3-29-2011

Propeller: A Scalable Metadata Organization for A Versatile Searchable File System

Lei Xu
University of Nebraska-Lincoln, lxu@cse.unl.edu

Hong Jiang
University of Nebraska - Lincoln, jiang@cse.unl.edu

Xue Liu
University of Nebraska-Lincoln, xueliu@cse.unl.edu

Lei Tian
University of Nebraska-Lincoln, tian@cse.unl.edu

Yu Hua
University of Nebraska-Lincoln, yhua@cse.unl.edu

See next page for additional authors

Propeller: A Scalable Metadata Organization for A Versatile Searchable File System

Lei Xu
University of Nebraska-Lincoln
lxu@cse.unl.edu

Hong Jiang
University of Nebraska-Lincoln
jiang@cse.unl.edu

Xue Liu
University of Nebraska-Lincoln
xueliu@cse.unl.edu

Lei Tian
University of Nebraska-Lincoln
tian@cse.unl.edu

Yu Hua
University of Nebraska-Lincoln
yhua@cse.unl.edu

Jian Hu
University of Nebraska-Lincoln
jhu@cse.unl.edu

ABSTRACT

The exponentially increasing amount of data in file systems has made it increasingly important for file systems to provide fast file-search services. The quality of the file-search services is significantly affected by the file-index overhead, the file-search responsiveness and the accuracy of search results. Unfortunately, the existing file-search solutions either are so poorly scalable that their performance degrades unacceptably when the systems scale up, or incur so much crawling delays that they produce acceptably inaccurate results. We believe that the time is ripe for the re-designing of a searchable file system capable of accurate and scalable system-level file search.

The main challenge facing the design and implementation of such a searchable file system is how to update file indices in a *real-time* and scalable way to obtain accurate file-search results. Thus we propose a lightweight and scalable metadata organization, *Propeller*, for the envisioned searchable file system. Propeller partitions the namespace according to file-access patterns, which exposes massive parallelism for the emerging manycore architecture to support future searchable file systems. The extensive evaluation results of our *Propeller* prototype show that it achieves significantly better file-indexing and file-search performance (up to 250×) than a centralized solution (MySQL) and only incurs negligible overhead (< 16%) to the normal file I/O operations and faster direct access performance (16.6×) on a state-of-the-art file system (Ext4). Furthermore, the 100% recall accuracy ensures Propeller offering a feasible metadata scheme for the system-level file-search services.

1. INTRODUCTION

Facing the challenges of managing the explosive growth in digital contents in file systems [30, 33], users and administrators alike expect the systems to be able to answer complex file queries, such as *"Which are the recently modified virtual machine images in our cloud infrastructure?"* or *"Where are the papers and their supplemental materials about SSDs published in SOSP in the last five years?"*, fast and accurately. Even more interestingly, the applications in such systems can also benefit from file-search functions to fast gather their desired file sets to operate on, e.g, *"A snapshot program that locates all recently changed files"* or *"A runtime support that prefetches files related to Program P to accelerate its execution"*, without having to traverse the entire file system, which is usually time consuming and error prone.

Unfortunately, the conventional hierarchical file systems have failed to efficiently support file-search functions, especially as the number of files they manage scales up [28, 29, 32], because the hierarchical directory structures were not designed to optimize file-search functionalities [20, 34]. The general principle behind effectively organizing such hierarchical directory structures is to divide the namespace into sub-directories manually with appropriate names. However, this approach to categorizing files is unlikely to always match the patterns of incoming file-search requests. For instance, a user may classify his/her papers by the order of {*conference, year, title*}, e.g., "/SOSP/2009/a-kernel-paper.pdf". Such a directory structure is not capable of efficiently answering a question like "*Which papers discussed about manycore kernel designs?*" without resorting to brute-force traversals in the entire paper repository. To better understand this problem of mismatch between current organization of files (i.e., the hierarchical directory structure) and its ability to easily locate files (i.e., serving file-search queries), and the prevalence of the problem experienced by file-system users, we would like the readers of this paper to ponder, from your own experiences, such questions as: *Have you ever hesitated or agonized in choosing representative names for your files? Have you ever found yourself at a loss trying to recall the location of a file off the top of your head even though the file is named appropriately?* From the authors' own experiences, we believe that most of the readers would agree that they have encountered to a variant degree such file name mismatch problems. In addition to this mismatch problem, the hierarchical directory structures also degrade the I/O performances by incurring unnecessary overheads in directory lookup [10] and lock contention [13]. We will explore these issues in more detail in Section 2.1.

As an open problem, the issue of providing efficient file-search capability has been addressed in many commercial products [8, 21, 37] and research prototypes [20, 23, 28, 32] to various extents. Unfortunately, these file-search solutions are either not scalable due to their use of centralized query-optimized data structures [32], or unable to overcome the essential limitations of the hierarchical directories [10, 34] because these solutions run as user applications on top of hierarchical file systems. Furthermore, none of the existing solutions has explicitly considered the applications (i.e., non-human users), which lack the necessary intelligence possessed by human users to adjust and tolerate inaccurate results. Therefore, we argue that for the applications the file-

search functionalities are required to provide consistent and up-to-date results. On the other hand, due to resource concerns, most file-search engines apply a *post-process* model (or called *offline* model) to aggregate the new changes into their file indices after these changes have been flushed to disks. For example, Spotlight [8] take advantages of file system notification mechanisms [6, 35] to gather changed files and then update their corresponding file indices. This post-process model introduces an inevitable window of delay in which file-search engines must catch up with file modifications. During this window, file-search engines cannot guarantee the accuracy and consistency of results. Unfortunately, this inaccuracy-inducing window will only be further widened with the increasing scale of file systems, workload intensive and complexity, etc.. Thus it is reasonable to argue that the existing solutions are not likely to guarantee the accuracy of file-search results and provide search functionalities to applications.

To address the aforementioned shortcomings of the existing file-search solutions and the hierarchical namespaces, we believe that the time is ripe for a new type of file system, which we call a *Searchable File System*, designed with the important features of *fast direct file access*, *versatile system-level search API* and *high scalability*.

As we have indicated earlier, the main challenge facing such a searchable file system is to keep file indices always up-to-date (a.k.a *inline* model) to guarantee the consistency and accuracy of search results, which is usually considered a very costly requirement. Fortunately, the emerging manycore architectures [27] are poised to provide potentially sufficient amount of CPU power to make *inline* file-index feasible if the file systems can embrace such a highly-parallel architecture [9, 46] to effectively exploit its massive computational parallelism. In this paper, we propose a scalable and light-weight metadata organization to support such a searchable file system, called *Propeller*, that focuses on taking advantages of manycore processors to provide fast file-index performance, by exposing and exploiting parallelism in metadata-update and file-index operations, without sacrificing the normal file I/O performance. Propeller is designed based on the following observations:

- For *inline* file-index model, the high file-index overhead stems primarily from: (1) the increasing scale of file indices, and (2) the I/O operations that trigger file re-indexing.
- Files can often be clustered and isolated into independent and small groups according to their corresponding access patterns and semantic correlations and to possible hints from their users.
- The file-search related operations are much less frequent than the actual I/O operations.

Thus, the proposed metadata organization is expected to offer versatile file-search functionalities on manycore-based systems. In addition to the existing file-search solutions that only consider the file content [8,21,37] or the namespace [28, 32], Propeller takes into account the various file system I/O characteristics and processing parallelism of the manycore processor architecture.

The main contributions of this paper include:

1. A comprehensive discussion on the design principles of a scalable, searchable and versatile file system and its metadata organization, in light of the data-explosion and manycore trends;
2. The development of several novel namespace clustering and indexing techniques, including *Semantic File Grouping*, *Lazy Indexing* and *Versatile Intra-Partition Indexing*, designed to enable the proposed lightweight, scalable metadata organization for a envisioned searchable file system so as to drastically reduce the file-index and file-search latencies;
3. The prototyping of Propeller and its extensive evaluation showing that Propeller significantly outperforms an existing database-based and centralized approach (MySQL) in file-index and file-search and incurs negligible overhead to the normal file I/O operations of a state-of-the-art file system (Ext4).

The rest of this paper is organized as follows. Section 2 presents the necessary background and related work to motivate the work on a file-search-oriented metadata organization scheme, Propeller, of an envisioned searchable file system. Section 3, describes the design and implementation of Propeller, with a focus on its optimizing approaches. We evaluate the Propeller performance in Section 4, and finish with concluding remarks in Section 5.

2. BACKGROUND AND MOTIVATION

This section presents the necessary background and elaborates on our observations that help motivate the Propeller research.

2.1 The Need for File-Search Capabilities

The amount of data in modern storage systems is growing explosively [24, 30]. While the distributions of file sizes have not changed significantly [5], the scale of file systems (the number of files) has increased so dramatically that it makes managing such large file namespace an increasing challenge for end-users and system administrators [34]. Moreover, the data generated today contain much more semantics and are significantly more inter-related among files than ever before [40], which further increases the complexities of file namespace. As a result, the aforementioned complex and large namespace led to the demands for efficient file-search functionalities. While both academia and industry have a consensus that file-search is vital for human users, we argue that it is equally important to the systems and applications. One evidence that supports the urgent need for file-search functionalities from applications is the many popular applications, such as iPhoto [7], that have implemented their own special-purpose search engines and search-driven intuitive GUIs to organize and present data. We believe that, with a well-designed system-level search API, systems and applications will be encouraged to take advantages of this OS file-search service to improve their usability and accelerate IO performance. In Table 1, we present several examples to illustrate how file-search can benefit both human users and applications for three basic types of file-search functionalities: **1)** *Attribute Search*, search for files with certain defined values, which include but are not limited to file attributes (e.g uid, size, atime, file type etc.) [28,32]; **2)** *Keyword Search*, search for files by their contents [8, 20, 21, 23, 36] ; and **3)** *Relation Search*, search for related files based on the extracted semantic relationships among them [37, 40].

Target	Search	Examples
Human Users	Attribute	*find all files updated recently (mtime < 1day) by user john (uid = 1001)*
	Keyword	*find the all papers that has keyword "**SSD**" in their titles*
	Relation	*find all supplemental materials of one paper and the contact cards of its authors*
Applications	Attribute	***Backup Service**: finds the recent changed files and create incremental snapshots*
	Keyword	***IDE**: records building dependency of a C++ project by analysing the call graph*
	Relation	***Prefetching**: once a file being read, fetch the related files that will be accessed later*

Table 1: File Search Examples

Unfortunately, the hierarchical namespace, which has been the standard file system interface for decades [16], is insufficient to systematically organize files and efficiently serve the above file-search functionalities. It uses *directories* to categorize files into a top-to-bottom namespace, in which each file is identified *exclusively* by its *full path* that consists of a sequence of directory names and one file name. In such a namespace, users and applications tend to embed search hints within file or directory names for the sake of future retrieval. On the one hand, this naming approach unavoidably increases the complexity of namespace and the system management cost. On the other hand, the mismatch between the various file-search requests and the sole representation of file name (*full path*) renders the efforts on classifying files useless, because a time-consuming brute-forced directory traversal must still be resorted to. To make things worse, the file path of a file in current systems is no longer fully relevant to its content [23, 34], which defeats the purpose of the hierarchical namespace: a system that classifies and locates files by names.

For the purpose of quantitatively measuring the complexity of representing file names in real hierarchical file systems, we have collected statistical information of file system namespace from 26 graduate students' working machines, one high-performance cluster (HPC) and one public website (an open-source community website). These statistical data include the total number of words in the full path of each file. The distribution of operating systems used in this sample is listed in Table 2.

Operating System	Users	# of Files
Windows	9	2,122,851
Mac OSX	12	10,273,253
Linux	5	2,624,107
CentOS (HPC)	Unknown	19,349,069
FreeBSD (Web)	Unknown	7,861,380

Table 2: Operating Systems Distribution

As illustrated in Figure 1(a), the average number of words used to describe a file in personal computers is around $15 \sim 25$ words, which means that an end user must use about 20 words to identify one file. Figure 1(b) shows a higher average complexity of filename representation ($20 \sim 30$ words) in the HPC cluster because it serves multiple physicists and chemists for scientific computing, that is, they share their home directories and tend to use longer filenames to describe the experimental conditions of results [24]. Even in a public web-hosting server (Figure 1(c)), in which there is no personal data, the complexity of filename representation ranges from 10 to 20 words. These figures clearly represent a management nightmare: end users and system administrators must devote a great deal of effort to naming the files in order to keep their digital contents appropriately organized for indexing and future retrieval.

As a result, the above observations inescapably lead us to the conclusion that the current hierarchical namespace is insufficient in effectively organizing and retrieving files. Additionally, maintaining such hierarchical namespace introduces huge performance overheads for large scale systems. In most file systems (e.g., Ext4/NTFS/ZFS), file systems must read all parent directory files (or blocks) to locate one file. It introduces several disk IOs (and seeks for hard drives) and/or several rounds of network lookups [10], a well-recognized bottleneck for storage systems. Moreover, to keep the consistency among directories, the parent directory must hold locks when the sub-directories are being accessed [19, 34], which limits the scalability and parallel performance of the system, especially on manycore-based systems [13].

Given the limitations of the hierarchical namespace described above, we believe that the next-generation file system namespace should provide:

- A *direct file-access API* that eliminates the lookup overheads of the hierarchical directory in accessing files with known locations.
- flexible and separate *system-level search API* for fast gathering desired files according to the incoming file-search patterns.

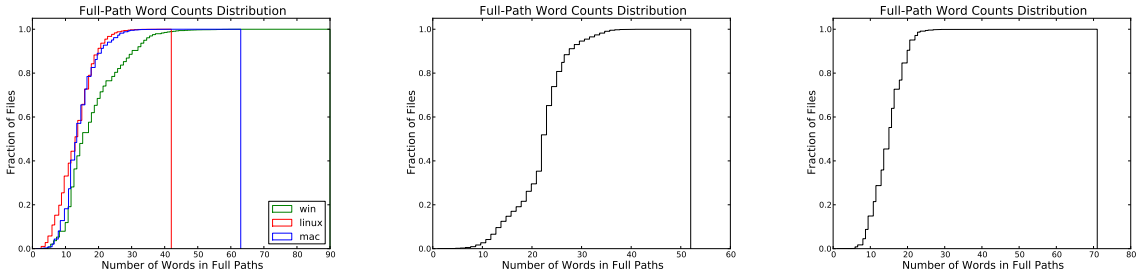
To achieve the aforementioned file-search functionalities, various techniques are required to extract and index the appropriate and useful information from files. One of our primary design goals for Propeller is to flexibly support versatile file-search functionalities at *the system level*. We will discuss the design considerations in Section 3.

To achieve the aforementioned file-search functionalities, various techniques are required to extract and index the appropriate and useful information from files. One of our primary design goals for Propeller is to flexibly support versatile file-search functionalities at *the system level*. We will discuss the design considerations in Section 3.

2.2 Existing File-Search Solutions

To address the management issue of large file systems, several commercial file-search applications have been introduced and deployed to either small-scale file systems [8, 21, 36] or large-scale file systems [22]. Meanwhile, as an active research area, numerous research prototypes aim to bring file-search functionalities into file systems [20, 28, 32, 34]. These file-search commercial applications and research prototypes can be broadly divided into two categories:

- *File systems with system-level search functionalities* that are built on top of either centralized query-optimized data structures [20,28] or relational databases (RDBMS) [37].
- *File-search engines* that run on top of existing file systems. Most recent desktop search products and research prototypes [8, 21, 32] fall into this category.



(a) End Users' Computers (b) High-Performance Cluster (c) A Public Website

Figure 1: Name Complexity Distribution

The centralized index structures (e.g B+Tree [15] for Semantic File System [20], R-Tree [25] for SmartStore [28] and RDBMS for WinFS [37]) used in the former come at the cost of very high maintenance overheads, which limits the overall I/O performance of a file system. Moreover, these centralized data structures are often the source of poor scalability, especially for manycore-based systems, in which the shared data structures modified by multiple cores can lead to huge performance degradations [13, 46, 47].

Furthermore, most production-level file-search solutions [8, 21, 36, 37] typically use RDBMS to store and index metadata. Unfortunately, RDBMSs were originally designed for very different purposes, such as transaction processing in banks, which requires RDBMSs to support heavy locks and transactions, guaranteeing the ACID properties (atomicity, consistency, isolation, durability). In addition, for specific application optimizations, an RDBMS usually trades off between update and search costs [3]. Therefore, the mismatch between file system I/O patterns and typical RDBMS applications prevents file indexing from being optimized for performance [34, 41, 42].

The latter category of file-search solutions run as an additional service on top of existing file systems, meaning that they use *offline* approaches that rely on crawling processes or specific system notification mechanisms (e.g, *inotify* [35] or *fsevent* [6]) to gather new file modifications and then merge them into the file index. In this case, *inotify* and *fsevent* must maintain a table to map a file system event to its corresponding file or directory, which will be considerably resource consuming in large systems. Particularly, inotify in Linux uses an in-RAM buffer to communicate with the applications. As a result, if the incoming file changes arrive at a faster pace than the indexing speed of the file-search engine, this buffer will likely overflow so quickly and frequently that the file-search engine will lose part, if not most, of the changes, which is particularly true when the system is heavily loaded. Additionally, the *offline index* mode employed by the application-level file-search solutions, owing to its separation from the processes that manipulate data, needs to issue extra IOs to locate, read and parse changed files, and thus misses the opportunity to capture crucial execution contexts [40] that are useful for enhancing the quality of file-search results.

More importantly, the inevitable window of possible inconsistency caused by the crawling and re-indexing delays results in outdated and inaccurate search results. Unfortunately, this inaccuracy-inducing window is the main challenge facing the existing file-search solutions when directly serving systems and applications that *lack the necessary intelligence possessed by human users to adjust and tolerate inaccurate results.* Based on this observation, we argue that:

- The **inline index mode** that *performs real-time updating on file-search indices* is the key to providing *system-level file-search functionalities* to systems and applications.

However, applying *inline index* on large-scale file-search products is usually difficult, because it results in poor indexing performances and adds considerable overheads along the I/O critical path. Our Propeller is designed to overcome the performance bottleneck of inline indexing on large-scale searchable file systems.

Another interesting fact about these file-search solutions is that, even with them already being integrated or installed within the systems, applications tend to embed their own application-specific search engines, because the existing file-search products are not flexible and extensible enough to meet applications' specific requirements. However, the main difference between the system-wide file-search engines and application-specific file-search engines lies only in the content to be stored and indexed, but not in the basic index data structures (e.g. B+Tree or hash table). For instance, to index and sort MP3 files according to their albums, a music player application only needs to build a B+Tree on the "album" attributes of these MP3 files. But the general-purpose file-search engines, like Spyglass [32] and SmartStore [28], only support file-search based on file's inode attributes and are not extendable to supporting arbitrary attributes. Besides developers putting needless redundant efforts, application by application, these separated application-specific file-search engines prevent information from being efficiently retrieved at the system level.

Since the existing file-search solutions are not able to overcome the weaknesses of hierarchical file systems, we argue that it is time to design a file-search-oriented metadata organization to support the envisioned searchable file systems, which offers:

- *Realtime Inline Index.* The metadata and file indices must be kept always up-to-date to guarantee the accurate file-search results.
- *Direct Access.* It allows files to be opened directly without having to issue multiple directory lookup operations, by separating the file-locating function from the file-access APIs.
- *High Scalability.* The architecture of the proposed metadata organization must embrace the emerging trends of

the explosive growth of data and the increasing computation power brought by manycore processors to provide highly scalable file index, search and access performance.

- *Versatile System-Level Search Capabilities.* Besides supporting the *Attribute-Based Search*, *Keyword-Based Search* and *Relation-Based Search* described in Section 2.1, the proposed metadata organization must also provide a flexible and extensible infrastructure for users and applications to index customized file attributes.

The main challenge facing such a file-search-oriented metadata organization is to overcome the expensive file-indexing overheads. In order to eliminate the performance bottleneck caused by complex file-index structures while providing effective and scalable search capabilities in file systems, we propose a metadata organization scheme, called Propeller, with *Semantic File Grouping* that accelerates file-index and file-search performances by isolating the IOs based on access correlations. This access-based isolation of IOs helps expose and exploit the I/O parallelism and thus fully utilize the parallel processing capability of manycore processors, which will be elaborated next.

2.3 Semantic File Grouping

In general, the centralized query-optimized file-index structures in large file systems are considered one of the root causes of the poor indexing performance. The limitations of centralized data organization (e.g., poor scalability, single-point failures, lack of parallel accesses, etc.) have been well studied in the storage research community for decades. One of the common remedies for this problem is to partition the namespace [28, 32, 38, 44, 45] to improve the file indexing performance by narrowing the scope of the structure to be operated on. Moreover, in the *inline index* scenario, our analysis and ongoing experiments suggest that *the frequent re-indexing triggered by the IOs in the existing namespace-partitioning approaches adds significant overheads to the I/O critical path*, because each IO results in changed data (e.g., inode attributes, such as mtime, or the file content) that need to be indexed immediately to ensure the consistency of search results.

To overcome this inefficiency and potential performance bottleneck of the existing namespace-partitioning schemes, we must take into account the I/O patterns of the applications in the design of such schemes to avoid unnecessary overhead. This key observation significantly impacts the design of our Propeller metadata organization.

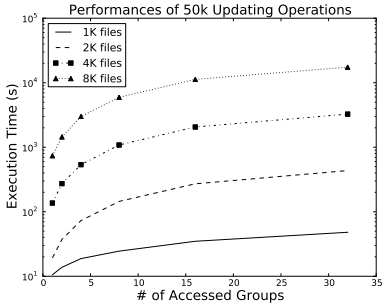


Figure 2: Metadata-Update Performance

To quantitatively assess the performance impact of namespace partitioning and the inter-partition I/O operations, we developed a C++ program that issues 50,000 update operations sequentially to different numbers of partitions, where each partition contains a different number of files, to simulate the execution of a particular application. The sizes of the partitions range from 1,000 to 8,000 files, and the total number of groups of files accessed for each run ranges from 1 to 32. Each partition maintains a B+tree for file records (inode attributes), a hash table for fast file-name lookup and a K-D-Tree [12] for multi-dimensional attributes search. The performance results, shown in Figure 2, indicate that both the partition size and the number of accessed partitions significantly impact the updating and indexing performance, because the failure to explicitly define the boundaries among files being accessed leads to expensive multi-partition file-index.

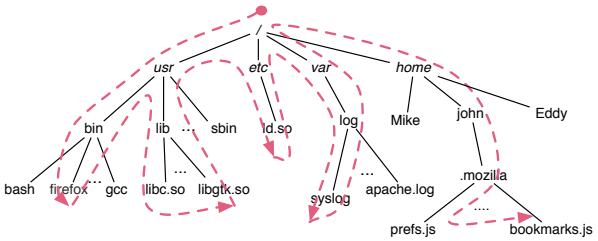


Figure 3: Firefox Execution Flow

As a result, to a achieve good I/O and file-indexing performance, the metadata organization must partition the namespace in such a way that, in addition to keeping each partition at a small scale, it isolates the IOs from different applications to avoid inter-partition IOs so that *it significantly reduces the amount of the expensive updates to file indices.* While the combined I/O pattern of a system highly depends on the contexts of all running programs, the I/O stream for each individual program usually follows a relatively stable pattern [31]. For example, during the execution of a program, say, the Firefox Web browser, only certain files will be accessed, such as executable files, libraries, resource files, configurations and temporary files, as illustrated in Figure 3. This stable access pattern, including the files being accessed and the corresponding access flow (the red dashed line in Figure 3), is determined by and also reveals the semantics of a particular program execution. Therefore, by grouping files according to the access patterns, this execution-related semantic knowledge can be captured and then utilized to boost the performance of the index and search operations.

Fortunately, the accessed files from different programs are largely isolated and independent from one another. To understand how the I/O boundaries may be identified based on the program executions, we recorded accessed files from four program executions that represent the following four different workloads on a Linux machine: Apt-get [18] (system upgrading), Firefox (web browsering), OpenOffice (document editing) and Linux Kernel Building (building the kernel). Table 3 shows that the common files accessed by any two different executions only account for a very small fraction of all accessed files. Most of these common files are correlated to their parent processes (Bash) setting up the execution environments. It is noteworthy that the relatively large number of commonly accessed files between the Firefox and OpenOffice executions are the same GUI library they rely on to render the graphical interfaces, which can be extracted

as an individual partition. Therefore, it is clear that most files can be partitioned in an isolated way based on their access patterns and the programs that access them during program executions.

Execution Name	Apt-get	Firefox	OpenOffice	Linux Kernel
Accessed Files	279	2279	2696	19715
Apt-get	N/A	31 (1.36%)	62 (2.29%)	29 (0.15%)
Firefox	31 (11.1%)	N/A	464 (17.2%)	48 (0.24%)
OpenOffice	62 (22.2%)	464 (20.3%)	N/A	45 (0.22%)
Linux Kernel	29 (10.3%)	48 (2.11%)	45 (1.69%)	N/A

Table 3: Common Files Accessed by Executions of Different Programs

From the aforementioned observations, we conclude that the access-based isolation of files offers an opportunity to avoid inter-partition index, limit the scale of file indices , and expose the parallelism among the program executions, which enables the parallelization of the costly intra-group file-index maintenance tasks to accelerate the file-index and file-search performance. Thus, we introduce the notion of "Semantic File Group", defined as a set of *access-correlated files*, because the files in such groups represent the semantics of program executions. Propeller uses *Semantic File Group* to organize files and namespace, which we will elaborate in Section 3.

2.4 Unique File System Characteristics

As discussed earlier, the file systems built on top of RDBMS are not a one-size-fits-all solution [34, 41], because the mismatch between RDBMS and file system I/O characteristics prevents the file-search and file-index operations from being optimized. In this section, we discuss several unique file system I/O characteristics that can be leveraged to improve the file-search and file-index performance.

Locality. Both file system IOs [5, 17] and file-search requests [11, 32] have very strong locality. In addition to the traditional directory cache used in operating systems, we have found that the execution-induced I/O locality as observed in Section 2.3 can be exploited to further improve the file-search engine performance.

Lopsided File-Search and I/O-Operation Distribution. In a typical file system, the file-search requests account for only a small fraction of the overall I/O operations. Since applications in hierarchical file systems generally rely on brute-forced search to locate desired files, one file-query request usually consists of a sequence of LOOKUP and corresponding GETATTR operations. We analyzed the distribution of the Lookup operations in the NFS trace [17]: there are 65,930 Lookup operations and 875,565 normal I/O operations that change the attributes or the contents of files (e.g., Write/Setattr/Rename, etc.), with the former accounting for 7% of the total I/O requests. Considering that multiple lookup operations usually belong to a single file-search request, the actual file search requests should account for a much smaller percentage than 7%. This unique property allows us to loosen the restrictions imposed by the ACID requirement from RDBMS and strike a sensible trade-off between file-IO and file-index operations to offer a good I/O performance without sacrificing the consistency guarantee.

Access Parallelism. Generally, there are multiple processes running in the operating system at any given time, whose number is poised to steadily increase given the emerging manycore trend in processors. Thus, massive I/O parallelism is clearly on the horizon. As addressed in [13], centralized in-memory data structures, such as directory cache and super block, become the bottleneck for highly parallel accesses on manycore systems. Furthermore, in the RDBMS-based solutions, the RDBMS, as the central and sequential point of operation in the entire system, quickly becomes the performance bottleneck as the system scales up [32, 41, 42]. Therefore, to improve the scalability and exploit parallelism, the proposed searchable file system and its metadata organization must isolate IOs to distributed data structures.

In the next section, we will discuss how these I/O characteristics affect the Propeller design.

3. DESIGN AND IMPLEMENTATION

In this section, we present the design and implementation of Propeller, with more emphasis on reducing file-indexing latency and improving system scalability.

3.1 Propeller Architecture

Our envisioned searchable file system aims to provide versatile file-search capabilities at the system level without sacrificing the I/O performance, particularly on manycore-based systems. Thus this searchable file system is expected to meet the goals of flexible file-search functionality, scalable file-index architecture and good IO performance. We use these goals to guide the design of Propeller, the metadata management organization of such a searchable file system.

Namespace. As discussed in Section 2.3, to obtain high metadata-update and file-index performance, Propeller first partitions the namespace into *Semantic File Groups (SFGs)*, defined as *a set of files that have access correlations*, then builds versatile file indices within the groups. Accordingly, the file system namespace is composed of two layers: 1) *Global Layer*, presenting a global view of semantic file groups; 2) *Semantic File Group* layer, offering an intra-group flat namespace for direct file accesses, as illustrated in Figure 4. In such a two-layer namespace, a particular program execution first locates the corresponding group from the global layer by a given "group id", then it communicates with the corresponding semantic file group for all future IOs. Therefore, except for the first semantic file group lookup operation, each subsequent file access involves only one intra-group file lookup.

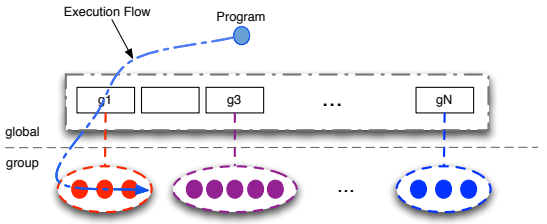


Figure 4: Two-Layer Namespace

The distinctions between the envisioned namespace supported by Propeller and the conventional file systems will likely change the way the existing applications communicate with systems. However, we argue that with sufficient performance (i.e., faster file lookups) and usability (i.e., being

easier to organize/search desired files) advantages, as well as other potential advantages(e.g., iOS uses application-specific namespace to implement sandbox [1] for security), it may be worthwhile to change the way applications interact with file systems [34]. More specifically, by incorporating *Semantic File Groups*, a searchable file system will be augmented with a number of desirable features.

First, it provides explicit boundaries of I/O accesses for certain workloads. By building file indices within each group, there are two advantages to the file-index performance: 1) the scale of the file index is bounded by the scale of the corresponding file group, which is significantly smaller than the scale of the system; 2) there is no expensive inter-group file-index updating operation. As a result, the overhead of file-indexing can be dramatically reduced.

Second, due to the I/O isolation among the semantic file groups, it is able to index and search files within each semantic file group in parallel. Furthermore, it allows the issuing of global file search by aggregating the results from parallel intra-group search operations to fully unleash the computation power of manycore processors.

Third, in addition to improving the query accuracy and efficiency, the semantic-aware nature of *semantic file groups* make it possible for them to facilitate future operating systems in improving systems functionalities such as disk layout, prefetching [31], buffering, data deduplication [43], etc.

Group Type	File Types in Group
Programs	All related execution files, resource files, configuration files, log files.
Source codes	All source files, document files and generated files belonging to the same project.
Music/Videos	Music in the same album, videos belonging to the same episode or event.
Photos	Photos taken in the same event.
Documents	All files stored in the same directory.
Others	Adjust manually.

Table 4: File Group Detection Rules. Note: photos/music/videos were pre-processed by MacOSX's iTunes and iPhoto.

In order to examine the distribution of semantic file groups existing in current systems in a quantitative manner, we developed a simple program to traverse the directory tree in a file system. The program collects the distribution of semantic file groups based on the specific group detection rules listed in Table 4. Figure 5 presents the results collected from one author's development machine. It illustrates that the vast majority (i.e., 95.2%) of semantic file groups are of size of 400 or fewer files. Given such a small group size, it is feasible to achieve a very good file-indexing performance within each group. Furthermore, with all files in one group being related to the same program, it becomes flexible to customize program-specific index in each group to support versatile file-search functionalities.

A Parallel Architecture. To fully leverage the processing parallelism of manycore systems [9, 14, 46, 47], the services of the envisioned searchable file system, shown in Figure 6, are rendered by a set of *File System Threads (FSTs)* running on different CPU cores. Each FST, created on demand, takes charge of one semantic file group and processes all operations on it, including file IOs, metadata updates, file indexing and intra-group searches, where there is no shared data among the FSTs and thus it is conducive to these FSTs' parallel and concurrent executions. Furthermore, a particular file system thread, called *Group Master Thread*, main-

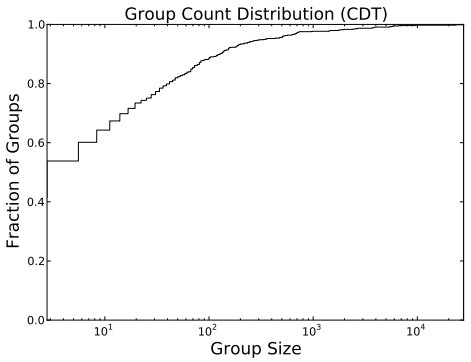


Figure 5: Semantic File Group Distribution

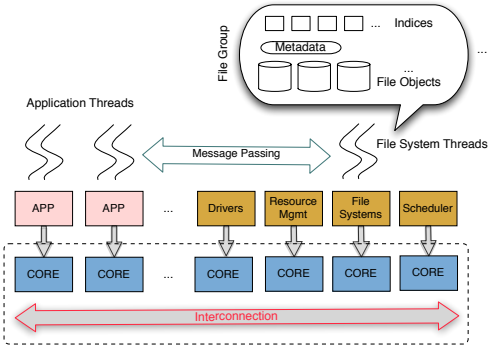


Figure 6: A Searchable File System Designed for Manycore-based Systems

tains the locations of all semantic file groups and several group indices to search groups. The user applications and system services communicate with the file system threads through a message-passing approach to reduce the overheads caused by accessing shared memory data structures [9, 46].

File System Interfaces. There are four types of built-in file system APIs : *Direct Access API*, *Search API*, *POSIX API* and *Search Hooks*, as illustrated in Figure 7. First, the direct access API offers the capability to directly access file data with a given (group id and file id) pair, where the "file id" represents the intra-group file name. Second, the search API is the primary API to serve file-locating requests. Additionally, for the purpose of backward-compatibility, a POSIX layer is under development as a thin translation layer on top of the direct access API and search API to present a hierarchical view of files. For example, it translates a lookup of "/com.mozilla.firefox/firefox.bin" to a group/file ID pair ("com.mozilla.firefox", "firefox.bin"). Lastly, human users and applications can define *Search Hooks* to customize file indices and search semantics.

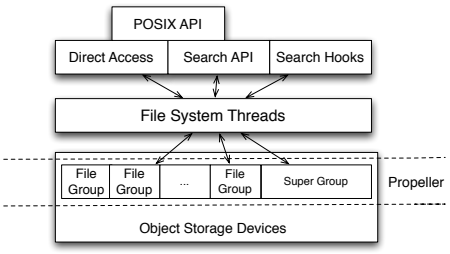


Figure 7: System Stack

Group Master Thread (GMT) runs as the global group

management service in the envisioned file system. When the operating system boots up, GMT loads the group locations from the *Super Group*, a special file group that stores the metadata about all the semantic file groups, and keeps them in the memory to answer clients' group-locating requests. GMT stores group summaries locally, which include the description of each group, such as its location, access privilege and file membership summary. These group summaries are very small and can easily fit in memory for fast accesses. For example, in our 100-million-file dataset, the total space consumed by all group summaries is only 14 MB. Furthermore, to accelerate the group-locating process, a hash table mapping group ids to their locations is built in the memory as well.

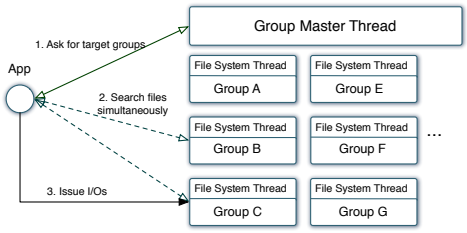


Figure 8: System Work-Flow

Parallel File Index and Search. As one of its design goals, Propeller leverages the processing parallelism of manycore processors by issuing file-search and file-index requests in parallel. As illustrated in Figure 8, a typical search request starts from a client application asking GMT for the groups that might have the targeted files. Then the client issues parallel file-search requests only to the selected SFGs that are in turn simultaneously managed by the FSTs. FSTs search the files within their intra-group file indices and return the file results to the client. Finally, the client accomplishes this file search request by aggregating the files returned from the selected FSTs. Furthermore, since the file metadata and indices are stored within each *Semantic File Group*, *Metadata Updates* and *File Re-Indexing* occur in the subsequent metadata operations or IOs that are executed **locally** and isolated from other groups. Consequently, the file re-indexing operations can be carried out in parallel by assigning each group to a different FST. Moreover, the FSTs only report the group descriptions to *Group Master Thread* when the groups are changed.

Since the Propeller design focuses on the metadata management part of the envisioned searchable file system, the remainder of the paper will concentrate only on metadata related designs.

3.2 Semantic File Group Design

The core component in the Propeller metadata organization is the *Semantic File Group (SFG)*. The central data structure of an SFG is a *Group File Table* that stores file metadata and location information. Additionally, to support versatile file-search capabilities, SFG introduces the notion of *Named Indices* that are identified by a character string and can be customized to extract content from files and to use specific data structure to index files by a given *Search Hook*. In our current design, Propeller can support up to 16 named indices beyond the four basic indices of Naming Index, Attribute Index, Content Index, and Relation Index. Moreover, the traditional "directories" are replaced by *Views*, which define the file queries that gather sub-sets of files in the semantic file groups. This will be elaborated later in this section.

Group File Table. Inspired by the Master File Table in NTFS [39], a *Group File Table (GFT)* stores all metadata of files and indices in the group. Each entry in GFT is 512-byte long. Entries 0 and 1 contain respectively the group description and additional metadata of this group, such as access control list (ACL) and keywords. Other GFT entries contain the IDs of files with the corresponding file metadata. There are several types of files in SFG: regular files, file indices, file views, as shown in Table 5. Entry 2 points to the journal file containing metadata changes. Entries 3 $\sim$ 6 contain the metadata of basic index files, which will be described shortly. Entries 8 $\sim$ 23 are able to support up to 16 named indices. Starting from Entry 32, GFT stores the metadata for regular files and file views, with Entry 32 being the root view of the entire file group.

Group Description includes the name and access privilege (owner and mode) of the group and *Membership Markers* describing the group membership of files. Propeller uses this description to accelerate global search by checking whether a search request has the privilege to access this group or the targeted files *might* be within this group.

Id	Name	Description
0	Group Desc.	(uid,gid,mode, group name, membership markers and etc)
1	Secondary Group Desc.	Additional group descriptions (e.g ACL)
2	Log	Metadata Journal Support
3	Naming Index	A hash table to fast directly access
4	Attribute Index	A K-D-Tree index for inode attributes
5	Content Index	A keyword index
6	Relation Index	A relation index
7	Reserved	For future use
8 $\sim$ 23	Named Indices	Customized Indices
25 $\sim$ 31	Reserved	For future use
32	Root View	Define the root view of files
33 $\sim$	Files & Views	Other regular files and views

Table 5: Group File Table

File Indices. To achieve fast and versatile search, Propeller maintains various file indices in each semantic file group. These indices fall into two categories:

Basic Indices. Basic Indices including naming index (hash table), attribute index (K-D-Tree [12]), content index (inverted list) and relation index (directed graph), are supported in every file group. In particular, the naming index is a hash table used to provide fast direct accesses, while other basic indices support major search functionalities described in Section 2.1.

Named Index defines the content of index and the data structure (e.g., B+Tree, K-D-Tree [12], hash table, inverted list, directed graph and SQL) to be used as the underlying indexing mechanism based on different performance characteristics.

Views. In Propeller, a *View* describes a given search, which is conceptually similar to the virtual directory used in Semantic File System [20]. However, unlike the virtual directory in [20] that only supports content-aware navigations, a file view in Propeller can define search conditions on top of any combination of in-group indices or define a set of files explicitly. Particularly, the POSIX layer will rely on *views* to

offer the "directory" concept in the hierarchical file systems. For instance, a POSIX directory "/org.propeller.src/?ext:.o" might conceptually represent a query "SELECT * FROM files WHERE groupid = 'org.propeller.src' and extension = '.o' "

3.3 Implementation Issues

Having described the overall architecture of the proposed searchable file system and its metadata organization Propeller, the focus of this paper, we are ready to present in this section several important implementation techniques aimed at enhancing the performance and the flexibility of Propeller.

Lazy Indexing. As we mentioned earlier, the file-search requests issued by typical workloads account for only a small fraction of all I/O requests. Nevertheless, the cost of re-indexing the indices to keep them constantly up-to-date is expensive even when the indices are very small. Therefore, inspired by and analogous to the concept of lazy consistency in the distributed shared memory model, we introduce an optimization technique, called *Lazy Indexing* to significantly reduce the re-indexing overhead. To implement lazy indexing, we design a write buffer for each index to log the updates before their commits. All changes captured in the write buffer are only committed to the index when: 1) after a pre-determined time interval from the last commit, also called a "timeout" or 2) upon the arrival of the next search request, whichever comes first. As a result, the lazy-indexing technique hides the re-indexing latency from regular file IOs without sacrificing the consistency guarantee that the most up-to-date and accurate file set be returned to each search request.

Search Hooks. Search Hooks offer a flexible mechanism to allow applications to feed the file index when they are manipulating data, which is the *inline index* approach mentioned in Section 2.2. In addition to the performance advantages of inline index approach, search hooks offer the feasibility of capturing the application execution semantics that are useful in enhancing file search [40], which the existing offline file-search engines are unable to.

Although in our current implementation we have built the Propeller prototype on a single machine, it will be straightforward to extend the Propeller model to a clustered or distributed environment. The *File System Threads* can be implemented as distributed processes to achieve load balancing, in addition to exploiting parallelisms. The semantic information exposed by the file-grouping methods will likely be conducive to optimizing data placement in a distributed or Cloud computing environment.

3.4 Limitations and Future Work

To respond to the emerging and increasing needs of efficient file-search functionalities in file systems, we are targeting an envisioned searchable file system and the corresponding new namespace scheme. Propeller, as *the metadata organization* of such an envisioned searchable file system, is our first but significant step toward realizing this searchable file system. We approach this final goal by evaluating the performance tradeoffs among several core functionalities: direct file lookup, maintaining up-to-date file indices, and performing file search. While Propeller has been designed and prototyped, a FUSE-based searchable file system prototype that sits on top of Propeller is currently under active development. As a result, there are some open problems left as our future work to investigate: 1) What would be an elegant and effective file-search API to systems and applications? Should it be incorporated within the POSIX API or not? 2) What is the overhead and accuracy tradeoff between capturing file semantics dynamically (e.g., monitoring I/O activities) and statically (e.g., using pre-defined groups); 3) Is there any other dynamic methodology besides monitoring the I/O activities that can be used to form the semantic file groups; and 4) Under what circumstance will the GMT be the system bottleneck? We plan to fully investigate the above problems during the development of the searchable file system.

4. PROOF-OF-CONCEPT EVALUATION

We evaluate the Propeller prototype using representative datasets and workloads. In the experiments, we examine the performance metrics in terms of metadata-update performance, file-query performance, query accuracy, system overhead and scalability in order to assess how effectively our Propeller metadata organization will likely perform as the core component of the envisioned searchable file system, as well as the prospect of such a searchable file system's effectiveness in comparison to the existing file-search solutions. Without loss of generality, in all the following experiments, every semantic file group maintains three intra-group indices: a B+tree for Group File Table, a hash table for fast direct access, and a K-D tree for attribute index, to prove the concept of Propeller.

The Propeller Prototype. It was written in C++ as a user-space program on Linux (6000 SLOC). We use Berkeley DB for the B+Tree, hash table and inverted list implementations and use libkdtree++ [26] for the K-D tree implementation. The keywords of each file group are extracted from the group's full-path name.

Experimental Setup

	MySQL	Ext4
Description	Centralized Offline Index	State-of-the-art I/O Performance
Data structure	Centralized B+Tree	Hierarchical directory tree
Realtime Update	No	Yes
Fast file-search	Yes	No
Inaccurate Results	Maybe	No
Require Crawling	Yes	No
Similar Systems	Google Desktop Search [21], Spotlight [8], WinFS [37]	Ext3, NTFS, ZFS, HFS+ etc.
What to compare	Realtime file-index, Scalability, Recall	Overheads to the I/O critical path

Table 6: Baseline Characteristics (MySQL and Ext4)

We run our experiments on a machine with an Intel Quad-Core Xeon X3440 (4 Cores, 8M Cache, 2.53GHz) CPU with 16GB RAM running Ubuntu Linux Server 10.10. All metadata files are stored as regular files on an Ext4 partition on two Hitachi HDS72202 2TB, 7200 RPM and 32MB-Cache hard drives configured as RAID-0. Due to the lack of a query API in the Google Desktop Search Linux version, we are not able to directly compare Propeller with Google Desktop. Instead, we compare the performance of Propeller with MySQL (ver. 5.1.49), which is used to reflect the perfor-

mance characteristics of centralized offline file-search solutions (including Google Desktop). We also compare Propeller with Ext4, which represents the typical I/O performance of the state-of-the-art file systems. Table 6 describes the detailed characteristics of the two baselines and the production system we intend for them to represent (directly or indirectly). The database is located on the same partition to obtain results for fair comparisons. We configured a 512MB RAM as the query cache for MySQL and built an index based on the file full-paths. By building an index to it, the MySQL solution is optimized and its performance is improved by up to 3 orders of magnitude over the naÃŕve MySQL solution.

4.1 Performance of File-Grouping

Our first set of experiments evaluate the performance overheads incurred by system-level file-search functionalities to normal file system I/O operations.

Dataset (C-files/all-files)	Ext4	Propeller 1	Propeller 2	MySQL
Linux Kernel (27496/36013)	27.87	26.18 (93.9%)	32.28 (116%)	1668.91 (5984.9%)
PostgreSQL (1526/4089)	3.98	2.83 (71.1%)	2.90 (72.8%)	238.71 (5997.7%)
Git (414/2131)	1.50	1.11 (74.1%)	1.22 (81.3%)	89.12 (5941.3%)
Apache Httpd (372/3015)	3.24	2.58 (79.7%)	2.355 (72.7%)	85.69 (2644.8%)

Table 7: Group-Creation Time (Seconds): Propeller 1: creation of groups each with an attribute index; Propeller 2: creation of groups each with an attribute index and a content index. The percentage value of each result represents the speed over the Ext4 execution.

Group Creation. In this evaluation, we choose four representative C-language-based projects (Linux Kernel, PostgreSQL, Git and Apache Httpd) to assess the overheads of inline file-indexing, because they represent a wide range of file sizes and directory structures. We compare the group creation times to two scenarios of normal file system I/O operations: 1) copying data by using the command "cp -r" on the Ext4 file system; and 2) Using MySQL to perform offline file-indexing, which includes the three steps of (a) copying data on Ext4, (b) extracting metadata (e.g., inode attributes, C function names etc.), and (c) importing these metadata into MySQL. To perform inline content-indexing, we implemented a simple C/C++-language parser to extract function names when Propeller is copying C/C++ source files. Moreover, all tests run on an existing 5-million-file namespace that contains 150-million keywords. Therefore, by comparing Propeller's group-creation time with the aforementioned two scenarios, we can evaluate the file-indexing overheads under a write-heavy, metadata-intensive workload that contains a large amount of file-allocation, file-write and attribute-setting operations. As shown in Table 7, where the results are normalized to the Ext4 execution times, Propeller adds less than 16% overhead when performing both attribute and content indexing, to the normal I/O operations. In most executions, even with the costly content extraction in content indexing, Propeller still performs up to 28.9% faster than the Ext4 execution. Considering that Propeller is a user-level program that runs on top of an Ext4 partition and maintains four indices (Group File Table, naming index, attribute index and content index), this performance boost can be attributed mainly to Propeller's intra-group flat namespace that eliminates the costly lookup operations in the hierarchical namespace, which we will examine in more detail next. During the MySQL execution, we observed two major performance overheads: 1) the offline metadata extraction process that must re-open and read the files and thus generates significant IOs; and 2) the expensive insertions of records into a large MySQL database. As a result, the MySQL solution is up to 82× slower than Propeller in the 5-million-file namespace and it is reasonable to expect this performance gap to be widened with the scaling of file systems. In summary, Propeller offers a highly scalable and light-weight inline file-indexing capability to the file systems, and the *Lazy-Indexing* technique effectively hides the index latency from the normal I/O operations.

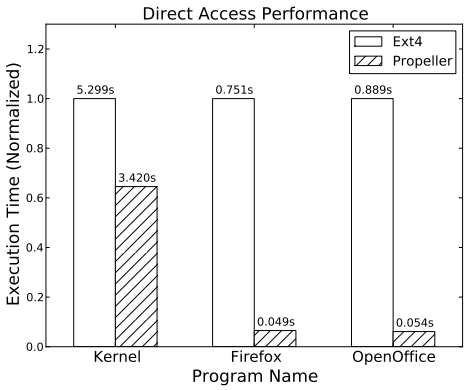


Figure 9: Direct Access Times

Direct Access. To evaluate the direct-access performance of Propeller, we feed three real-world execution-driven traces recorded from the executions on the three datasets of 1) *Kernel*(9402 files/1741338 accesses) (i.e., building the Linux kernel), 2) *Firefox*(2581 files/8836 accesses) (i.e., opening Firefox and browsing the www.nytimes.com website), and 3) *OpenOffice*(3000 files/11297 accesses) (i.e., opening OpenOffice and editing a new document). Then we issue Lookup operations to the recorded files as fast as possible on Ext4 and Propeller to assess the file-locating acceleration brought by Propeller. The performance results are normalized to the Ext4 execution times as well, as shown in Figure 9. The results show that the direct-access performance in Propeller is 1.53 $\sim$ 16.6 times faster than that in the Ext4 file system. Note that in the Linux-kernel building case, the same files and sub-directories are repeatedly accessed during the entire execution, thus the directory cache has been properly warmed up, which dramatically improves its Lookup performance on Ext4. However, the direct-access performance in Propeller is still superior to Ext4 in all cases. The reason behind this performance advantage of Propeller is that each particular program execution in Ext4 must issue multiple directory Lookup requests, which introduce numerous costly disk IOs to read the corresponding directory files. Whereas, in the Propeller case, as a result of the *Semantic File Group* being defined by the I/O boundaries of the particular executions, exactly three disk IOs, one to read the *Group File Table* and two to read file indices, will be required to serve most of the lookup operations in the execution. Furthermore, the intra-group naming index provides an $O(1)$ time overhead for in-group file lookups, which is more efficient than the hash-indexed directory structure in Ext4 that only

accelerates the lookup speed for each individual directory.

In summary, we conclude that Propeller adds a very small amount of overhead to write-intensive operations, while providing much better direct-access performances to read intensive workloads than the Ext4 file system. Considering that Propeller offers flexible and real-time file-search functionalities, the small I/O overheads are acceptable. It is noteworthy that, unlike the centralized file-index data structures used in the existing file-search solutions, the above performance results are scalable due to the fact that all operations take place within the groups locally, a complexity that is only proportional to the scale of each group instead of the scale of the entire file system. This performance result supports our argument that it is worthwhile to change the way applications interact with file systems. Moreover, the above results reveal the potential opportunities for optimizing the disk-layout and prefetching algorithms by exploring the semantics of execution that has been captured and stored in the *Semantic File Groups*.

4.2 Performance of Parallel File Index

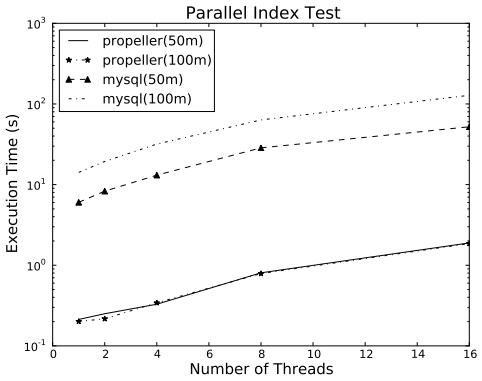


Figure 10: Parallel Index Times on 50-million-file and 100-million-file datasets

We next turn our attention to assessing the parallel file-index performance of Propeller. We compare the Propeller results to those of MySQL that represents a typical workload for current RDBMS-based file-search engines, to show the advantages of the partitioning and the parallel file-index techniques. We start by feeding a sequence of parallel file updates to both Propeller and MySQL on different scales of datasets (50-million-file and 100-million-file). In this experiment, we create from 1 to 16 threads to issue 10,000 update requests to Propeller and MySQL, respectively, and measure the execution times for all threads accomplishing their IOs. In the Propeller experiment, each thread issues IOs within one individual semantic file group. In the MySQL experiment, each thread issues IOs to file entries with the same group id. We have observed that the experimental results are not sensitive to the group size, thus we only present the results for the 1000-file-group experiment. As shown in Figure 10, the file-indexing performance of Propeller is $30 \sim 60$ times better than MySQL. Note that, in both data sets, the file-indexing performance of Propeller is similar, while in the MySQL case, it degrades significantly ($2\times$) from the 50-million-file dataset to the 100-million-file dataset. Thus this experimental result indicates that the Propeller file-indexing performance is scalable, because it is only determined by the size of each group, but not the scale of the entire file system. Furthermore, the reason for the performance decrease in the Propeller experiment is that the user-level Propeller threads issue parallel I/O requests to different files on the underlying Ext4 file system, resulting in mostly small and random IOs that are known to decrease performance for the HDD-based storage system and can make it a performance bottleneck. It is also noteworthy that the Propeller file-indexing performance degrades slower when there are fewer than 4 threads. This is because our current testbed has only four hyper-threading cores and, for each active semantic file group, there are currently two threads processes updates, a foreground thread that updates the Group File Table and the naming index table, and a background thread that carries out the lazy indexing scheme. As a result, there will be increasing competition for the quad-core processor as the number of threads grows beyond four, which can diminish the overall I/O performance.

4.3 Mixed Workloads

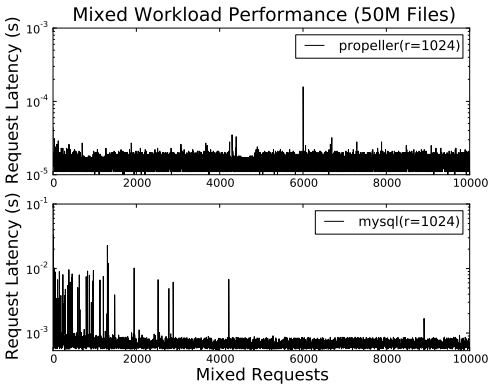


Figure 11: Mixed Workload (50m Files)

As shown in Section 4.1, the *Lazy Indexing* technique effectively hides the file-indexing latency from the normal file IOs. However, this technique increases the latency of the file-search requests, because semantic file groups must commit all modifications into the intra-group file indices before issuing a file search in order to guarantee the consistency of file-search results. Thus it is desirable to mix I/O operations with file-search requests in the I/O workloads to obtain a deeper understanding of the Propeller performance. Unfortunately, to the best of our knowledge, none of the existing I/O traces has combined file-search patterns, since the search functionality by and large is non-existent in current production file systems. To explicitly show the impact of file-search requests, we feed a synthetic workload including 10,000 updates combined with file-attribute-search requests, to one semantic file group (1,000 files) on a 50-million-file dataset on both Propeller and MySQL, where there is one file-search request for every 1,024 updates. And the background re-indexing is trigged by every 500 updates to simulate the "timeout" effect in the lazy-indexing technique. As shown in Figure 11, the average latency of normal metadata update and file re-indexing operations in Propeller ($15.6\mu s$) is $250\times$ faster than that in MySQL ($3,980.9\mu s$). This result proves that, with the Semantic File Group, the performance penalty of synchronous commit modifications before each file-search is very small due to the significantly reduced scale of a file group, while in the MySQL solution, the update operations occur in the global namespace, which results in an extremely high latency. Furthermore, the Lazy-Indexing technique not only hides most of the re-indexing overheads

from the normal I/O operations but also reduces the number of modifications to be merged for the file-search requests due to the "background" merges triggered by the "timeout" mechanism (recall "Lazy Indexing" in Section 3.3). In summary, with *Semantic File Group* and *Lazy Indexing*, Propeller guarantees the consistency of file-search results with very little overheads.

4.4 Scalable Global File Search

Name	Description	# of files
Safari	Web browser	3908
Firefox	Web browser	264
Microsoft Word	Word processor	1251
VMWare Image	Virtual machine	1244
Boost	C++ library	29664
libkdtree++	C++ K-D tree	234
django	Python web framework	7119
Photos	Photo library	30398
Musics	Music library	3083
Papers library	Documents	589

Table 8: Sample File Groups

Propeller partitions the namespace into *Semantic File Groups* (SFG) based on the file-access patterns. Unfortunately, none of the current publicly accessible file-system snapshots [4, 5] is usable for building such a namespace. Due to security and privacy concerns, these publicly accessible static snapshots have their file names encrypted and consequently lose the semantic-correlation information that can otherwise be partially mined from file names. Thus, we choose a set of well-known applications and open-source projects, as shown in Table 8 as sample semantic file groups, because they are representative of typical real-world workloads and publicly accessible. The group sample set also includes music albums, personal photo collections, proceedings and presentation slides of several technical conferences. To obtain a dataset of a desired scale, we duplicate these samples with an appropriate scaling factor, with the important exception that each "duplicated" semantic file group is given a *different group ID*. This implies that each namespace is also scaled up accordingly since files in the newly duplicated groups each have a different full-path with a distinct group ID. Additionally, the keywords of each group are extracted from the directory names from all sub-directories of the sample set.

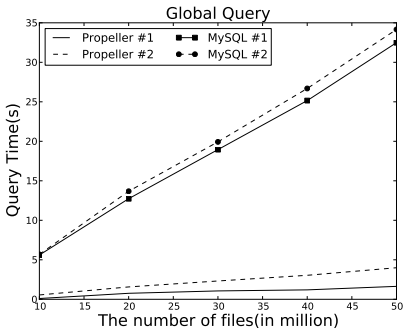


Figure 12: Global File Search: Query #1: size > 1 GB & mtime < 1day ; Query #2: keyword "**firefox**" & mtime < 1 week.

We compare the global file-search performances of Propeller and MySQL on the synthetically scaled-up namespaces. The namespaces are assumed static in order to eliminate the impact of continuous metadata updates. We define two queries (Figure 12) to evaluate the global-search performances of the two systems. The results shown in Figure 12 indicate that these two queries in Propeller are on average 9.0 and 26.3 times faster than in MySQL, respectively.

In our current Propeller prototype implementation, each global file search must load the attribute index files of all query-targeted *semantic file groups* into the memory before issuing *intra-group* attribute-based queries. Thus the main performance cost for the global file-search is the I/O operations that load the targeted file indices from the disks. As a result, the total amount of metadata, approximately estimated to be proportional to the number of *semantic file groups*, must be limited to achieve a scalable performance. Due to the limitations of our synthetically generated namespaces, the global file-search performance suffers from the homogeneity of the namespace in that the search requests are issued to duplicated groups (i.e., identical file groups with different group IDs). This has an effect of underestimating the real performance and scalability of Propeller's global file-search. This is because, in a real environment, we expect the file groups to be more highly diversified (i.e., heterogeneous) and concerns of user privacy will likely limit the number of file groups a global file-search request can access, thus significantly reducing the search space. Furthermore, an optimized *membership-marker* algorithm and more precise *keywords* will help further reduce the search space. For MySQL, since each SQL query is issued to the entire namespace and, typically for a multi-dimensional attribute-based file-search, multi-column comparisons are required, its global-search performance will likely further degrade with more realistic and scaled up datasets.

4.5 Query Accuracy

We now measure the accuracy of file-search results from Propeller (inline model) and the MySQL solution (offline model). In the MySQL solution, we use *inotify* to capture the file changes and then insert these changes into MySQL as fast as possible. The experiments run on the namespaces of two different scales, with 10 million files and 25 million files respectively. The Propeller namespaces are constituted by the 1000-file scale semantic file groups, while in the MySQL solution, we use the directory that contains 1,000 files, to represent the corresponding concept of a "file group". Thus, there are 10,000 and 25,000 file groups in each of the two namespaces. We issue 20,000 IOs to 20 randomly chosen semantic file groups in Propeller and 20 corresponding directories in the MySQL namespace. Then we perform a file-search operation that answers the question: "SELECT * FROM files WHERE mtime > now - 1 minute". Furthermore, we apply the "Recall" notion [2] from the Information Retrieval field to quantitatively evaluate the correctness of the file-search results, where the "Recall" notion in our scenario is defined as follows:

$$recall = \frac{|\{\text{changed files}\}| \cap |\{\text{search results}\}|}{|\{\text{changed files}\}|}$$

Because of the crawling delays, the recall of file-search results is a function of the crawling time in the MySQL solution. In other words, the more time is allowed to elapse the more files will have their last changes accurately reflected in the file index by the crawling process, thus the more accurate the file-search results. Therefore, to evaluate the impact of the crawling delay window on the file-search accuracy, we continuously perform SQL queries on MySQL with an inter-

	Recall Accuracy(Time)				
Nth Search	1st	2nd	3rd	4th	5th
Propeller (10m-file)	100% (0.152s)	100%	100%	100%	100%
Propeller (25m-file)	100% (0.178s)	100%	100%	100%	100%
MySQL (10m-file)	0.01% (4.3s)	38.1% (7.7s)	80.7% (11.1s)	87.3% (14.5s)	87.3% (15.7s)
MySQL (25m-file)	0.25% (9.5s)	35.4% (16.5)	72.7% (23.7s)	87.92% (30.4s)	87.92% (31.8s)

Table 9: Recall as a function of time(s)

val of 1 second, until the recall of the results becomes stable. Table 9 shows the recall values of file-search results of each query and its completion time for Propeller and MySQL in two different namespaces. Propeller returns a 100% accurate result from the first search since the *Lazy Indexing* mechanism ensures that Propeller flushes all changes into file indices before issuing a file-search to guarantee the return of the most up-to-date results. Whereas, the MySQL solution needs $15 \sim 32$s to capture the changes, during which it induces notable errors on file-search results. It is worth noting that the MySQL solutions in both namespaces fail to catch up with all the file modifications ($recall = 100\%$) because of the buffer overflow caused by the fast incoming file-change events as mentioned in Section 2.2. In summary, Propeller is able to completely avoid the inaccuracy-inducing crawling window of the existing file-search approaches and further offer the system-level search facilities to the systems and the applications.

5. CONCLUSION

This paper presents the metadata organization, called Propeller, for highly scalable searchable file system. By applying a novel file-clustering mechanism, called *Semantic File Grouping*, and several optimization techniques, Propeller offers faster direct file access than the Ext4 file system, and outperforms an optimized MySQL solution by $2 \sim 3$ orders of magnitude in the file-index and file-search performance. Moreover, Propeller guarantees the consistency of file-search results with very small overhead to normal file IOs. By invesgating the performance impacts of Propeller metadata scheme, we have obtained more insights of the design principles for our ongoing searchable file system design.

6. REFERENCES

[1] iOS - The Application Sandbox. `http://developer.apple.com`.

[2] Precision and recall. `http://en.wikipedia.org/wiki/Precision_and_recall`.

[3] Abadi, D. J., Madden, S. R., and Hachem, N. Column-stores vs. row-stores: how different are they really? In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2008), SIGMOD '08, ACM, pp. 967–980.

[4] Agrawal, N., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Generating realistic impressions for file-system benchmarking. In *Proccedings of the 7th conference on File and storage technologies* (Berkeley, CA, USA, 2009), USENIX Association, pp. 125–138.

[5] Agrawal, N., Bolosky, W. J., Douceur, J. R., and Lorch, J. R. A five-year study of file-system metadata. In *In Proceedings of the 5th USENIX Conference on File and Storage Technologies. USENIX Association* (2007).

[6] Apple Inc. File system events programming guide.

[7] Apple Inc. iPhoto. `http://www.apple.com/ilife/iphoto/`.

[8] Apple Inc. Spotlight. `http://www.apple.com/macosx/what-is-macosx/spotlight.html`.

[9] Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., and Singhania, A. The multikernel: a new os architecture for scalable multicore systems. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (New York, NY, USA, 2009), ACM, pp. 29–44.

[10] Beaver, D., Kumar, S., Li, H. C., Sobel, J., and Vajgel, P. Finding a needle in haystack: Facebook's photo storage. In *Proceddings of the 9th USENIX Symposium on Operating Systems Design and Implementation* (2010), OSDI '10.

[11] Beitzel, S. M., Jensen, E. C., Chowdhury, A., Grossman, D., and Frieder, O. Hourly analysis of a very large topically categorized web query log. In *SIGIR '04: Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval* (New York, NY, USA, 2004), ACM, pp. 321–328.

[12] Bentley, J. L. Multidimensional binary search trees used for associative searching. *Commun. ACM 18*, 9 (September 1975), 509–517.

[13] Boyd-Wickizer, S., Clements, A. T., Mao, Y., Pesterev, A., Kaashoek, M. F., Morris, R., and Zeldovich, N. An Analysis of Linux Scalability to Many Cores. In *OSDI 2010: Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation*.

[14] Colmenares, J. A., Bird, S., Cook, H., Pearce, P., Zhu, D., Shalf, J., Hofmeyr, S., AsanoviÄĞ, K., and Kubiatowicz, J. Resource management in the tessellation manycore os. In *Proceddings of the 2nd USENIX Workshop on Hot Topics in Parallelism* (2010), HotPar '10.

[15] Comer, D. Ubiquitous b-tree. *ACM Comput. Surv. 11*, 2 (1979), 121–137.

[16] Daley, R. C., and Neumann, P. G. A general-purpose file system for secondary storage. In *AFIPS '65 (Fall, part I): Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I* (New York, NY, USA, 1965), ACM, pp. 213–229.

[17] Ellard, D., Ledlie, J., Malkani, P., and Seltzer, M. Passive nfs tracing of email and research workloads. In *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2003), USENIX Association, pp. 203–216.

[18] Fernandez-Sanguino, J. The debian gnu/linux faq chapter 8 - the debian package management tools.

http://www.debian.org/doc/FAQ/ch-pkgtools.en.html.

[19] Ghemawat, S., Gobioff, H., and Leung, S.-T. The google file system. *SIGOPS Oper. Syst. Rev. 37*, 5 (2003), 29–43.

[20] Gifford, D. K., Jouvelot, P., Sheldon, M. A., and O'Toole, Jr., J. W. Semantic file systems. In *SOSP '91: Proceedings of the thirteenth ACM symposium on Operating systems principles* (New York, NY, USA, 1991), ACM, pp. 16–25.

[21] Google.com. Google Desktop Search. http://desktop.google.com/.

[22] Google.com. Google Search Applicance. http://www.google.com/enterprise/search/gsa.html.

[23] Gopal, B., and Manber, U. Integrating content-based access mechanisms with hierarchical file systems. In *Proceedings of the third symposium on Operating systems design and implementation* (Berkeley, CA, USA, 1999), OSDI '99, USENIX Association, pp. 265–278.

[24] Gray, J., Liu, D. T., Nieto-Santisteban, M., Szalay, A., DeWitt, D. J., and Heber, G. Scientific data management in the coming decade. *SIGMOD Rec. 34*, 4 (2005), 34–41.

[25] Guttman, A. R-trees: a dynamic index structure for spatial searching. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 1984), ACM, pp. 47–57.

[26] Harris, P. libkdtree++. http://libkdtree.alioth.debian.org/.

[27] Held, J., Bautista, J., and Koehl, S. From a Few Cores to Many: A Tera-scale Computing Research Overview. Tech. rep., 2006.

[28] Hua, Y., Jiang, H., Zhu, Y., Feng, D., and Tian, L. Smartstore: a new metadata organization paradigm with semantic-awareness for next-generation file systems. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (New York, NY, USA, 2009), ACM, pp. 1–12.

[29] Huang, H. H., Zhang, N., Wang, W., Das, G., and Szalay, A. S. Just-in-time analytics on large file systems. In *FAST '11: Proceedings of the 5th USENIX conference on File and Storage Technologies* (2011), USENIX Association.

[30] John, F. G., Chute, C., Manfrediz, A., Minton, S., Reinsel, D., Schelichting, W., and Toncheva, A. The diverse and exploding digital universe: An update forecast of worldwide information growth through 2011. Tech. rep., 2008.

[31] Lei, H., and Duchamp, D. An analytical approach to file prefetching. In *Proceedings of the annual conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 1997), USENIX Association, pp. 21–21.

[32] Leung, A., Shao, M., Bisson, T., Pasupathy, S., and Miller, E. L. Spyglass: Fast, scalable metadata search for large-scale storage systems.

[33] Lyman, P., and Varian, H. R. How much information. http://www2.sims.berkeley.edu/research/projects/how-much-info-2003/.

[34] Margo, S., and Nicholas, M. Hierarchical file systems are dead. In *HotOS '09: 12th Workshop on Hot Topics in Operating systems* (2009), USENIX.

[35] McCutchan, J., Love, R., and Griffis, A. Inotify - get your file system supervised. http://inotify.aiken.cz/.

[36] Microsoft. Windows Search. http://www.microsoft.com/windows/products/winfamily/desktopsearch/default.mspx.

[37] Microsoft. WinFS: Windows Future Storage. http://en.wikipedia.org/wiki/WinFS.

[38] Patil, S., and Gibson, G. Scale and concurrency of GIGA+: File system directories with millions of files. In *FAST '11: Proceedings of the 5th USENIX conference on File and Storage Technologies* (2011), USENIX Association.

[39] Russinovich, M. E., Solomon, D. A., and Ionescu, A. *Windows Internals, Fifth Edition.* Microsoft Press, 2009.

[40] Soules, C. A. N., and Ganger, G. R. Connections: using context to enhance file search. *SIGOPS Oper. Syst. Rev. 39*, 5 (2005), 119–132.

[41] Stonebraker, M., and Cetintemel, U. "One Size Fits All": An idea whose time has come and gone. *Data Engineering, International Conference on 0* (2005), 2–11.

[42] Stonebraker, M., Madden, S., Abadi, D. J., Harizopoulos, S., Hachem, N., and Helland, P. The end of an architectural era: (it's time for a complete rewrite). In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases* (2007), VLDB Endowment, pp. 1150–1160.

[43] Tan, Y., Jiang, H., Feng, D., Tian, L., Yan, Z., and Zhou, G. Sam: A semantic-aware multi-tiered source de-duplication framework for cloud backup. *Parallel Processing, International Conference on 0* (2010), 614–623.

[44] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E., and Maltzahn, C. Ceph: a scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 307–320.

[45] Welch, B., Unangst, M., Abbasi, Z., Gibson, G., Mueller, B., Small, J., Zelenka, J., and Zhou, B. Scalable performance of the panasas parallel file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2008), FAST'08, USENIX Association, pp. 2:1–2:17.

[46] Wentzlaff, D., and Agarwal, A. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev. 43* (April 2009), 76–85.

[47] Wickizer, S. B., Chen, H., Chen, R., Mao, Y., Kaashoek, F., Morris, R., Pesterev, A., Stein, L., Wu, M., Dai, Y., Zhang, Y., and Zhang, Z. Corey: An Operating System for Many Cores. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)* (San Diego, California, December 2008).