

DAFT: Disk Geometry-Aware File System Traversal

Fanglu Guo Tzi-cker Chiueh
Symantec Research Labs
Culver City, CA 90230

Abstract—Bulk file access is a read access to a large number of files in a file system. Example applications that use bulk file access extensively are anti-virus (AV) scanner, file-level data back-up agent, file system defragmentation tool, etc. This paper describes the design, implementation, and evaluation of an optimization to modern file systems that is designed to improve the read efficiency of bulk file accesses. The resulting scheme, called DAFT (Disk geometry-Aware File system Traversal), provides a bulk file access application with individual files while fetching these files into memory in a way that respects the disk geometry and thus is as efficient as it can be. We have successfully implemented a fully operational DAFT prototype, and tested it with commercial AV scanners and data back-up agents. Empirical measurements on this prototype demonstrate that it can reduce the elapsed time of enumerating all files in a file system by a factor of 5 to 15 for both fragmented and non-fragmented file systems on fast and slow disks.

I. INTRODUCTION

File system provides the abstractions of file, file attribute, and directory/folder on top of raw disk blocks to simplify the design and implementation of applications accessing data stored on disk. However, this simplification sometimes carries a significant performance price, because the existence of a file system layer insulates these applications from the disk and prevents them from accessing their disk-resident data in the most efficient way. For example, many applications require the ability to access every file in a file system or in a selective set of directories in a file system. A natural way to develop such applications is to go through every file in the target list, open it, and read in its contents. Depending on the order in which the files in the target list are traversed, the end-to-end elapsed time required to access them could vary significantly. Conventional file systems are not designed to optimize for such *bulk file access* pattern because (1) they don't provide an API for applications to express their intentions to access a target file set, which could be a set of directories or an entire file system, and (2) they lack a multi-file disk scheduling mechanism to retrieve the requested file blocks in a way most friendly to the disk geometry.

One example application with bulk file access pattern is a file-level data back-up application that retrieves individual files on a host's file system and transfers them to a back-up media server for persistent storage. To overcome the poor performance problem of file-level data back-up, an alternative back-up method, block-level data back-up, is invented. Block-level data back-up directly reads the disk volume underlying a file system and bypasses the file system layer completely. Because this approach accesses disk volumes sequentially, its performance is excellent. However, because block-level data

back-up cannot capture per-file information, it cannot provide several important benefits associated with file-level data back-up, such as more restore flexibility [1], better deduplication efficiency [2], etc.

As a result, despite being slower, file-level data back-up is still commonly supported in many commercial back-up applications, which in turn require fast bulk file access.

Another example application with bulk file access pattern is the AV (Anti-Virus) scanner. When an AV scanner performs a full-system scan, it needs to retrieve each file and scans the file content to determine if a file is infected. Because AV scanners need to parse individual files, e.g., PE executable file, MPEG file, etc., it is impossible for them to operate directly on a raw disk volume.

This paper describes the design, implementation and evaluation of a file system extension for Microsoft's NTFS file system called DAFT (Disk geometry-Aware File system Traversal), which is specifically designed to speed up bulk file accesses. The goal of the DAFT project is to develop a bulk file access mechanism that provides the same flexibility of file-level bulk file access and the performance benefit of block-level access, thus achieving the best of both worlds. Through a bulk file access API, an application specifies a set of files that it needs. Then DAFT delivers each individual file to the application by reading the file's content directly from the disk. As a result, DAFT is able to make the best use of the raw disk transfer bandwidth while returning the target file set on a file by file basis.

More concretely, given a target file set, DAFT first accesses the file system's metadata to identify the disk location of every file fragment to be accessed, sorts their disk locations in an increasing order, and fetches the requested file fragments according to this order. Therefore, only one sequential pass through the disk is needed to retrieve the target file set. As file fragments are brought into memory, DAFT associates them with their corresponding files, completes the assembly of a file as soon as all of its file fragments become available in memory, and delivers each completely assembled file to the requesting application. Consequently, DAFT is able to enumerate individual files in an application-specified file set with the same efficiency as block-level bulk file access. At the same time, application can still operate on each individual file for the purpose of AV scanning or file-level data back-up.

We have successfully implemented the first DAFT prototype on Microsoft's NTFS file system. Compared with the original version of the NTFS file system, DAFT is able to improve the sustained throughput of bulk file access by a factor of 5 to

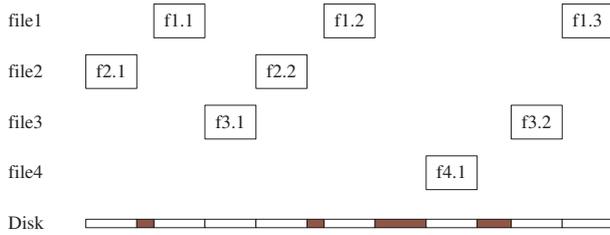


Fig. 1. An example to show how DAFT works when the file is fragmented. Each file may have several fragments and the fragment location on the figure represents where the fragments are physically on the disk. The dark areas on the disk represent blocks not processed by DAFT.

15, depending on the degree of fragmentation in the test file systems and the speed of the underlying disks.

II. DAFT DESIGN

In this section, we describe the design of DAFT in the context of Microsoft’s NTFS file system.

A. Bulk File Access API

DAFT provides a new API, `BulkFileRead(FileList, CallbackFunction)`, for a bulk file access application to specify a list of files or directories that it intends to access, and a call-back function that DAFT should call when it brings each requested file in the list into memory. `FileList` could be a list of directories and represents all the files recursively under these directories. Through this API, a bulk file access application indicates to DAFT that the order in which files are brought into memory is unimportant and is thus left to DAFT for performance optimization. In addition, this API dictates that the control flow of the calling application be structured as *data-driven*, i.e., the processing logic associated with a file, e.g., AV scanning or data back-up, is triggered only upon the file’s arrival at memory.

B. How DAFT Works

DAFT divides the file access to two phases. In the first phase, all the files’ metadata are brought into memory and analyzed. In the second phase, DAFT accesses the files’ data according to the analysis result from the metadata. In this two-stage approach, both accesses to file metadata and file data are sequential.

In the first phase, DAFT determines the disk location of every fragment of every file in the target file set. On NTFS, a file’s fragment information is described by one or more MFT records. An NTFS file system’s MFT records are stored in a special file called `$MFT`, which is usually stored in a reserved disk region and is itself rarely fragmented. DAFT reads the `$MFT` file into memory, parses the MFT records, and puts the file fragment location information to a fragment list.

The total memory requirement for file fragment disk location information is relatively modest, because most files have a small number of fragments. It costs DAFT 20 bytes to maintain each file fragment’s disk location: 8 bytes for the fragment’s

location, 4 bytes for the fragment’s length, 4 bytes for the fragment’s offset in the file, and 4 bytes for pointing back to the file to which the fragment belongs. For a 150,000-file NTFS system, the total amount of memory required to record its file fragments’ disk location information is about 10 Mbytes, which suggests the average number of file fragments per file is smaller than 4.

When the number of files in `FileList` is so large that the file fragment information cannot be fitted in memory, we can divide the files to several small subsets and process one subset at a time.

In the second phase, DAFT first sorts the fragments of the files in the target set according to their disk location, and then read them into memory in the sorted order. Figure 1 shows the physical location of several file fragments on the disk. DAFT first reads in the first fragment of file2, fragment f2.1, and then fragment f1.1. DAFT ignores the file boundary, and does not need to fetch a file completely before reading in fragments of another file. When fragments are sufficiently close to each other, they can be merged into one disk read request. For example, f2.1 and f1.1 in Figure 1 can be fetched in one physical disk access because they are sufficiently close to each other. The version of DAFT with this optimization is called **DAFTM** (DAFT with merging adjacent fragments), and greatly improves DAFT’s disk access efficiency, especially on disks that do not support automatic coalescing of physical disk access requests.

When the first fragment of a file is read into memory, DAFT allocates memory for the entire file, not just for that file fragment. When subsequent fragments of the file are read into memory, they are stored in the corresponding locations of the file’s allocated memory. When all fragments of a file are read to memory, DAFT invokes the application-specific call-back function `CallbackFunction` and frees the memory occupied by the file. DAFT continues to fetch all the file fragments until all fragments in the sorted fragment list are brought into memory.

To summarize, the key ideas in DAFT that make it more efficient for bulk file access are

- DAFT ignores the file and file fragment boundary when accessing the target file set of a bulk file access. This allows DAFT to use large disk reads in most cases, and reduces the seek distance between consecutive disk reads.
- After bringing file fragments into memory through a raw disk interface, DAFT transparently assembles file fragments into complete files, essentially converting inefficient random disk accesses to more efficient random memory accesses. This allows DAFT to present to bulk file access applications with individual files.

The basic idea behind DAFT could be applied and ported to arbitrary file systems, as long as it is possible to access the file system’s metadata that contains the mapping from a file to the disk locations of its file fragments. In the case of Unix-like systems, this file mapping information is stored in an inode table. In the case of Microsoft’s NTFS file system, it is stored in the Master File Table (MFT). DAFT can easily

be implemented on any files system as user level library and benefit any bulk file access applications. Furthermore, DAFT doesn't need to defragment the disk or have other run time component. It doesn't introduce any overhead to a system but can provide improvement at the spot where it is needed.

C. Re-assembly Buffer Memory Management

Although DAFT drastically improves the throughput of bulk file access, it achieves this feat at the expense of larger memory requirement. In the process of fetching file fragments according to the sorted order, DAFT needs to buffer in memory those files that are waiting to be completed. We define the range between the first and the last physical fragment that belong to a file as the file's **physical range**. For example, the physical range of file1 in Figure 1 is the range between f1.1 and f1.3. The set of files that need to be buffered in memory when the i th file system fragment is fetched will be all the files whose physical range covers the i th file system fragment. For example, when DAFT reads fragment f2.2, file1, file2, and file3 need to be buffered in memory because their physical range covers fragment f2.2.

Accordingly, the buffer memory requirement when the i th file system fragment is fetched is the sum of the sizes of all those files whose physical range covers it, and the memory requirement of a DAFT run will be the maximum of the buffer memory requirements across all file fragments specified in the target file set.

Because the amount of buffer memory available on machines where DAFT runs is typically limited, we need to adapt DAFT to work with a fixed amount of buffer amount, even at the expense of some performance degradation. More concretely, given a fixed buffer memory budget M , DAFT partitions the file fragment list into a number of sub-lists so that each sub-list can be fetched into memory via one sequential pass through the disk, the buffer memory requirement of each such pass is lower than M . Intuitively, the fewer the number of passes required, the lower the total elapsed time. Therefore the main goal for DAFT's buffer memory management design is to minimize the total number of sequential disk passes needed in a bulk file access.

This problem can be mapped exactly to the Dynamic Bin Packing (DBP) problem [3], whose goal is to pack a sequence of items that arrive and depart at arbitrary times into fixed-sized bins so that the total number of bins required is minimized. Here a bin of size M corresponds to a DAFT pass through the disk with a buffer memory budget of M , and an item corresponds to a file whose life time is the file's physical range and whose size is the file's size. The Dynamic Bin Packing problem is a well researched problem and Coffman, Garey and Johnson [3] proved that the lower bound on the competitive ratio of any on-line DBP algorithm is 2.3881 and the first fit heuristic is very effective and is 2.788-competitive.

Given a target file set, the **first fit** algorithm *statically* considers the files in an order according to the disk location of their first fragment and partitions them into multiple bins (passes) as follows. When the first fragment of a file is

encountered, the algorithm checks the first bin to see if it has enough free space to hold the file. If it has, the file is put to the first bin. Otherwise, the algorithm checks subsequent bins the same way and creates a new bin if no existing bin can hold the file. After the last fragment of a file is fetched, the algorithm removes the file from its bin and frees up the memory it previously occupied. Files with disjoint life spans could take up the same buffer memory area even when they are assigned to the same pass. As an example in Figure 1, when f2.1, f1.1, f3.1, and f4.1 are read, the first bin is always checked first. Assume f3.1 cannot be fit in the first bin, file3 will be put into other bins. When f4.1 is about to be read, since file2 is completed, file4 may be put to the first bin again if this bin has enough space to hold file4 after the memory taken by file2 is freed up.

Surprisingly our experiments show that minimizing the number of sequential disk passes, as is done in the first fit algorithm, may not result in the best end-to-end throughput. Instead, a simple greedy algorithm, **on-demand cleaning**, actually performs better and processes files in the target file set in the same order as follows. Before the first fragment of a file is read, the algorithm checks if there is enough space in the buffer memory to hold the entire file. If yes, the algorithm allocates the memory required for the file; otherwise, it enters a clean-up mode. In the clean-up mode, the algorithm identifies and sorts the set of file fragments that need to be fetched to complete the currently pending files. Then it reads in all the remaining fragments of the pending files in the sorted order, and completes all the pending files. No memory limit violation is possible in the clean-up mode because no new files are admitted. After completing all the pending files, the algorithm returns to the normal mode and proceeds from the last fetched fragment that triggers the on-demand cleaning which it just completes.

Although on-demand cleaning appears greedy, it actually corresponds to a solution to a batched version of the dynamic bin packing problem, which imposes a minimal size limit on the item(s) that are assigned to a bin. That is, whenever a bin accepts new items, the algorithm assigns a batch of items at a time rather than one file at a time, and the total size of these items must be greater than or equal to a threshold S . In the context of DAFT, if S is too small, physically adjacent files may be assigned to different bins/passes, and each pass may thus incur additional disk seeks; if S is too large, having to process each batch of files separately decreases the probability of picking up all relevant file fragments in a disk area as the disk head sweeps through the area, and in turn may require more disk passes than necessary. On-demand cleaning requires S to be the size of the entire buffer memory, whereas file-by-file dynamic bin packing does not have any such constraint, i.e. $S = 0$.

D. Special Files

There are some special files that DAFT currently does not handle. First, DAFT treats large files specially by separately bringing them into memory and handing them to the applica-

tion. The rationale of this design is the additional performance gain from including large files into the standard DAFT flow does not justify the extra memory requirement they introduce. Second, for files compressed or encrypted by the file system, DAFT cannot decompress or decrypt them properly before delivering them to the application, because DAFT retrieve them from disk without going through the file system. This may be acceptable for some applications, such as data backup, but is not acceptable for other applications, such as AV scanning, because AV scanners cannot make sense of a file if it is in compressed or encrypted form.

One solution to this problem is to access these files through the standard file system API. In the first phase, if the file size is over a certain limit, or the file is compressed or encrypted, the file is put into a special file list. The file fragment information will not be collected. In the second phase, after all fragments are processed, the files in the special file list are handed over the application one by one with a flag to notify the application that the file content cannot be read directly from memory. Instead, the file content should be read from disk with standard file system API. Fortunately, large files or compressed/encrypted files are only a small portion of the files on a typical file system. As long as DAFT can handle over 90% of the files, it is expected to produce significant performance improvement when compared with existing file systems without any bulk file access optimization.

E. Consistency Issue

Because DAFT bypasses the file system layer, the metadata and data it reads may not be up to date or consistent with the file system's current image. For example, if a file is being opened for write by an application at the time when DAFT reads it from the disk, the version that DAFT brings into memory may not be up to date with respect to the version in the file system cache, let alone that in the application's address space. As another example, when DAFT reads the file system metadata to build up the requested file fragment list, it does not go through the file system cache, and therefore may miss some pending file system metadata updates that should be but have not been committed to disk.

Fortunately, this inconsistency problem has already been addressed in the back-up practice. Windows Volume Shadow Copy Service (VSS) [4] provides a framework for applications to register application-specific quiesce functions. Before a back-up agent takes a back-up, it asks VSS to create a snapshot, which is a point-in-time image of the underlying disk volume. Before creating the snapshot, VSS automatically quiesces the applications through application-provided quiesce call-back functions and flushes the file system cache. The quiesce step forces all applications to flush their cache and stop modifying files they open. Because a snapshot captures an instant image, there is no consistency issue at all when DAFT works with an VSS snapshot.

Yet another solution to the consistency issue is to run DAFT through the file system cache. In this solution, DAFT accesses the requested file fragments and caches them at the

disk volume level, and then reads each requested file through the standard file system interface when it is fully assembled at the disk volume level. By retrieving each requested file through the file system layer, DAFT effectively masks any metadata and data inconsistency problems. Because everything DAFT accesses goes through the file system, this version of DAFT can seamlessly work with files that are encrypted or compressed by the file system layer without any modification. Finally, because in most cases DAFT already caches a requested file's fragments at the disk volume level, most of its file system reads are serviced from this cache and do not require any physical disk accesses.

F. Other Miscellaneous Issues

Disk geometry-aware access. One concern regarding disk geometry is that it is hard to get the exact disk geometry. Spare sectors are provisioned to replace defective sectors. The RAID technology can map a volume to multiple disks. Fortunately DAFT doesn't need to have accurate disk geometry information. The only assumption is that sequential disk access will be much faster than random access. By sorting file fragments and accessing them sequentially, we can get more throughput from a disk than access the file fragments randomly. This is true most of the time even if RAID mapping or sector remapping renders the logical volume view drastically different from the physical disk view.

Other concurrent accesses to the disk. DAFT improve performance by accessing the volume sequentially. When there are other concurrent accesses to the disk, DAFT's sequential access may be interrupted by those requests. This might shrink the performance gain of DAFT. But DAFT is still much better than standard file access in that when a disk serves DAFT requests, they correspond to sequential accesses, which are still much more efficient than random accesses in standard file system traversal.

Asynchronous access. In theory, asynchronous file system access interface may already be good enough for file system traversal. In practice, asynchronous access interface only slightly improves the overall performance. When applications use asynchronous interface to issue multiple requests to a disk, it helps in that the disk has more freedom to schedule the requests by using some form of elevator algorithm to improve the access efficiency. This has the same effect as the bulk access feature of DAFT: when more freedom is given, DAFT gets a chance to reorder the requests and makes them more efficient. But the similarity ends here. In practice, asynchronous requests are inferior to DAFT for the following reasons. First, asynchronous accesses don't exploit file fragment boundary information. As a result, random access inside each request cannot be avoided. Second, it is difficult to issue all the requests associated with a file system traversal through the asynchronous access interface in one shot. Thus the disk scheduler may not have the broader view that DAFT has. Finally, to allow the disk scheduler to have more freedom to reorder requests, many small asynchronous access requests

need to be issued. The overhead of these system calls and asynchronous signal processing is substantial.

III. PERFORMANCE EVALUATION

A. Methodology

To evaluate the performance gain of DAFT when an application accesses a large set of files, we vary the following configuration parameters in the test runs:

- Speed of the test disk: the slow disk is a normal desktop disk with a throughput ranging from 30 MB/sec to 56 MB/sec. The fast disk has a throughput ranging from 76 MB/sec to 128 MB/sec.
- Degree of fragmentation: we used a real-world fragmented file system and a synthetic non-fragmented file system.
- Maximum file size: we varied the maximum size of the files used in the test runs.
- Buffer size: we varied the size of the memory buffer used in re-assembling file fragments into files.

We compared three different methods to access a target file set on the Windows platform. The first method, **Vanilla**, corresponds to the conventional way, which uses standard Win32 API, i.e.,

`FindFirstFile` and `FindNextFile` to traverse the directories and files, and `CreateFile` and `ReadFile` to access each file's content. The performance of the Vanilla method serves as the base case for comparison. The second method is DAFT without merging adjacent fragments (**DAFTNM**), which traverses and parses the MFT, collects and sorts the file fragments associated with the target file set, and reads these file fragments according to the resulting order. That is, even if two needed file fragments are adjacent to each other, DAFT reads each of them using a separate disk read operation. The third method is DAFT with adjacent fragments merged (**DAFTM**), which is different from the second method in that it fetches adjacent fragments in the sorted list that are sufficiently close to one another on the disk using a single disk read operation. The performance difference between these two DAFT versions lies in the tradeoff between the decrease in the number of disk read operations required and the increase in the total number of bytes retrieved from disk.

All evaluation tests are carried out on the same test computer to make it easier to compare results from different tests. The test computer used has a Xeon 2.4 GHz CPU with 1 GB memory. The specifications of the two disks used in these tests are shown in Table I. The slow disk is a normal desktop disk accessible through the Parallel ATA interface. The fast disk is a high performance SCSI disk that is connected to an SCSI adaptor through an Ultra 320 SCSI channel with maximum throughput of 320 MB/sec. The SCSI adaptor is connected to the computer's 64-bit, 133-MHz PCI-X bus. The raw disk throughputs and access times reported in Table I are measured using the program HD Tune 2.55 [5].

File Size	4 KB	16 KB	64 KB
Fast Disk	15.5	5.5	2.1
Slow Disk	6.2	3.9	2.1

TABLE II

The throughput improvement of DAFTM over the Vanilla method for the non-fragmented test file system. The improvement is more significant for small files because DAFTM is less affected by the file size than the Vanilla method.

B. Non-Fragmented File System

In this test, we evaluate the performance gain of DAFT on an ideal non-fragmented file system, in which (a) files are not fragmented, (b) files in a directory are clustered physically on disk, and (c) the order in which the file system enumerates a directory's files is the same as they are laid out on disk. Note that a file system output by a standard disk defragmentation tool is NOT an ideal non-fragmented file system because disk defragmentation typically only eliminates intra-file fragmentation (case (a) in the above), but not necessarily inter-file fragmentation (case (b) and (c) in the above). DAFT is immune to inter-file fragmentation but the Vanilla method is not. Therefore, the performance gain of DAFT over the Vanilla method is expected to be higher on a real-world defragmented file system than on an ideal non-fragmented file system.

To synthesize a non-fragmented file system, we start with an empty disk, and use a program to populate it with 1 million files in the following way. The program creates the first directory, and generates 1000 files under that directory. Then it repeats the same process to create another 999 directories one by one, each containing 1000 files. Three file sizes are used in the test runs: 4 KB, 16 KB, and 64 KB. The maximum file size is limited to 64 KB because the capacity of the fast disk is only around 70 GB. We created 1 million files on the fast disk, and then copied the fast disk's image to the slow disk using an imaging software (Symantec's Backup Exec System Restore [6]). The imaging software insures that both disks have the same file system layout.

Figure 2 shows the throughput of using the three bulk file access methods, Vanilla, DAFTNM, and DAFTM, to access the entire synthesized file system, when the file size is varied and the disk used in the tests is the fast disk or the slow disk. As expected, the performance of DAFTM is better than that of DAFTNM, which in turn is better than that of the Vanilla method.

When the file system being enumerated is an ideal non-fragmented file system, the sorted list in DAFT does not help much, and there should not be any performance difference between Vanilla and DAFTNM. However, Figure 2 shows that there is actually a substantial performance gap between Vanilla and DAFTNM, because the disk head needs to move between the MFT and the accessed files in the case of Vanilla, whereas the disk head just needs to move among the accessed files in the case of DAFTNM.

The performance gap between DAFTM and DAFTNM, on the other hand, mainly comes from higher physical disk access efficiency as a result of larger disk access request size, because both use the same sorted fragment order. For example,

	Model	Capacity (GB)	Speed (RPM)	Throughput (MB/sec)	Access Time (msec)
Fast Disk	Seagate ST373455LW	73	15000	76 to 128	5.8
Slow Disk	Maxtor 6Y160P0	163	7200	30 to 56	19.8

TABLE I
The hardware specifications of the fast and slow disks used in this study.

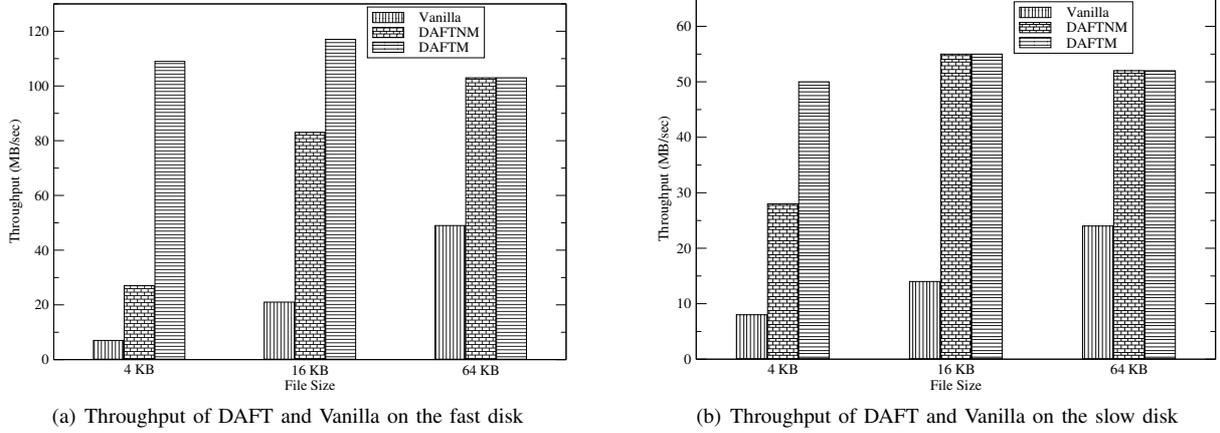


Fig. 2. The throughput of DAFTM (DAFT with adjacent fragments merging) and DAFTNM (DAFT without adjacent fragments merging) on the non-fragmented test file system. DAFTM is able to deliver close to the raw disk throughput regardless of the file size and improve over the Vanilla method by a factor of 2 to 15.

File Size Limit	128 KB	512 KB	1024 KB
File Percentage (%)	87.9	95.1	97.3
Fragment Ratio	1.11	1.21	1.28
Memory Requirement (MB)	84	387	718

TABLE III

The characteristics of the fragmented test file system and the total buffer memory requirement needed to run DAFT through different target file sets, each of which corresponds to the set of files in the fragmented test file system that are smaller than the file size limit.

File Size	128 KB	512 KB	1024 KB
Fast Disk	6.7	5.3	4.4
Slow Disk	7.0	5.7	4.3

TABLE IV

The throughput improvement of DAFTM over the Vanilla method for the fragmented test file system. The improvement is more significant for small files because DAFTM is less affected by the file size than the Vanilla method.

when the file size is 4KB, the throughput of DAFTNM is 4 times smaller than that of DAFTM on the fast disk, but is only less than twice smaller than DAFTM on the slow disk. The performance impact of disk access request size is more pronounced on the fast disk than on the slow disk, because the relatively performance overhead associated with disk head seeks and rotations is more significant for the fast disk than for the slow disk. This explains why the performance difference between DAFTNM and DAFTM decreases with the file size, and is generally more substantial on the fast disk than on the slow disk. For example, there is no difference between DAFTNM's throughput and DAFTM's throughput on the slow disk when the file size is 16KB or 64KB, or on the fast disk when the file size is 64KB.

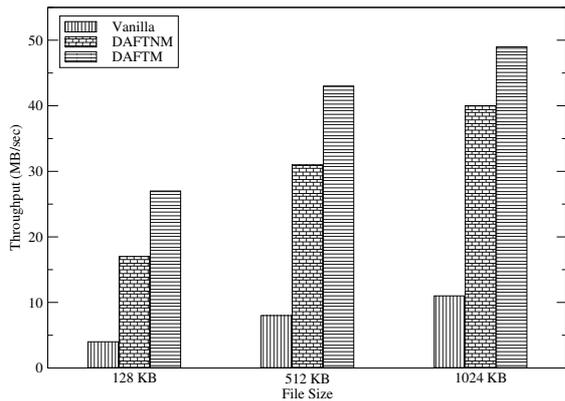
Table II shows that the performance gain of DAFTM over Vanilla under different file size and on different disks ranges from 2 to 15. This performance gain decreases when the file size increases because the relative weight of disk seek and rotational delay decreases.

C. Fragmented File System

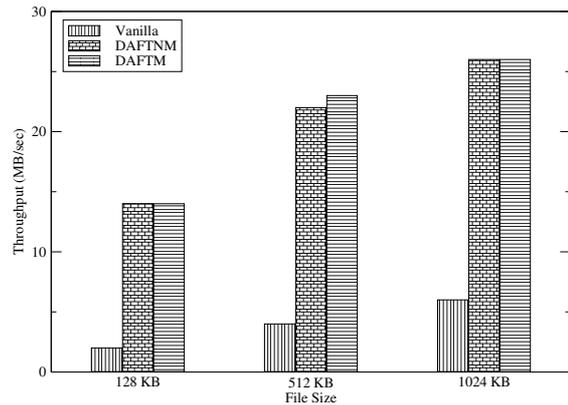
To evaluate the performance of DAFT over Vanilla for fragmented file systems, we used the imaging software (Symantec's Backup Exec System Restore [6]) to copy a fragmented disk's image to the fast disk and the slow disk to insure that the file system layouts on the two test disks is identical. We have used several fragmented disks in this study, and they all show similar results. The following presents the results of one of the fragmented disks.

To stress-test DAFT, we limited DAFT's file re-assembly buffer to 50 MB in the following experiments. We extracted three different target file sets from the test file system, each defined by the size of the largest file in the set. That is, given a file size limit X , we chose a target file set each member of which is smaller than X . Table III shows the file percentage, the fragmentation ratio, and the memory requirement for three file sizes 128KB, 512KB, and 1024KB.

The file percentage is the ratio between the number of files whose size is smaller than the specified file size limit and the total number of files in the test file system. The fragmentation ratio is the ratio between the number of file fragments and the number of files. The memory requirement is the maximum



(a) The throughput of DAFT and Vanilla on the fast disk



(b) The throughput of DAFT and Vanilla on the slow disk

Fig. 3. The throughput of DAFTM (DAFT with adjacent fragments merging) and DAFTNM (DAFT without adjacent fragments merging) on the fragmented test file system. DAFTM is able to improve the throughput of Vanilla by a factor of 4 to 7 even when the amount of buffer memory available for re-assembling pending files is only 50 MB.

amount of memory that DAFT needs to buffer and re-assemble pending files. Increasing the file size limit has very little impact on the fragmentation ratio. Most small files on an NTFS file system do not seem to have intra-file fragmentation. But the memory requirement increase roughly proportionally to the file size limit. Finally, the file percentage is high even for file size 128 KB. More than 95% of files are smaller than 512 KB. This suggests that DAFT can enumerate a significant portion of a file system without incurring an exceedingly large buffer memory requirement. We evaluated the same three bulk file access methods as in the fragmented file system case. Figure 3 shows the throughput of these three methods for both the fast disk and the slow disk and under different file size limit. The results are very similar to those obtained from the non-fragmented file system test.

A key difference between DAFTM’s performance under a non-fragmented file system and that under a fragmented file system is that DAFTM cannot fully exploit the raw disk throughput because when target file fragments are retrieved, they may not reside in contiguous areas on the disk and thus might require additional disk seeks and/or rotations. These additional disk seeks and rotations prevent DAFT’s throughput from reaching the raw disk throughput and in turn decrease DAFTM’s improvement over Vanilla.

The performance gain of DAFTM over Vanilla is shown in Table IV. Although intra-file fragmentation takes a toll on the effectiveness of DAFTM, DAFTM still out-performs Vanilla by a factor of 4 to 7, because DAFTM is largely immune from inter-file fragmentation, which remains as a significant factor that slows Vanilla down.

As expected, the performance gain of DAFTM over Vanilla decreases with the file size limit because the benefit of reduced disk seeks and rotations is less significant for larger files than for smaller files. Moreover, this performance gain of DAFTM over Vanilla is smaller on the faster disk than that on the slower disk, which is the opposite of that from the non-fragmented case, because the additional holes/gaps among files

hurt DAFTM’s performance more on the fast disk than on the slow disk.

Interestingly, although the peak re-assembly buffer memory requirement for enumerating all files in the fragmented file system that are smaller than 1024 KB is 718 MB, given a buffer memory budget of 50 MB, DAFT is still able to deliver more than 4 times as much throughput improvement over Vanilla on both the fast and slow disks.

D. Comparison among Buffer Reclamation Methods

Because DAFT is typically given a limited amount of memory to re-assemble pending files, how it frees up buffer memory used by pending files when it exhausts its buffer memory budget has a significant impact on DAFT’s performance. In this section, we compare the effectiveness of the following two buffer free-up methods:

On-demand cleaning: Whenever the first file fragment of a new file appears and there is not enough memory in the buffer to accommodate the entire new file, this algorithm stops admitting new files, fetches only file fragments required to complete all the pending files at that instant and frees up all the buffer memory they occupy. After fully assembling and delivering the pending files, this algorithm resumes in the normal mode with the new file that triggers the current clean-up transaction.

Dynamic bin packing using first fit: This algorithm uses a dynamic bin packing formulation to determine a priori the number of passes required to bring in a bulk file access’s target file set into memory based on their physical range, and uses a first fit heuristic to solve it. The given buffer memory represents a bin, and files assigned to one sequential pass of the disk must be fit into the bin. Whenever a new file cannot fit into a pass’s bin, the algorithm either fits it to other bins/passes or creates a new bin to hold the file.

Figure 4 shows the throughput comparison among Vanilla, DAFT with limited buffer memory managed by these two buffer reclamation algorithms, and DAFT with unlimited

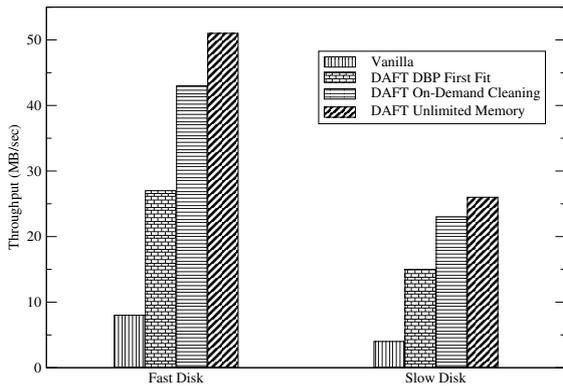


Fig. 4. The throughput comparison among Vanilla, DAFTM with limited buffer memory managed by two different buffer reclamation algorithms, and DAFT with unlimited buffer memory. In the limited buffer memory cases, the amount of buffer memory is set to 50 MB. The target file set consists of all files on the test fragmented file system that are smaller than or equal to 512 KB.

buffer memory. In the limited buffer memory case, the amount of buffer memory is set to 50 MB. The target file set consists of all files on the test fragmented file system that are smaller than or equal to 512 KB. Tests on other target file sets produce similar results. It is surprising that the overall throughput of the dynamic bin packing method is actually worse than that of on-demand cleaning, although the number of passes through the disk required by the dynamic bin packing method (8) is smaller than that of on-demand cleaning (12). A first-order explanation of this counter-intuitive result is that each pass in on-demand cleaning does not incur the same overhead as that in the dynamic bin packing method. When on-demand cleaning triggers a clean-up operation, which is counted as a distinct pass, it only fetches the set of file fragments that belong to the pending files and are yet to be picked up. So each such clean-up operation does not need to sweep the entire disk.

A deeper analysis reveals that a more subtle performance factor is at play. The first fit algorithm assigns files to bins on a file by file basis. After bins are filled up, every time some space is freed up in a bin, the space is only large enough to hold one or two new files before it is reused. As a result, physically adjacent files are likely to be assigned to different bins after existing bins start to be filled up. This means that there are more holes between neighboring files that are assigned to the same bin/pass, and therefore more disk seeks are needed in each pass. In contrast, on-demand cleaning holds off all new files before completing current pending files when the buffer memory is used up. That is, this algorithm only starts assigning files to a bin when the bin is empty, stops assigning more files to the bin when the bin is filled up, and repeats the process only when the bin becomes empty again. By holding off insertions of new files into a bin until it has enough free space, in this case the entire buffer memory, on-demand cleaning ensures that files which are physically adjacent are more likely to be retrieved in one disk pass, and thus improves the overall disk access efficiency, which is the main reason why an ostensibly

greedy algorithm like on-demand cleaning actually performs better than file-by-file dynamic bin packing.

We are currently studying how to derive the optimal S value for the batched version of the dynamic bin packing problem, which we believe is highly dependent on the input workload. As a preliminary result, for file enumeration in the fragmented test file system using a 50-MB memory buffer, We found there is a significant performance jump from $S = 0MB$ to $S = 12.5MB$, but not much performance difference between $S = 12.5MB$ and $S = 50MB$ (on-demand cleaning).

E. Impact of Buffer Memory Size

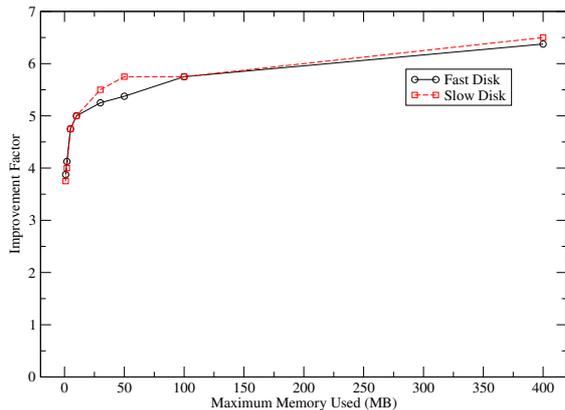
Intuitively, the more buffer memory is available, the more performance improvement DAFT can provide over Vanilla. However, it is not clear what the minimal buffer memory requirement is in order for DAFT to deliver a substantial performance improvement over Vanilla. This subsection is meant to answer this question.

Figure 5(a) shows the improvement factor of DAFTM with on-demand cleaning running on the fast and slow disk when the available buffer memory size is varied. The target file set consists of all files on the fragmented test file system that are smaller than or equal to 512 KB. The rightmost data point in the figure is DAFT with an infinite amount of buffer memory. In this cases, no buffer reclamation is needed during the run, because there is enough memory to hold all pending file in the target file set simultaneously.

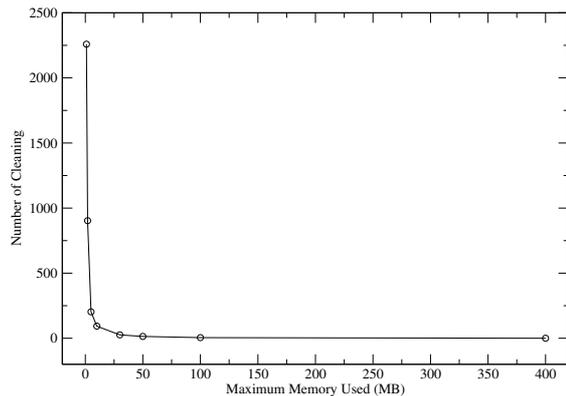
Even when the amount of re-assembly buffer memory is as small as 1 MB, DAFT can already deliver a throughput improvement over Vanilla of 3.8 and 3.7 for the fast and the slow disk, respectively. With this amount of buffer memory, the number of buffer clean-up operations is over 2000, as shown in Figure 5(b). However, these many clean-up operations do not diminish DAFT's performance gain over Vanilla, because the required file fragments to complete pending files in each buffer clean-up operation tend to be relatively close to one another on the disk, and consequently the actual cost of each buffer clean-up operation is much smaller than that of a full pass over the disk.

DAFT's throughput improvement over Vanilla increases drastically when the amount of available buffer memory increases from 1 MB to 10 MB. After 10 MB, the net benefit of adding memory is no longer that significant and the performance improvement saturates. As shown in Figure 5(b), the number of buffer clean-up operations decreases drastically when the amount of available buffer memory increases from 1 MB to 10 MB. After 10 MB, only a handful of buffer clean-up operations. These results suggest that the marginal performance benefit of adding more buffer memory is strongly tied to the number of additional buffer clean-up operations it can further eliminate.

The performance benefit of more buffer memory is more pronounced for the fast disk than for the slow disk, as evidenced by the steeper slope of the fast disk curve. This behavior arises from the fact that compared with a slow disk, a fast disk is much better at sequential access and is



(a) The improvement factor of DAFTM with on-demand cleaning



(b) The number of on-demand clean-up operations

Fig. 5. The throughput of DAFTM with on-demand cleaning running on the fast and slow disk and the required number of on-demand clean-up operations, as the amount of available re-assembly buffer memory is varied. The target file set consists of all files on the fragmented test file system that are smaller than or equal to 512 KB.

only slightly better at random access. Therefore, as more buffer memory increases the percentage of sequential access in servicing a bulk file access request, the amount of performance improvement is also more significant for the fast disk than for the slow disk.

IV. RELATED WORK

Modern file systems are designed to access large files efficiently and can routinely read/write large files at a throughput very close to the raw disk throughput. Unfortunately existing mainstream file systems are not equipped to provide efficient access to small files. On normal desktop or laptop machines, the observed throughput of accessing small files (less than 1 MB) ranges from 1 MB/sec to 10 MB/sec even though the raw disk throughput can be over 50 MB/sec. But it is known small files dominate a typical file system in terms of the number of files. Ganger and Kaashoek [7] reported that 79% of all files on their file servers are smaller than 8 KB. Mandagere et al. [2] reported that around 80% of the files in a representative backup and archival data set are files smaller than 32 KB. Vogels [8] reported that files smaller than 64 KB account for the majority of files accessed on a Windows-based file server.

The performance problem associated with small files has long been known and researched. Most previous research efforts attempted to solve this problem by co-locating related file objects and reading/writing related objects using a single disk I/O. The approaches proposed varied in terms of where and at which level the co-location is done.

Mullender and Tanenbaum [9] proposed the idea of *immediate files*, which expand a file's inode to the size of a logical disk block so as to include the first part of the associated file. When a file is small, the file's content and its metadata will be stored together inside its inode. This scheme is also implemented in Microsoft's NTFS file systems. When a file's data can be fit into its MFT record, they are stored inside the MFT record directly. This idea corresponds to co-location of a file's data and metadata at the individual file level.

The fast file system (FFS) [10] places a file's inode and its data blocks in the same cylinder group. This corresponds to co-location of data and metadata. But this approach only reduces the seek time between a file's inode and its data blocks, but not the rotational delay and command processing delay. Even with this optimization, the resulting seek time could still be significant because even a track-to-track seek incurs a non-negligible delay and increases quickly for slightly longer seek distances [11].

Ganger and Kaashoek [7] proposed C-FFS (Co-locating Fast File System) to address the shortcomings of FFS. C-FFS includes two novel ideas. First, the inode of a file is stored directly in the directory containing the file. This is an improvement over FFS because it completely removes the time between accessing a directory and the inodes of the files in it. Second, data blocks of multiple small files in a given directory are allocated adjacently and read from the disk as a single disk I/O unit in most cases. These two ideas together correspond to co-location of directory, file metadata, and file data.

Both FFS and C-FFS depend on enough free space to co-locate related objects together. When there is not enough free space, their capability to co-locate related objects degrades. Defragmented file system (DFS) [12] proposed to relocate data at run time to eliminate two kinds of fragmentation. First, fragments of small files (intra-file fragmentation) are relocated to a contiguous disk area. Second, related files in a directory (inter-file fragmentation) are relocated together to a contiguous disk area. The key advantage of DFS is that it piggybacks its defragmentation operation with normal file accesses. More concretely, DFS directly uses cached file fragments to relocate them so as to reduce the read overhead during relocation. To leverage cached file fragments, it needs to modify the OS to secure an exclusive access to them, lest it should corrupt the data if they are being accessed by other processes concurrently.

All the above ideas on file system object co-location obviously help improve the performance of bulk file accesses.

But to co-locate directories or inodes requires the OS to be modified. If such co-location operates only on files, it may not be necessary to modify the underlying OS because most OSs already provide some form of defragmentation API. But moving files could incur a significant performance overhead and is not always an acceptable option on all systems. A major advantage of DAFT is that it could be implemented as a user-level library used by any applications without requiring modifying the OS or incurring additional overheads such as DFS.

The idea of performing block-level data back-up by reading a disk volume sequentially for better performance is well known. Unfortunately, it is difficult to restore individual files with block-level back-up. To produce the file-level restore capability, Matze et al. [13] proposed to identify the blocks related to directories and inodes and store the metadata at the beginning of the back-up tape. When doing file-level restore, the metadata is consulted to obtain the location of the file data. But this approach results in poor restore performance because file data may be fragmented and scattered on the tape. Moreover, this approach cannot provide applications with individual files while backing up blocks of a disk volume. Therefore it cannot satisfy the requirements of many bulk file access applications such as AV scanners or file-based deduplication. DAFT is unique in that it can provide applications with complete files while sequentially reading disk blocks in a volume. That is, DAFT achieves the best of both worlds: the flexibility of file level information and the performance gain of block level disk access.

Patterson et al. [14] advocated the use of application hints to prefetch files before an application accesses them. In some sense, the bulk file access mechanism of DAFT is a generalization of informed prefetching: A DAFT application also specifies beforehand a set of files it is going to access. However, DAFT not only prefetches them, but also brings them into memory in the most efficient manner, in terms of both main memory requirements and disk bandwidth utilization.

V. CONCLUSIONS

Most existing file systems assume that when an application reads the files it needs, it demands to access them in a specific order or at certain points during its execution, and accordingly provide a *control-driven* access interface for an application to order their read operations or to issue read operations at proper moments. This paper advocates a *data-driven* interface that allows an application to express its intention to access a set of files without imposing any ordering and timing constraints. This interface is data-driven because fetching individual files into memory triggers the core application logic, rather than the other way around. Although the data-driven file access interface is less general or flexible than the more traditional control-oriented file access interface, it has many important applications, such as AV scanning, data back-up, data scrubbing, software build, disk defragmentation, etc.

Once an application specifies the set of files it needs through this data-driven interface, the proposed system, called DAFT, fully leverages the absence of ordering/timing constraints and optimizes the performance of accesses to these files in a way that respects the disk geometry and thus is highly efficient. The DAFT prototype demonstrates that it can consistently improve the end-to-end elapsed time required to bring a large set of small files into memory by at least an order of magnitude. Moreover, even with a small main memory budget, DAFT is still able to achieve at least a factor of 5 improvement in the end-to-end elapsed time. In summary, the DAFT project makes the following specific research contributions:

- A bulk file access interface for an application to explicitly specify its intention to access a set of files in any order that the file system decides to bring them into memory,
- An on-the-fly file re-assembly algorithm that uses a minimal number of sequential passes through the disk(s) to bring a target file set into memory when the total amount of buffer memory is fixed, and
- A fully operational DAFT prototype that successfully demonstrates its dramatic performance improvement over file systems without support for bulk file access.

REFERENCES

- [1] N. C. Hutchinson, S. Manley, M. Federwisch, G. Harris, D. Hitz, S. Kleiman, and S. O'Malley, "Logical vs. Physical File System Backup," in *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*. Berkeley, CA, USA: USENIX Association, 1999, pp. 239–249.
- [2] N. Mandagere, P. Zhou, M. A. Smith, and S. M. Uttamchandani, "D3: Demystifying Data Deduplication," University of Minnesota.
- [3] E. G. Coffman, M. R. G. Jr., and D. S. Johnson, "Dynamic Bin Packing," *SIAM Journal on Computing*, vol. 12, no. 2, 1983.
- [4] Microsoft, "Volume Shadow Copy Service (VSS)," Windows Sysinternals, 2008. [Online]. Available: [http://msdn.microsoft.com/en-us/library/bb968832\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/bb968832(VS.85).aspx)
- [5] , "HD Tune," 2008. [Online]. Available: <http://www.hdtune.com>
- [6] Symantec, "Symantec Backup Exec System Recovery," 2008. [Online]. Available: <http://www.symantec.com/business/back-up-exec-system-recovery-desktop-edition>
- [7] G. R. Ganger and M. F. Kaashoek, "Embedded Inodes and Explicit Grouping: Exploiting Disk Bandwidth for Small Files," in *In Proceedings of the 1997 USENIX Annual Technical Conference*, 1997, pp. 1–17.
- [8] W. Vogels, "File System Usage in Windows NT 4.0," in *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 1999, pp. 93–109.
- [9] S. J. Mullender and A. S. Tanenbaum, "Immediate Files," *Softw. Pract. Exper.*, vol. 14, no. 4, pp. 365–368, 1984.
- [10] M. K. Mckusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, "A Fast File System for UNIX," *ACM Transactions on Computer Systems*, 1984.
- [11] B. L. Worthington, G. R. Ganger, and Y. N. Patt, "Scheduling Algorithms for Modern Disk Drives," 1994, pp. 241–251.
- [12] W. H. Ahn, K. Kim, Y. Choi, and D. Park, "DFS: A De-Fragmented File System," *miscots*, vol. 0, p. 0071, 2002.
- [13] J. E. G. Matze and D. L. Whiting, "System for Backing up Computer Disk Volumes with Error Remapping of Flawed Memory Addresses," U.S. Patent 5 907 672, 1999.
- [14] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed Prefetching and Caching," in *In Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. ACM Press, 1995, pp. 79–95.