an ON state is purely from the standpoint of minimizing the risk of disk failure. Studies have shown that if a disk drive is in an OFF or Standby state for too long, moisture from humidity can set in and increase the probability of disk failure on the next transition. [3] overcomes this issue by using a similar technique called Disk Jogging.

6. Simulation Evaluation

In order to quantify the performance of our algorithms and compare them against standard techniques, we have built a detailed simulation of the GreenStor storage subsystem. Various Storage System components were modeled using a discrete event simulation package called OPNET [15]. Since we are mainly focused on large-scale storage systems with possibly hundreds of disks, for practical considerations, we decided to use event-driven simulation instead of actual prototyping.

Table 1. Disk Parameters

	MAID Disk	Cache Disk
Avg. Access Time	8 ms	4ms
Number of Platters	4	1
Disk Capacity	146 GB	36 GB
I/O Block Size	64 KB	64 KB
Avg. Disk Spin-up Time	10 s	10 s
Rotational Speed	10000 RPM	15000 RPM
Idle State Power	6 w	7 w
Busy State Power	10 w	12 w
Standby State Power	1 w	1 w
Spin-up Peak Power	15 w	15 w

The storage subsystem used for these experiments consists of 12 servers, an interconnection network, a virtualization device, and two disk arrays. Each server is, in turn, composed of multiple asynchronous applications that generate read, write, and prefetch requests. All energy measurements mentioned henceforth include energy consumption of disk drives and cache drives only. This study does not include the power consumption of the storage controllers or servers. At this point, we do not have a way of quantifying the increase in server energy consumption due to the additional task of prefetch/hint generation.

Disk Drive Modeling: Table 1 shows the disk drive and cache drive parameters that we model. The disks modeled here are single-speed disks with support for Active (Busy), Idle, Standby, and Shutdown (Sleep) states. All transitions between different states are modeled very accurately both with respect to timing and power parameters. A simple First Come First Served scheduling algorithm is implemented for these disks.

Disk Array Modeling: For our simulation setup we model a storage subsystem with two disk arrays, each with 60 regular disks and 6 cache disks (per array). The disk array controller simulates multiple logical volumes across these disks. This disk array controller also actively monitors disk access in order to determine idle periods. All the intelligence about disk transitioning is embedded in this controller. If any disk is not scheduled to be accessed beyond its break-even point (i.e., the point at which the transition costs are surpassed by the power savings), the controller makes the disk transition into Standby mode.

6.1. Comparison of Scheduling Schemes

For the purpose of comparison and quantification, we compare our Opportunistic Scheduling scheme with a pseudo Prefetch Horizon scheme. We call it a pseudo prefetch horizon scheme because the simulation is conceptually similar, but not identical, to the original scheme. We do not consider a First Come First Served scheme because, as expected given the delayed/lazy disk scheduler, the power consumption of this scheme was found to be the same as that of our scheduling scheme. FCFS fails when it comes to fairness, however, as it does not arbitrate based on

deadlines while scheduling a request. We do not consider traditional MAID systems (like Copan MAID) for the comparison as prefetching in response to application hints is not a part of these original MAID designs and hence would not be a fair comparison. Hereafter we consider only the prefetch horizon scheme for comparison. For all of the following experiments, we keep the simulation workload constant as performance numbers are gathered. All the results reported here are averaged over five runs. The standard deviations are very low; hence, the graphs do not include this measure.

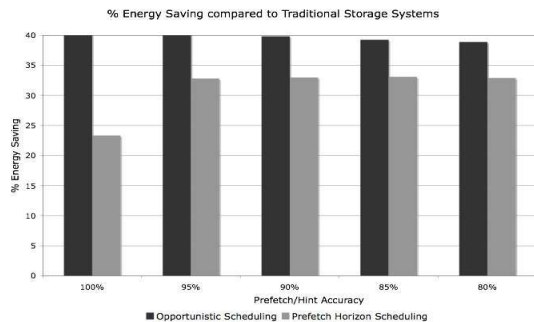


Figure 5. Percent Energy Savings

Figure 5 shows the plot of percent energy savings for the two schedulers with varying prefetch/hint accuracy. In this experiment, we measure the energy consumption of a traditional storage subsystem when executing the fixed workload and then perform the same measurements with the two scheduling schemes. As expected, for our scheme, the case with 100% accurate predictions yields the maximum energy savings. Our opportunistic scheduling scheme coupled with delayed scheduling yields close to 40% energy savings. The prefetch horizon scheme results in an energy savings of about 23%. As the prefetch/hint accuracy reduces, the energy savings also drop slightly for our scheduling scheme. This is exactly as expected, because with a reduction in prefetch/hint accuracy, more read requests need to be serviced by the dormant disks instead of the cache disks. This reduces the time these dormant disks spend in their Standby/Sleep state and thereby increases power consumption. However, in the case of the prefetch horizon scheduling scheme, the energy savings actually increase as the prefetch/hint accuracy reduces. This is slightly counter-intuitive. Upon further investigation, we find that the main reason for this increasing energy savings is that the number of disk restarts in this case decreases with the reduction in prefetch/hint accuracy.

Deeper examination of the disk states and time spent in each state reveals several interesting facts. We classify the time spent by the disks into Idle state, Busy state, and Sleep/Standby state. The time spent transitioning from Sleep/Standby to Idle/Busy and vice versa is counted as part of the time spent in Sleep/Standby state. We find that

our scheduling scheme minimizes the time spent in the Idle state (i.e., the state where disks are spinning but inactive) at the cost of making more disk transitions, or restarts, to and from Sleep/Standby. This is mainly attributed to the opportunistic behavior of the scheduler, which gives the disks more time to service a request. Another interesting result is the fact that our scheduling scheme minimizes the time spent in the Idle state to less than 20%. Ideally, one would expect to see much higher energy savings in Figure 5 than what is shown. The reason for this behavior is the large number of disk transitions that occur with our approach, as shown in Figure 6. Disk restarts have a very high energy penalty. Typically, the Idle state power consumption is about 6-8 watts, while the peak spinup power consumption is about 15-20 watts. In addition, the cost of keeping the set of cache disks always powered on further reduces the energy savings. In the prefetch horizon case, the percent of time spent in the Idle state is still very significant, about 50%. The main reason for this is that the prefetch horizon scheme tends to dispatch a constant stream of requests to the disk and does not allow the disk much flexibility in serving these requests. One interesting behavior to note is that, in prefetch horizon, with a decrease in prefetch/hint accuracy, the percent of time spent in the Idle state increases. The main reason for this behavior is that the disk gets fewer chances to transition into its Sleep/Standby state because both unhinted and hinted prefetch requests arrive at the disk at a steady rate that needs to be serviced immediately.

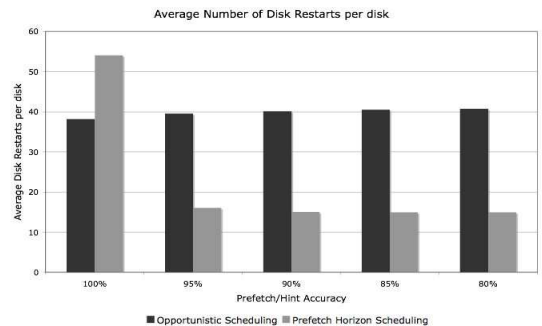


Figure 6. Average # of Disk Restarts

Figure 6 shows the average number of restarts, or transitions, per disk for each of the scheduling schemes. With 100% accurate predictions, the number of disk restarts is about 50 for the prefetch horizon scheme. This is slightly higher than that of our scheduling scheme, where the average number of disk restarts is about 40. This is to be expected as our scheme tries to dispatch requests well in advance in order to maximize the time spent in the Sleep/Standby state. When moving away from perfect prefetch/hint accuracy, the number of disk restarts in the prefetch horizon case drops drastically, whereas it is relatively constant in the case of our scheduling scheme. This,

in turn, leads to less time spent in the Standby/Sleep state when using the prefetch horizon scheme. Because the energy saved by avoiding disk restarts is considerably larger than the energy lost in this reduction of sleep time, we see this interesting behavior where the prefetch horizon scheme’s energy savings increases as the hints become less accurate.

Figure 7 shows the read performance of the system with varying prefetch/hint accuracy. For the purpose of this study, our main interest with respect to performance is determining if a request was served from a cache disk, a dormant disk that was on, or a dormant disk that was off. Hence, we classify the read responses into two categories; namely, requests with response time in the milliseconds range and requests with response time in the seconds range. With 100% accurate predictions, all the read responses seem to be completed in the millisecond range for both approaches. As prefetch/hint accuracy decreases, the percentage of requests completed in the millisecond range is smaller in the case of our scheduling scheme compared to the prefetch horizon method. Again, as shown in Figure 6, the disks spend less time in a Sleep/Standby state with the prefetch horizon scheme as a consequence of a smaller number of restarts. Hence, when an unhinted request arrives, the probability that the corresponding disk is not in the Sleep/Standby state is higher for the prefetch horizon scheme, and indirectly results in better read performance than our approach.

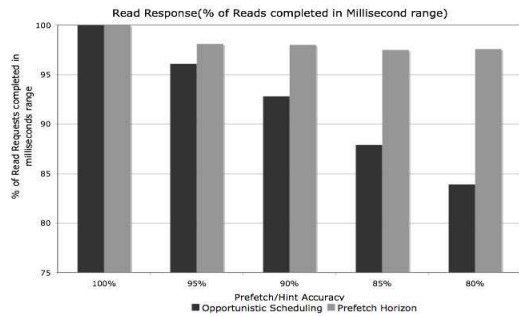


Figure 7. Read Response Times

6.2. Comparison of Metadata Management

Figure 8 shows a comparison of different cache/prefetch block allocation schemes. We compare our extent-based allocation scheme, which uses CQ to dynamically determine the size of the extents, with a traditional one-to-one mapping scheme and with a scheme that uses a larger fixed granularity; i.e., it stores fixed-size groups of 4, 8, or 16 blocks (labeled FG(4), etc., in the figure). For this simulation, we generate a synthetic workload consisting of logical block requests with an average cluster size of eight contiguous logical blocks. Out of this synthetic workload we randomly pick a logical block cluster and try to allocate ap-

propriate space in cache using different allocation schemes. This random sampling process is continued until the cache is full, at which point we calculate the performance results.

Since our approach makes use of a heuristic estimate of contiguity within any cache group, we also inject different percentages of error into our contiguity estimates to simulate real system behavior where access patterns can change over time. The size of the mapping structure is calculated based on the number of extents, or groups, that fit in the cache. The one-to-one method uses an inverted mapping, so its mapping structure size is directly proportional to the number of blocks that fit in the cache. The FG(4) method creates one fourth as many mapping entries, so its normalized size is 0.25, as is seen in Figure 8. However, because unneeded blocks are sometimes brought into the cache when a fixed-size group is used, the cache utilization is worse. This pattern continues for FG(8) and FG(16). Our CQ-based extent mapping method stores more information in the Global Mapping Structure, so each cache mapping entry is assumed to be about twice as large as the one-to-one mapping entry (16 bytes vs. 8 bytes). The figure shows that our mapping structure uses a little more space than a fixed-size group of four blocks. However, our cache utilization, which is above 90% even with slightly erroneous CQ estimates, is higher than any of the fixed-size methods. This shows that we are able to substantially reduce the size of the mapping structure without losing very much in terms of cache utilization. For a more detailed version of this work the reader should refer [16].

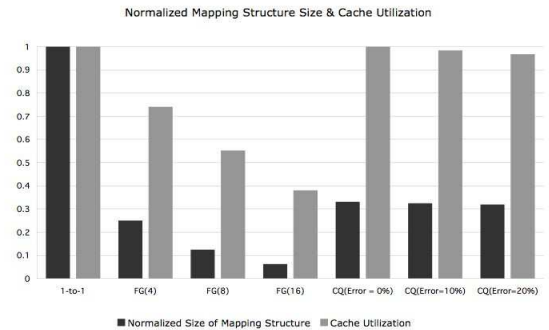


Figure 8. Metadata Management Schemes

7. Conclusion & Future Work

In this work we have clearly shown that energy savings can be substantial when application hinting is used on top of MAID storage systems. On average, the results show that the energy savings achieved by using even the existing scheduling techniques is greater than 25%. Scheduling technique used drastically impacts the amount of energy consumed by the system. On average, our GreenStor storage system using its opportunistic deep prefetcher provides energy savings of an extra 10% or more compared

to other scheduling techniques. When prediction accuracy is very high, a system with our scheduling technique consumes about 40% less power than traditional storage systems without any prefetching.

This is an ongoing project and we are currently investigating a variety of other issues in the domain. One important point to note is that the energy consumption results that we have presented do not include the extra energy consumed by the servers due to the additional task of hint generation. We plan to work on estimating this additional server-side energy cost. In this work we do not consider any constraints on disk transitions. It has recently come to our attention that, in MAID systems, disks are packed every closely to minimize the impact of humidity on disk lifetime. Specifically, if disks are not turned on for a long period, the probability of failure on the next spinup increases due to the collection of moisture from humidity. Hence, MAID manufactures pack disks closely to minimize the effects of humidity. In our proposed design, this close packing of disks could lead to overheating in certain regions of the disk array if not properly arbitrated. We are currently investigating approaches by which intelligent arbitration of disk spinups could be used to minimize the problem of overheating.

8. Acknowledgments

We would like to thank all our sponsors - DISC member companies, Department of Energy's Los Alamos National Labs and National Science foundation for their continued support.

References

- [1] A. P. Corporation(APC). (2004) Determining total cost of ownership for data center and network room infrastructure. [Online]. Available: www.apcmedia.com/salestools/CMRP-5T9PQG_R3_EN.pdf
- [2] M. Hopkins. (2004) The onsite energy generation option, the data center j. [Online]. Available: http://datacenterjournal.com/News/Article.asp?article_id=66
- [3] D. Colarelli and D. Grunwald, "Massive arrays of idle disks for storage archives," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Los Alamitos, CA, USA, 2002, pp. 1–11.
- [4] K. Rajamani and C. Lefurgy, "On evaluating request-distribution schemes for saving energy in server clusters," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, CA, USA, 2003, pp. 111–122.
- [5] R. H. Patterson, G. A. Gibson, and M. Satyanarayanan, "Using transparent informed prefetching to reduce file read latency," in *Proceedings of the IEEE/NASA Mass Storage Systems*, 1992, pp. 329–342.
- [6] D. Rochberg and G. Gibson, "Prefetching over a network: early experience with ctip," *SIGMETRICS Perform. Eval. Rev.*, vol. 25, no. 3, pp. 29–36, 1997.
- [7] E. Pinheiro and R. Bianchini, "Energy conservation techniques for disk array-based servers," in *Proceedings of the 18th international conference on Supercomputing*, New York, USA, 2004, pp. 68–78.
- [8] Q. Zhu, Z. Chen, L. Tan, Y. Zhou, K. Keeton, and J. Wilkes, "Hibernator: helping disk arrays sleep through the winter," *SIGOPS Oper. Syst. Rev.*, vol. 39, no. 5, pp. 177–190, 2005.
- [9] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Franke, "Drpm: dynamic speed control for power management in server class disks," in *Proceedings of the 30th international symposium on Computer architecture*, New York, USA, 2003, pp. 169–181.
- [10] F. Chang and G. A. Gibson, "Automatic I/O hint generation through speculative execution," in *Proceedings of the 3rd symposium on Operating System Design and Implementation*, CA, USA, 1999, pp. 1–14.
- [11] T. C. Mowry, A. K. Demke, and O. Krieger, "Automatic compiler-inserted I/O prefetching for out-of-core applications," in *Proceedings of the 1st symposium on Operating System Design and Implementation*, 1996, pp. 3–17.
- [12] A. Weissel, B. Beutel, and F. Bellosa, "Cooperative I/O: a novel I/O semantics for energy-aware applications," *SIGOPS Oper. Syst. Rev.*, vol. 36, pp. 117–129, 2002.
- [13] A. Tomkins, R. H. Patterson, and G. Gibson, "Informed multi-process prefetching and caching," in *Proceedings of the ACM SIGMETRICS*, NY, USA, 1997, pp. 100–114.
- [14] P. Cao, E. W. Felten, A. R. Karlin, and K. Li, "A study of integrated prefetching and caching strategies," *SIGMETRICS Perform. Eval. Rev.*, vol. 23, no. 1, pp. 188–197, 1995.
- [15] (2007, May). [Online]. Available: www.opnet.com
- [16] N. Mandagere, J. Diehl, and D. Du, "Greenstor: Application-aided energy-efficient storage," Digital Technology Center, Univ of Minnesota, Tech. Rep., 2007. [Online]. Available: <http://www.dtc.umn.edu/disc/publications.shtml>