Frog: A Framework for Context-Based File Systems

JI ZHANG, Auburn University
XUNFEI JIANG, Earlham College
XIAO QIN and WEI-SHINN KU, Auburn University
MOHAMMED I. ALGHAMDI, Al-Baha University

This article presents a framework, Frog, for Context-Based File Systems (CBFSs) that aim at simplifying the development of context-based file systems and applications. Unlike existing informed-based context-aware systems, Frog is a unifying informed-based framework that abstracts context-specific solutions as views, allowing applications to make view selections according to application behaviors. The framework can not only eliminate overheads induced by traditional context analysis, but also simplify the interactions between the context-based file systems and applications. Rather than propagating data through solution-specific interfaces, views in Frog can be selected by inserting their names in file path strings. With Frog in place, programmers can migrate an application from one solution to another by switching among views rather than changing programming interfaces. Since the data consistency issues are automatically enforced by the framework, file-system developers can focus their attention on context-specific solutions. We implement two prototypes to demonstrate the strengths and overheads of our design. Inspired by an observation that there are more than 50% of small files (<4KB) in a file system, we create a Bi-context Archiving Virtual File System (BAVFS) that utilizes conservative and aggressive prefetching for the contexts of random and sequential reads. To improve the performance of random read-and-write operations, the Bi-context Hybrid Virtual File System (BHVFS) combines the *update-in-place* and *update-out-of-place* solutions for read-intensive and write-intensive contexts. Our experimental results show that the benefits of Frog-based CBFSs outweigh the overheads introduced by integrating multiple context-specific solutions.

Categories and Subject Descriptors: H.2 [**Information Systems**]: Information Storage Systems

General Terms: Performance of Systems

Additional Key Words and Phrases: File systems, context aware, multiview

ACM Reference Format:
Ji Zhang, Xunfei Jiang, Xiao Qin, Wei-Shinn Ku, and Mohammed I. Alghamdi. 2015. Frog: A framework for context-based file systems. ACM Trans. Storage 11, 3, Article 11 (July 2015), 28 pages.
DOI: http://dx.doi.org/10.1145/2720022

1. INTRODUCTION

Context-aware computing allows modern storage systems to adjust I/O schemes and mechanisms according to specific contexts (e.g., read-intensive contexts). Taking context information into account enables storage systems to dynamically optimize I/O performance, energy efficiency, and so forth. For instance, quFiles create multiple

The work reported in this article was supported by the US National Science Foundation under Grants CCF-0845257(CAREER), CNS-0757778 (CSR), CCF-0742187 (CPA), CNS-0917137 (CSR), CNS-0831502 (CyberTrust), CNS-0855251 (CRI), and OCI-0753305 (CITEAM).
Authors' addresses: J. Zhang, X. Qin, and W.-S. Ku, Department of Computer Science and Software Engineering, 3101 Shelby Center, Auburn University, AL 36849-5347; emails: {jizhang, xqin, weishinn}@auburn.edu; X. Jiang, Department of Computer Science, Earlham College, 801 National Road West, Richmond, IN 47374-4095; email: jiangxu@earlham.edu; M. I. Alghamdi, Department of Computer Science, Al-Baha University, Al-Baha City, Kingdom of Saudi Arabia; email: mialmushilah@bu.edu.sa.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2015 ACM 1553-3077/2015/07-ART11 $15.00
DOI: http://dx.doi.org/10.1145/2720022

physical representations of files for data-specific contexts [Veeraraghavan et al. 2010]. Prefetching can be optimized (e.g., adjusting the number of prefetched blocks) by identifying and modeling future access patterns in a real-time manner [Soundararajan et al. 2008]. When it comes to energy saving, a server may have two operating contexts: a full-utilization mode during daytime and an energy-saving mode during nights [Leverich and Kozyrakis 2010]. To reduce SSD maintenance costs, an HDD–SSD hybrid file system may have two contexts: a performance critical context and an SSD lifetime-extending context [Soundararajan et al. 2010]. Both contexts can be implemented on SSDs and HDDs.

Context-aware techniques have been investigated in previous studies. However, the following four problems make the existing context-aware techniques impractical in file systems. First, high accuracy of context identification is critical to context-aware systems. Various heuristic and informed approaches have been proposed to identify contexts in particular scenarios. Appropriately selecting an approach for a context-aware file system requires analysis expertise and years of experience. Second, existing informed-based approaches expose a rich set of interfaces to applications [Soundararajan et al. 2008; Patterson et al. 1995]. Migrating them from one approach to another is challenging. Third, existing heuristic-based approaches inevitably trigger false positives, which may have negative impacts on I/O operations [Kroeger and Long 1999, 2001; Liang et al. 2007]. Last, the heuristic-based approaches suffer from back-end analysis, which is likely to incur unnecessary overheads during I/O operations in file systems.

In this article, we propose a framework, Frog, for Context-Based File Systems (CBFSs) that encapsulates a number of solutions, which are dedicatedly designed for particular contexts. The context-specific solutions are abstracted as views in the internal representations of Frog-based CBFSs. Since the views are independent of each other, a new context can be supported in CBFSs by creating a new view. The Frog-based CBFSs interact with applications through virtual file system (VFS) interfaces.

Frog is a unifying informed-based framework for statically mapping contexts to appropriate solutions in CBFSs. The framework simplifies the interfaces between file systems and applications. Rather than propagating hints through a dedicated interface (i.e., *ioctl* [Patterson et al. 1995] or newly developed *ctx_action* [Soundararajan et al. 2008]), the framework allows applications to choose contexts by inserting view names in file-path strings. Without the knowledge of extra I/O interfaces, application developers are able to write view-aware codes.

Frog does not dynamically identify any context based on hints, thereby avoiding overheads of back-end analysis and shortening I/O response times. The contexts maintained in Frog-based CBFSs are configured offline; view-aware applications are in charge of context selections. The performance of programs running in the CBFSs are automatically optimized by appropriate context-specific solutions implemented in the form of views. Frog takes care of the details of integrating multiple solutions, keeping consistency among views (see Section 3.7). This allows system developers to focus on context-specific solutions without having to pay attention to the integration issues.

Frog is compatible with existing nonview-aware applications. The compatibility is enforced by setting a default view for nonview-aware applications that generally do not represent any specific context. Moreover, the default view is an important abstraction, on which the consistency mechanism relies.

Frog offers two approaches to integrating multiple context-specific solutions by considering metadata and physical data management. In the shared-data approach, physical data is shared among views; in the shared-nothing approach, each view maintains a replica of files. We also illustrate flexible interactions between Frog and applications in two concrete examples. Applications can access a file through different views based

on their contexts. In addition, an application can update a file through a view while another application can read the file through another view later.

We implement two prototypes of the shared-data and shared-nothing approaches for two practical scenarios, respectively. Motivated by the rapid growth of the number of files in a file system, in which more than 50% of the files are smaller than 4KB, we create a Bi-context Archiving Virtual File System (BAVFS) that organizes the small files in a big chunk to improve sequential reads with slight overhead on random reads. In the sequential view, BAVFS applies an aggressive prefetching technique under an assumption that all data will be loaded in sequence; the random view utilizes a conservative strategy, by which data will be loaded by demand. The second prototype, a Bi-context Hybrid Virtual File System (BHVFS), investigates random reads and writes in a combination of the *update-in-place* (e.g., fast file system, FFS [McKusick et al. 1984]) and an *update-out-of-place* (e.g., log-structured file system, LFS [Rosenblum and Ousterhout 1992]) technique. In particular, read operations are processed by the FFS view, because all data in a file is continuously managed in the FFS view; while write operations are processed by the LFS view, appending written data at the end of the log file.

The rest of the article is organized as follows. Basic concepts of context-based file systems are described in Section 2. Our design is illustrated in Section 3. Section 4 describes the prototype implementation. The experimental results are discussed in Section 5. Related works are presented in Section 6. Discussions about potential extensions of current work are provided in Section 7. Conclusions are given in Section 8.

2. CONTEXT-BASED FILE SYSTEMS

In this section, we first introduce basic concepts of context-based file systems (CBFSs). Then, we demonstrate the advantages of CBFS by comparing it with context-aware file systems. Last, we highlight how to use the Frog framework to support context-specific solutions and applications.

2.1. Basic Concepts

An I/O Context is a set of interrelated conditions under which I/O operations are performed. A file system may leverage access-pattern information exploited from an I/O context (i.e., conditions) to improve I/O performance. For instance, data can be prefetched before being required by applications in a sequential-read context.

An I/O Context-Specific Solution is a set of techniques or mechanisms deployed to maximize benefits (e.g., performance and energy efficiency) of I/O operations in a given context. Regularly, a context-specific solution that fits one context may be inadequate for another. For example, a file system may benefit from a prefetching solution in a sequential-read context, whereas the solution can cause performance degradation in a random-read context due to retrieving unused data.

A Context-Based File System (CBFS) is a file system that encapsulates an array of I/O context-specific solutions. A CBFS exposes the solutions to and receives context selections from applications. When a context is selected by an application, I/O operations will be processed by the context-specific solution of CBFS.

A Context-Based Application is a program that is able to identify its contexts, perceive contexts provided by systems, and select the best context leading to good performance. Differing from context-aware applications, context-based applications can interact with context-based systems to select contexts based on both I/O operations and available contexts offered by the systems.

2.2. Context-Aware vs. Context-Based File Systems

In this section, we show differences between CBFSs and context-aware file systems. Context-aware file systems can be roughly divided into two categories: heuristic-based

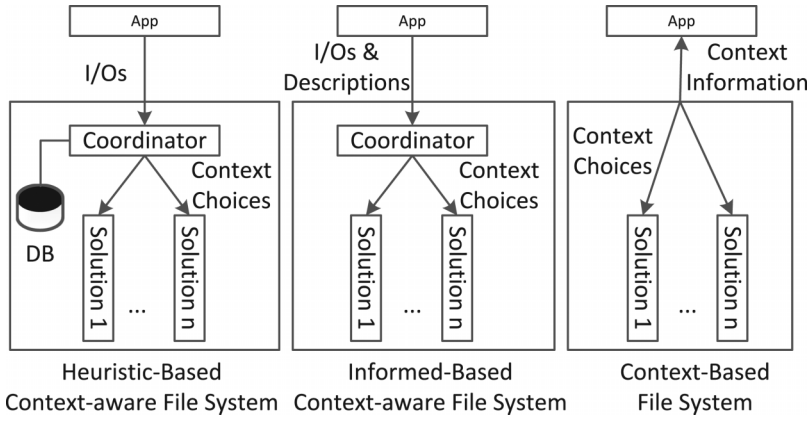

Fig. 1. A comparison of context-aware systems.

and informed-based systems. Figure 1 illustrates that three types of file systems encapsulate a set of context-specific solutions. The methods of context identification and solution selection are completely different among the three approaches.

In a heuristic-based file system, a coordinator keeps track of system-wide I/O operations in an internal database. By analyzing historical I/O accesses, the coordinator identifies a context for current I/O operations. The advantage of heuristic-based systems is that context processing is transparent to applications. Any application can be supported by heuristic-based systems without modifying the application's source code. For example, Partitioned Context Modeling (PCM) is a predictive prefetching algorithm that reliably predicts future I/O accesses by modeling historical accesses [Kroeger and Long 2001]. The method is transparent to applications, because it is seamlessly integrated with file systems. On the other hand, heuristic-based systems have three major drawbacks. First, maintaining I/O records increases the complexity of system design, and incurs extra costs on behavior analysis. Second, accuracy of context selections relies heavily on the analysis. Because applications may exhibit different I/O patterns, recording and analyzing their behaviors may have negative impacts on context identification. Third, applications with random behaviors (e.g., triggered by events) produce noises to the context identification process.

In an informed-based file system, a coordinator receives I/O operations with their description (e.g., represented by data structures) through the dedicated interfaces. For example, an access pattern can be described by a pair $< file, pattern >$, where $file$ specifies a file; *pattern* indicates an access pattern of future requests (e.g., a sequential read or reads described by an order list of $< offset, length >$ intervals [Patterson et al. 1995]. The context hint is disclosed through *ioctl* system call. Then, the context description is transformed into context choices based on particular rules. The informed-based systems do not suffer from the overhead of back-end analysis; the accuracy of context identification depends on applications. However, the systems have to incorporate additional interfaces to collect extra information from applications. Currently, the interfaces are abstracted in various ways. The lack of consistent information propagation prevents the informed approach from widespread deployment.

CBFSs aim to overcome the aforementioned problems in context-aware file systems. A CBFS can be envisioned as a special informed-based system without relying on context analysis. The context selections are propagated from applications. In addition, a CBFS provides unifying interfaces between file systems and context-based applications, meaning that the applications can easily and freely migrate among context-based file systems.

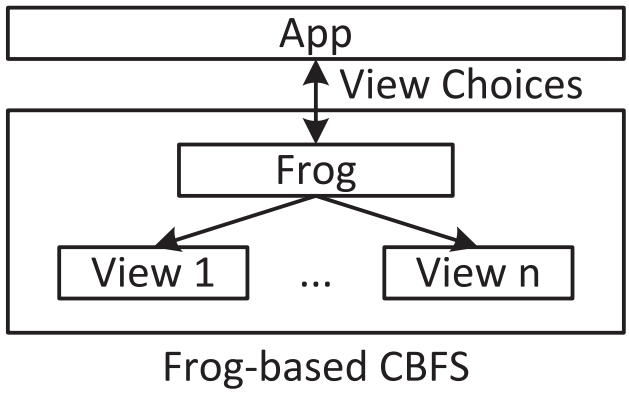

Fig. 2. The framework of a Frog-based CBFS.

Context-aware applications, being executed on informed-based file systems, can exchange context information with file systems. Context hints are generated by the context-aware applications, whereas the context selections are facilitated by the file systems. The context-based applications are the decision makers, which propagate the context selections to the file systems.

2.3. The Frog Framework for Context-Based File Systems

Encapsulating multiple context-specific solutions in a file system is challenging. The methods of organizing metadata and physical data in the solutions may be different, implying that a potential conflict may allow only one method to be applied due to the fact that most systems have only one copy of metadata and physical data. If a CBFS maintains multiple copies of data, each of which is managed by solutions individually, other issues are raised by data duplication. Take the consistency issue, for example; once data handled by a solution has been updated, other solutions have to change their managed data accordingly.

To simplify CBFS design, we propose a general framework, Frog, that provides (1) a unifying interaction mechanism between systems and applications, and (2) a solution encapsulation mechanism. Figure 2 shows that the context-specific solutions are abstracted as views in Frog. Frog is implemented as an intermediate layer between applications and views. Frog and applications exchange the view information through a unifying interface at runtime. Propagating a view from applications to Frog indicates that the corresponding context is selected by the applications. From the perspective of application development, the unifying interface significantly reduces the costs of solution migrations.

Importantly, Frog makes it easy to create individual spaces for context-specific solutions, each of which maintains data in its own space. Separating data spaces can allow a solution to be plugged in as if there were only one solution in the file system. Frog is in charge of solution maintenance issues such as consistency. Due to the independence among solutions, one or all of the solutions can have their own heuristic-based methods, for example,, tracking and analyzing I/O behaviors in specific contexts.

3. DESIGN

In Frog, we creatively introduce the view concept into file systems. Three essential attributes—file metadata management, physical data management, and I/O operations—are encapsulated in views. In this section, we present an overview of Frog as well as new concepts by illustrating Frog-based CBFSs. Next, due to the diversity

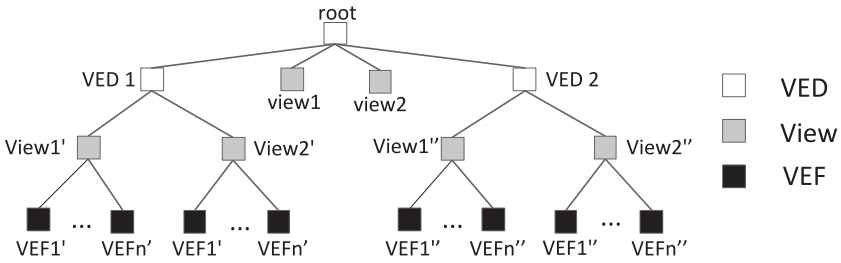

Fig. 3. Overview of a Frog-based CBFS.

of physical data management, two Frog-based CBFS designs are proposed. Finally, we discuss extra overheads introduced by Frog-based CBFSs.

Since our study mainly focuses on a framework design for CBFSs, we do not demonstrate context-specific solutions in details. In real CBFS development, existing solutions can be implemented in views. We will show two concrete prototypes as CBFS examples in the next section. In BAVFS, the implementation of aggressive prefetching can be found in Patterson et al. [1995]. The details of the FFS view and LFS view in BHVFS can be found in McKusick et al. [1984] and Rosenblum and Ousterhout [1992].

3.1. Overview

Figure 3 shows the hierarchical structure in the metadata space of Frog-based CBFS. Directories, views, and files are three types of abstractions represented by white, gray, and black boxes, respectively. These directories and files are called View-Enhanced Directories (VEDs) and View-Enhanced Files (VEFs). To concisely and accurately illustrate our design, we focus on the differences between Frog-based CBFSs and traditional file systems. We do not discuss the design principles of Frog-based CBFSs that are similar to those of the traditional file systems.

A Frog-based CBFS is organized in a tree structure, the root of which is an instance of a VED. A VED manages other VEDs and a set of views, but does not directly contain any VEF. The number of views in a VED can be dynamically configured. For simplicity, the number of views in the current design of our CBFS is fixed. Note that views under a VED are the instances of different view types. Figure 3 illustrates an example of two view types configured in a Frog-based CBFS. *view1* and *view2* are two instances of different types in the root node. *view1*, *view1'* and *view1"* are three instances of a view type, and *view2*, *view2'* and *view2"* are instances of another view type.

Views are an intermediate layer between VEDs and VEFs. A view is a container that organizes only VEFs rather than VEDs or other views. Thus, views do not recursively construct the hierarchy of the file systems. The views under a VED expose logically identical interfaces and metadata of VEFs (e.g., file names and file sizes) to applications; however, the implementation details (e.g., managing VEFs in a B-tree or a hash table) of views may vary.

VEFs are basic units referring to physical data on devices. In traditional file systems, a file is a logically unique unit that embraces only one copy of metadata and physical data. In Frog, by contrast, multiple physical representations are integrated in a file system. Associating each representation of a file with a unique file name in a directory is an intuitive approach to making all the representation accessible, but this method breaks the backward compatibility. Instead, each possible representation of a file is abstracted as a child in a view. Applications can access all the representations of a file by giving the corresponding view name in its file-path string. VEFs are always the leaf nodes in the tree structure.

In this article, we assume that the number of views in our framework is fixed for simplifying our presentation, but they can be dynamically changed in real applications. In general, the dynamic configuration (e.g., add or delete a view) raises an issue of how the data of a view can be copied from an existing view or removed from the Frog-based CBFS. The dynamic configuration module is similar to the data duplication management in distributed storage systems. When the number of replicas is changed, a distributed storage system will generate extra copies or remove redundant copies accordingly [Ghemawat et al. 2003; Shvachko et al. 2010]. The data duplication management methods are applicable to our Frog-based CBFS. One of the existing views can be locked temporarily during view creation and removal. Any implementation and optimization of these methods will be further investigated in our future study.

3.2. Operations

Now we discuss the operations of VEDs, views, and VEFs at the file-system level. We set up default views in VEDs to show compatibility with existing applications.

3.2.1. Frog-based CBFS Initialization/Finalization. Let us start the operation description with the initialization/finalization processes in Frog-based CBFSs when the file systems are mounted/unmounted at a point in the naming space. Before deploying a Frog-based CBFS, a format program should be executed to construct a super block, set up metadata and data partitions, and create a root VED and its subviews. When the CBFS is mounted, the root VED and subviews are loaded into the main memory. Then, the CBFS is ready to serve I/O requests. When the CBFS is unmounted, all metadata should be permanently serialized to disks.

3.2.2. VED Operations. VED operations supported by Frog-based CBFSs are directory creation, deletion, and listing. These operations differ from those of traditional file systems in that Frog must consider views in addition to directories. In the creation process, a new VED is created and appended to a child list of its parent. Views under the new VED are also generated. By default, an empty VED has two special directories (“.” and “..”) and a number of empty views. In the deletion process, a VED can be deleted if it does not contain any other VEDs (except for “.” and “..”) and all subviews are empty. Its subviews are automatically removed.

The directory listing operation (through *readdir* system call) will be discussed in the default view section (see Section 3.2.5), where we show how to make a trade-off between supporting compatibility for existing applications and exposing views to view-aware applications.

3.2.3. View Operations. In Frog, view supports creation, deletion, and listing operations. The listing operation can be individually issued, but the creation and deletion have to be committed by VED creation and deletion operations. The listing operation returns all the VEF names in the view.

The view names are predetermined by their types. For example, in Figure 3, *view1*, *view1’*, and *view1”*, three instances of a view type, share a name (e.g., “view1”) without any name conflicts because they are in different VEDs. If the names of view types are different, views in a VED cannot conflict with each other as well; however, similar to “.” and “..” directories, the view names pollute the naming space in VEDs. If any name conflict is detected, the creation operation will be denied.

3.2.4. VEF Operations. Due to metadata duplication across views, creating/deleting a file in a view must trigger the view synchronization in order to keep metadata consistent. This synchronization process takes place in any operation in which the metadata will be modified.

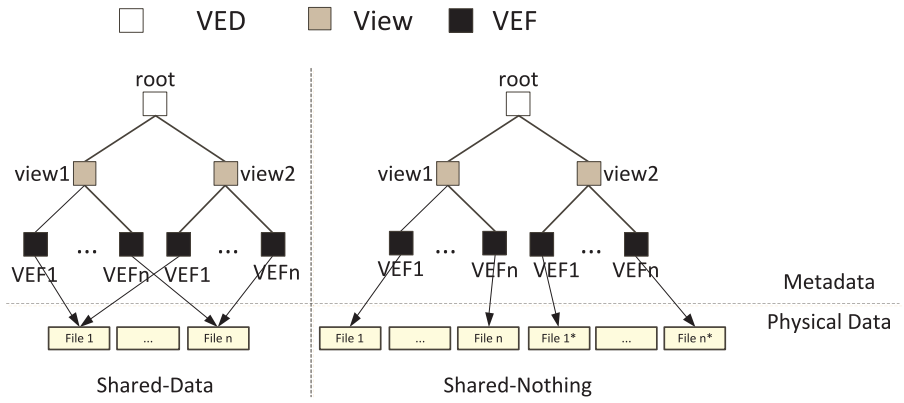

Fig. 4. Frog-based CBFS designs.

3.2.5. Default View Configuration. In our design, we classify applications into two categories: view-aware and nonview-aware applications. Existing applications are nonview-aware. The challenge of supporting both types of applications is that views are invisible in nonview-aware ones. To solve the compatibility issues, we set up a view instance of a particular type as Default View in each VED, which is used when view names are missing from file paths. The view-awareness of a file-path string is indicated by its view name next to the file name. Because the file name must be the last item in the string and view names in the Frog-based CBFS are predetermined, the view-aware file-path string can be easily identified by checking the name of the item next to the last item in the string. We describe the following typical scenarios as examples to demonstrate how the default views work.

File paths of two types of applications are different. In Figure 3, *VEF1'* under *VED1* is referred to as the path "/root/VED1/view1'/VEF1'" in view-aware applications, and "/root/VED1/VEF1'" in nonview-aware applications. The view name "view1'" is given in view-aware applications.

The default views are used for nonview-aware applications. For example, when *open* system call is issued on "/root/VED1/VEF1'," the last entity in the path string must be the file name *VEF1'*, and the next to the file name can be either a VED or a view, which can be determined by the name because view names are predetermined. If the entity is a VED, the default view name (either *view1'* or *view2'*) will be used to form a view-aware path.

The semantics of directory listing are different between view-aware and nonview-aware applications. When listing a VED, view-aware applications expect to retrieve the names of sub-VEDs and subviews, whereas nonview-aware applications obtain the names of subdirectories and files. In case existing applications cannot be modified, our design adapts view-aware applications to nonview-aware semantics. View names will not be exposed in directory listing. Another reason that we compromise the view-aware semantics is that similar to "." and ".." directories, the view names are predetermined. It is reasonable to assume that view names are known as a priori in applications.

3.3. Two Frog-based CBFS Designs

Metadata management has been discussed in the previous sections. If we take into account physical data organizations, there are two potential designs of CBFSs (see Figure 4).

A Shared-Data CBFS shares one copy of physical data among multiple views. *VEF1*s in *view1* and *view2* refer to the address of *File1* on disks. A shared-data CBFS simplifies physical data management by sharing data among views. A shared-data CBFS is applicable to the integration of context-specific solutions that are not conflicting in physical data management. As a good example of a shared-data CBFS, a BAVFS integrates aggressive and conservative prefetching in two views. Theoretically, these two prefetching techniques do not interfere with physical data management. Instead, the techniques address the issues of read operations (e.g., how much data is prefetched each time) by duplicating metadata.

A Shared-Nothing CBFS advocates a data duplication approach, in which each copy is managed by a view; thus, VEFs in views refer to the addresses of data in separate copies. A typical example of a shared-nothing CBFS is a BHVFS, which maintains data replicas for both *update-in-place* and *update-out-of-place* strategies that conflict with each other in data organization. Compared with a shared-data CBFS, a shared-nothing CBFS has to pay extra overheads to physical data synchronization among views.

3.4. Flexibility of View Design

Frog provides a unifying framework to cope with two issues: (1) to facilitate directory and view management for a CBFS, and (2) to separate context-specific solutions in the form of views. When a new solution is emerging, developers can either recycle parts of an existing view or design a brand new view. Reusing metadata structure and physical data layout is cost-effective under the Frog framework. For example, a default view can take responsibility for I/O operations that make changes in metadata and physical data, thereby allowing other views to only issue read-only operations. Moreover, I/O operations can be shared at the source-code level. Views and VEFs may share their operations by setting function pointers in inodes.

View development differs from traditional file system development. The only necessity for Frog developers to deal with is to focus on file organization and management in a particular view. Design of directory management and interactions among views can be outside the radar scope of the developers. After implementing view and VEF interfaces, the developers can link the new view to their file systems.

It is worth noting that the Frog framework contains a mechanism for solution integration; nevertheless, it is developers' responsibility to decide what contexts and solutions should be integrated in a CBFS according to application scenarios. For example, a BAVFS is a file system that manages millions of small files. Two typical operations issued by applications include entire file system scans and random reads. Thus, a file system developer may integrate aggressive and conservative prefetching solutions for sequential and random reads, respectively, in a BAVFS. In addition, the performance of an operation largely depends on context selections by applications and solution implementation by file system developers. In this study, we assume that context and its solution are appropriately selected.

3.5. Interactions with Applications

Besides the diversity of Frog-based CBFS designs, Frog is incredibly flexible to support a wide range of applications. One or multiple views in Frog can be chosen by applications. Figure 5 displays two typical scenarios of interactions between processes and a Frog-based CBFS. Application 1 concurrently communicates with both views. Another example is that Application 2 writes data through *view1*, which can be retrieved by Application 3 through *view2*.

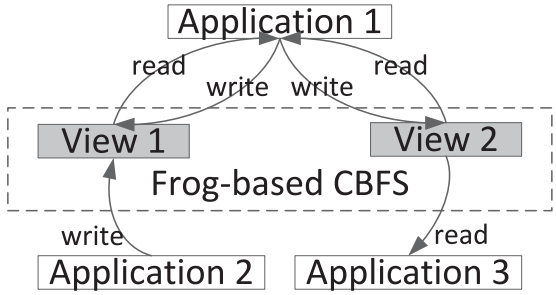

Fig. 5. Interactions between processes and Frog.

3.6. Application Comparison

We further illustrate differences between context-based and context-aware file systems by comparing their impacts of solution migration at applications. Applications running on heuristic-based systems are context-unaware applications, which do not identify and disclose context information. On the other hand, applications running on context-aware and context-based systems are context-aware.

The ease of context-specific solution migration estimated by the efforts spent on solution replacement is different. Because heuristic-based applications are transparent to solutions, no application modification is necessary for solution migration; however, context-aware and context-based applications are aware of solution replacements. Because of a unifying interface provided by Frog, context-based applications change only view names, whereas context-aware applications may have to shift to a new interface applied by the new solution.

Figure 6 illustrates a concrete example in which an application must be modified in the case of a solution replacement. In this example, a prefetching solution proposed by Patterson et al. [1995] has been employed in the file system. For some reasons (e.g., hardware upgrade), the solution needs to be replaced by another prefetching solution (e.g., Soundararajan et al. [2008]). In this case, both the file system and application have to be modified accordingly.

An intuitive modification method is to remove codes related to an old solution, and to write new codes for a new solution. The old codes are indicated by strike-through texts; the new codes are highlighted using gray background. In the context-aware applications (on the left-hand side in Figure 6), there are at least 10 lines of codes removed and 7 lines newly written, which is more than 4 line changes in the context-based applications (on the right-hand side in Figure 6). More important, the changes in the context-aware applications appear at three places (i.e., a global definition and two functions), probably in different source code files, thereby increasing the complexity of the solution migration. The modification may become more strenuous in solution migration for real-world applications due to complicated system design.

3.7. Data Consistency Overhead

Due to data duplication, Frog introduces data consistency overheads. In Frog, views are managed in separate spaces. Once an update is issued in a view, the update will be synchronized with other views in order to keep all of the views consistent.

Due to replicated metadata, the two designs mentioned earlier have to address the issue of metadata synchronization. The more views configured in Frog, the higher the consistency maintenance overhead is. Importantly, the overhead of physical data synchronization differs significantly among the designs. Shared-data CBFSs do not

~~code~~ Codes related to the old solution

code Codes related to the new solution

```
//initialize hints
HINT_STRU hints;
//initialize context parameter
structure context ctx;

void determine_access_pattern(char* file_str)
{
   if (this is a sequential read)
   {
      hints.filename = file_str;
      hints.pattern = SEQ_READs;
      ...
      //initialize the structure ctx;
   }
   else
   {
      hints.filename = file_str;
      hints.pattern = RAM_READs;
      ...
      //initialize the structure ctx;
   }
}
void context_aware_read(char* file_str)
{
   char buf[20];
   determine_access_pattern(file_str);
   int fd = open (file_str, ...);

   ioctl (fd, REQ, &hints);
   read(fd, buf, sizeof(buf));
   ctx_action(CTX_BEG);
   ctx_pread(ctx, buf);

   // Process data

   ctx_action(CTX_ENG);
   close(fd);
}
```

```
char* determine_view(char* file_str)
{
   if (this is a sequential read)
   {
      return "view1";
      return "view3";
   }
   else
   {
      return "view2";
      return "view4";
   }
}
void context_based_app(char* file_str)
{
   char buf[20];
   char* view_name = determine_view(file_str);
   //insert view_name into file_str
   int fd = open(file_str, ...);
   read(fd, buf, sizeof(buf));
   close(fd);
}
```

Fig. 6. Application modification comparison.

conduct any physical data synchronization, whereas shared-nothing CBFSs have to synchronize all updates among the views.

When a modification operation is issued, the Frog framework will conduct the operation to a view specified by view-based applications. The operation will be first processed by the default view in nonview-based cases. After the completion of the operation in the view, the operation result is returned to applications. Moreover, the modification is synchronized to other views by the Frog framework. A flag is set in each view to indicate the type of CBFS (i.e., shared-data CBFS or shared-nothing CBFS). In a shared-data CBFS, metadata changes will be processed during the data synchronization, since the specified or default view handles the changes on the physical data. The views will have to make decisions on how to complete the operations by their implementation. The order of views, which operations are synchronized to, is fully transparent to views.

Like journaling file systems, the Frog framework can apply a journaling mechanism to facilitate system recoveries from crash. In the journaling mechanism, all modification operations are recorded by either the Frog framework or views before committing the operations to disks. Multiple views in file systems give rise to a complicated journaling process, because updates are processed in different views before the updates are

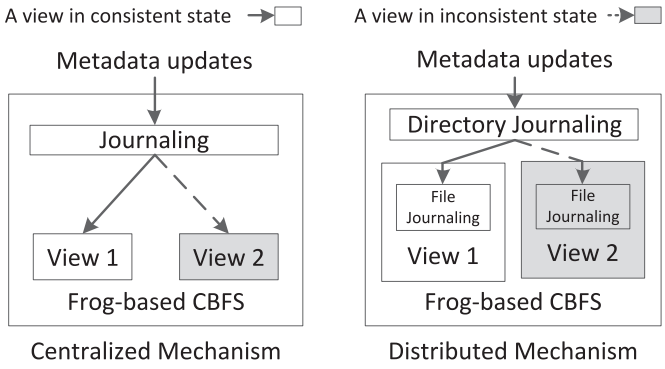

Fig. 7. Two journaling mechanisms.

synchronized. If a file system crashes during the synchronization process, views in the file system are likely in inconsistent states.

To recover file systems from a crash, there are two candidate journaling solutions supported by the Frog framework. *Centralized Journaling* encapsulates the journaling process in the framework (see Figure 7). A *journal* file or a dedicated region is created to record metadata updates before forwarding the updates to views [Seltzer et al. 2000]. During a system recovery, the Frog framework first checks the state of each update (e.g., using checksum) in views. If an update completely exists in one of the views, the update could be recovered by synchronizing it to all other views; otherwise, the update is ignored.

Centralized journaling reduces the difficulty of system development by virtue of a unifying journaling mechanism. Developers may focus only on the functionalities of context-specific solutions. Such simplicity comes at the cost of the flexibility of system design. Specifically, all metadata updates have to be journaled, preventing the system design from integrating a nonjournaling solution (probably for performance considerations). In addition, the entire file system is running under a journaling mechanism. Other metadata protection methods, such as soft update [McKusick and Ganger 1999], cannot be utilized in the file system.

The other mechanism is *Distributed Journaling*, which offers developers with flexible journaling choices. A distributed journaling mechanism separates file updates from directory updates, because there is only one copy of directory metadata in the Frog framework. In distributed journaling, directory updates are journaled by Frog in the same way as handled by the centralized journaling mechanism; file updates are directly forwarded to views, protecting metadata individually. In this case, an additional record is needed for each update in a central area, which indicates the view name that the update is committed to. During a system recovery, Frog first retrieves update records from the central area, followed by checking the consistency of each update in the corresponding view in the record. The update could be recovered if the update is completed in the view; otherwise, Frog ignores the update. Moreover, an alternative and simple method is to always check the consistency of updates in the default view. In this approach, Frog does not need to keep any global information in the central area; however, this scheme increases the possibility of data loss when a system crashes due to the delay of data synchronization to the default view.

4. CASE STUDIES

We conduct two case studies to evaluate the effectiveness, generality, and flexibility of the Frog framework. In the first case study, we outline the implementation issues of

BAVFS—the shared-data design of a context-based file system that applies a dual-mode (i.e., aggressive and conservative) prefetching for the context of sequential and random small reads. In the second case study, we focus on BHVFS—the shared-nothing design that separates read-intensive contexts from write-intensive ones, in each of which either the *update-in-place* or *update-out-of-place* strategy is employed to improve I/O performance.

The prototypes of a BAVFS and BHVFS are implemented on the top of the FUSE file system [FUSE 2014], which is popular in the file systems research community [Jain et al. 2008; Sundararaman et al. 2011; Weil et al. 2006]. Directories are used to emulate views in a hierarchy structure. Our prototypes enforce the differences between the two abstractions. The operation interfaces implemented in the prototypes include: *open*, *create*, *unlink*, *read*, *write*, *release*, *mkdir*, *rmdir*, *readdir*, *getattr*, and *lock*.

We may further optimize the prototypes. For example, considering alphabetical order in the metatada management of a BAVFS, applications can sort data returned from *readdir*. Nevertheless, we implement the prototypes in a straightforward way while focusing on the strengths and overheads of context-based file systems.

We developed 1,671 lines of codes in the Frog framework. A BAVFS has 822 lines of codes in the aggressive prefetching view and 294 lines of codes in the conservative prefetching view. A BHVFS has 505 lines of codes in the *update-out-of-place* view and 287 lines of codes in the *update-in-place* view.

4.1. BAVFS

4.1.1. Overview. The data explosion problem has been tackled in the past few years. The number and diversity of files are growing rapidly [Gantz 2008]. Recent studies [Agrawal et al. 2007; Tanenbaum et al. 2006] show that more than 50% of all files in a file system are small (e.g., smaller than 4KB); thus, it is critical to develop modern file systems that can efficiently process a large number of small files [Carns et al. 2009; Patil et al. 2007; Schindler et al. 2011].

Two common types of data-access patterns in I/O-intensive applications are sequential and random reads. For example, all small files in a directory are sequentially scanned by antivirus software, whereas one of these files is simultaneously opened by an editor. Sequential data retrieval can be improved by prefetching techniques [Kotz and Ellis 1991; Li et al. 2007; Patterson et al. 1995]; however, the unpredictability of future accesses is a natural barrier to adopting the prefetching techniques. Unnecessary prefetched data can significantly degrade system performance.

We implement a dual-mode I/O prefetching in a file system. A similar dual-mode prefetching mechanism can be found in the realm of instruction retrievals [Lee et al. 1993]. Aggressive prefetching is applied in the sequential context, thereby avoiding unnecessary prefetches due to random data retrievals; on the other hand, conservative prefetching is employed in the random context. A BAVFS is different from existing prefetching approaches in that a BAVFS provides a context-based prefetching approach, in which data not only in operating files but also files that will be accessed in future can be prefetched based on the contexts. Aggregating small files into large files would increase the overhead of update operations on these files, but small-file aggregation can optimize block usage as well as sequential-read performance in read-intensive scenarios (e.g., archival storage systems and the Hadoop Har File System [White 2009]).

4.1.2. Implementation. To shorten relocation times in the sequential view, small files in a directory are aggregated into large files. As shown in Figure 8, the physical data of files are merged into big *.data* files. The *.meta* files store the metadata in the views. The *.data* and *.meta* files appear in a pair, and the order of metadata in *.meta* is identical to the corresponding *.data* file. When a file is required, its metadata will be loaded in

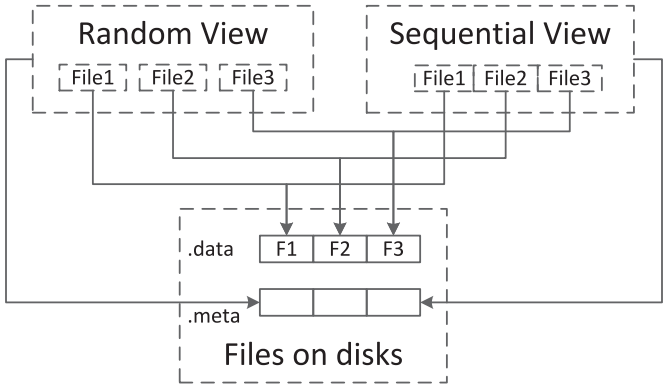

Fig. 8. Structure of a BAVFS.

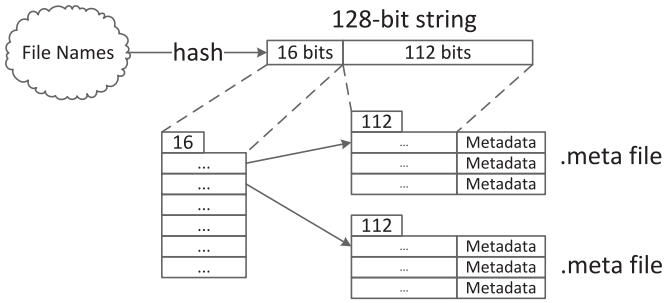

Fig. 9. Name processing in two views.

memory under the random view, whereas the entire *.meta* file will be loaded in the sequential view.

An extendible hash method is used in file-name mapping [Fagin et al. 1979] (see Figure 9). In both views, 128-bit hash values are generated from file names. The first 16b form the names of *.meta* files; the remaining 112b represent a unique index number in the *.meta* files. Each *.meta* file manages a set of entities, each of which is made up of an index number and file metadata information, including the location of physical data (i.e., *.log* file name, offset, and file size).

An update operation may be expensive. For instance, if the current block of a file is not sufficient to accommodate an update due to an increase in file size, the entire file must be moved to another location. Though such data movement incurs extra I/O costs, the design is reasonably effective for write-once-read-many access patterns [Shvachko et al. 2010]. Moreover, although two copies of metadata are managed in the two views, they actually share one copy of metadata on disk. The difference in metadata management in the views exists only in memory.

A BAVFS maintains data consistency as follows. When a write operation is issued, the specified view or default view will write changes to disks, whereas another view will cache the changes in memory and write the cached data to disks later. To respond to a read request, the view returns data if it is in memory. Changes in memory will be written back to the disk when either a write operation on the changes is issued or a time threshold is reached. The time threshold can be set as a time interval to periodically activate a back-end process to clean "dirty" data in the memory.

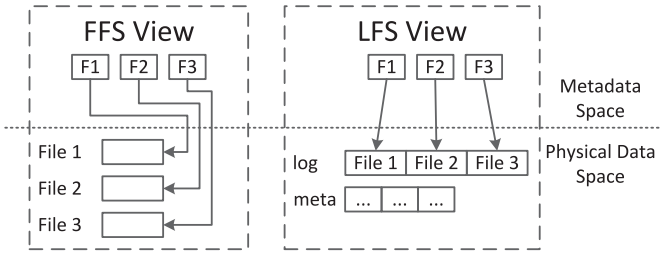

Fig. 10. Structure of a BHVFS.

When a read operation is issued, 30KB of data is prefetched and cached in memory in the sequential view. On the other hand, no data is prefetched in the random view. The reads are directly forwarded to the underlying file system. The random view is set as the default view of the BAVFS.

4.2. BHVFS

4.2.1. Overview. The efficiency of data retrieval and storage is critical in modern file systems. Two prevailing file systems, FFS [McKusick et al. 1984] and LFS [Rosenblum and Ousterhout 1992], embrace optimization techniques for reads or writes; however, few existing file systems are able to provide competitive performance on both reads (like FFS does) and writes (like LFS does).

To maximize the strengths of LFS and FFS, hFS [Zhang and Ghose 2007] incorporates them for both types of operations. Since either the LFS or FFS technique is selected in hFS for a file according to the file size, the file can only benefit from one technique rather than two. In addition, quFile proposes a method that encapsulates physical representations of files for contexts at the file level. Enjoying the benefits from FFS and LFS cannot be addressed by quFile at the file system level. The aforementioned limitations of hFS and quFile motivate us to develop a BHVFS that can exhibit the strengths of both FFS and LFS. Their two completely different strategies are adopted in read-intensive and write-intensive views, each of which shows the strengths and weaknesses of either FFS or LFS. If the views are appropriately selected by view-aware applications, the strengths of both views are maximally leveraged to achieve high performance for both reads and writes.

4.2.2. Implementation. Figure 10 shows that both views individually manage metadata in the metadata space. File metadata in a directory is organized in the form of a balanced search tree. In the physical data space, the FFS view applies the *update-in-place* strategy that overwrites data in the corresponding files; on the other hand, the LFS view uses the *update-out-of-place* strategy that appends updates to big *log* files. In addition, the LFS view maintains *meta* files to record the metadata information.

A cleaner program is repeatedly (e.g., once every hour) invoked to reclaim disk space. Improving I/O performance through physical data placement in Frog is less important than traditional log-structured file systems, because read operations are processed by the FFS view in a BHVFS.

5. EVALUATION

In this section, we examine the performance of CBFSs by evaluating a BAVFS and BHVFS using our testbed. We mainly focus on (1) investigating strengths derived from context decomposition, and (2) quantifying overheads incurred by introducing multiple views. BAVFS-NP (BAVFS Nonprefetching) and BAVFS-AP (BAVFS Aggressive Prefetching) denote conservative and aggressive prefetching views in a BAVFS.

BHVFS-FFS and BHVFS-LFS denote FFS and LFS views in a BHVFS. To fairly compare BAVFSs and BHVFSs with existing file systems, we have to consider the negative impacts imposed by the FUSE module and VFSs. Rather than minimizing the negative impacts of FUSE and VFSs, we create the following four baseline file systems using FUSE. FS-NP and FS-AP are traditional file systems that apply identical metadata management, data layout, and I/O operations with BAVFS-NP and BAVFS-AP, respectively. The only difference is that FS-NP and FS-AP do not contain views. FS-FFS and FS-LFS are fuse-based FFS and LFS. We use the distributed journaling mechanism without directory journaling in the prototypes because the underlying file systems (Ext 4 in our experiments) will take care of metadata protection.

We evaluate these file systems on a Ubuntu desktop computer with a 2.2GHz Intel Celeron CPU, 1GB memory and two 160GB Sata disks [WD1600AAJS 2013]. The Linux kernel version is 2.6.35. Ext 4 [Mathur et al. 2007] is configured as the underlying file system on which FUSE-based file systems are running. In the experiments, the BAVFS is running on a single disk, whereas two disks are utilized for the BHVFS, in which views maintain their data in separate disks.

We conduct the following three groups of experiments.

Metadata Operations. The first group is designed to measure the performance of file creation and deletion operations.

Random and Sequential Reads in BAVFS. We evaluate the impacts of aggressive and conservative prefetchings on two common types of operations, sequential and random reads, and their mixed workloads in a BAVFS.

Random Reads/Writes in BHVFS. We quantify the strengths and weaknesses of LFS and FFS in the context of a BHVFS by issuing two common types of operations, random reads and writes, as well as their mixed workloads, to the file systems.

During each experiment, we focus on testing a single view instead of the entire file system. This testing strategy allows us to demonstrate the performance of basic I/O operations in light of multiple views. In the figures shown in the following sections, BAVFS-NP, BAVFS-AP, BHVFS-LFS, and BHVFS-FFS indicate results of issuing operations on the views. In order to show the performance of a BAVFS and BHVFS, we conduct the experiments by switching the default views that support nonview-aware applications with appropriate view selection. Note that the existing benchmarks and applications are nonview-aware. In the two prototypes, write operations in the default view are pushed to disks by *fsync*, and update operations on the other view are processed by a thread at the back end.

From the comparison with FS-AP and FS-NP, a BAVFS, as an example of shared-data context-based file systems, can achieve 10% to 20% improvement in random and sequential read, respectively. The overhead of view maintenance in file creation, deletion, and path-string parsing is less than 3%. On the other hand, the BHVFS, working as a shared-nothing CBFS, always achieves more improvement in either reads or writes than the overhead of data synchronization.

5.1. File Creation/Deletion Rate

To test the file creation/deletion rate, we use PostMark [Katcher 1997]—a file system benchmark—with an initial set of 1,000,000 small files. We set file sizes anywhere between 1 to 2KB. FileBench (version 1.4.9) is used to generate mixed creation/deletion workloads [FileBench 2014]. We use the default settings of FileBench to create and delete 50,000 files in 16 threads. The mean file size is set to 16KB. The ratio of creation-to-deletion operations is set to 1.

From Figure 11, which reveals the file creation/deletion rates (i.e., the number of files created/deleted per second) without other I/O transactions, and Figure 12, which shows the evaluation of a BAVFS using mixed workloads, we observe that the BAVFS suffers

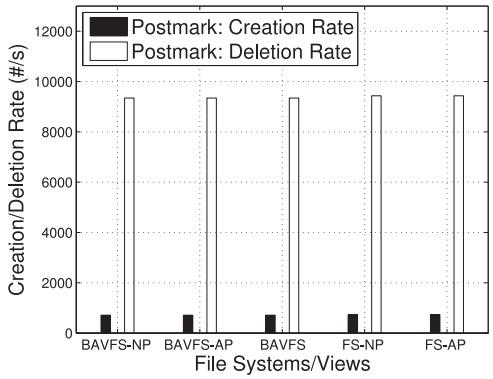

Fig. 11. File creation/deletion evaluation in a BAVFS.

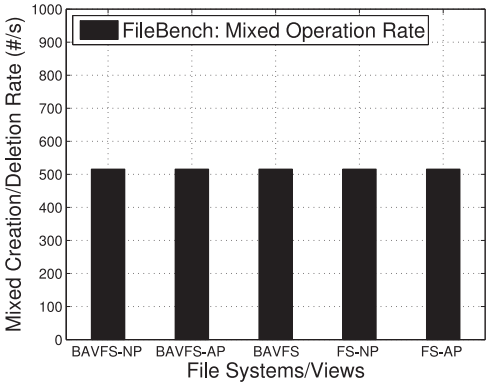

Fig. 12. Mixed creation/deletion evaluation in a BAVFS.

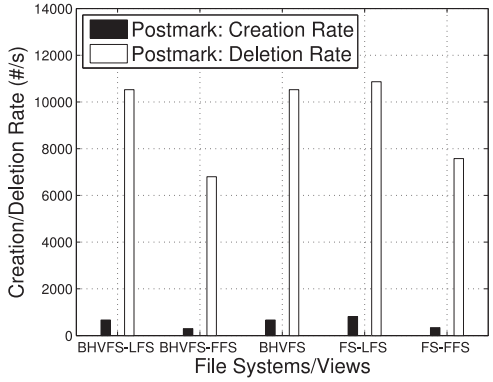

Fig. 13. File creation/deletion evaluation in a BHVFS.

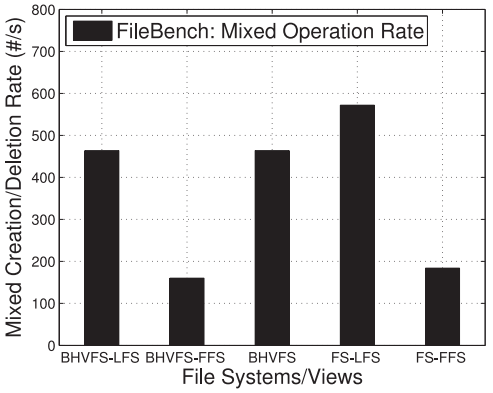

Fig. 14. Mixed creation/deletion evaluation in a BHVFS.

only from little overhead in file creation and deletion. In Figure 11, the creation/deletion rates of BAVFS-NP (713/9345) and BAVFS-AP (712/9345) are 3% lower than those of FS-NP (727/9433) and FS-AP (726/9433), because BAVFS does not actually maintain any replica on disks; even metadata are simply duplicated in memory. When a file is created or deleted, only one update is committed to disks. In the mixed-workloads evaluation, all the file systems can complete around 510 operations per second; the performance difference is below 2%.

From the evaluation on the BHVFS, FS-LFS and BHVFS-LFS are the better view to conduct file creation and deletion, because LFS appends updates only at the end of *.meta* files rather than generating and removing files. As Figure 13 shows, the file creation/deletion rates of BHVFS-LFS are 664/10526, which are 18%/3% slower than that of FS-LFS (809/10869). Compared with FS-FFS (339/7575), BHVFS-FFS (299/6802) suffers little (i.e., 11%/10%) performance degradation in file creation/deletion. Thanks to committing updates to separate disks, the overhead is hidden by the back-end synchronization process. Under the mixed workloads, a similar performance pattern is observed in Figure 14. BHVFS-LFS and FS-LFS can process 464 and 572 operations per second, respectively, whereas BHVFS-FFS and FS-FFS can issue 160 and 184

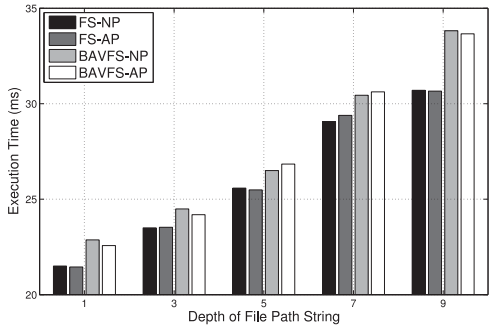

Fig. 15. The overhead of path-string parsing in a BAVFS.

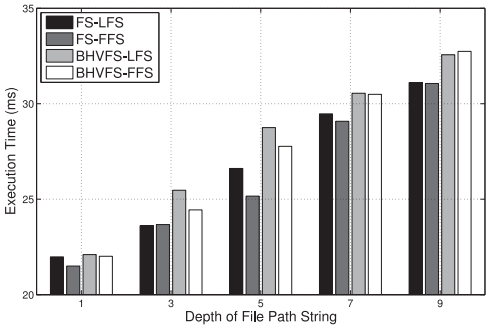

Fig. 16. The overhead of path-string parsing in a BHVFS.

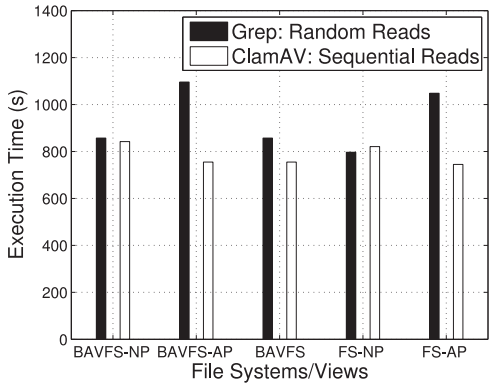

Fig. 17. Random and sequential read evaluation in a BAVFS.

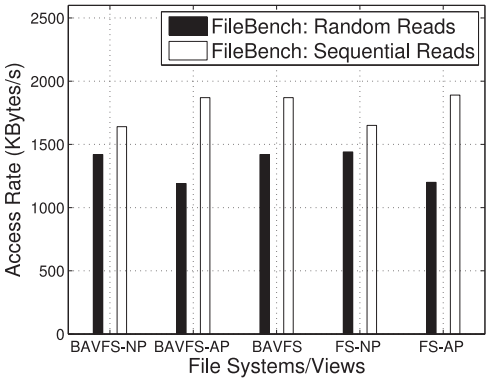

Fig. 18. Mixed random and sequential read evaluation in a BAVFS.

operations, respectively. In short, the BHVFS performs better than FS-FFS in both file creation and deletion, since the BHVFS derives the performance from the LFS view. If BHVFS-LFS is selected by applications, then the BHVFS performs 19% slower than FS-LFS and 150% faster than FS-FFS.

To evaluate the overhead of views in the file path string parsing, we use a synthetic program that invokes "open" and "close" system calls on random files. Figure 15 and Figure 16 show the relationship between the execution time and the depth of file path strings in the experiments. The program repeatedly makes the system calls 10K times on views of the BAVFS and BHVFS. As the experimental results show, the overhead of making 10K system calls in BAVFSs and BHVFSs is approximately as low as 1ms, which is negligible in any I/O operation, because the execution time of each system call is smaller than those of read/write operations by multiple orders of magnitude (see Figures 17, 18, 19, and 20).

5.2. BAVFS

Small random and sequential reads are evaluated in a BAVFS, FS-NP, and FS-AP. We create 2,000,000 files of size 1-2KB by a modified PostMark that does not execute transactions or delete files after file creation. We use ClamAV [ClamAV 2014], an open-source antivirus engine, as a representative application with sequential access pattern, and Grep [Grep 2014] as an example with random access pattern, searching

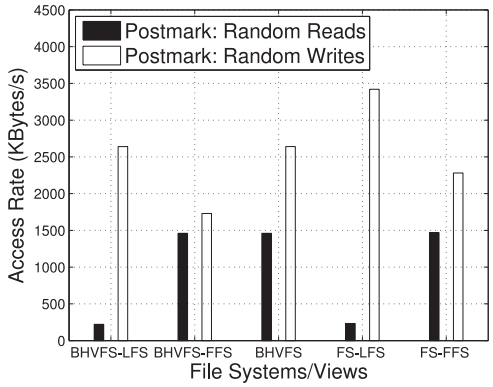

Fig. 19. Random read-and-write evaluation in a BHVFS.

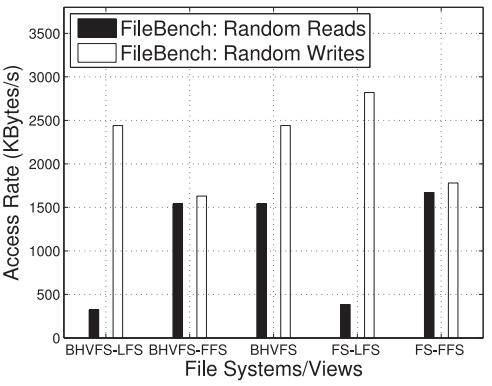

Fig. 20. Mixed random read-and-write evaluation in a BHVFS.

a keyword "Frog" from 200,000 files in a random order. Moreover, FileBench with the default settings is used for evaluating mixed sequential and random reads in a BAVFS. FileBench first creates a big file (1GB), and issues sequential read operations that retrieve data from the beginning of the file to the end, and random read operations that randomly read data until 128MB of data is accessed.

Figure 17 displays the execution time of ClamAV and Grep. FS-NP (797/821) can complete I/O operations faster than BAVFS-NP (857/842); FS-AP (1048/745) performs better than BAVFS-AP (1096/755) for random and sequential access patterns, because view-maintenance costs adversely affect the performance of the BAVFS. In addition, random reads are more efficient in BAVFS-NP and FS-NP. Conservative prefetching does not prefetch unnecessary data for random accesses. On the other hand, high hit rates of aggressive prefetching make BAVFS-AP and FS-AP perform better than BAVFS-NP and FS-NP in the case of sequential accesses.

If views are selected appropriately, the BAVFS performs 9% better than FS-NP with respect to sequential reads, and 21% better than FS-AP when it comes to random reads. The performance degradation is lower than 8% for random reads with FS-NP, and 1% for sequential reads with FS-AP, because the BAVFS experiences extra overhead induced by views. Thus, our empirical study shows that the BAVFS outperforms FS-NP and FS-AP in terms of overall performance.

In the mixed workload evaluation shown in Figure 18, BAVFS-NP (1420/1640) and BAVFS-AP (1190/1870) perform close to FS-NP (1440/1650) and FS-AP (1200/1890) in both operations. If views are selected appropriately, the BAVFS performs 18% better than FS-AP on random reads, and 13% better than FS-NP. The penalty of introducing a BAVFS for random and sequential reads is less than 1%.

5.3. BHVFS

We compare the random read-and-write performance of the BHVFS with those of FS-LFS and FS-FFS. To reduce the data consistency overhead, we deploy two disks in this group of experiments. Two views maintain their data on two separate disks. We use PostMark to create 20,000 files, each of which is 100-200KB in size. There are 2,000,000 writes and 20,000 reads issued to the tested file systems. The buffer size in each transaction is 4KB. Since PostMark creates files in a sequential way, we modify the file creation process so that data are randomly appended to initial files. In doing so, data among files are interleaved in a big *log* file in the LFS view. In the mixed workload evaluation, random read-and-write operations are generated by FileBench

with the default setting. FileBench creates a big file (5GB) and issues random reads and writes to the file.

From our experimental results, a BHVFS can achieve more advantage from the existence of two views if views are appropriately selected. Figure 19 reveals the access rate (measured in KB/s) of I/O operations. The write rates of BHVFS-LFS and FS-LFS are much higher than those of BHVFS-FFS and FS-FFS because written data are appended to the big *log* files. No seek is needed for individual files on the disks. In the BHVFS, updates in the FFS view are committed to another disk by the back-end process; the rate of BHVFS-LFS is not as low as the half of FS-LFS. The overheads are attributed to the synchronization process as well as resource contentions.

Due to a continuous organization, the read rates of BHVFS-FFS and FS-FFS are much higher than those of BHVFS-LFS and FS-LFS. When a read is issued to BHVFS-LFS or FS-LFS, the data is retrieved from a number of pieces triggering multiple seeks. Moreover, the difference in read rates between BHVFS-LFS and FS-LFS, as well as BHVFS-FFS and FS-FFS, is negligible. The way of retrieving data in a BHVFS is similar to that in traditional file systems. The only difference is that a BHVFS has to handle one more level (i.e., views emulated by directories in our experiments) to reach files.

More important, the read rate of the BHVFS (1460) is six times better than that of FS-LFS (223), and the write rate of the BHVFS (2640) is 30% worse than that of FS-LFS (3420). The read rate of the BHVFS is 1% worse than that of FS-FFS (1470); whereas the write rate is 16% better than that of FS-FFS (2280). After a comprehensive comparison, we conclude that the BHVFS is better than both FS-LFS and FS-FFS; the benefits obtained by the BHVFS outweigh its overhead.

From the mixed workload evaluation displayed in Figure 20, we observe that a BHVFS can achieve better random reads in BHVFS-FFS (1540) and random writes in BHVFS-LFS (2440), which is consistent with the nature of FFS and LFS. If views are selected appropriately, BHVFS performs 300% better than FS-LFS in random reads, and 37% faster than FS-FFS in random writes. The penalty of introducing a BHVFS is less than 13%.

5.4. Overhead of Data Synchronization

To quantitatively evaluate the overhead of data synchronization, we create a shared-nothing CBFS with alternative numbers of FFS views. Each view runs on a dedicated disk. The mixed workload of random reads and writes is generated from FileBench using the default settings.

Figure 21 displays the access rate of file systems with increasing numbers of views. We observe that the file systems slow down when the number of views increases. When an additional view is deployed, the access rate of read-and-write operations reduces by approximately 7%. This is because the Frog framework has to make extra data replicas to accommodate additional views, which induces overhead on their logic processing.

6. RELATED WORK

6.1. Context-Aware Systems

Context-aware computing provides opportunities to improve the performance of adaptive systems. For example, in a context-aware mobile computing system, context-sensitive data (e.g., locations and users' activities) can be simply collected by sensors. With the context-sensitive information, mobile devices can identify contexts and adjust device behaviors accordingly [Krause et al. 2006; Siewiorek et al. 2003].

Collecting context-sensitive data is nontrivial at storage server ends. Many researchers have investigated mechanisms to infer future access patterns based on

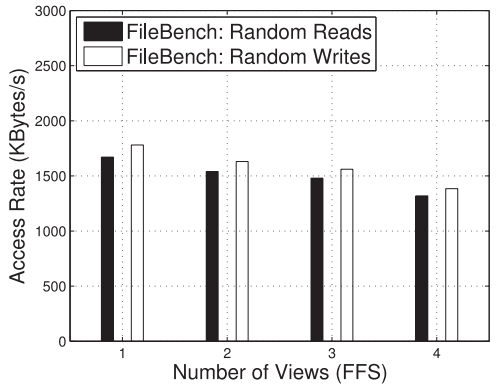

Fig. 21. Mixed random read-and-write evaluation.

historical accesses. For example, Kroeger and Long [1999, 2001] developed and extended Partitioned Context Modeling (PCM) that models historical file accesses to reliably predict future requests. Liang et al. developed a Sequentiality and Thrashing dEtection based Prefetching (STEP) scheme that detects sequential accesses and disk thrashing. Liang's prefetching techniques are powered by an analysis model using access history, disk access costs and data access locality [Liang et al. 2007]. While these heuristic approaches suffer from back-end analysis, the overhead of which greatly varies by the granularity of predication models, the number of factors considered include the model and timeline of records. Kroeger and Long [2001] discovered that the overhead of their modeling and prefetching modules is 0.05s, which account for up to 27% of elapsed time reduction obtained by their advanced methods. The detailed analysis of the overhead of back-end prefetching can be found in Kroeger [2000]. On the other hand, context information can be provided and propagated as hints by applications through dedicated programming interfaces. Soundararajan et al. [2008] developed a novel lightweight and minimally intrusive method, QuickMine, that improves prefetching performance by discovering block correlations. In addition, a new system call, *ctx_action*, and a new structure *context* are designed to indicate start or end of a particular context. Patterson et al. [1995] proposed aggressive and proactive mechanisms that exploit I/O parallelism and optimization on file buffer allocation by using application-disclosed hints. The hints consist of a file descriptor and an access pattern specifier describing either a sequential read from beginning of a file to end, or a read according to an ordered list of <offset, length> intervals. The hints are issued through *ioctl* system call, and additional data structures are designed for hint representation. Rather than relying on hints, our Frog-based CBFS handles context selections offered by applications that are aware of future I/O accesses. If future access patterns are unpredictable, the default view may be used without any penalty. CBFS can avoid the overheads of context analysis on storage servers.

A number of context-aware middleware systems were proposed to make context-aware computing possible [Chu et al. 2011; Gu et al. 2004; Román et al. 2002]. Our approach is different from these systems in that we focus on context-based file systems, in which view-aware applications can achieve context-specific I/O optimizations.

6.2. View-Enhanced Systems

Conceptual view models have been widely used in software design. Along with multiple views, data can be logically presented in diverse manners. A typical example is a relational database management system, in which views constructed on top of hierarchical

structures are available for applications [Ramakrishnan and Gehrke 2002]. Gehani et al. [1994] proposed a file system interface for an object-oriented database in which objects can be accessed through both file manipulations and database operations.

The Multiview Storage System (MVSS) introduces a multiview model for active storage systems to provide flexible services by dynamically generating in-memory views. A set of views are mounted at distinct points in a file system namespace [Ma and Reddy 2001]. Unlike an MVSS, which implements multiple views at the storage system level, Frog supports multiple views at the file system level, where it takes care of details of integrating multiple solutions in a file system. Frog handles metadata and physical data management, as well as consistency issues, which can be applied to a native file system (see our two case studies in Section 4). Moreover, Frog statically duplicates metadata or physical data across views, avoiding overheads of dynamic methods (e.g., making decisions about when and what types of views are created).

The issues of context-aware adaptation were addressed in quFiles [Veeraraghavan et al. 2010]. In particular, various contexts in quFiles are accessed in the form of views, in which an optimal policy is selected for a specific I/O context. To provide flexible and transparent services, quFiles dynamically or statically creates views. One view is set as the default view. Compared with quFiles, Frog diversifies not only physical representations but also metadata management as well as I/O operations. This salient feature of Frog makes it possible to integrate multiple I/O optimizations (e.g., prefetching) for particular contexts. More important, we show how to make use of Frog to implement feasible designs of CBFSs.

6.3. File Systems

Two completely different update strategies were implemented in file systems: *update-in-place* and *update-out-of-place*. An *update-in-place* file system (e.g., FFS [McKusick et al. 1984], IBM's JFS [JFS 2014], SGI's XFS [XFS 2014] and Ext 4 [Mathur et al. 2007]) usually commits an update at a place where data is stored. On the other hand, an *update-out-of-place* file system (e.g., LFS [Rosenblum and Ousterhout 1992]) appends updates in a log file, which could be reorganized by a separate cleaner program in a batch manner.

Hybrid file systems integrate multiple I/O techniques in a single file system. By identifying the characteristics of a file system workload, the multistructured file system (MFS) separates data into control, data, and log storage [Muller and Pasquale 1991]. The control and data storage manage data that has not been recently modified. The log storage is designed for the crash recovery purpose. DualFS is a journaling file system that separates metadata from data using partitions or devices [Piernas et al. 2002]. *Update-in-place* and *update-out-of-place* strategies are applied to the metadata and data partitions, respectively.

HyLog is a hybrid file system that incorporates the FFS and LFS strategies [Wang et al. 2004]. To avoid the cleaning overhead of LFS, HyLog divides disk space into fixed-size segments. Hot segments, containing hot pages, are organized in LFS to achieve high write performance. Cold segments, containing cold pages, are managed in FFS. When hot pages become cold, HyLog moves the cold pages to the cold partition to avoid extra cleaning overhead. The nature of pages, regardless of hot or cold, is determined by a separating algorithm.

Similar to HyLog, hFS is a hybrid file system combining FFS and LFS [Zhang and Ghose 2007]. In hFS, files are classified into two categories: large and small files. Large files are stored in FFS to offer competitive read performance by avoiding fragmentations incurred by small files. Metadata and small files are organized in the LFS fashion because read operations accessing small amounts of data are likely to respond by retrieving one or two blocks.

Unlike the existing file systems, our Frog-based CBFS allows applications to interact with file systems. The CBFS exhibits not only file information but also a number of contexts in which the context-specific solutions are applied. View-aware applications are able to benefit from the context-specific solutions by appropriately selecting views.

Stackable file systems were proposed as an easier way of implementing and extending file systems [Zadok et al. 1999; Zadok and Nieh 2000]. By using Unionfs [Wright et al. 2006], files in several directories appearing to be merged are actually managed separately. Our design is different from these systems in the sense that Frog-based CBFS combines multiple context-specific solutions by duplicating either a part of or an entire file system. In light of contexts, applications can pick views that offer optimization solutions to meet the needs of the applications.

7. DISCUSSIONS

CBFS, the context-based file system, is a broad research topic that cannot be fully covered in this article. We make an effort to raise a number of intriguing issues that are important for future studies. In this section, we first present the applicability of the novel CBFSs. Then we propose two potential extensions that implement CBFSs in native file systems and distributed environments. Finally, we discuss context exposure methods.

7.1. Applicability of Context-based File Systems

CBFSs encapsulate multiple file system solutions in views, which provide ample opportunities for systems and application developers to optimize performance. System developers are allowed to select I/O optimization approaches for a particular context without imposing adverse impacts in other contexts, because the contexts are independently integrated in the Frog framework. Moreover, the optimization approaches are beneficial to applications. Applications can perceive contexts and dynamically select appropriate contexts based on their performance requirements.

The context-aware adaptation techniques have been applied to multiple systems. For example, Odyssey is a context-aware tool monitoring resource availability in operating systems; Odyssey notifies applications when the statuses of corresponding resource changes [Noble et al. 1997]. Based on the resource notification from Odyssey, applications can adjust their data generation or consumption policies. Two case studies of context-aware file systems are presented in Veeraraghavan et al. [2010]. A photo viewer is extended to support context-aware adaptation by the virtue of Odyssey and quFiles. The file system automatically generates an additional four copies with different lower-fidelity representations of new images. The image copies with low fidelity are used when the photo viewer lists a directory, whereas the original copies are read when they are displayed in the photo viewer. The second case study, platform-dependent video display, applies the method in a similar fashion. Alternative video formats are generated and selected based on the platforms of clients (e.g., a format with a smaller screen size for tablets and the original format for laptops).

Due to data duplication in existing systems, the overhead of data synchronization increases proportionally to the number of data replicas. We summarize the following three scenarios, in which the overhead can be hidden by either the nature of applications or parallel I/Os. First, let us consider the read-intensive or write-once-read-many scenario commonly found in numerous systems like multimedia or archival systems. Most requests issued by applications are read-only without incurring any data synchronization. Thus, the overhead of data synchronization becomes marginal. The experimental results reported in Veeraraghavan et al. [2010] confirm that the overhead

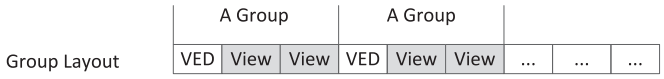

Fig. 22. VED and views allocation.

of generating extra copies on quFiles-based file systems is less than 1%. Second, utilizing multiple disks in a system helps in hiding data synchronization overhead through parallel I/Os. Ideally, if each view of a context-based file system is deployed on a disk, all data updates can be processed by views in parallel. In this scenario, the CBFSs and RAID-1 behave similarly, keeping two copies of data on two disks. Third, the model of CBFSs can be applied to distributed file systems. Since most of distributed file systems keep data replicas for reliability, the process of data synchronization becomes a fundamental functionality. Moreover, unlike data replicas that are identical in RAID-1 and distributed file systems, each copy of data in the CBFS has its own preference on physical representation for a particular scenario. The preference information is exposed to applications, allowing the applications to select appropriate views based on the applications' contexts.

7.2. View Management in Native File Systems

When it comes to CBFS implementation, we have to address a new issue, view allocation. As the experimental results suggest, a BAVFS is an appropriate candidate as a native file system, because overhead in metadata duplication is insignificant. For simplicity, we use directories to emulate views in our design; however, the following unique characteristics of views may provide us more opportunities of optimization in view management. First, unlike variable directory names, view names are predetermined during file system design. The sizes of metadata of views are fixed in length, which could simplify the view allocation. Second, view creation and deletion are bundled with directory creation and deletion. Instead of creating directories and views individually, the process can be combined in order to minimize overhead of view creation and deletion. For example, when a directory is created, extra fixed-length space can be allocated for views, as shown in Figure 22.

7.3. Work in Distributed Environments

Distributed environments help in reducing the complexity of metadata management, when two or more views are running on a single disk. In a distributed environment in which multiple computing resources and devices are coordinated, a CBFS is likely to have each view located in a standalone disk. More important, data is usually duplicated in distributed environments for fault-tolerance purpose (e.g., GFS [Ghemawat et al. 2003] and HDFS [Shvachko et al. 2010]). Overheads related to replica maintenance have been minimized in the distributed systems. Adopting existing replica-management facilities can considerably reduce the difficulty of the CBFS design. Thus, integrating Frog into a replica-enable distributed file system is attractive.

It is worth noting that after this integration, generating replicas in CBFS may be slightly different from that in distributed file systems. Rather than byte-to-byte duplications, each copy of replicas is independently managed by views. The content and format of a copy in a view may be different from the ones in other views. If the quality of the copies is not identical, the fault tolerance of the entire system will be compromised. For example, a bi-context file system has two copies of a video file. A copy in the high-resolution view is used for providing high-quality services to VIPs, whereas another copy in the low-resolution view is for regular customers. If the low-quality copy is lost, the copy can be recovered from its high-quality copy. However, the lost high-quality file cannot be recovered from the low-quality copy. The information

that does not exist in the low-quality file will be lost permanently. To address this problem, we can create the low-quality copy along with a complement file that stores the difference between high- and low-quality copies. In doing so, the high-quality copy can be recovered from the low-quality copy. The generation methods of complement files vary from application to application; these methods are out of the scope of our study.

7.4. Context Exposure

We discuss the methods of context exposure and perception in context-based systems and applications. In BAVFSs and BHVFSs, view set-ups in file systems are known as a priori during application development. The applications do not really perceive the contexts supported by the systems. However, context perception ability enables applications to be more adaptable in diverse computing environments. In what follows, we discuss two approaches to providing applications with context information in Frog.

The first approach is Context Detection, in which view-aware applications are able to detect the existence of particular views by issuing operations (e.g., stat) on the views. The systems return information on the views if they exist; otherwise, an error code (e.g., ENOENT) is returned. The view detection operations are invisible to view-unaware applications because they cannot issue any operation on views. Therefore, both types of applications can perform well in this approach. A limitation occurring in this method is that view names are required to be known in order to form a string path to views. In addition, detecting the existence of a number of views may cause multiple I/O operations that downgrade system performance.

The second approach is View Listing, in which views are listed through directory listing operations. In this approach, a new type value (e.g., DT_VIEW in d_type field of struct dirent) is added for views. After retrieving all objects under a directory, applications can find views by comparing their type values. As we mentioned in Section 3.2.5, the method exposes views to both types of applications, thereby possibly making view-unaware applications work incorrectly. For example, an application counts the number of files under a directory by calculating the number of objects that are not directories. If the views are returned along with files in the directory-listing operation, the application will retrieve incorrect information. Therefore, extra efforts on the correctness of view-unaware applications should be made in the case of view listing.

8. CONCLUSIONS

We present a general informed-based framework, Frog, for CBFSs, in which contexts are encapsulated in views. Frog integrates context-specific solutions that may conflict with each other in terms of metadata management, physical data organization, and I/O operations. We show the generality, transparency, diversity, and flexibility of Frog by implementing two Bi-Context File Systems, namely, the BAVFS and BHVFS. In the two case studies, we first demonstrate that the BAVFS optimizes performance of sequential and random reads on small files by virtue of dual-mode prefetching. Then, we illustrate that the BHVFS speeds up random reads and writes by incorporating *update-in-place* and *update-out-of-place* strategies. Our experimental results show that the benefits gained from CBFSs outweigh the overhead induced by creating and maintaining duplications for multiple views.

REFERENCES

N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. 2007. A five-year study of file-system metadata. *ACM Transactions on Storage (TOS)* 3, 3, Article 9, 32 pages.

P. Carns, S. Lang, R. Ross, M. Vilayannur, J. Kunkel, and T. Ludwig. 2009. Small-file access in parallel file systems. In *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*. 1–11.

Weijie Chu, Weiping Li, Tong Mo, and Zhonghai Wu. 2011. A context-source abstraction layer for context-aware middleware. In *8th International Conference on Information Technology: New Generations (ITNG)*. 1064–1065.

ClamAV. 2014. ClamAV. Retrieved July 3, 2015 from http://www.clamav.net.

R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. 1979. Extendible hashing—a fast access method for dynamic files. *ACM Transactions on Database Systems* 4, 3, 315–344.

FileBench. 2014. FileBench. Retrieved July 3, 2015 from http://sourceforge.net/projects/filebench/.

FUSE. 2014. File system in user space(FUSE). Retrieved July 3, 2015 from http://fuse.sourceforge.net/.

J. F. Gantz. 2008. The diverse and exploding digital universe. IDC White Paper 2, Framingham, MA, 1–16.

N. H. Gehani, H. V. Jagadish, and W. D. Roome. 1994. OdeFS: A file system interface to an object-oriented database. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94)*. San Francisco, CA, 249–260.

S. Ghemawat, H. Gobioff, and S. Leung. 2003. The Google file system. *SIGOPS Operating Systems Review* 37, 5, 29–43.

Gnu Grep. 2014. Gnu Grep. Retrieved July 3, 2015 from http://www.gnu.org/software/grep/.

T. Gu, H. K. Pung, and D. Q. Zhang. 2004. A middleware for building context-aware mobile services. In *Proceedings of IEEE Vehicular Technology Conference (VTC-Spring'04)*, Vol. 5. Milan, Italy, 2656–2660.

S. Jain, F. Shafique, V. Djeric, and A. Goel. 2008. Application-level isolation and recovery with solitude. *SIGOPS Operating Systems Review* 42, 4, 95–107.

JFS. 2014. The IBM JFS project. Retrieved July 3, 2015 from http://jfs.sourceforge.net/.

J. Katcher. 1997. PostMark: A new filesystem benchmark. *Technical Report TR3022,* 1–8.

D. Kotz and C. S. Ellis. 1991. Practical prefetching techniques for parallel file systems. In *Proceedings of the 1st International Conference on Parallel and Distributed Information Systems (PDIS'91)*. IEEE Computer Society Press, Los Alamitos, CA, 182–189.

A. Krause, A. Smailagic, and D. P. Siewiorek. 2006. Context-aware mobile computing: Learning context-dependent personal preferences from a wearable sensor array. *IEEE Transactions on Mobile Computing* 5, 2, 113–127.

Tom M. Kroeger. 2000. Modeling file access patterns to improve caching performance. PhD Dissertation. University of California, Santa Cruz.

Thomas M. Kroeger and Darrell D. E. Long. 1999. The case for efficient file access pattern modeling. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HOTOS'99)*. IEEE Computer Society, Washington, DC, 14–14.

Tom M. Kroeger and Darrell D. E. Long. 2001. Design and implementation of a predictive file prefetching algorithm. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, 105–118.

M. Lee, S. L. Min, C. Y. Park, Y. H. Bae, H. Shin, and C. S. Kim. 1993. A dual-mode instruction prefetch scheme for improved worst case and average case program execution times. In *Proceedings of the Real-Time Systems Symposium*. 98–105.

J. Leverich and C. Kozyrakis. 2010. On the energy (in)efficiency of Hadoop clusters. *SIGOPS Operating Systems Review* 44, 1, 61–65.

C. Li, K. Shen, and A. E. Papathanasiou. 2007. Competitive prefetching for concurrent sequential I/O. *SIGOPS Operating Systems Review* 41, 3, 189–202.

Shuang Liang, Song Jiang, and Xiaodong Zhang. 2007. STEP: Sequentiality and thrashing detection based prefetching to improve performance of networked storage servers. In *Proceedings of the 27th International Conference on Distributed Computing Systems (ICDCS'07)*. IEEE Computer Society, Washington, DC, 64–74.

X. Ma and A. L. N. Reddy. 2001. MVSS: Multi-view storage system. In *Proceeding of the 21st International Conference on Distributed Computing Systems, 2001.* 31–38.

A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. 2007. The new ext4 filesystem: Current status and future plans. In *Proceedings of the 2007 Linux Symposium*. 21–33.

M. K. McKusick and G. R. Ganger. 1999. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference (ATEC'99)*. USENIX Association, Berkeley, CA, 24–24.

M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. 1984. A fast file system for UNIX. *ACM Transactions on Computer Systems (TOCS)* 2, 3, 181–197.

K. Muller and J. Pasquale. 1991. A high performance multi-structured file system design. *Proceedings of the 13th ACM Symposium on Operating Systems Principles* 25, 5, 56–67.

Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. 1997. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP'97)*. ACM, New York, NY, 276–287.

S. V. Patil, G. A. Gibson, S. Lang, and M. Polte. 2007. GIGA+: Scalable directories for shared file systems. In *Proceedings of the 2nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing'07 (PDSW'07)*. ACM, New York, NY, 26–29.

R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. 1995. Informed prefetching and caching. *SIGOPS Operating Systems Review* 29, 5, 79–95.

J. Piernas, T. Cortes, and J. M. García. 2002. DualFS: A new journaling file system without meta-data duplication. In *Proceedings of the 16th International Conference on Supercomputing (ICS'02)*. ACM, New York, NY, 137–146.

R. Ramakrishnan and J. Gehrke. 2002. *Database Management Systems*. McGraw-Hill Science, New York, NY.

M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt. 2002. A middleware infrastructure for active spaces. *IEEE Pervasive Computing* 1, 4, 74–83.

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

J. Schindler, S. Shete, and K. A. Smith. 2011. Improving throughput for small disk requests with proximal I/O. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST'11)*. USENIX Association, Berkeley, CA, 10–25.

M. I. Seltzer, G. R. Ganger, M. K. McKusick, K. A. Smith, C. A. N. Soules, and C. A. Stein. 2000. Journaling versus soft updates: Asynchronous meta-data protection in file systems. In *Proceedings of the 2000 USENIX Annual Technical Conference (ATEC'00)*. USENIX Association, Berkeley, CA, 6–6.

K. Shvachko, H. Kuang, S. Radia, and R. Chansler. 2010. The Hadoop distributed file system. *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'10)*. 1–10.

D. Siewiorek, A. Smailagic, J. Furukawa, A. Krause, N. Moraveji, K. Reiger, J. Shaffer, and Fei Lung Wong. 2003. SenSay: A context-aware mobile phone. In *Proceedings of 7th IEEE International Symposium on Wearable Computers, 2003*. 248–249.

G. Soundararajan, M. Mihailescu, and C. Amza. 2008. Context-aware prefetching at the storage server. In *Proceedings of the USENIX 2008 Annual Technical Conference (ATC'08)*. USENIX Association, Berkeley, CA, 377–390.

G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber. 2010. Extending SSD lifetimes with disk-based write caches. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'10)*. USENIX Association, Berkeley, CA, 8–8.

S. Sundararaman, L. Visampalli, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. 2011. Refuse to crash with Re-FUSE. In *Proceedings of the 6th Conference on Computer Systems (EuroSys'11)*. ACM, New York, NY, 77–90.

A. S. Tanenbaum, J. N. Herder, and H. Bos. 2006. File size distribution on UNIX systems: Then and now. *SIGOPS Operating Systems Review* 40, 1, 100–104.

K. Veeraraghavan, J. Flinn, E. B. Nightingale, and B. Noble. 2010. quFiles: The right file at the right time. *ACM Transactions on Storage (TOS)* 6, Issue 3, Article 12, 28 pages.

W. Wang, Y. Zhao, and R. Bunt. 2004. HyLog: A high performance approach to managing disk layout. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST'04)*. USENIX Association, Berkeley, CA, 145–158.

WD1600AAJS. 2013. WD1600AAJS specification. Retrieved July 3, 2015 from http://wdc.custhelp.com/app/answers/detail/search/1/a_id/1400#.

S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. 2006. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*. USENIX Association, Berkeley, CA, 307–320.

T. White. June 5, 2009. Hadoop: The definitive guide. O'Reilly Media, Yahoo! Press, Sebastopol, CA.

Charles P. Wright, Jay Dave, Puja Gupta, Harikesavan Krishnan, David P. Quigley, Erez Zadok, and Mohammad Nayyer Zubair. 2006. Versatility and Unix semantics in namespace unification. *ACM Transactions on Storage (TOS)* 2, 1, 74–105.

XFS. 2014. The SGI XFS project. Retrieved July 3, 2015 from http://oss.sgi.com/projects/xfs/.

E. Zadok, I. Badulescu, and A. Shender. 1999. Extending file systems using stackable templates. In *Proceedings of the USENIX Annual Technical Conference (ATEC'99)*. USENIX Association, Berkeley, CA, 5–5.

E. Zadok and J. Nieh. 2000. FIST: A language for stackable file systems. *SIGOPS Operating Systems Review* 34, 2, 38–38.

Z. Zhang and K. Ghose. 2007. hFS: A hybrid file system prototype for improving small file and metadata performance. *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems* 41, 3, 175–187.

Received March 2013; revised September 2014; accepted January 2015