A Performance-oriented Energy Efficient File System

Dong Li (li@cse.unl.edu), Jun Wang (wang@cse.unl.edu)
Department of Computer Science and Engineering
University of Nebraska Lincoln, Lincoln, NE, 68588 USA

Abstract

Current general-purpose file systems emphasize the consistency of standard file system semantics and performance issues rather than energy-efficiency. In this paper we present a novel energy efficient file system called EEFS to effectively both reduce energy consumption and improve performance by separately managing those small-sized files with a good *group access locality*. To keep compatibility, EEFS consists of two working modules: a normal Unix-like File System (UFS) and a group-structured file system (GFS) that are transparent to user applications. EEFS contributes a new grouping policy that can construct files groups with group access locality and be used to migrate files between UFS and GFS. Comprehensive trace-driven simulation experiments show that EEFS achieves a great energy savings by up to 50% compared to that of the general-purpose UNIX file system, and simultaneously delivers a better file I/O performance by up to 21%.

Introduction

File systems play an important role in today's diverse low-power computer system designs, such as laptops, personal digital assistants and digital camera/video devices, in which energy should be managed as a first class OS resource [Zeng et al., 2002]. Current file systems bias on either performance improvement or energy conservation. To the best of our knowledge, none of them specifically addresses both factors.

There is some research work done on how to reduce energy consumption for disk-based file and storage systems. It is quite common that disk is one of I/O devices that consume much less power in an idle/standby state compared to that in an active state [Zedlewski et al., 2003]. Researchers try to reduce energy consumption by fully exploiting such feature and their methods are categorized mainly into two basic classes: a direct approach and an indirect approach. For the direct approach, researchers attempt to keep disk switch between high power state and low power state at the disk driver level by rescheduling disk requests and arbitrarily stretching the idle periods [Douglis et al., 1995, Greenawalt, 1994, Helmbold et al., 1996]. The indirect approach generally uses various prefetching and caching policies [Papathanasiou and Scott, 2004] working at I/O buffer cache level to increase the disk burstiness, therefore indirectly stretching the idle period for disk drives. Although the above two methods could result in a significant energy saving, they typically leads to a negative side effect on performance.

Recently, Zheng et.al [Zheng et al., 2003] show that improving file layout or scheduling background activities, which are used to improve read/write locality or burst ratio, can achieve energy-saving. Since disk accesses are non-uniform, optimizing file layouts increases file system spatial locality, and thus a system with less I/Os tends to have better performance and lower energy consumption [Smith and Seltzer, 1996]. LFS [Rosenblum and Ousterhout, 1992] developed an optimized disk layout policy to boost write performance but its garbage collector grants the disk driver little chance to be idle and significantly compacts disk idle periods. A similar problem is applied in Wang et al. [Wang et al., 1999] 's virtual log based file system deploying a fictional programmable disk that is expensive and difficult to implement.

The above two approaches motivate us to develop a new energy-efficient file system named EEFS by combining the advantages of the above strategies while addressing their limitations, and realize both high energy efficiency and high-performance by translating its performance gain and energy savings from the reduced I/Os. EEFS optimizes file data layout and conduct large I/O based prefetch by fully exploiting a new *group access locality*, which means if a file out of a group is accessed, other files from this group have a high probability to be accessed in the near future. As a result, EEFS can reduce the overall I/Os and indirectly increase disk burstiness. EEFS has two working modules: UFS (Unix File System) and GFS (Group-based File System). UFS manages files and metadata as general Unix file system does while GFS only manages those grouped files that have strong group access locality. A new dynamic, file grouping algorithm is developed to capture such locality and construct file groups. Although EEFS is designed for disk-based storage devices, it can be applied to any mobile storage devices (e.g. Flash PC cards and wireless LAN cards) that are satisfied with a common feature: a large I/O consumes a similar power to a small I/O and their latency differences are modest [Zheng et al., 2003]. EEFS can also cooperate with energy-efficient policies working on other levels different from file system, such as disk spin-down scheme, application-aware prefetch, to achieve more energy saving. Comprehensive experiments show that EEFS can realize a large energy saving by up to 50% compared to general-purpose UNIX file system

while simultaneously achieving a better I/O performance by up to 21%.

The Design and Implementation of EEFS

The fundamental goal of EEFS is to achieve a good energy efficiency and good I/O read/write performance at the same time.

EEFS System Architecture

The idea of EEFS is quite straightforward, deploying GFS to manage groups with small affiliated files to reduce the total number of I/O. To attain the POSIX compatibility, EEFS consists of two working modules that are transparent to applications: a UFS-like (Unix File System) module and a GFS (Group-based File System) module. A file is physically located in either module. All files go through some "early-bird" periods in UFS, including file creation and the very first access. Once EEFS finds a set of file groups with good group access locality, it migrates them from UFS to GFS. EEFS periodically invokes a *group shuffle* process to conduct this migration procedure between GFS and UFS. Similarly, when GFS evicts some files that do not follow group access locality any more, EEFS migrates them back to UFS. UFS maintains all directory related information and operations, while GFS is responsible for managing metadata and data of files migrated from UFS.

EEFS dispatches different type of requests to either GFS or UFS respectively. When a file I/O request arrives, GFS checks its file lookup table to see whether the requested file in GFS or not. If yes, the whole group containing the requested file is fetched into the memory, Otherwise, GFS hands over the request to UFS and the request is served as general-purpose UFS-like file system. Figure 1 illustrates the architecture of EEFS.

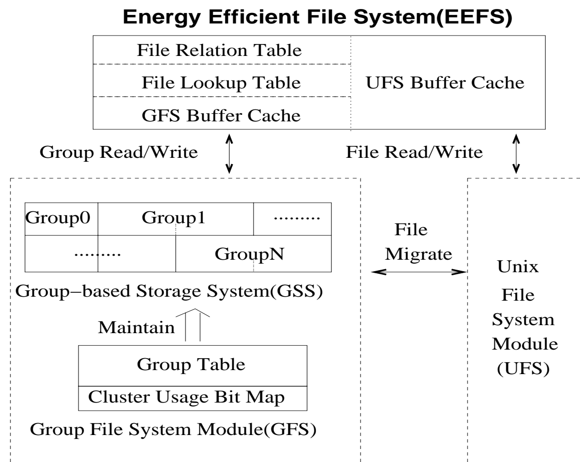

Figure 1: EEFS System Architecture

Buffer Cache and File Lookup Table

The I/O buffer cache of EEFS is logically divided into GFS buffer cache and UFS buffer cache. UFS buffer cache works the same as general I/O buffer cache in UNIX platform. GFS buffer cache stores recently accessed active groups in the memory.

A file lookup table is developed in GFS to maintain file metadata information to be used for fast GFS file lookup. It is implemented as a hash table saved on a reserved disk space in GFS, typically occupying two special disk clusters (a disk cluster is a storage unit in GFS, the total size of such a file lookup table is usually less than 128 KB). It is loaded into memory when EEFS is mounted. Figure 2 shows its detailed structure. The most important field is the file metadata that conforms to file inode structure used in UFS except the disk address field. The address field is truncated or added when the file is migrated between UFS and GFS. It is retrieved in many common traversal operations (e.g., ls and fstat). The size of each entry is generally less than 80 bytes.

Such a design has two significant advantages: (1)**Fast file lookup** In EEFS, the first step to locate a file is to check this table. A lookup hit saves the possible expensive directory pathname lookup in UFS (directory name lookup cache hit rate is typically only 70% in UFS). More importantly, most files are correctly prefetched into memory after one of their affiliated files from the same group gets accessed. For the workloads that show good access locality, most files lookup and file data can be loaded quickly. (2)**Reduced file metadata updates** In general Unix-like file system, frequent file metadata updates may result in very expensive I/Os because each file metadata is stored separately on disk and any metadata update leads to a possible disk I/O. In GFS, all the file metadata are saved in file lookup table. As a sequence, all metadata updates become an aggregated large disk write. More importantly, a lightweight checkpointing may further reduce the I/O traffic involved for only writing the updates of file lookup table to the disk.

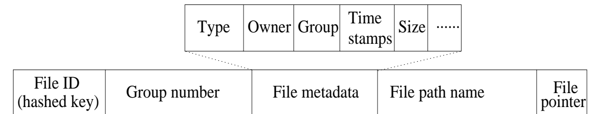

Figure 2: File Lookup Table

The Group-based Storage System (GSS)

Previous studies show cluster-based storage systems have achieved high I/O performance [Wang and Li, 2003, Shriver et al., 2001] for only disk reads. In GSS, we revolutionize the storage design in [Wang and Li, 2003] to handle both disk reads and writes more energy efficiently. GSS is developed to efficiently manage file groups on disk. We will discuss how to identify such groups in Section "The File Grouping Algorithm".

GSS divides its disk space into large, fix-size clusters. Each group contains at least one cluster. Large group may span several contiguous clusters. Group is the basic unit for any read and write.[1] Such a large I/O design maximizes the use of disk bandwidth and saves energy by

[1]From experiments, we found a cluster size of 64 KB or 128 KB achieves the best system performance.

the reduced small I/Os and asynchronous write buffering exploiting locality among multiple file group members.

Figure 3 shows the disk layout of GSS. All GFS file metadatas are organized by the file lookup table which is saved in a special two-disk-cluster-sized space and make the file metadata update more efficient. It may be noted that these files metadata have the same structure with file inode of UFS in order to smoothly migrate between GFS and UFS. GFS uses a Group Table to record the

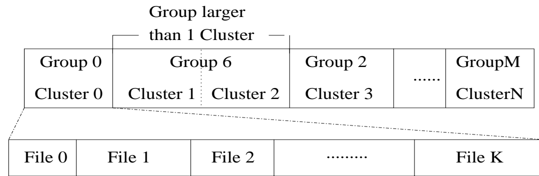

Figure 3: Disk layout of GFS

mapping relation from a logical group to physical cluster(s) in GSS. Another data structure used to maintain the status of disk cluster in GSS is Cluster Usage Bitmap. Cluster Usage Bitmap tracks the usage of all clusters in GSS. Since the cluster is the allocation unit in Cluster Usage Bitmap, we use a bitmap to track the cluster usage to save space. Searching in Cluster Usage Bitmap is not a problem because any group can occupy a limited number of clusters.

Notice all groups are subject to change for the variations of users file access pattern. When new groups are identified or the group access locality is changed in some groups, a file migration process between UFS and GFS is initiated (details in Section "File Migration and Group Shuffle").

Reads The read operation in GSS is very different from those in other file systems such as LFS [Rosenblum and Ousterhout, 1992] or FFS [McKusick et al., 1984]. Because groups are the basic I/O unit in GSS, when reading a file, the entire cluster that contains that group is read, in a single large disk I/O, into the memory. If the group occupies more than one clusters, all the clusters will be read into memory. The reason of choosing such a design is to realize affiliated large reads. Reading an entire group that contains many small files, instead of a single file, is in fact doing *prefetching* with very few extra overhead because the disk seek and rotational latency, which are independent of request sizes, dominate the disk access time for small disk request. Because many separate small disk requests are replaced with few large group disk access, the disk burstiness is increased, the disk idle time is stretched and therefore energy efficiency is attained.

Writes GFS employs append-only write policy to write all new information (updated groups or new groups) to GSS sequentially. A Cluster Pointer is recorded as the last cluster number after a most recently successful write to indicate current position. There are two major differences between GFS and LFS: (1) Group is the basic read/write unit in GFS and any time GFS only writes an integer-times number of cluster to GSS, and (2) When the Cluster Pointer reaches the end of GSS, it will turn around to write to the first empty cluster of GSS from the beginning.

When GFS writes an updated group g_{new} to the GSS, firstly GFS marks a "deleted" tag for those disk clusters occupied by group g_{old} that is the original group stored on disk before the update, and then writes the group g_{new} to disk. GFS doesn't reclaim the disk space of group g_{old} only until EEFS completes the next checkpoint write. After that, EEFS marks "reclaimable" for the invalid disk clusters in group g_{old}.

Invalidation When EEFS conducts a file deletion on GFS, namely GFS invalidates a file, GFS will first set a delete flag in its file inode of group table and then remove its corresponding entries in file relation table of GFS, file lookup table of GFS and its directory inode of UFS. If its host group to which the invalidated file belongs is in memory, a corresponding group update operation occurs when this group is flushed back to disk. If the host group is not in memory, GFS still conducts similar updates in related file system metadata structures except that it will not read this group into memory to do the group compaction only until its group gets another future access. This may introduce some negative effects such as cache pollution and wasted disk bandwidth since the invalidated file(s) may be read into memory along with other live files in the same group. But this penalty is limited to a narrow period and will not become a critical issue. There are two reasons: 1) In case of any future access to its affiliated live file in the same group, GFS will conduct group compaction in memory and the later accesses to the group will not have the problems at all. The only penalty here is the intra-fragmentation within a group; and 2) Since the group shuffle occurs periodically, those groups will be updated anyway. As a result, such intra-fragmentation problems will be restricted to every two group shuffles.

A distinct advantage of GFS invalidation is that it does not generate external fragmentation problems and not require garbage collection compared with LFS.

A New File Grouping Algorithm

One important factor to make GSS work well is to develop a good on-line grouping algorithm to accurately find and construct groups of files that follow a good group access locality. To capture such a locality in file system, we develop a component-based grouping algorithm to address the problem. A successor is simply a different file accessed immediately following the current file. The we define a *file group* as a set of files that are connected by file successor relationships and have group access locality.

We want to find such kind of file groups that have the following characteristics: (1) Group access locality, (2) Stable group members (i.e., file), (3) There are no overlap among any groups, (4) Grouped files will be frequently accessed in the near future.

Component-Grouping In order to find file groups, we firstly construct a directed graph, named file-relation graph to represent the file successor relationship. Let each unique file be a vertex of the directed graph. If file B is the successor of file A, we add a path from A to B. The weight of the edge (A,B) is defined by the successor prediction model. We adopted the Recent Popularity [Amer et al., 2002b] as our file successor prediction model to build this directed graph since this predictor has good prediction accuracy and is easy to maintain. Another important reason to select this model is that we don't want this background grouping activity pays too much computation. Recent popularity model works like this: for the list of k most recently observed successors of a file, a successor can become a prediction if it occurs at least j times in this list. For example, file B appears 4 times of 10 most recently observed successors of file A. If we let j be 3, B is a prediction of A and B's popularity as the successor of A is 3/10. Thus, the weight of edge (A,B) in the file-relation graph should be 3/10. Of course, this weight may be dynamically adjusted with the changing of file access flow.

After constructing the file-relation graph, we define a file group is a component in the file-relation graph. A component is a subgraph in which there exists at least one directed path between any two vertexes. To control the number of files in the file group, we set two thresholds, G_{max} and G_{min}. G_{max} is the maximum files which a file group can have and G_{min} is the minimum number of files a file group must have. They are important for GSS to layout groups. If group size is too big, then energy savings may be diminished in return in case some files prefetched are not accessed in the near future, reducing effective disk bandwidth and polluting the buffer cache, especially in heavy I/O workloads. To strike a good trade-off, if the group size is larger than G_{max}, GFS will break the group into some sub-groups through a depth-first search algorithm. On the other hand, if the group size is too small, for example only two files, the overhead to maintain the group might consume more energy than the energy saving gained from asynchronous write, burstiness and layout locality. Experiments indicate the system works well when G_{min} is three.

This algorithm is executed on every group shuffle. Group shuffle is not a frequent action because usually user access pattern does not change very frequently, as backed in our experimental results: most produced-groups in EEFS have very few changes even after a few hours.

Comparison with Other Grouping Policies There are some other algorithms provided by previous research [Amer et al., 2002a, Kroeger and Long, 1999] to do file grouping. The differences between their grouping strategies and ours are detailed as follows. (1) They used file grouping to improve buffer cache performance, while our goal is energy efficiency. As a background activity, our algorithm must be a light-weight algorithm. (2) They use the file group only for prefetching while we take the file group as a basic storage unit and become the core design component in our new flat name space structured storage system GSS. (3) They find a file group according to some leading files accessed and other files in the file group are auxiliary. While in our file group, each file has the same role as others. (4) Overlapping among file groups are permitted by current grouping methods, while it is not allowed in EEFS because overlapping induces multi-copies and consistency-control overhead. This is why we emphasize the selection of components in the file relation graph. (5) Specific accuracy [Amer et al., 2002b] is more important than general accuracy in our grouping strategy.

Interaction between GFS and UFS

File Migration and Group Shuffle EEFS maintains only one copy for a single file, either stored on UFS or on GFS. Since GFS aims to optimize file I/O access pattern and work only for those files with inside group access affinity, file migration between UFS and GFS is unavoidable. Such a single-copy policy simplifies the design and implementation issues in EEFS.

Files may migrate from UFS to GFS only under one condition, group shuffle. A group shuffle will invoke a grouping procedure based on component-grouping algorithm. For new generated groups, some group members may not be in GFS and need to migrate from UFS to GFS at that time. GFS copies these files along with their file inodes from UFS, constructs corresponding entries in file lookup table and group table and then removes the file from UFS. Notice directory is a special type of file that are always maintained in UFS. When UFS removes the file to be migrated to GFS, it reclaims its file blocks and inode space but still keeps the file's entry in its parent directory (e.g the file inode number is changed to a negative number to indicate the file is in GFS). Such a design provides fast directory traversal operations.

Files may migrate from GFS to UFS under two situations: (1) files become too big (e.g >64 KB), (2) after a group shuffle, some files are no longer in any GFS group. To migrate a file from GFS to UFS, the following operations are executed: copy the file to its original directory of UFS, create a file inode by copying the file metadata in GFS and then remove this file from GFS.

Crash Recovery and File Consistency GFS keeps critical metadata in RAM in order to achieve high I/O performance. It is important to safely save file system metadata, such as group table and file lookup table, along with new data writes onto disk. The reliability problem can be addressed by saving the above important system data structures in the non-volatile memory space. While for low-cost design without non-volatile memory or storage devices installed, similar to LFS, GFS will employ a two-pronged approach to deliver a fast recovery by writing metadata updates periodically to the reserved checkpoints. Checkpoints are used to maintain file system consistency while the roll-forward method is adopted to recover information written since the last checkpoint. The append-only write policy protects old data from being invalidated before new data reach disk. Compared to the crash recovery in LFS, GFS employs a *lighter weight checkpoint* that has much

less metadata to write because GFS manages small files on a fixed-size segment storage unit and stores all file metadata together on some fixed-address disk clusters.

For the GFS files, since their directory inodes are stored and managed by UFS, EEFS needs to coordinate the sequence of related file metadata updates in both GFS and UFS. EEFS applies a journaling technique to guarantee file system integrity for EEFS. All related file metadata updates are written to a log for EEFS crash recovery. EEFS has a faster crash recovery compared to existing UNIX-like file systems because its GFS crash recovery is faster and the load in UFS crash recovery is largely reduced for only a portion of EEFS files.

Experimental Methodology

Figure 4 shows the experimental framework: feed file system traces into the file system simulator, collect all output disk requests and feed into a validated disk power consumption simulator, and finally collect results of the energy consumption.

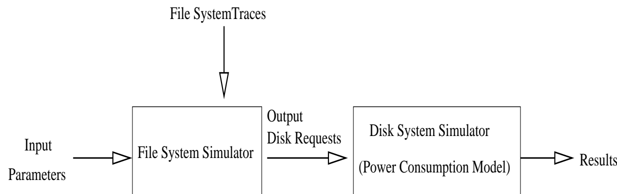

Figure 4: *Illustration of the Experimental Framework*

Four file system traces were used in our simulation experiments. To test how EEFS works under different environment and conduct a comprehensive evaluation, we selected traces from different applications. Two of them were originally collected from University of California, Berkeley, called INS and RES [Roselli et al., 2000].[2] To see how EEFS works well under different I/O intension workloads, we generated two synthetic traces from HP file system traces [Riedel et al., 2002] which was collected in late 2000. One is a light file I/O trace (Light I/O for short), which is produced by selecting only 2 users I/O requests. The other one is a heavy file I/O trace (Heavy I/O for short), which is generated by merging 50 users I/O requests. The characteristics of all traces are described in Table 1.

We develop two simulators, a Fast File System (FFS) simulator that is ported from the BSD UNIX distribution as the baseline system, and a complete EEFS simulator by composing a GFS module with a UFS module and coding the interaction between GFS and UFS. FFS simulator was validated in our TFS [Wang and Li, 2003] project. EEFS simulators implement all data structures and algorithms discussed in Section 2. Both simulators are built on top of the disksim [Ganger and Patt, 1998], which is a widely used, comprehensive disk simulator. To measure the energy consumption, we adopted a disk power modeling simulator Dempsy developed by Princeton University [Zedlewski et al., 2003]. We used a 5 GB Toshiba II PC Card HDD as the hard disk drive in all experiments.

[2] INS is a collection from a group consisting of 20 machines located in labs for undergraduate classes. RES was attained from 13 desktop machines of a research group. INS and RES were recorded over 112 days form September 1996 to December 1996. We randomly chose one day's trace from INS and RES collections, with a November 2nd, 1996 for INS and November 1st, 1996 for RES.

Experiment Results and Analysis

In this section we compare both energy consumption and I/O performance of EEFS with that of FFS, a baseline file system. To justify the grouping efficiency, we use the accumulative grouped file access rate to see how many files get accessed after the group-based prefetch. For energy measurement metrics, we collect the energy consumption during different active states (disk Seek, Write, Read, Rotation) and the idle state. For the performance comparison, we use the average response time per file request as the metric.

Grouping Efficiency

Figure 5 illustrates the percentage of files in the trace that are accessed after their groups have been accessed in the first time. Notice the results are collected between two continuous group shuffles. Both group shuffles are randomly selected. From Figure 5 we can see, among all files from the same group, 49.3% files are accessed after the first file of the group is visited in INS, while the number in RES is 72.3%. There are 85.9% of files in INS and 75.8% of files in RES being accessed within five minutes since the group gets visited at the first time. The results tell that our grouping algorithm works extremely well, when any one file of a group gets accessed, most of other files in the same group are accessed very soon.

When EEFS conducts grouping, only those prediction files are selected, and as a result, we see only 70% to 89% of all file accesses between two group shuffles. Through experiments, we found the number of files in most groups (group size for short) is between three and eight which explains why we set G_{min} and G_{max} to be three and eight respectively.

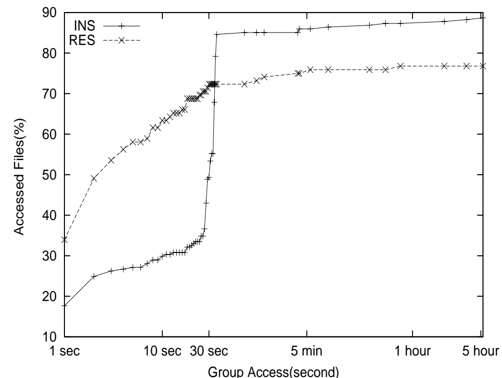

Figure 5: Group Access

It is also useful to see how many files changed per group after each group shuffle because we would like

Table 1: The Characteristics of Four Traces

Features	INS	RES	Light I/O	Heavy I/O
Total Requests	5,583,587	2,917,247	86,154	8,400,764
Data Read (MB)	3,052	1,701	101	4,732
Data Write (MB)	543	455	18	965
Distinct Bytes (MB)	132	133	44	202
R/W ratio	5.6	3.7	5.6	5.1
File Size (<16KB)	82.8%	63.0%	81.5%	76.9%
File Size (16KB+)	17.2%	34.0%	18.5%	23.1%

to construct stable groups without frequent file member changes so that the group shuffle overhead is kept at a modest level. Table 2 shows the result of how groups change over the future group shuffle periods. It shows the results when doing group shuffle after every 100,000 requests and every 500,000 requests, which correspond to around one and four hours in INS and RES traces respectively. We randomly chose two immediate group shuffles, shuffle 1 and shuffle 2. We collected the number of groups (see the 2nd and 3th column in Table 2) generated at group shuffle 1 and group shuffle 2 respectively. The 4th column presents the number of groups that see little changes (stable). The average number of group size is shown in 5th column. The 6th column tells the percentage of files present at shuffle 1 but disappearing at shuffle 2, namely those files being migrated from GFS to UFS. The 7th column tells the percentage of files not present at shuffle 1 but appearing at shuffle 2, namely those new files being migrated from UFS to GFS. The results of this table indicate that 1) most produced groups are quite stable, and 2) only a small number of files need to migrate between GFS and UFS at each group shuffle.

Another fact we find is that, in both traces, the number of grouped files in total is not affected by the interval time between every two group shuffles. For example, the number of grouped files is around 200 to 250 no matter the group shuffle interval time is either every 100,000 requests or every 500,000 requests. The reason is that, the file access patterns of users usually do not change a lot often within an one-day period.

Energy Efficiency

Table 3 compares the energy consumption of EEFS to that of FFS. Through this table we can see that the total number of disk requests in EEFS is only around one fourth to that in FFS. The reason is that EEFS saves multiple small files together on the disk, and thus combines many small file I/O requests into few large file I/O requests by doing group prefetch and batching writes for both file data and file metadata. The results shown from 4th column to 9th column are calculated as the percent of energy consumption difference comparing EEFS to FFS. Under two real-world traces, EEFS saves the energy consumption 46.5% for RES and 50.2% for INS. Given two synthetic traces, the saving ratio still reaches 33.4% for light I/O and 12.3% for heavy I/O. From the breakdown of energy consumption at each state, we can see the reduced disk rotation and seek times of EEFS make all the contributions for energy savings. The reason why EEFS achieves the biggest energy savings under INS is that, EEFS stretches the idle period for 25.5%, which contributes a significant energy saving for disk drives. For I/O reads and writes, EEFS consumes a little more energy from 11.6% to 18.6% compared to FFS mainly because of its large I/O latency. The group shuffle process also contributes part of energy consumption of I/O reads and writes.

It may be noted that, under the light file I/O workload, EEFS saves less energy than under INS and RES. The reason is that less benefits can be obtained from group optimization in GFS among the infrequent file I/O requests. Given a heavy file I/O trace, the file I/O requests are intensive, and thus the disk drive is most likely kept always busy and does not have a long idle time. Even in such a case, EEFS still can save 12% energy compared to FFS.

Performance Improvement

The comparison of the file I/O performance in terms of average response time per file request is shown in Figure 6 (LFIO means light file I/O trace and HFIO means heavy file I/O trace). From these two pictures we can see that EEFS achieves better write and read performance in INS and RES by up to 21% than that of FFS. This can be explained by the hit rate of file system buffer cache. Given a 128MB file system buffer cache, the hit rates of EEFS, 84.30% for RES and 77.43% for INS, are higher than those of FFS, 80.20% for RES and 69.76% for INS.

In EEFS, one disk write can generate less than one disk write because several disk write might be replaced by one large group write. However, this does not make much contribution in light I/O workload. Under a light I/O workload, an entire group may have to be updated only because one or few files from this group have been changed, and therefore the write response time might be stretched. While the read response time of EEFS may still be better than that of FFS if the group prefetch strategy works well. Figure 6 confirms our expectations. In a heavy file I/O workload, FFS shows a better average read performance than EEFS. Because in EEFS, it executes a group shuffle process every 100,000 requests. Given the intensive requests in a heavy file I/O workload, EEFS runs a group shuffle more frequently than it does in a light file I/O workload. In addition, user's file

Table 2: Group stability

Shuffle Interval	Trace	# of groups at shuffle 1	# of groups at shuffle 2	# of stable groups	Avg. group size	files disappear	new files
100,000	INS	28	26	17	7	14.6%	16.7%
100,000	RES	28	27	15	5	11.2%	9.5%
500,000	INS	27	28	12	9	17.7%	13.5%
500,000	RES	27	37	11	6	8.3%	21.6%

Table 3: Energy Consumption Comparison

Trace	# of disk reqs in FFS	# of disk reqs in EEFS	Read	Write	Rotation	Seek	Idle	Total
INS	482,947	151,901	+12.6%	+11.8%	-78.5%	-78.1%	+25.5%	-50.2%
RES	252,325	79,107	+14.4%	+17.4%	-75.7%	-75.6%	+21.9%	-46.5%
Light I/O	7,072	2,217	+11.6%	+12.0%	-56.5%	-42.2%	+15.5%	-33.4%
Heavy I/O	630,057	415,026	+18.6%	+12.0%	-39.7%	-40.6%	+7.7%	-12.3%

access pattern changes faster, which may also increase the chance of file migration between GFS and UFS.

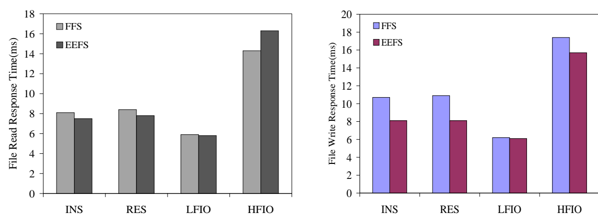

Figure 6: Average File Read and Write Response Time

Related Work

Energy management schemes can work at different levels. At disk level, numerous scheduling algorithms using either a static or adaptive time-out policy to predict future requests have been developed based on different statistical models of disk requests [Douglis et al., 1995, Greenawalt, 1994, Helmbold et al., 1996]. Most of these policies improve energy efficiency at the cost of I/O performance degradation. ECOSystem [Zeng et al., 2002] and Coop-I/O [Weiel et al., 2002] are energy-efficient at the OS level. The drawbacks of their approaches are the complexity of system implementation and poor compatibility. Lu et al. [Lu and Micheli, 1999] presented a design that allowed the applications be involved in energy management. EEFS may use this method to help the decision-make process of group shuffle more accurate. Recently, some energy-efficient schemes [Zhu et al., 2004, Gurumurthi et al., 2003] have been presented based on fictional multi-speed disks.

Grouping is a common approach used to improve file system performance. Griffioen and Appleton [Griffioen and Appleton, 1994] presented a file prefetching scheme relying on graph-based relationships. By comparing the predictive performance of the Last Successor predictor [Lei and Duchamp, 1997] to that of Griffioen and appleton's scheme, Kroeger and Long [Kroeger and Long, 1999]introduced more effective schemes based on context modeling and data compression. Recently more successor predictors are introduced such as Noah and Recent Popularity [Amer et al., 2002b] which motivated us to develop a new component-based grouping algorithm for EEFS by considering both prefetch and disk layout.

EEFS shares some insights with TFS [Wang and Li, 2003], a lightweight temporal file system to boost I/O performance for Internet servers. But EEFS revolutionizes TFS in two factors: 1) TFS works only for the read-only file system workloads, but EEFS is a general-purpose kernel-level file system that deals with both read and write; and 2) TFS works on performance only while EEFS emphasizes both energy efficiency and performance.

Conclusions and Future Work

This paper presents a novel energy efficient file system called EEFS which realizes an energy efficiency by translating its I/O performance gains, namely the reduced I/Os to a more bursty arrival I/O request pattern. EEFS works in two modules: a normal Unix-like File System mode (UFS) and a group-structured file system (GFS) mode. EEFS incorporates a new grouping policy that can capture *group access locality* used to construct a group of files and migrate between UFS and GFS. Comprehensive experiments show that EEFS can realize a large energy savings of up to 50% compared to general-purpose UNIX file system while simultaneously achieving an up to 21% better file I/O performance. In the future, we will implement EEFS and then compare it with other modern file systems, such as XFS and ReiserFS, by real evaluation instead of simulations.

References

[Amer et al., 2002a] Amer, A., Long, D. D. E., and Burns, R. C. (2002a). Group-based management of distributed file caches. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, pages 525–533, Vienna, Austria.

[Amer et al., 2002b] Amer, A., Long, D. D. E., Paris, J.-F., and Burns, R. C. (2002b). File access prediction with adjustable accuracy. In *Proceedings of the International Performance Conference on Computers and Communi-cation (IPCCC 2002)*, Phoenix, Arizona.

[Douglis et al., 1995] Douglis, F., Krishnan, P., and Bershad, B. (1995). Adaptive disk spin-down policies for mobile computers. In *Proceedings of the 2nd Symposium on Mobile and Location-Independent Computing (MLICS'94)*, pages 121–137, Berkeley, CA, USA. USENIX Association.

[Ganger and Patt, 1998] Ganger, G. R. and Patt, Y. N. (1998). Using system-level models to evaluate I/O subsystem designs. *IEEE Transactions on Computers*, 47(6):667–678.

[Greenawalt, 1994] Greenawalt, P. (1994). Modeling power management for hard disks. In *Proceedings of the Symposium on Modeling and Simulation of Computer and Telecommunication Systems(MASCOTS 1994)*, pages 62–66.

[Griffioen and Appleton, 1994] Griffioen, J. and Appleton, R. (1994). Reducing file system latency using a predictive approach. In *Proceedings of USENIX Summer Technical Conference*, pages 197–207, Boston, Massachusetts.

[Gurumurthi et al., 2003] Gurumurthi, S., Sivasubramaniam, A., Kandemir, M., and Franke, H. (2003). DRPM: dynamic speed control for power management in server class disks. In DeGroot, D., editor, *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA-03)*, volume 31, 2 of *Computer Architecture News*, pages 169–181, New York. ACM Press.

[Helmbold et al., 1996] Helmbold, D. P., Long, D. D. E., and Sherrod, B. (1996). A dynamic disk spin-down technique for mobile computing. In *Mobile Computing and Networking*, pages 130–142.

[Kroeger and Long, 1999] Kroeger, T. and Long, D. (1999). The case for efficient file access pattern modeling. In IEEE, editor, *The Seventh Workshop on Hot Topics in Operating Systems: [HotOS-VII]: 29–30 March 1999, Rio Rico, Arizona*, pages 14–19, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA. IEEE Computer Society Press.

[Lei and Duchamp, 1997] Lei, H. and Duchamp, D. (1997). An analytical approach to file prefetching. In *Proceedings of the USENIX Annual Technical Conference (USENIX-97)*, pages 275–288, Berkeley. Usenix Association.

[Lu and Micheli, 1999] Lu, Y.-H. and Micheli, G. D. (1999). Adaptive hard disk power management on personal computers. In *Proceedings of the IEEE Great Lakes Symposium*, pages 50–53.

[McKusick et al., 1984] McKusick, M. K., Joy, W. N., Leffler, S. J., and Fabry, R. S. (1984). A fast file system for UNIX. *Computer Systems*, 2(3):181–197.

[Papathanasiou and Scott, 2004] Papathanasiou, A. E. and Scott, M. L. (2004). Energy efficient prefetching and caching. In *Proceedings of USENIX Annual Technical Conference.*

[Riedel et al., 2002] Riedel, E., Kallahalla, M., and Swaminathan, R. (2002). A framework for evaluating storage system security. In *Proceedings of the FAST '02 Conference on File and Storage Technologies (FAST-02)*, pages 15–30, Berkeley, CA. USENIX Association.

[Roselli et al., 2000] Roselli, D., Lorch, J. R., and Anderson, T. E. (2000). A comparison of file system workloads. In *Proceedings of the 2000 USENIX Conference*, pages 41–54, San Diego, California.

[Rosenblum and Ousterhout, 1992] Rosenblum, M. and Ousterhout, J. K. (1992). The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26 – 52.

[Shriver et al., 2001] Shriver, E., Gabber, E., Huang, L., and Stein, C. A. (2001). Storage management for web proxies. In *Proceedings of the 2001 USENIX Annual Technical Conference (USENIX-01)*, pages 203–216, Berkeley, CA. The USENIX Association.

[Smith and Seltzer, 1996] Smith, K. A. and Seltzer, M. (1996). A comparison of FFS disk allocation policies. In *Proceedings of the USENIX Annual Technical Conference*, pages 15–26, Berkeley. Usenix Association.

[Wang and Li, 2003] Wang, J. and Li, D. (2003). A lightweight, temporary file system for large-scale Web servers. In *Proceedings of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems(MASCOTS-03)*, pages 96–103, Florida.

[Wang et al., 1999] Wang, R. Y., Anderson, T. E., and Patterson, D. A. (1999). Virtual log based file systems for a programmable disk. In *Proceedings of the 3rd Symposium on Operating Systans Design and Implementation (OSDI-99)*, pages 29–44, Berkeley, CA. Usenix Association.

[Weiel et al., 2002] Weiel, A., Beutel, B., and Bellosa, F. (2002). Cooperative io - a novel io semantics for energy-aware applications. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA.

[Zedlewski et al., 2003] Zedlewski, J., Sobti, S., Garg, N., Zheng, F., Krishnamurthy, A., and Wang, R. (2003). Modeling hard-disk power consumption. In *Proceedings of the Second Conference on File and Storage Technologies FAST'03*, pages 217–230.

[Zeng et al., 2002] Zeng, H., Ellis, C. S., Lebeck, A. R., and Vahdat, A. (2002). ECOSystem: managing energy as a first class operating system resource. *ACM SIGPLAN Notices*, 37(10):123–132.

[Zheng et al., 2003] Zheng, F., Garg, N., Sobti, S., Zhang, C., Joseph, R., Krishnamurthy, A., and Wang, R. (2003). Considering the energy consumption of mobile storage alternatives. In *Proceedings of the 11th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, Orlando, Florida.

[Zhu et al., 2004] Zhu, Q., David, F. M., Devaraj, C. F., Li, Z., Zhou, Y., and Cao, P. (2004). Reducing energy consumption of disk storage using power-aware cache management. In *Tenth International Symposium on High Performance Computer Architecture (HPCA-10)*, Madrid, Spain.