quFiles: The Right File at the Right Time

KAUSHIK VEERARAGHAVAN and JASON FLINN
University of Michigan, Ann Arbor
EDMUND B. NIGHTINGALE
Microsoft Research, Redmond
and
BRIAN NOBLE
University of Michigan, Ann Arbor

A quFile is a unifying abstraction that simplifies data management by encapsulating different physical representations of the same logical data. Similar to a quBit (quantum bit), the particular representation of the logical data displayed by a quFile is not determined until the moment it is needed. The representation returned by a quFile is specified by a data-specific policy that can take context into account such as the application requesting the data, the device on which data is accessed, screen size, and battery status. We demonstrate the generality of the quFile abstraction by using it to implement six case studies: resource management, copy-on-write versioning, data redaction, resource-aware directories, application-aware adaptation, and platform-specific encoding. Most quFile policies were expressed using less than one hundred lines of code. Our experimental results show that, with caching and other performance optimizations, quFiles add less than 1% overhead to application-level file system.

Categories and Subject Descriptors: D.4.3 [**Operating Systems**]: File Systems Management; D.4.7 [**Operating Systems**]: Organization and Design

General Terms: Design, Performance

Additional Key Words and Phrases: Context-aware file systems, data management, distributed storage, copy-on-write versions

12

J. Flinn was supported by NSF CAREER award CNS-0346686. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, the University of Michigan, Microsoft, or the U.S. government.
Authors' addresses: K. Veeraraghavan, J. Flinn, B. Noble, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109-2122; email: {kaushikv, jflinn, bnoble}@umich.edu; E. B. Nightingale, Microsoft Research, Redmond, WA 98052; email: ed.nightingale@microsoft.com.

DOI 10.1145/1837915.1837920 http://doi.acm.org/10.1145/1837915.1837920

ACM Reference Format:
Veeraraghavan, K., Flinn, J., Nightingale, E. B., and Noble, B. 2010. quFiles: The right file at the right time. ACM Trans. Storage 6, 3, Article 12 (September 2010), 28 pages.
DOI = 10.1145/1837915.1837920 http://doi.acm.org/10.1145/1837915.1837920

1. INTRODUCTION

It has become increasingly common for new storage systems to implement *context-aware adaptation*, in which different representations of the same object are returned based on the context in which the object is accessed. For instance, many systems transcode data to meet the screen-size constraints of mobile devices [Bila et al. 2007; Fox et al. 1996]. Others display reduced fidelity representations to meet constraints on resources such as network bandwidth [de Lara et al. 2001; Noble et al. 1997]; and battery energy [Flinn and Satyanarayanan 1999]; display redacted representations of data files when they are viewed at insecure locations [Lopresti and Lawrence 2005; Yumerefendi et al. 2007]; and create different formats of multimedia data for diverse devices [Peek and Flinn 2006].

These systems, and many others, have been successful at addressing specific needs for adapting the representation of data to fit a given context. However, they suffer from several problems that inhibit their wide-scale adoption. First, building such systems is time-consuming. Most required several person-years to build a prototype; porting them to mainstream environments would be difficult at best. Second, each system presents a different abstraction and interface, so each has a learning curve. Third, these systems typically present only a single logical view of data, making it difficult for users to pierce the abstraction and explicitly choose different presentations.

Why are there so many systems that share the same premise, yet have completely separate implementations? The answer is that, as a community, we have failed to recognize that there is a fundamental abstraction that underlies all these systems. This simple abstraction is the ability to view different representations of the same logical data in different contexts.

In this article, we argue that this new abstraction, which we refer to as a quFile, should be implemented as a first-class file system entity. A quFile encapsulates different physical representations of the same logical data. Similar to a quBit (quantum bit), the particular representation of the logical data displayed by a quFile is not determined until the moment it is needed. The representation returned by a quFile is specified by a data-specific policy that can take into account context such as the application requesting the data, the device on which data is accessed, screen size, and battery status.

quFiles provide a *mechanism/policy split*. In other words, they provide a common mechanism for dynamically resolving logical data items to specific representations in different contexts. A common mechanism reduces the time to develop new context-sensitive systems; developers only need to write code that expresses their new policies because quFiles already provide the mechanism.

A common mechanism also makes deploying new systems easier. Since the file system provides a unifying mechanism, a new policy can be inserted simply by creating another quFile.

quFiles provide *transparency* for quFile-unaware users and applications. Each quFile policy defines a *default view* that makes the observable behavior of the file system indistinguishable from the behavior of a file system without quFiles that happens to contain the correct data for the current context. This transparency has a powerful property: no application modification is required to benefit from quFiles. The default view also provides *encapsulation* by hiding the messy details of the physical representation and exporting only a context-specific logical view of the data.

For users and applications that are quFile-aware, a single logical representation of the data is often not enough. For instance, some users may wish to view the data in the quFile as it is actually stored or see a different logical presentation of data than the one provided by default. quFiles support this functionality through their *views* interface. All quFiles export a *raw view* that allows the physical representation of data within a quFile to be directly viewed and manipulated. In addition, quFile policies may define any number of *custom views*, each of which is an alternate logical representation of the data contained within the quFile. Users and applications select views using a special filename suffix, an interface that allows users to select views even when using unmodified commercial-off-the-shelf (COTS) applications.

How good is the quFile abstraction? We demonstrate its generality by implementing both ideas previously proposed by the research community (application-aware adaptation, copy-on-write file systems, location-aware document redaction, and platform-specific caching) and new ideas enabled by the abstraction (using spare storage to save battery energy and resource-aware directories). Our experience suggests a "natural fitness" for implementing context-aware policies using quFiles: compared to the multiple developer-years required to implement each of the existing systems described above, a single graduate student implemented each new policy in less than two weeks using quFiles. Further, policies required only 84 lines of code on average. Our results show that, with caching and other performance optimizations, quFiles add less than 1% overhead to application-level file system benchmarks.

2. RELATED WORK

A quFile is a new abstraction that encapsulates different physical representations of the same logical data and dynamically returns the correct representation of the logical data for the context in which it is accessed.

quFiles are not an extensibility mechanism. Instead, they are an abstraction that uses safe extensibility mechanisms (Sprockets [Peek et al. 2007] in our implementation) to execute policies. Thus, quFiles could use previously-proposed operating system extensibility mechanisms such as Spin [Bershad et al. 1995]; Exokernel [Engler et al. 1995]; or Vino [Seltzer et al. 1996]; as well as file system extensibility mechanisms such as Watchdogs [Bershad and Pinkerton 1988] or FUSE [FUSE 2009]. Compared to Watchdogs and FUSE,

quFiles present a minimal interface that focuses on contextual awareness; this results in policies that can be expressed in only a few lines of code.

A quFile can be thought of as the file system equivalent of a materialized view in a relational database [Gupta and Mumick 1995]. Unlike materialized views, quFiles return different data, depending on the context in which they are accessed, and they operate on file data, which has no fixed schema. Similarly, OdeFS [Gehani et al. 1994] presents a transparent file system view of data stored in a relational database. However, unlike quFiles, OdeFS objects are always statically resolved to the same view. MVSS [Ma and Reddy 2003] provides multiple in-memory views of a base file via distinct mount points in the file system to MVSS-aware applications. quFiles differ from MVSS, as they benefit unmodified applications and also allow dynamically-generated file names and content to be persistently stored on disk.

Multiple systems adapt the fidelity of data presented to clients. Since a full discussion of this body of work is outside the scope of this article, we only list here those systems that directly inspired our quFile case studies. These include systems that transcode data to meet screen-size constraints [Fox et al. 1996]; network bandwidth limitations [de Lara et al. 2001; Noble et al. 1997]; battery energy constraints [Flinn and Satyanarayanan 1999]; format decoding limitations [Peek and Flinn 2006], or storage restrictions [Pillai et al. 2004]. These previous systems either require application or operating system modification or the addition of an intermediary proxy that performs data adaptation. With quFiles, we propose a unified mechanism within the file system that can dynamically invoke any adaptation policy.

To simplify data management across multiple devices, Cimbiosys [Ramasubramanian et al. 2009]; PRACTI [Belaramani et al. 2006]; and Perspective [Salmon et al. 2009] allow clients to specify which files to replicate with query-based filters. quFiles could complement filters by adding context-awareness to replication policies.

Some file systems allow limited dynamic resolution of file content. Mac OS X Bundles [Bundle 2009] are file system directories that resolve to a platform-specific binary when accessed through the Mac OS X Finder. Similarly, AFS [Howard et al. 1988] has an "@sys" directory that resolves to the binary appropriate for a particular client's architecture. quFiles are a more general abstraction that capture these specific instances that embed particular resolution policies into the file system. NTFS has Alternate Data Streams [Russinovich and Solomon 2005] that support multiple representations of data within a file. However, unlike quFiles, NTFS does not currently support safe execution of arbitrary application policies to determine which representation should be accessed.

We describe one metadata edit policy for low-fidelity files. Other quFile policies could be implemented to support adaptation-aware editing [de Lara et al. 2003]. One possible approach is to layer updates separately from the data they modify and reconcile the high-fidelity original with the edit layer at a later time [Phan et al. 2004].

Past approaches such as Xerox's Placeless Documents [Dourish et al. 2000] and Gifford's Semantic File Systems [Gifford et al. 1991] suggest semantic

or property-based mechanisms to better organize and manage data in a file system. quFiles share the same goals of improving organization and simplifying management, but we have chosen a backward-compatible design that works within existing file systems, rather than requiring a system rewrite. The Semantic File System provides virtualized directories of files with similar attributes, whereas quFiles virtualize name and content of data within a directory based on context.

Schilit et al. advocate context-aware computing applications [Schilit et al. 1994] and identify four major categories of applications. Of these, quFiles support context-triggered actions, as well as contextual information and command-based applications. While Schilit et al. focus on usability and the graphical user interface, quFiles focus on supporting different views of the data in the file system. Building on these ideas, context-aware middleware [Kjær 2007] allows applications to modify the presentation of data depending on access context. However, these systems require application modification, for example, to subscribe to context events. quFiles provide similar functionality transparently to unmodified applications by manipulating the file system interface.

3. DESIGN GOALS

We next describe the goals that we aimed to achieve with our design of quFiles.

3.1 Be Transparent to the quFile-Unaware

We designed quFiles to be transparent by default. quFiles hide their presence from users and applications unaware of their existence. We say quFiles are transparent if *the observable behavior of a file system containing quFiles is indistinguishable from the behavior of a file system without quFiles that contains the correct data for the current context.* Consider a quFile that contains multiple formats of a video and returns the one appropriate for the media player that accesses the data. In this case, the application need not be aware of the quFile. It perceives that the file system contains a single instance of the video that happens to be one it can play. In general, a quFile may dynamically resolve to zero, one, or many files located in the directory in which it resides; we refer to this logical representation as the quFile's *default view*.

The default view provides the backward compatibility required to use COTS applications. Without modification, such applications must be quFile-unaware, so the context-specific presentation of data must be accomplished by presenting the illusion of a file system without quFiles that contains the appropriate data. The default view also reduces the cognitive load on the user by removing the need to reason about which representation of data should be accessed in the current context. Instead, the policy executed by the quFile mechanism makes this decision transparently.

Note that our definition of transparency applies to any specific point in time. When context changes, the appropriate representation to return may also change. This implies that a quFile-unaware user or application may observe that the contents of the file system change over time. This behavior is the same as that seen when another application or user modifies a file. For

instance, a quFile may redact files to remove sensitive content when data is accessed at insecure locations. A user will necessarily notice that the contents of the file change after moving from home to a coffee shop. However, the quFile *mechanism* itself remains transparent, so the same application can display the file in both contexts.

3.2 Don't Hide Power from the quFile-Aware

A quFile does not hide power from users and applications that wish to view and manipulate data directly. Instead, a quFile allows them to select among different *views*, each of which is a different presentation of its data. In addition to the default view described in the previous section, each quFile also presents a *raw view* that shows the data within the quFile as files within a directory. The raw view might include, for example, an original object, all materialized alternate representations of that object, as well as the links to policies that govern the quFile. quFile-aware utilities typically use the raw view to manipulate quFile contents directly.

The raw and default views represent the two endpoints on the spectrum of transparency. In between, a quFile's policy may define any number of additional *custom views*. A custom view returns a different logical representation of the data than that provided by the default view. A quFile-aware user or application can specify the name of a custom view when accessing a quFile to switch to an alternate representation. In effect, the name of the custom view becomes an additional source of context.

For example, consider a quFile that keeps old versions of a file for archival purposes along with the file's current version. The quFile's default view returns a representation equivalent to the file's current version. In the common case, the file system is as easy to use as one that does not support versioning because its outward appearance is equivalent to that of one without versioning. However, when a backup version is needed, the user should be able to see all the previous versions of the file and select the correct representation. The quFile policy therefore defines a `versions` custom view that shows all past versions in addition to the current one. Another custom view (a `yesterday` view) might show the state of all files as they existed at midnight of the previous day, and so on. Finally, a utility that removes older versions to save disk space may need to see incremental change logs, not just checkpoints, so that it can compact delta changes to reduce storage use. This utility uses the quFile's raw view.

quFiles distinguish between application transparency and user transparency. In the above example, a user may view previous versions of a file using `ls` or a graphical file browser. The user is quFile-aware, but the file browser is quFile-unaware. This scenario is tricky because the user must pass quFile-specific information through the unmodified application to the quFile policy. We solve this dilemma by using the file name, which is generally treated as a black box by applications to encode view selection. Specifically, for a directory `papers`, the user may select the versions custom view by specifying the name `papers.quFile.versions` or the raw view by specifying `papers.quFile`, which is shorthand for `papers.quFile.raw`.

3.3 Support Both Static and Dynamic Content

quFiles support both static and dynamic content. When data is read from a quFile, the file names and content returned might either be that of files stored within the quFile or new values generated on-the-fly. Storing and returning static content within the quFile amortizes the work of generating content across multiple reads. Static content can also reduce the load on resource-impoverished mobile devices; for example, rather than transcode a video on demand on a mobile computer, we pre-transcode the video on a desktop and store the result in a quFile. On the other hand, dynamic content generation is useful when all context-dependent versions cannot be enumerated easily. For instance, our versioning quFile dynamically creates checkpoints of files at specific points in time from an undo log of delta changes.

3.4 Be Flexible for Policy Writers

quFiles support not just the resolution policies that we have implemented so far, but also resolution policies that we have yet to imagine. We provide this flexibility by allowing resolution policies to be specified as short code modules in libraries that are dynamically loaded when a quFile is accessed. Each quFile links to the specific policies that govern it: a `name` policy that determines its name(s) in a given context; a `content` policy that determines its contents in a given context; and an `edit` policy that describes how its contents may be modified. A quFile may optionally link to two `cache` policies that direct how its contents are cached. These policies are easy to craft; the policies for our six case studies average only 84 lines of code.

Executing arbitrary code within the file system is dangerous, so policies are executed in a user-level sandbox. Our current implementation can use Sprocket [Peek et al. 2007] software fault isolation to ensure that buggy policies do not damage the file system or consume unbounded resources (e.g., by executing an infinite loop); other safe execution methods should work equally well.

4. IMPLEMENTATION

4.1 Overview

To illustrate how quFiles work, we briefly describe one quFile we developed. This quFile returns videos formatted appropriately for the device on which the video is viewed. When a new video is added to the file system, a quFile-aware transcoder utility learns of the new file through a file system change notification. The transcoder creates alternate representations of the video, sized and formatted for display on the different clients of the file system. It then creates a quFile and moves the original and alternate representations into the quFile using the quFile's raw view.

The transcoder also sets specific policies that govern the behavior and resolution of the quFile. A `name` policy determines the name of a quFile in a given context. If the quFile dynamically resolves to multiple files, the policy returns all resolved names in a list. For example, one author owns a DVR that displays

only TiVo files, which must have a file name ending in `.TiVo`. The `name` policy thus returns `foo.TiVo` when a video is viewed, using the DVR and `foo.mp4` otherwise.

A `content` policy determines the content of the quFile in a given context. This policy is called once for each name returned by a quFile's `name` policy. In the video example, the `content` policy returns the alternate representation in the TiVo format when the quFile is viewed on the DVR; an alternate representation for a smaller screen size when the quFile is viewed on a Nokia N800 Internet tablet; and the original representation when the quFile is viewed on a laptop. Note that the example quFile resolves to the same name on the N800 and the laptop, yet it resolves to different content on each device. Thus, COTS video players see only the video in the format they can play. Users who are quFile-unaware see the same video when they list the directory, but a quFile-aware power user could use the raw view to see all transcodings.

An `edit` policy specifies whether specific changes are allowed to the contents of a quFile. For instance, the user may modify the metadata of a lower-fidelity representation on the N800. In this case, the video transcoder is notified of the edit, and it makes corresponding modifications to the metadata of the other representations. However, changes to the actual video are disallowed, since there is no easy way to reflect changes made to a low-fidelity version to higher-fidelity representations.

Two optional `cache` policies specify context-aware prefetching and cache eviction policies for the quFile and its contents. These policies help manage the cache of distributed file systems [Howard et al. 1988; Kistler and Satyanarayanan 1992; Nightingale and Flinn 2004] that persistently store data on the disk of a file system client. For the example quFile, the `cache` policies ensure that only the format needed for a specific device is cached on that device.

4.2 Background: BlueFS

The quFile design is sufficiently generic so that quFile support can be added to most local and distributed file systems. For our prototype implementation, we added quFile support to the Blue File System [Nightingale and Flinn 2004] (BlueFS) because BlueFS targets mobile and consumer usage scenarios for which quFiles are particularly useful and because we were familiar with the code base. BlueFS is an open-source, server-based distributed file system with support for both traditional computers and mobile devices such as cell phones. Additionally, BlueFS can cache data on a device's local storage and on removable storage media to improve performance and support disconnected operation [Kistler and Satyanarayanan 1992]. BlueFS has a small kernel module that manages file system data in the kernel's caches. The kernel module redirects most VFS operations to a user-level daemon. To support quFiles, we made small modifications to both the kernel module and daemon, while the file server remained unchanged. For simplicity, we also use BlueFS' persistent query [Peek and Flinn 2006] mechanism to deliver file change notifications.

```
name_policy            (IN list of quFile contents,
                        IN view name (if specified),
                        OUT list of file names, OUT cache lifetime);
content_policy         (IN filename, IN list of quFile contents,
                        IN view name (if specified),  OUT fileid,
                        OUT cache lifetime);
edit_policy            (IN fileid, IN edit type, IN offset, IN size,
                        OUT enum {ALLOW, DISALLOW, VERSION})
cache_insert_policy    (IN list of quFile contents,
                        OUT list of fileids to cache)
cache_eviction_policy  (IN fileid, OUT enum {EVICT, RETAIN})
```

Fig. 1. quFile API.

4.3 Physical Representation of a quFile

Logically, a quFile is a new type of file system object. A quFile is similar to a directory in that they both contain other file system objects. The difference between quFiles and directories is their resolution policies. Directory resolution policies are *static*: given the same content, a directory returns the same results. quFile resolution policies are *dynamic*: the same content may resolve differently in different contexts. Further, users and applications must be aware of directories, since they add another layer to the file system hierarchy, whereas quFiles can hide their presence by simply adding resolved files to the listing of their parent directories.

Using this observation, we reduce the amount of new code required to add quFiles to a file system by having the physical (on-disk and in-memory) representation of a quFile be the same as a directory, but we redefine a quFile's VFS operations to provide different functionality than that provided by a directory. We segment the namespace to differentiate quFiles from regular directories. All quFiles have names of the form `<name>.quFile`. While we considered other methods of differentiating the two, such as using a different file mode, a special filename extension allows quFile-aware utilities to manipulate quFiles without changing the file system interface. For example, the video transcoder simply issues the commands `mkdir foo.quFile` and `mv /tmp/foo.mp4 foo.quFile` to create a quFile and populate it with the original video. The only disadvantage of namespace differentiation is the unlikely possibility that a quFile-unaware application might try to create a directory that ends with `.quFile`. Note that the quFile-aware transcoder uses the quFile's raw view to manipulate its contents; this allows it to use COTS file system utilities such as `mv`. Video players will see the default view, since they will not use the special `.quFile` extension. When they list the directory containing the quFile, they will see an entry for either `foo.mp4` or `foo.TiVo`.

4.4 quFile Policies

Figure 1 shows the programming interface for all quFile policies. Policies are stored in shared libraries in the file system. When a quFile is created, utilities

such as the video transcoder create links in the quFile to the libraries for its specific policies. Links share policies across quFiles of the same type, simplifying management and reducing storage usage.

4.4.1 *Name Policies.* A `name` policy lets a quFile have different logical names in different contexts. To make the existence of a quFile transparent to quFile-unaware applications and users, a VFS `readdir` on the parent directory of a quFile does not return the quFile's name; instead, it returns the names of zero-to-many logical representations of the data encapsulated within the quFile. quFiles interpose on the parent's `readdir` because that is when the filenames of the children of a directory are returned to an application.

If `readdir` encounters a directory entry with the reserved `.quFile` extension, it makes a downcall to the BlueFS daemon, which runs the `name` policy for that quFile. The kernel reads the quFile's static contents from the page cache and passes the contents to the daemon.

The user may optionally specify the name of a view for the `name` policy. For example, instead of typing `ls foo`, a user could type `ls foo.quFile.versions` to show a directory listing that contains all versions retained by the quFiles in the directory. The view name is passed to the `name` policy without interpretation by the file system. This allows a quFile-aware user to use a COTS application such as `ls` to list file versions when desired. As mentioned previously, the syntax `ls foo.quFile` returns the raw view of the quFile, which shows the quFile and all its contents as a subdirectory within `foo`. This syntax allows quFile-aware utilities and users to directly manipulate quFile contents and policies.

The `name` policy returns a list of zero-to-many logical names. The kernel module then calls `filldir` for each name on the list to return them to the application reading the directory. If no names are returned by the policy, the kernel does not call `filldir`. This hides the existence of the quFile from the application.

In addition to returning the name of existing representations encapsulated in a quFile, a `name` policy may also dynamically instantiate new representations by returning filenames that do not currently exist within the quFile. To ensure that such names do not conflict with other directory entries or names returned by other quFiles within the directory, each quFile reserves a portion of the directory namespace. For instance, the names returned by `foo.quFile` must all start with the string `foo` (e.g., `foo.mp3`, `foo.bar.txt`, etc). Directory manipulation functions such as `create` and `rename` ensure that the claimed namespace does not conflict with current directory entries. For example, creating a quFile `foo.quFile` is disallowed if there currently exists within the directory a file named `foo.txt` or another quFile named `foo.tex.quFile`.

To improve performance, a `name` policy may specify a cache lifetime for the names it returns—the kernel will not reinvoke the `name` policy for this time period. By default, the kernel module does not cache entries if no lifetime is specified, so the policy is reinvoked on the next `readdir`, and may return different entries if context has changed. Cache lifetimes are useful for policies that depend on slowly-changing context such as battery life.

4.4.2 *Content Policies.* A `content` policy lets a quFile have different content in different contexts. After reading a directory, an application that is unaware of quFiles will believe that there are one or more files with the logical names returned by the quFile's `name` policy within that directory. Thus, it issues a VFS `lookup` for each logical name. Since no such file exists, we modify `lookup` to return an inode of a file containing the logical content associated with the name in the given context.

The modified BlueFS `lookup` operation checks whether the name being looked up resides within the directory namespace reserved by a quFile. If this is the case, it makes a downcall to the BlueFS daemon, passing the filename being looked up, a list of the quFile's contents, and a view name if one was specified. The daemon calls the quFile's `content` policy, which returns the unique identifier of a file containing the appropriate content. The kernel module `lookup` operation instantiates a Linux `dentry` with the inode specified by the fileid returned by the policy.

This implementation allows quFiles to create content dynamically. A `content` policy can first create a new file and populate it with content, then return the newly created file to the kernel. Like `name` policies, `content` policies may also specify a cache lifetime for the content they return. If a lifetime is not specified, the kernel does not cache the resulting `dentry`, which forces a new `lookup` the next time the content is accessed.

4.4.3 *Edit Policies.* An `edit` policy specifies which modifications to a quFile's contents are allowed. Currently, quFiles support three actions: the modification can be allowed, disallowed, or force the creation of a new version. We modified VFS operations such as `commit_write` and `unlink` to make a downcall to the daemon when a quFile representation is modified. The daemon runs the `edit` policy, passing in the unique identifier of the file being modified and the type of the modifying operation. For write operations, it also specifies the region of the file being modified. The policy returns an enum that specifies which action to take.

If the edit is allowed, the modification proceeds as normal. If it is disallowed, the kernel returns an error code to the calling application specifying that the file is read-only. If the edit should cause a new version, we modify the representation in place, but also save the previous version of the modified range in an undo log. We chose to log changes rather than create a new copy of the file for each version because many consumer files are large (e.g., multimedia files) and are only partially modified (e.g., by updating an ID3 header). Modifications that delete files such as `unlink` and `rename` cause the current version of the file to be saved as a log checkpoint.

4.4.4 *Cache Policies.* Our final two policies control the caching of quFile data in the BlueFS on-disk cache. For a distributed file system, the decision of what files to cache locally significantly impacts user experience when disconnected.

quFiles may optionally specify two cache policies. A `cache insert` policy is called when a quFile is read, and may specify which of its contents to cache on

Table I. quFile Context Library

Function	Returns
getUserName	char* username
getUserGroupId	uid_t uid, gid_t gid
getProcessName	char* procname
getHostname	char* hostname
getOSname	char* osname
getOSversion	char* release, char* version
getMachine	char* family
getCPUvendor	char* vendor, char* model
getCPUspeed	double cpuSpeed
getCPUutil	double utilization
getMemUtil	double utilization
getPowerState	enum{A/C, Battery}
getLocation	double latitude, double longitude
getServerBandwidth	double bandwidth
getServerLatency	double latency

disk. Files specified by the cache insert policy are kept on a per-cache list by the BlueFS daemon and are fetched and stored when the daemon periodically prefetches data for the cache. For instance, when a quFile containing the recent episode of a favorite TV show is prefetched to a portable video player, its `cache insert` policy might specify that the video formatted for the video player, a representation that resides in that quFile, should also be prefetched. In contrast, when the same policy runs on a laptop, it would specify that the full-quality video should be fetched and cached instead. Thus, the policy ensures that only the data needed to play the video on each device is actually cached on the device's disk.

A `cache eviction` policy is called when the file system needs to reclaim disk space. The policy specifies whether or not cached contents should be evicted. Cache policies complement type-specific caching mechanisms in mobile storage systems [Peek and Flinn 2006; Ramasubramanian et al. 2009; Salmon et al. 2009] by adding the ability to make cache decisions based on dynamic context such as battery state or location.

4.5 Context Library

Through the Sprocket interface, quFiles have read-only access to all information available to the BlueFS daemon. Thus, in principle, policies can extract arbitrary user-level context information in order to determine which representations to return. However, for convenience, we have implemented a library against which policies may link. This library contains the functions shown in Table I that query commonly-used context.

4.6 Performance Optimizations

We perform several optimizations to reduce quFile overhead. First, the kernel reads the names of a quFile's contents from the page cache and passes this information to the daemon when a policy is invoked. This eliminates the need for the daemon to read this data from disk or the server. Second, the kernel

caches quFile `name` and `content` results for the specified cache lifetime. We use the Linux `dentry` cache for `content` results, and we implemented a new cache in the BlueFS-specific inode data for `name` results.

Finally, we use compound RPCs to batch the retrieval of quFile objects. As in NFSv4 [Shepler et al. 2003], BlueFS compound RPCs allow multiple operations to be done in a single roundtrip. Since we know that all quFiles in a given directory will be read during a `readdir`, we fetch multiple quFiles together. In contrast, we do not fetch quFile representations in a batch because we cannot know if a file will be read until it is opened.

4.7 File System Requirements for quFiles

Since our current implementation leverages BlueFS, it is useful to consider what features of BlueFS would need to be supported by a file system before we could port quFiles to that file system. First, quFiles require a method to notify applications when files are created or modified. While OS-specific notification mechanisms such as Linux's inotify [Love 2005] would suffice for a local file system, BlueFS persistent queries are useful in that they allow notifications to be delivered to any client of the distributed file system. Second, quFiles require a method to isolate the execution of extensions. This could be as simple as a user-level daemon process, or we could leverage existing extensibility research [Bershad et al. 1995; Engler et al. 1995; Seltzer et al. 1996]. Finally, quFiles reuse existing file system directory support, as defined by POSIX.

5. CASE STUDIES

The best way to evaluate the effectiveness and generality of a new abstraction is to implement several systems that use that abstraction to perform different tasks. Thus, in this section, we describe six case studies that use quFiles to extend the functionality of the file system. We have used these quFile case studies within our research group. The primary author of the article has used quFiles for the last 12 months, while others have used quFiles for the past 6 months.

5.1 Resource Management

One of the primary responsibilities of an operating system is to manage system resources such as CPU, memory, network, storage, and power. While several research projects have shown that context can be used to craft more effective policies, almost every new proposed policy has resulted in a new system being built [Anand et al. 2003; de Lara et al. 2001; Noble et al. 1997].

quFiles simplify resource management in two ways. First, they execute policies in the file system—thus, developers need not create new middleware or modify applications or the operating system. Second, developers only need to write resource management policies; quFiles take care of the mechanism.

Our case study allows a mobile computer to save battery energy by utilizing its spare storage capacity. Music playback is one of the most popular

applications on mobile devices. Most mobile devices store music in a lossy, compressed format, such as the mp3 format, to conserve storage space and reduce network transfer times. However, decoding compressed music files requires significantly more computational power than playing uncompressed versions. For instance, the experimental results in Section 6.6 show a battery lifetime cost of 4 to 11% across several mobile devices. Further, we conducted a small survey to determine the amount of unused storage on cell phones and mp3 players: 13 of 45 mp3 players were over half empty; 18 were 50 to 90% full; and 14 were over 90% full; 15 of 29 cell phones were over half empty; 10 were 50 to 90% full; and 4 were over 90% full.

Our quFile uses the spare storage on a mobile computer to store uncompressed versions of music files and then transparently provides those uncompressed versions to music players to save energy. We built a quFile-aware transcoder that is notified when a new mp3 file is added to the distributed file system. The transcoder generates an uncompressed version of the music file with the same audio quality as the original, creates a quFile, links it to our policies, and moves both the compressed and uncompressed versions of the music file into the quFile using its raw view. Since persistent queries provide the ability to run the transcoder on any BlueFS client, we generate alternate transcodings on a wall-powered desktop computer. This shows one benefit of statically storing alternate representations in a quFile rather than generating them on-demand: we can avoid performing work on a resource-constrained device. In contrast, dynamically generating transcodings on a mobile device could substantially drain its battery.

The quFile `cache` policies ensure that only otherwise unused storage space is used to store uncompressed versions of music files. Using the normal BlueFS mechanisms, a music file is cached on a client either when it is first played or when it is prefetched by a user-specified policy (e.g., that all music files should be cached on a cell phone [Peek and Flinn 2006]). Since the music file is contained within a quFile, the file system's `lookup` function must always read the quFile before reading the music file. At this time, the quFile's `cache insert` policy is run. The policy queries the amount of storage space available on the device and adds the uncompressed representation to the prefetch list if space is available.

Later, when BlueFS does a regularly scheduled prefetch of files for the mobile client, it retrieves files on the prefetch list from the server if the mobile computer is plugged in, has spare storage available, and has network connectivity to the server. It adds these prefetched files to its on-disk cache. When BlueFS needs to evict files from the cache, it executes the quFile's `cache eviction` policy, which specifies that the uncompressed version is always evicted before any other data in the cache.

The `name` and `content` policies return the name and data for the uncompressed version of the music file if the mobile device is operating on battery power and the uncompressed version is cached on local storage, thereby improving battery lifetime. If the uncompressed version is not cached on the device, the original file is returned.

This case study demonstrates how quFiles achieve application and user transparency. All actions described above run automatically, without explicit user involvement and without application modification.

5.2 Versioning: A Copy-On-Write File System

Copy-on-write file systems such as Elephant [Santry et al. 1999] and ext3cow [Peterson and Burns 2005] create and retain previous versions of files when they are modified. Users can examine previous versions and revert the current version to a past one when desired. However, these systems are monolithic implementations, and the need to use new file systems has hindered their adoption. Thus, we were curious to see if quFiles could be used to add copy-on-write functionality to an existing file system.

We created a copy-on-write quFile that adds the ability to retain past versions of files. A user may choose to version any individual file, all files of a certain type, or all files in a particular subtree of the file system. For instance, a user might version all LaTeX source files. A quFile-aware utility uses BlueFS persistent queries to register for notifications when a file with the extension `.tex` is created. When it receives a notification (e.g., that `foo.tex` is being created) it creates a new quFile with the name `foo.tex.quFile`. It then uses the quFile's raw view to move the LaTeX file into the quFile and link the quFile to the copy-on-write policies.

In addition to the current version of the file, each copy-on-write quFile may contain possibly many older versions of the file. A past version may be represented as either a *checkpoint*, which is a complete past version of the file, or a *reverse delta*, which captures only the changes needed to reconstruct that version from the next most recent one. The reverse delta scheme is effectively an undo log that reduces the storage space needed to store past data; for instance, a change to the header of a 1 GB video file can be represented by a delta file only one block in size. While reverse deltas save storage, generating a complete copy of a past version incurs additional latency when one or more deltas are applied to a checkpoint or the current version.

The quFile's `name` and `content` policies simply return the current version of the file for the default view. The quFile's edit policy specifies that a new version should be created on any modification, that is, whenever a file is closed, deleted, or renamed. Thus, when the user opens a file and issues one or more writes, the old data needed to undo his changes is saved to a new delta file within the quFile. The modifications are written to the current version of the file stored within the quFile. Because the default view exposes only the current version, these actions and the presence of past versions are completely transparent.

Versioning the data overwritten by file writes often consumes less storage and takes less time than creating a full checkpoint. To further reduce the cost of versioning, quFiles create new versions at the granularity of file `open` and `close` operations, rather than at each individual write. Unlike `write`, operations such as `rename` and `unlink` affect the entire file. For these operations, the current version is moved to a checkpoint within the quFile. Since there is no current

version remaining, the quFile's `name` policy does not return a filename for the default view, giving the appearance that the file has been deleted. However, the old data can still be accessed via the raw view or a custom view.

When the user wishes to view prior versions, she uses the `versions` custom view (the `.quFile.versions` extension). This allows the use of COTS applications such as `ls` and graphical file system browsers to view versions. Whereas the default view only shows a single file, `foo.tex`, in a directory, the custom view may additionally show several past versions, for example, `foo.tex`, `foo.tex.ckpt.monday`, `foo.tex.ckpt.last_week`, and so on. When the `name` policy receives the `versions` keyword, it returns the names of any past versions found in the quFile's undo log. A user may use the `versions` keyword to specify all versions within a subtree; for example, `grep bar -Rn src.quFile.versions` searches for bar in all versions of all files in all subdirectories of `src`.

To conserve storage space, we dynamically generate checkpoints of past versions when they are viewed using the `versions` view. The quFile's `content` policy receives one of the names returned by the `name` policy. It dynamically creates a new checkpoint file within the quFile by applying the reverse deltas in succession to the next most recent checkpoint or the current version of the file. In addition to saving storage space, dynamic resolution also saves work in the common case where the user never inspects a past version. The performance hit of instantiating a previous checkpoint is taken only in the uncommon case when a user recovers a past version.

We have also implemented a quFile-aware garbage-collection utility that runs as a cron job and removes older versions to save disk space. One sample policy maintains all prior versions less than one-day old, one version from the previous day, one from the prior two days, and one additional version from each exponentially increasing number of days.

5.3 Availability: Resource-Aware Directories

Distributed file systems typically make no visible distinction between data cached locally and data that must be fetched from a remote server. Unfortunately, the absence of this distinction is often frustrating. For instance, a directory listing might reveal interesting multimedia content that the user tries to view. However, the user subsequently finds out that the content cannot be viewed satisfactorily because it is not cached locally and the network bandwidth to the server is insufficient to sustain the bit rate required to play the content.

To address this problem, we created a resource-aware directory listing policy that uses quFiles to tailor the contents of the directory to match the resources available to the computer. Our policy currently tailors directory listings to reflect cache state and network bandwidth. We can imagine similar policies that tailor listings to match the availability of CPU cycles or battery energy.

If a multimedia file is cached on a computer, the `name` policy's default view returns its name to the application. Otherwise, the policy returns the name of the multimedia file only if the network bandwidth to the server is greater than the bit rate needed to play the file.

The effect of the `name` policy is that a multimedia file is not displayed by directory listings or media players if there is insufficient network bandwidth to play it. Thus, a media player that is shuffling randomly among songs will not experience a glitch when it tries to play an unavailable song. A user will not have to experiment to find out which songs can be played and which cannot.

However, our experience using this policy revealed that we sometimes want to see files that are currently unavailable when we list a directory. For instance, a video player may support buffering, and we are willing to tolerate a delay before we watch a video. We therefore altered the `name` policy to support a custom view that simply changes the name of a file from `foo` to `foo_is_currently_unavailable` when the file is unplayable. The custom view is selected using the keyword `all`; e.g., `ls MyMusic.quFile.all` shows `foo_is_currently_unplayable`, while `ls MyMusic` does not show an entry for that file.

5.4 Security: Context-Aware Data Redaction

Mobile computers may be used at any location, including those that are insecure. For this reason, information scrubbing [Ioannidis et al. 2006] has been proposed to protect, isolate, and constrain private data on mobile devices. For instance, a user may not want to view her bank records or credit card information in a coffee shop or other public venue because others may observe personal or sensitive information by glancing at the screen. To help such users, we created a quFile that shows only redacted versions of files with sensitive data removed when data is viewed at insecure locations. The original data is displayed at secure locations.

This case study redacts only the presentation of data, not the bytes stored on disk. Thus, it guards against inadvertent display of data on a mobile computer, but not against the computer being lost or stolen.

We first created a quFile-aware utility that redacts XML files containing sensitive data. This utility is notified when files that may contain sensitive data are added to the file system. While our utility can redact any XML file using type-specific rules, we currently use it only for GnuCash, a personal finance program that stores data in a binary XML format. GnuCash [2009] runs on Linux and is compatible with the Quicken Interchange Format.

Our utility parses each GnuCash file and generates a redacted version. The general-purpose redactor uses the Xerces [Xerces 2009] XML parser to apply type-specific transformation rules that obfuscate sensitive data. Our current rules obfuscate details such as account numbers, transaction details and dates, but leave the balances visible. Finally, the utility creates a quFile and moves both the original and redacted files into the quFile using its raw view. The redactor generates these two static representations each time the file is modified.

When an application reads this quFile, our context-aware declassification policy determines the location of the mobile computer using a modified version of Place Lab [Nicholson and Noble 2008; Sohn et al. 2006]. If the computer is at a trusted location, as specified by a configuration file, the original version

is returned. Otherwise the redacted version is displayed. Since the file type of the original and redacted versions are the same, the `name` policy returns the same name in all locations; however the data returned by the `content` policy may change as the user moves.

We did not need to modify GnuCash, since it uses the transparent default view. GnuCash simply displays the original or redacted values in its GUI, depending on the location of the mobile computer. A quFile-aware user may override the `content` policy and view a different version using the quFile's raw view; for example, by specifying `/bluefs/credit_card.quFile/credit_card.xml` instead of `/bluefs/credit_card.xml`.

5.5 Application-Aware Adaptation: Odyssey

Odyssey [Noble et al. 1997] introduced the notion of application-aware adaptation, in which the operating system monitors resource availability and notifies applications of any relevant changes. When notified by Odyssey of a resource-level change, applications adjust the fidelity of the data they consume. A drawback of Odyssey is that both the operating system and applications must be modified. However, we observe that almost all application modification is due to implementing the adaptation policy and mechanism inside the application. Thus, we decided to re-implement the functionality of Odyssey using quFiles. Unlike Odyssey, our quFile implementation requires no application modification. The adaptation policy can be removed from the application and cleanly specified using the quFile interface.

Our Odyssey implementation replicates Odyssey's Web (image viewing) application. A similar policy could be used for other Odyssey data types such as speech, maps [Flinn and Satyanarayanan 1999]; and 3-D graphics [Narayanan et al. 2000].

We created a utility that is notified when new JPEG images such as photos are added to the file system. The utility creates four additional lower-fidelity representations of the photo with varying JPEG-quality levels. It creates a quFile, links in our Odyssey policies, and moves the lower-fidelity representations and the original image into the quFile using its raw view.

When a photo viewer lists a directory containing an image quFile, the Odyssey `name` policy returns the name of the original image file. However, when the content of the image is read, the quFile's `content` policy returns the best-quality representation that can be displayed within one second.

The `content` policy uses the context library to determine the client's current bandwidth to the server. It reads the size of each representation in the quFile, starting with the highest-fidelity, original, representation and proceeding to the lowest. If a representation is cached locally or can be fetched from the server in less than a second, the `content` policy returns the inode for that representation. If no representation can meet the service time requirement, the lowest fidelity representation is returned.

The `edit` policy returns a context-specific value. It allows all modifications to the original image, since the quFile-aware transcoder will be notified to regenerate alternate representations from the modified original. However, the

policy disallows modifications to multimedia data in low-fidelity representations because it is unclear how such modifications can be reflected back to the original and other representations. This behavior is similar to the one users see in other arenas (e.g., when they try to save an Office document in a reduced-fidelity format such as ASCII text).

After experimenting with this policy, we made two further refinements. First, we realized that most edits to multimedia files change only the metadata header, which is identical across formats and quality levels. Thus, we modified our policy to allow editing of metadata for low-fidelity representations. The transcoder propagates metadata changes to other representations.

We also realized that some image editors rewrite the entire image instead of just modifying its metadata. We therefore modified our `edit` policy to allow writes outside the metadata region if the data written is identical to the data in the file. With these changes, all edits we attempted to make to low-fidelity versions succeeded. Of course, this is just one policy, and different applications may craft other policies such as allowing edits to low-fidelity data or creating multiple versions.

5.6 Platform-Specific Video Display

Section 4.1 gave a brief overview of our last case study, which transcodes videos to meet the resource constraints of file system clients. The authors currently use TiVo DVRs, N800 Internet tablets, and laptop computers to display videos. When a new `.TiVo` file is recorded and stored in BlueFS, a quFile-aware utility generates a full-resolution `.mp4` for the laptop and a lower-fidelity `.mp4` representation for the Nokia N800. Since the N800 has a lower screen resolution, we can save storage space on that device by producing a video formatted specifically for the N800's smaller display. The utility creates a quFile and populates it with the original and transcoded videos for each computer type described above. If we were to use additional types of clients, our transcoder could produce versions for those devices.

The `name` and `content` policies query the machine type on which they are running using the context library described in Section 4.5. The `name` policy returns a name ending with `.TiVo` when the video is read by the DVR, as determined by seeing that the name of the requesting application is a TiVo-specific utility. Otherwise, the `name` policy returns a name ending with `.mp4`. The `content` policy determines the type of client using the context library and returns the encoding appropriate for that type. The `cache_insert` policy ensures that each device only caches the video encoding it will display. We use BlueFS' type-specific affinity to prefetch such encodings to each device. quFiles hide this manipulation from video display applications, which therefore do not need to be modified. In practice, we found that this cached store of videos on the N800 made many a busride more enjoyable! We also implemented a simple eviction policy: when the device is running out of storage space, all prefetched recordings are deleted before content the user has explicitly cached.

6. EVALUATION

While the case studies in the previous section illustrate the generality of quFiles, we also verified that quFiles do not add too much overhead to file system operations and that the amount of code required to implement quFile policies is reasonable.

Unless stated otherwise, we evaluated quFiles on a Dell GX620 desktop with a 3.4 GHz Pentium 4 processor and 3 GB of DRAM. The desktop runs Ubuntu Linux 8.04 (Linux kernel 2.6.24). The desktop runs both the BlueFS server and client, and the BlueFS client does not use a local disk cache.

We executed each experiment in three scenarios. In the *warm client* scenario, the kernel's page cache contains all BlueFS data read during the experiment (the working sets of all experiments fit in memory). In the *cold client* scenario, no client data is in the kernel's page cache, but all server data is initially in the page cache. Thus, the first time an application reads a file page or attributes, an RPC is made to the server but no disk access is required. In the *cold server* scenario, no data is initially in any cache. On the first read, an RPC and a disk access are required to retrieve the data.

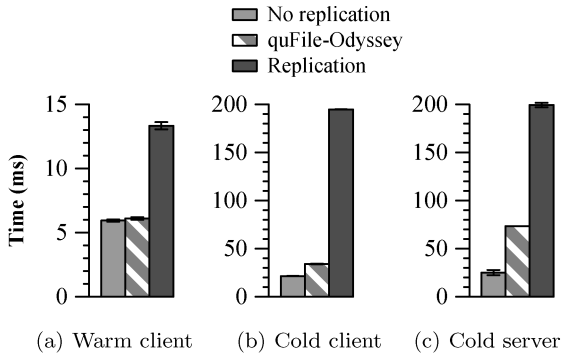
6.1 Directory Listing

Our first experiment evaluates the performance overhead of quFiles for common file system operations by measuring the time to list the files in a directory and their attributes with the command `ls -al`. This is a worst-case scenario for using quFiles, since the listing incurs the overhead of retrieving a quFile and executing both the `name` and `content` policies to determine which attributes to return for each file. Yet there is minimal additional work to amortize this overhead because the directory listing requires that only the attributes of the file being listed be retrieved.

In our experiment, a directory contains 100 JPEG images. Each image is placed in a quFile that contains four additional low-fidelity representations and returns the appropriate one for the available server bandwidth using the Odyssey policy in Section 5.5.

The first bar for each scenario in Figure 2 shows a lower performance bound generated by assuming that Odyssey-like functionality is completely unsupported. Each value shows the time to list a directory without quFiles that contains only the original 100 JPEG images.

The second bar in each scenario shows the time to list the directory using quFiles. The Odyssey `name` and `content` policies return the name and content of the original image since server bandwidth is abundant. If the client cache is warm (which we expect to be the common case for most file system operations), quFiles add less than 3% overhead for this experiment (roughly 1.6 μs per file). If the client cache is cold, quFiles add 59% overhead. For each file, quFiles execute two policies. There is a measured overhead of 28 μs per policy, almost entirely due to user-level sandboxing. An additional 70 μs per file is required to fetch quFile attributes and contents from the server. If both the client and server caches are cold, the server performs two disk reads per file to read the quFile attributes and data. In this case, quFiles impose slightly less than a



Each value is the mean of 10 trials; error bars are 90% confidence intervals. Note that the scales of the three graphs differ.

Fig. 2. Time to list a directory with 100 images.

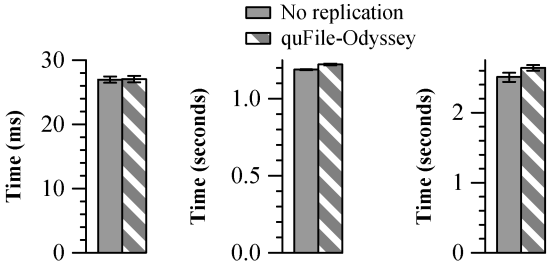
3x overhead because disk reads are the dominant cost and three reads per file are performed with quFiles while only one read is performed without quFiles. However, it should be noted that even when both caches are cold, quFiles impose only 0.48 ms of overhead per file in this worst-case scenario. Note that the relative overhead of quFiles would decrease if file accesses were more random, since as directories, quFiles can be placed on disk near the files they contain (minimizing seeks).

While the first bar in each scenario in the figure provides a lower bound on performance, a fairer comparison for Odyssey with quFiles is one in which all representations are stored together in the same directory. Odyssey uses this storage method for video, map, and speech data [Noble et al. 1997; Flinn and Satyanarayanan 1999]. Thus, there are 500 files in the directory. As the last bar in each scenario in Figure 2 shows, listing the directory takes over twice as long without quFiles in the warm client and cold server scenarios, and over five times as long in the cold client scenario. Because each quFile encapsulates many representations but returns only one; quFiles fetch less data than a regular file system when a naive storage layout policy is used.

Overall, we conclude that quFiles add minimal overhead to common file system operations, especially when the client cache is warm. Compared to naive file system layouts, quFiles can sometimes improve performance through their encapsulation properties.

6.2 Reading Data

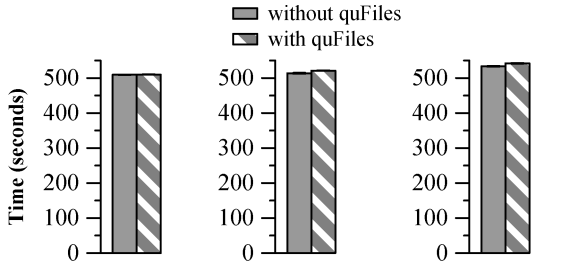
Often, users and applications will read file data, not just file attributes. We therefore ran a second microbenchmark that measures the time taken by the `cat` utility to read all images in our test directory and pipe the output to */dev/null*. As Figure 3 shows, quFile overhead is negligible in the warm client scenario; 3% in the cold client scenario; and 5% in the cold server scenario. Although the total overhead of quFile indirection remains the same as in the



(a) Warm client (b) Cold client (c) Cold server

Each value is the mean of 10 trials; error bars are 90% confidence intervals. Note that the scales of the three graphs differ.

Fig. 3. Time to read 100 images.



(a) Warm client (b) Cold client (c) Cold server

Each value is the mean of 5 trials; error bars are 90% confidence intervals.

Fig. 4. Time to make the Linux kernel.

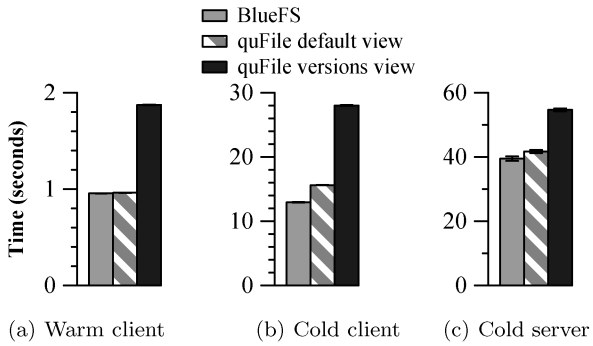
previous experiment, that overhead is now amortized across more file system activity. Thus, relative overhead decreases substantially.

6.3 Andrew-Style Make Benchmark

We next turned our attention to application-level benchmarks. We started with a benchmark that measures quFile overhead during a complete `make` of the Linux 2.6.24-2 kernel. Such benchmarks, while perhaps not representative of modern workloads, have long been used to stress file system performance [Howard et al. 1988].

We compare the time to build the Linux kernel on BlueFS with and without quFiles. For the quFile test, we created a kernel source tree in which all source files (ending in .c, .h, or .S) are versioned using the copy-on-write quFile described in Section 5.2. The kernel source tree contains 23,062 files, of which 19,844 are versioned. Each quFile contains the original file and a checkpoint of approximately the same size as the original.

As Figure 4 shows, quFiles add negligible overhead in the warm client scenario and 1% overhead in the cold client and cold server scenarios. Even though



Each value is the mean of 5 trials; error bars are 90% confidence intervals. Note that the scales of the three graphs differ.

Fig. 5. Time to search through the Linux kernel.

kernel source files are quite small (averaging 11,663 bytes per file), many files such as headers are read multiple times, meaning that the extra overhead of fetching quFile data from the server can be amortized across multiple file reads. Further, computation is a significant portion of this benchmark, reducing the performance impact of I/O.

6.4 Kernel Grep

We next ran a read-only benchmark that stresses file I/O performance. We used `grep` to search through the Linux source tree described in the previous section to find all nine occurrences of "remove_wait_queue_locked".

The first bar in each scenario of Figure 5 shows the time to search through the Linux source without quFiles. The second bar in each scenario shows the time to search through the source with quFiles using the default view. In this case, each quFile returns only the current version of each source file. Thus, the results returned by the two `grep` commands are identical.

In the warm client scenario, the performance of `grep` with quFiles is within 1% of the performance without quFiles. As we would expect, the overhead is larger when there is no data in the client cache: 21% in the cold cache scenario and 6% in the cold server scenario.

quFiles, however, allow greater functionality than a regular file system. For instance, we can search through not only the current versions of source files but also all past versions by simply executing `grep -Rn linux.quFile.versions` where `linux` is the root of the kernel source tree. This command, which uses the `versions` view of the copy-on-write quFile, searches through twice as much data and returns 18 matches.

The last bar in each scenario shows the time to execute `grep` using the `versions` view. Since approximately twice as much data is read, the version-aware search takes approximately twice as long as a search using the default view in the warm client scenario. However, in the cold server scenario, the search takes only 31% longer since quFile representations are located close to each other on disk, reducing seek times.

Table II. Lines of Code for quFile Policies

Component	Name	Content	Edit	Cache	Total
Resource mgmt.	32	18	8	36	94
Versioning	29	18	8	n/a	55
Security	20	33	8	n/a	61
Availability	64	26	8	n/a	98
Odyssey	23	27	32	n/a	82
Platform spec.	31	30	8	43	112

Table III. Power Savings Enabled by quFiles

Device	Power to Play mp3 Files (mW)	Power with quFiles (mW)	Battery Life Extension
HP4700 iPAQ	1549	1401	11%
Nokia N95-1	962	914	5%
Nokia N95-3	454	437	4%

This table compares the power used to play mp3 files on 3 mobile devices with the power required to play the uncompressed versions returned by quFiles.

This scenario shows that even when there is little data or computation across which to amortize overhead, performance is still reasonable, especially when data resides in the kernel's page cache. Further, quFiles enable functionality that is unavailable using regular file systems.

6.5 Code Size

We measure the effort required to develop new policies by counting the lines of code for the quFiles used in each of our six case studies. As Table II shows, almost all policies required less than 100 lines of code. Compared to the code size of their monolithic ancestors, these numbers represent a dramatic reduction. For instance, the base Odyssey source is comprised of 32,329 lines of code, while ext3cow requires a 18,494 line patch to the Linux-2.6.20.3 source tree. Our quFile implementation added 1,515 lines of code to BlueFS (BlueFS has 28,788 lines of code without quFiles). Further, all policies were implemented by a single graduate student. All policies took less than two weeks to implement; later policies required only a few days as we gained experience.

6.6 Energy Saving Results

To evaluate the effectiveness of our case study in Section 5.1 that plays uncompressed music files to save energy, we measured the power used to play the uncompressed version of music files returned by quFiles and the power used to play the equivalent mp3 files. Table III shows results for three mobile devices: an HP4700 iPAQ handheld, Nokia N95-1, and N95-3 smart phones. The iPAQ runs Familiar v8.4, with OpiePlayer as its media player while the N95-1 and N95-3 ran their factory-installed operating system and media players.

We directly measured the power consumed on the iPAQ by removing its battery and connecting its power supply cable through a digital multimeter. Unfortunately, the Nokia smart phones cannot operate with their battery unplugged, so instead we used the Nokia Energy Profiler [Nokia] to measure

playback power. Our tests show that quFiles can increase the battery lifetime of these devices by 4 to 11% when they are playing music. Given the importance of battery lifetime for these devices, this is a nice gain, especially since only spare resources are used to achieve it.

7. FUTURE WORK

Our current system only supports one resolution policy per quFile. While this suffices for the case studies we have described, a user might wish to combine multiple policies to suit his needs. For instance, a user might associate both the Odyssey (Section 5.5) and copy-on-write (Section 5.2) policies with his image quFiles so he can see the highest resolution image given his network quality and also maintain a past version if he happens to edit the image. To support this usage model, we plan to make quFile policies *composable*. Since each quFile policy effectively filters an input set of representations to an appropriate output set based on the current context, policies can be chained together so that the output of a policy is the input to the subsequent one.

Composing quFile policies is difficult because of our design goal that quFiles support both static and dynamic content. Specifically, we designed quFiles so the `name` and `content` policies can dynamically instantiate new filenames and content as desired (Section 3.3); while the `edit` policy dynamically decides whether a modification to the content is allowed, disallowed, or forces the creation of a new version. The challenge in implementing composability is that quFile policies might not always *commute*, that is, the final state of a quFile's content might be dependent on the order in which individual policies are executed. For instance, when the user edits an image, either the Odyssey `edit` policy (which only permits modifications to the image's metadata) or the copy-on-write `edit` policy (which versions the entire image) might govern how the edit is handled, depending on the order in which the policies are applied. In our current system, the final state of the resulting image file is undefined. One way to address this challenge is to require that policies be prioritized so they have a defined ordering, or perhaps also to introduce a stricter requirement that composable policies cannot dynamically change a quFile's content.

We would also like to integrate quFiles with desktop search engines [Google Desktop 2008; Windows Desktop Search 2008; Spotlight 2007]. Desktop search engines maintain a custom index of all files on a computer and provide users with a simple search facility as an alternate mechanism for accessing their data. We intend to implement a new *search* view that presents a custom view of the quFile, one that can be easily indexed, to desktop search engines. The challenge here is in determining which of a quFile's representations ought to be exposed to the search engine so that the user's expectation is met for all access contexts. A simple solution would be to only expose the most constrained representation (which might not always exist); but this is unoptimal, as it hides the richer content of a full-fidelity representation that might be available given the user's access context. It is plausible that good integration can only be achieved if the search engine is made quFile-aware, so that it searches over different indexes based on the access context.

8. CONCLUSION

The quFile abstraction simplifies data management by providing a common mechanism for selecting one of several possible representations of the same logical data, depending on the context in which it is accessed. A quFile also encapsulates the messy details of generating and storing multiple representations and the policies for selecting among them. We have shown the generality of quFiles by implementing six case studies that use them.

ACKNOWLEDGMENTS

We thank Mona Attariyan, Dan Peek, Doug Terry, Benji Wester, Kim Keeton, our shepherd Karsten Schwan, and the anonymous reviewers for comments that improved this article. We used David A. Wheeler's SLOCCount to estimate the lines of code for our implementation.

REFERENCES

ANAND, M., NIGHTINGALE, E. B., AND FLINN, J. 2003. Self-tuning wireless network power management. In *Proceedings of the 9th Annual Conference on Mobile Computing and Networking*. 176–189.

BELARAMANI, N., DAHLIN, M., GAO, L., NAYATE, A., VENKATARAMANI, A., YALAGANDULA, P., AND ZHENG, J. 2006. PRACTI replication. In *Proceedings of the 3rd Symposium on Networked System Design and Implementation*. 59–72.

BERSHAD, B., SAVAGE, S., PARDYAK, P., SIRER, E., FIUCZYNSKI, M., BECKER, D., CHAMBERS, C., AND EGGERS, S. 1995. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. ACM, New York, 267–284.

BERSHAD, B. B. AND PINKERTON, C. B. 1988. Watchdogs—Extending the UNIX file system. *Comput. Syst. 1*, 2.

BILA, N., RONDA, T., MOHOMED, I., TRUONG, K. N., AND DE LARA, E. 2007. PageTailor: Reusable end-user customization for the mobile web. In *Proceedings of the 5th International Conference on Mobile Systems, Applications and Services*. 16–29.

BUNDLE. 2009. *Bundle Programming Guide*. http://developer.apple.com/documentation/CoreFoundation/Conceptual/CFBundles/CFBundles.html.

DE LARA, E., KUMAR, R., WALLACH, D. S., AND ZWAENEPOEL, W. 2003. Collaboration and multimedia authoring on mobile devices. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*. 287–301.

DE LARA, E., WALLACH, D. S., AND ZWAENEPOEL, W. 2001. Puppeteer: Component-based adaptation for mobile computing. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*. 159–170.

DOURISH, P., EDWARDS, W. K., LAMARCA, A., LAMPING, J., PETERSEN, K., SALISBURY, M., TERRY, D. B., AND THORNTON, J. 2000. Extending document management systems with user-specific active properties. *ACM Trans. Inform. Syst. 18*, 2, 140–170.

ENGLER, D., KAASHOEK, M., AND J. O'TOOLE, J. 1995. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. 251–266.

FLINN, J. AND SATYANARAYANAN, M. 1999. Energy-aware adaptation for mobile applications. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*. 48–63.

FOX, A., GRIBBLE, S. D., BREWER, E. A., AND AMIR, E. 1996. Adapting to network and client variability via on-demand dynamic distillation. In *Proceedings of the 7th International ACM Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York,160–170.

FUSE. 2009. Filesystem in userspace. http://fuse.sourceforge.net/.

GEHANI, N. H., JAGADISH, H. V., AND ROOME, W. D. 1994. OdeFS: A file system interface to an object-oriented database. In *Proceedings of the 20th International Conference on Very Large Databases*. 249–260.

GIFFORD, D. K., JOUVELOT, P., SHELDON, M. A., AND O'TOOLE, J. W. 1991. Semantic file systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*. ACM, New York, 16–25.

GNUCASH 2009. GnuCash: Free accounting software. http://www.gnucash.org.

GOOGLE DESKTOP. 2008. Google desktop. http://desktop.google.com.

GUPTA, A. AND MUMICK, I. S. 1995. Maintenance of materialized views: Problems, techniques and applications. *IEEE Q. Bull. Data Eng; 18*, 2, 3–18 (Special issue on materialized views and data warehousing).

HOWARD, J. H., KAZAR, M. L., MENEES, S. G., NICHOLS, D. A., SATYANARAYANAN, M., SIDEBOTHAM, R. N., AND WEST, M. J. 1988. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst. 6*, 1, 51–81.

IOANNIDIS, S., SIDIROGLOU, S., AND KEROMYTIS, A. D. 2006. Privacy as an operating system service. In *Proceedings of the 1st Conference on USENIX Workshop on Hot Topics in Security*. USENIX Association, Monterey, CA, 45–50.

KISTLER, J. J. AND SATYANARAYANAN, M. 1992. Disconnected operation in the Coda file system. *ACM Trans. Comput. Syst. 10*, 1.

KJÆR, K. 2007. A survey of context-aware middleware. In *Proceedings of the IASTED International Conference on Software Engineering*. 148–155.

LOPRESTI, D. P. AND LAWRENCE, S. A. 2005. Information leakage through document redaction: Attacks and countermeasures. In *Proceedings of Document Recognition and Retrieval XII - International Symposium on Electronic Imaging*. 183–190.

LOVE, R. 2005. Kernel Korner: Intro to inotify. *Linux J. 139*, 8.

MA, X. AND REDDY, A. L. N. 2003. MVSS: Multiview storage system. *IEEE Trans, Parall. Distrib. Syst. 14*, 10, 993–1005.

NARAYANAN, D., FLINN, J., AND SATYANARAYANAN, M. 2000. Using history to improve mobile application adaptation. In *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*. IEEE, Los Alamitos, CA, 30–41.

NICHOLSON, A. J. AND NOBLE, B. D. 2008. BreadCrumbs: Forecasting mobile connectivity. In *Proceedings of the 14th International Conference on Mobile Computing and Networking*. 46–57.

NIGHTINGALE, E. B. AND FLINN, J. 2004. Energy-efficiency and storage flexibility in the Blue file system. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*. 363–378.

NOBLE, B. D., SATYANARAYANAN, M., NARAYANAN, D., TILTON, J. E., FLINN, J., AND WALKER, K. R. 1997. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*. ACM, New York, 276–287.

NOKIA. Nokia Energy Profiler. http://www.forum.nokia.com/main/resources/development_ process/power_management/nokia_energy_profiler/.

PEEK, D. AND FLINN, J. 2006. EnsemBlue: Integrating distributed storage and consumer electronics. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. 219–232.

PEEK, D., NIGHTINGALE, E. B., HIGGINS, B. D., KUMAR, P., AND FLINN, J. 2007. Sprockets: Safe extensions for distributed file systems. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, Monterey, CA, 115–128.

PETERSON, Z. N. J. AND BURNS, R. 2005. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Trans. Storage 1*, 2, 190–212.

PHAN, T., ZORPAS, G., AND BAGRODIA, R. 2004. Middleware support for reconciling client updates and data transcoding. In *Proceedings of the 2nd International Conference on Mobile Systems, Applications and Services*. 139–152.

PILLAI, P., KE, Y., AND CAMPBELL, J. 2004. Multi-fidelity storage. In *Proceedings of the ACM 2nd International Workshop on Video Surveillance and Sensor Networks*. 72–79.

RAMASUBRAMANIAN, V., RODEHEFFER, T. L., TERRY, D. B., WALRAED-SULLIVAN, M., WOBBER, T., MARSHALL, C. C., AND VAHDAT, A. 2009. Cimbiosys: A platform for content-based partial replication. In *Proceedings of the 6th Symposium on Networked System Design and Implementation*. 261–276.

RUSSINOVICH, M. E. AND SOLOMON, D. A. 2005. Advanced features of NTFS. In *Microsoft Windows Internals*, 719–721.

SALMON, B., SCHLOSSER, S. W., CRANOR, L. F., AND GANGER, G. R. 2009. Perspective: Semantic data management for the home. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies*. 167–182.

SANTRY, D. S., FEELEY, M. J., HUTCHINSON, N. C., VEITCH, A. C., CARTON, R. W., AND OFIR, J. 1999. Deciding when to forget in the Elephant file system. *SIGOPS Oper. Syst. Rev. 33*, 5, 110–123.

SCHILIT, B., ADAMS, N., AND WANT, R. 1994. Context-aware computing applications. In *IEEE Workshop on Mobile Computing Systems and Applications*. IEEE, Los Alamitos, CA, 85–90.

SELTZER, M. I., ENDO, Y., SMALL, C., AND SMITH, K. A. 1996. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*. 213–227.

SHEPLER, S., CALLAGHAN, B., ROBINSON, D., THURLOW, R., BEAME, C., EISLER, M., AND NOVECK, D. 2003. Network file system (NFS) ver. 4 protocol. Tech. rep. RFC 3530, IETF.

SOHN, T., GRISWOLD, W. G., SCOTT, J., LAMARCA, A., CHAWATHE, Y., SMITH, I., AND CHEN, M. 2006. Experiences with Place Lab: An open source toolkit for location-aware computing. In *Proceedings of the 28th International Conference on Software Engineering*. 462–471.

SPOTLIGHT. 2007. Spotlight overview. http://developer.apple.com/documentation/Carbon/ Conceptual/MetadataIntro/MetadataIntro.pdf.

WINDOWS DESKTOP SEARCH. 2008. Windows desktop home page. http://www.microsoft.com/windows/desktopsearch.

XERCES. 2009. Xerces-C++ XML Parser. http://xerces.apache.org/xerces-c/.

YUMEREFENDI, A. R., MICKLE, B., AND COX, L. P. 2007. TightLip: Keeping applications from spilling the beans. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation*. 159–172.

Received April 2010; revised May 2010; accepted June 2010