Energy-Efficiency and Storage Flexibility in the Blue File System

Edmund B. Nightingale and Jason Flinn
Department of Electrical Engineering and Computer Science
University of Michigan

Abstract

A fundamental vision driving pervasive computing research is access to personal and shared data anywhere at anytime. In many ways, this vision is close to being realized. Wireless networks such as 802.11 offer connectivity to small, mobile devices. Portable storage, such as mobile disks and USB keychains, let users carry several gigabytes of data in their pockets. Yet, at least three substantial barriers to pervasive data access remain. First, power-hungry network and storage devices tax the limited battery capacity of mobile computers. Second, the danger of viewing stale data or making inconsistent updates grows as objects are replicated across more computers and portable storage devices. Third, mobile data access performance can suffer due to variable storage access times caused by dynamic power management, mobility, and use of heterogeneous storage devices. To overcome these barriers, we have built a new distributed file system called BlueFS. Compared to the Coda file system, BlueFS reduces file system energy usage by up to 55% and provides up to 3 times faster access to data replicated on portable storage.

1 Introduction

Storage technology for mobile computers is evolving rapidly. New types of storage such as flash memory offer performance and energy characteristics that differ substantially from those of disk drives. Further, the limited battery lifetime of mobile computers has made power management a critical factor in the design of mobile I/O devices. Finally, a dramatic improvement in storage capacity is enabling users to carry gigabytes of data in a pocket-sized device. Collectively, these changes are causing several of the assumptions made in the design of previous mobile file systems to no longer hold.

In this paper, we present the Blue File System (BlueFS), a distributed file system for mobile computing that addresses the new opportunities and challenges created by the evolution of mobile storage. Rather than embed static assumptions about the performance and availability of storage devices, BlueFS has a *flexible cache hierarchy* that adaptively decides when and where to access data based upon the performance and energy characteristics of each available device. BlueFS's flexible cache hierarchy has three significant benefits: it extends battery lifetime through energy-efficient data access, it lets BlueFS efficiently support portable storage, and it improves performance by leveraging the unique characteristics of heterogeneous storage devices.

BlueFS uses a *read from any, write to many* strategy. A kernel module redirects file system calls to a user-level daemon, called Wolverine, that decides when and where to access data. For calls that read data, Wolverine orders attached devices by the anticipated performance and energy cost of fetching data, then attempts to read information from the least costly device. This lets BlueFS adapt when portable storage devices are inserted or removed, when the dynamic power management status of a device changes, or when wireless network conditions fluctuate.

For calls that write data, Wolverine asynchronously replicates modifications to all devices attached to a mobile computer. Wolverine aggregates modifications in per-device write queues that are periodically flushed to storage devices. Asynchrony hides the performance cost of writing to several devices, while aggregation saves energy by amortizing expensive power state transitions across multiple modifications.

We have implemented BlueFS as a Linux file system that runs on both laptops and handhelds. Our results show that BlueFS reduces file system energy usage by up to 55% and interactive delay by up to 76% compared to the Coda distributed file system. BlueFS also provides up to 3 times faster access to data replicated on portable storage. Further, BlueFS masks the performance impact of disk power management and leverages new types of storage technology such as flash memory to save power and improve performance.

We begin in the next section with a discussion of related work. Sections 3 and 4 outline BlueFS's design goals and implementation. We present an evaluation of BlueFS in Section 5 and discuss our conclusions in Section 6.

2 Related work

BlueFS distinguishes itself from prior distributed file systems by using an adaptive cache hierarchy to reduce energy usage and efficiently integrate portable storage.

Energy management has not been a primary design consideration for previous distributed file systems; however, several recent projects target energy reduction in the domain of local file systems. Researchers have noted the energy benefits of burstiness for local file systems [6, 15] and have provided interfaces that enable applications to create burstier access patterns [24]. Rather than defer this work to applications, BlueFS automatically creates burstiness by shaping device access patterns. BlueFS further reduces client energy usage by dynamically monitoring device power states and fetching data from the device that will use the least energy. Finally, BlueFS proactively triggers power mode transitions that lead to improved performance and energy savings.

Several ideas that improve file system performance may also lead to reduced energy usage. These include the use of content-addressable storage in systems such as LBFS [12] and CASPER [21], as well as memory/disk hybrid file systems such as Conquest [23]. We believe that these ideas are orthogonal to our current design.

Previous file systems differ substantially in their choice of cache hierarchy. NFS [13] uses only the kernel page cache and the file server. AFS [8] adds a single local disk cache — however, it always tries to read data from the disk before going to the server. xFS [3] employs cooperative caching to leverage the memory resources of networked workstations as a global file cache. Each static hierarchy works well in the target environment envisioned by the system designers precisely because each environment does not exhibit the degree of variability in device access time seen by mobile computers. In contrast, BlueFS's use of an adaptive hierarchy is driven by the expectation of a highly dynamic environment in which access times change by orders of magnitude due to dynamic power management, network variability, and devices with different characteristics.

The file system most similar to BlueFS is Coda [9], which is also designed for mobile computing. Like AFS, Coda uses a single disk cache on the client — this cache is always accessed in preference to the server. BlueFS borrows several techniques from Coda including support for disconnected operation and asynchronous reintegration of modifications to the file server [11].

Unlike BlueFS, Coda has not made energy-efficiency a design priority. Although Coda did not originally support portable storage, recent work by Tolia et al. [20] has used lookaside caching to provide limited, read-only support for portable devices. Lookaside caching indexes files on a portable device by their SHA-1 hash. If Coda does not find a file in its local disk cache, it retrieves the attributes and SHA-1 hash for that file from the server. If a file with the same hash value exists in the index, Coda reads the file from portable storage.

Tolia's work examines how one can best support portable storage while making only minimal changes to an existing file system and its usage model. In contrast, our work integrates support for portable storage into the file system from the start. Our clean-sheet approach yields two substantial benefits. First, BlueFS support is not read-only. BlueFS keeps each portable device up to date with the latest files in a user's working set and with the latest version of files that have affinity for the device. Since all modifications written to a portable device are also written to the file server, BlueFS does not lose data if a portable device is misplaced or stolen. Second, BlueFS provides substantial performance improvements by maintaining consistency at the granularity of storage devices rather than clients. This lets BlueFS access files on portable storage devices without fetching attributes or hashes from the server.

PersonalRAID [17] uses portable storage to synchronize disconnected computers owned by the same user. The target environment differs from that of a distributed file system in that data is only shared between users and computers through synchronization with the portable device. Also, one must have the portable device in order to access data on any computer. Since PersonalRAID does not target mobile computers explicitly, it does not make energy-efficiency a design goal.

Bayou [19], Segank [18], and Footloose [14] take a peer-to-peer approach to mobile computing. These systems allow cooperating computers to share data through pairwise exchange. Many of the principles employed in BlueFS such as adaptive caching to multiple devices and tight integration with power management could also be applied in such peer-to-peer systems.

3 Design goals

In this section, we outline the three primary design goals of the Blue File System. We also describe how BlueFS addresses each goal.

3.1 Change is constant

Our first design goal is to have BlueFS automatically monitor and adapt to the performance and energy characteristics of local, portable, and network storage. Mobile storage performance can vary substantially due to:

- *deployment of new types of storage devices*. In recent years, some handheld computers have used flash or battery-backed RAM for primary storage. Compared to disk storage, flash offers excellent performance for read operations but poor performance for small writes [25]. Battery-backed DRAM offers excellent performance for reads and writes, but can sacrifice reliability. Finally, the performance of new portable devices such as mobile disks often lags far behind that of their fixed counterparts [26].

- *the impact of power management*. Storage and network devices use aggressive power management to extend the battery lifetime of mobile computers. However, the performance impact of power management is often substantial. Consider disk drives that cease rotation during idle periods to save energy [5]. The next access is delayed several hundred milliseconds or more while the disk spins up. Similarly, when network interfaces are disabled or placed in power saving modes, the latency to access remote servers increases greatly [1]. The end result is that storage access latency can change by several orders of magnitude when a device transitions to a new power mode.
- *network variability*. Access times for remote storage are affected by fluctuations in network performance due to mobility. For example, when latency to a network server is low, the best performance will often be achieved by eschewing local storage and fetching data directly from the server. When latency is high, local or portable storage is best.

BlueFS adapts to storage variability by passively monitoring the performance of each local, portable, and remote storage option available to the mobile computer. It also monitors the power state of each device. When reading data, BlueFS orders available storage options by estimated performance and energy cost. It first tries to obtain needed data from the lowest cost device. If that option fails, it tries the next device. To mitigate the impact of searching devices that do not contain the desired data, it maintains an *enode cache* that records which devices hold recently used files.

On the write path, BlueFS places modifications in per-device operation logs and asynchronously writes changes to each device. This substantially improves performance when writing to devices with high latency such as flash memory and distant network storage.

One can reasonably expect that mobile storage will continue to evolve as new technologies such as MEMS-based storage [16] become available. A file system that embeds static assumptions about device performance may soon be outdated. In contrast, a flexible cache hierarchy offers the possibility of adapting to new devices.

3.2 Power to the people

BlueFS's second design goal is to extend client battery lifetime. Energy is often the critical bottleneck on mobile computers, especially for small devices such as handhelds. On one hand, manufacturers are adding additional capabilities such as ubiquitous network access through 802.11 interfaces, more powerful processors, and larger storage capacities. On the other hand, the amount of energy supplied by batteries is improving slowly [10]. Despite this disconnect, mobile computers have been able to maintain reasonable battery lifetimes by implementing more efficient power management strategies at all system layers.

The energy cost of ubiquitous data access can be high because distributed file systems make heavy use of power-hungry storage and network devices. However, the energy expended by current file systems is often much higher than necessary, simply because energy-efficiency was not an important goal in their design. In contrast, BlueFS contains several mechanisms that substantially reduce energy consumption.

First, BlueFS considers energy cost in addition to performance when deciding from which device it will read data. Second, BlueFS aggregates write operations to amortize the energy cost of spinning up disk drives and powering on network interfaces.

Third, BlueFS integrates with device power management strategies. It uses self-tuning power management interfaces [1, 2] to disclose hints about the devices that it intends to access. The operating system uses these hints to proactively transition devices to save time and energy.

Finally, BlueFS enables the use of more aggressive power management by masking the performance impact of device transitions. For example, many users disable disk power management due to the annoying delays that occur when accessing data after the disk has spun down. BlueFS hides these delays by initially fetching data from the network while it spins up the disk. By limiting the perceived performance impact of power management, BlueFS may increase the adoption of more aggressive power management policies.

3.3 You *can* take it with you

Our last design goal is transparent support for portable storage such as USB keychains and mobile disks. The capacity of these devices is currently several gigabytes and is growing rapidly. Given a wide-area connection that offers limited bandwidth or high latency, accessing data on portable storage is substantially faster than reading the data from a file server.

Yet, portable storage is not a panacea. Most users currently manage their portable storage with manual copying or file synchronization tools [22]. These methods

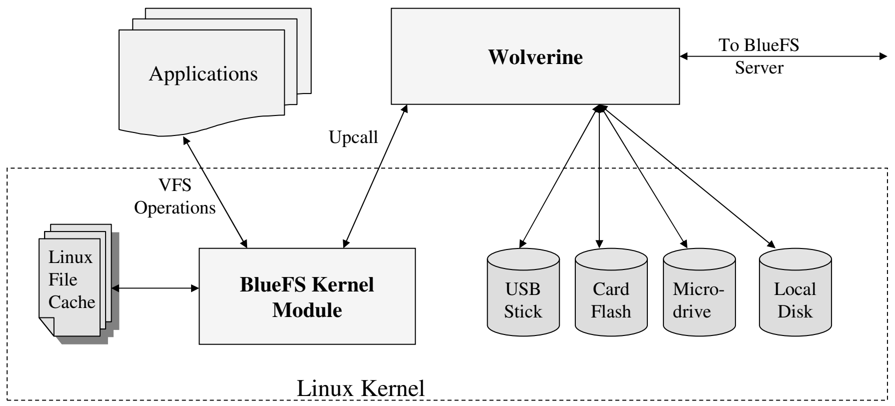


Figure 1. Blue File System architecture

have three potential disadvantages. First, since files on portable storage are not automatically backed up, vital data may be lost if a portable device is misplaced, damaged, or stolen. Second, data on portable storage cannot be accessed from remote computers or shared with other users when the device is not attached to a computer with a network connection. Third, the potential for accessing out-of-date versions or creating inconsistent copies of a file increases greatly when the user manually manages the consistency of file replicas.

BlueFS addresses these problems by having each portable device act as a persistent file cache. BlueFS presents a single system image across all computers and portable storage devices. The primary replica of each block of data resides on the file server. Client and portable storage devices store second-class replicas that are used only to improve performance and reduce energy usage. No data is lost when a portable device is lost, stolen, or damaged since the primary replica always resides on the server.

Blocks cached on portable storage devices are accessible through the BlueFS namespace. When a new device is inserted into a computer, BlueFS adds the device to its search path. On subsequent read operations, BlueFS transparently fetches files from the portable device if that device offers better performance and lower energy usage than other storage options.

When files are modified, BlueFS asynchronously writes modifications to all storage devices attached to the computer, as well as to the file server. Propagation of data to the file server greatly reduces the chance of data loss since files never reside solely on a portable device. Further, files are available from the server even when the portable device is disconnected. Of course, mobile users may wish to access cached data when disconnected from the network. In this case, BlueFS uses techniques pioneered by Coda [9] to allow disconnected operation.

For example, consider a user with a USB keychain who accesses data on computers at home and at work. When the user creates the file `/bfs/myfile` at work, BlueFS transparently replicates it on the keychain. The user brings the keychain home and inserts it into the computer there. When the file is next accessed, BlueFS fetches it from the keychain instead of from a distant server. The only change that the user notices is that performance is substantially better using the keychain. If the user leaves the keychain at work by mistake, the file is retrieved from the file server instead.

Compared to manual copy, BlueFS greatly reduces the potential for data inconsistency by making individual storage devices first-class entities in the file system. The BlueFS file server maintains per-device callbacks for each file. On modification, the server sends an invalidation to the client to which the portable device is currently attached. If the device is disconnected, the invalidation is queued and delivered when the device is next attached to a client. Note that BlueFS cannot eliminate all inconsistencies because it allows disconnected operation and writes modifications asynchronously to improve performance and energy efficiency. Concurrent updates to the same file may create conflicts where two versions of the file exist. As in many other systems that support weaker consistency, such conflicts are automatically detected and flagged for manual or automatic resolution [9].

Manual copy guarantees that a replica of the file, no matter how out of date, will exist on a device until it is deleted. BlueFS provides a stronger guarantee through *affinity*, which lets users specify that a particular device should always store the latest version of a file or subtree. When a file with affinity is invalidated, BlueFS automatically fetches the new version and stores it on the device.

4 Implementation

4.1 Overview

Figure 1 shows the BlueFS architecture. To simplify implementation, we perform the bulk of the client functionality in a user-level daemon called Wolverine. A minimal kernel module, described in the next section, intercepts VFS calls, interfaces with the Linux file cache, and redirects operations to Wolverine. Wolverine, described in Section 4.3, handles reading and writing of data to multiple local, portable, and remote storage devices. The BlueFS server, described in Section 4.4, stores the primary replica of each object (i.e. each file, directory, symlink, etc.)

BlueFS associates a 96-bit identifier with each object: the first 32 bits specify an administrative volume [8], and the remaining 64 bits uniquely identify the object within that volume. Caching in BlueFS is performed at the block-level. Operations that read or modify data operate on whole 4 KB blocks — this size was chosen to match the Linux page cache block size. As described in Section 4.3.6, cache consistency is maintained with version stamps and callbacks [8].

4.2 BlueFS kernel module

Similar to many previous file systems [8, 9], BlueFS simplifies implementation by performing most functionality in a user-level daemon. The BlueFS kernel module intercepts Linux VFS operations and redirects them to the daemon through an upcall mechanism. The kernel module also handles caching and invalidation of data and metadata in the Linux file cache. Operations that modify data such as `write`, `create` and `rename` are redirected synchronously to the daemon. This allows the daemon to support devices with different consistency semantics: for example, the daemon might reflect modifications to the server immediately to minimize the chance of conflicts, but it might delay writing data to disk to minimize expensive disk power mode transitions. The cost of this implementation is that a dirty block is double-buffered until it is written to the last storage device or evicted from the file cache. We plan to eliminate double-buffering in the future by moving more functionality into the kernel.

4.3 The Wolverine user-level daemon

4.3.1 Write queues

Wolverine maintains a write queue in memory for each local, portable, and network storage device that is currently accessible to the mobile computer. Initially, it creates a write queue for the server and one for each permanent storage device on the client. When a portable storage device is attached, Wolverine creates a new queue; when a portable device is detached, Wolverine deletes its associated queue. Write queues serve two purposes: they improve performance by asynchronously writing data to storage, and they improve energy-efficiency by aggregating writes to minimize expensive power mode transitions.

A write queue is essentially an operation log for a storage device. Whenever Wolverine receives a modification via an upcall, it adds a new record containing that modification to each write queue. Operations are written to storage in FIFO order to guarantee serializability; e.g. since modifications are reflected to the server in order, other clients never view inconsistent file system state.

BlueFS conserves memory by sharing records between write queues. Each record is reference counted and deleted when the last queue dereferences the record. For each queue, Wolverine maintains a per-file hash table of objects that have been modified by queued operations. Before reading data from a device, Wolverine checks its hash table to identify any operations that modify the data and have not yet been reflected to the device.

Each storage device specifies the maximum time that records may remain on its write queue before being written to the device. By default, BlueFS uses a maximum delay of 30 seconds for all write queues (equivalent to the default write-back delay for dirty pages in the Linux file cache). The user may change this value for each device.

Wolverine caps the maximum memory usage of all write queues to a user-specified value. New operations that would cause Wolverine to exceed this limit are blocked until memory is freed by writing queued operations to storage. To avoid blocking, Wolverine starts to flush write queues when memory utilization exceeds 75% of the allowed maximum. Each write queue may be explicitly flushed using a command-line tool.

Wolverine creates bursty device access patterns with a simple rule: whenever the first operation in a queue is written to storage, all other operations in that queue are flushed at the same time. This rule dramatically reduces the energy used by storage devices that have large transition costs between power modes.

For example, disk drives save energy by turning off electronic components and stopping platter rotation. Entering and leaving power-saving modes consumes a large amount of energy that is only recovered if the disk stays powered down for several seconds [5]. By burst-writing modifications, BlueFS creates long periods of inactivity during which the disk saves energy.

Write queues also reduce the energy used to write to remote storage. Wireless 802.11 networks use a power-saving mode (PSM) that disables the client interface during periods of light network utilization. However, the mobile computer expends more energy to transmit data in PSM [1] because PSM decreases throughput and increases latency. Ideally, the wireless network could achieve the best of both worlds by entering PSM during idle periods and disabling PSM before transmitting data. Unfortunately, since the time and energy cost of mode transitions is large, it is not cost effective to disable PSM before a small transmission and re-enable it after. BlueFS solves this problem by aggregating many small transmissions into a single large one — this makes power-mode toggling effective.

It seems counter-intuitive that writing data to more than one storage device could save energy. However, we find that writing data to multiple devices creates opportunities to save energy in the future. For example, if the only copy of an object exists on a disk drive, then that disk must be spun up when the object is next read. In contrast, if the object is also written to a remote server, then the file system may avoid an expensive disk transition by reading data from the server. Further, since writes can be aggregated to minimize the number of transitions while reads usually cannot, writing to multiple devices in BlueFS tends to *reduce* the total number of transitions that occur. When transitions are the dominant component of energy usage, this saves energy.

4.3.2 Reading data

The kernel module sends Wolverine an upcall whenever data it needs is not in the Linux file cache. Wolverine then tries to read the data from the storage option that offers the best performance and lowest energy cost.

Wolverine maintains a running estimate of the current time to access each storage device. Whenever data is fetched, Wolverine updates a weighted average of recent access times, as follows:

$$\textit{new est.} = (\alpha)(\textit{this msmt.}) + (1-\alpha)(\textit{old est.})$$

For the server, latency and bandwidth are estimated separately. For local storage, separate estimates are used for the first block accessed in a file (this captures disk seek time) and contiguous block accesses in the same file. Since storage behavior tends to be more stable than network access times, BlueFS sets α to 0.1 for network storage and 0.01 for local storage. We chose these values based on empirical observation of what worked best in practice.

One might reasonably expect that passive monitoring of access times alone would yield good estimates. However, we have found that monitoring device power states is also essential. In fact, the power state is often the dominant factor in determining the latency of the next access. For example, the time to read data from a Hitachi microdrive increases by approximately 800 ms when the drive is in standby mode. Also, server latency increases by up to 100 ms when the network interface enters its power-saving mode.

BlueFS passively monitors *both* access times and device power modes. It uses the OS power manager described in [2] to receive a callback whenever a network or storage device transitions to a new power mode. For each storage device, it maintains estimators for each possible power mode. When predicting access times, BlueFS uses the estimator that matches the current mode.

In addition to performance, Wolverine also considers energy usage when deciding which device to access. Because on-line power measurement is often infeasible, BlueFS relies on offline device characterization to predict energy costs. These characteristics need be determined only once for each type of I/O device — they are provided when the device is first registered with BlueFS. Often, relevant energy characteristics can be found in device specifications. Alternatively, we have developed an offline benchmarking tool that measures the energy used to read data, write data, and transition between power modes [1]. Given a device characterization, BlueFS predicts the energy used *by the entire system* to access data. This includes not just energy used by device I/O, but also the energy expended while the mobile computer waits for the request to be serviced.

The cost of accessing each device is the weighted sum of the predicted latency and energy usage. By default, BlueFS weights these two goals equally; however, the user may adjust their relative priority. BlueFS first tries to read the data from the least costly device. If the data is not stored on that device, BlueFS then tries the remaining devices in order. Since the server has the primary replica of each file, Wolverine will only fail to read the data on any device when the client is disconnected.

Before reading an object from a device, Wolverine checks the device's write queue hash table to see if operations that modify that item have not yet been reflected to storage. If this is the case, the request can often be serviced directly from the queued operation. For example, a queued store operation may contain the block of data being read. Sometimes Wolverine must replay queued operations that modify the requested object. For example, if the write queue contains operations that modify a directory, Wolverine first fetches the directory from the device, then replays the modifications in memory. Finally, it returns the modified directory to the kernel.

4.3.3 The enode cache

Early in our implementation, we found that BlueFS sometimes wasted time and energy trying to read data from a storage device that did not contain the needed data. We therefore considered how to minimize the number of fruitless reads that occur. We explored maintaining an in-memory index of cached data for each device. However, given the limited memory and rapidly increasing storage capacity of many small mobile devices, we felt that such an index would often be too large. Instead, we decided to strike a reasonable compromise by caching index information only for recently accessed files.

When BlueFS first accesses an object, it creates an *enode* that captures all the information it currently holds about the validity of that object on the storage devices attached to the mobile computer. For each device, the enode indicates whether the object's state is known or unknown. For an object with known state, the enode tells whether its attributes are valid, and whether none, some, or all of its data blocks are cached. Enodes are hashed by file id and stored in an *enode cache* managed by LRU replacement. The default size of the cache is 1 MB.

Wolverine checks the enode cache before trying to read an object from a device. It skips the device if an enode reveals that the object is not present or invalid. Whenever Wolverine tries to access an object on a new device, it updates the object's enode with its current status (i.e. whether it is present on the device, and if so, whether its data and attributes are valid).

4.3.4 Storage devices

Wolverine supports a modular VFS-style device interface for storage. Each device provides methods to read data and attributes, as well as methods to handle queued modifications. We have implemented two devices that use this interface. The *remote* device communicates with the server through a TCP-based RPC package. The *cache* device performs LRU-style caching of object attributes and data blocks.

The cache device is layered on top of a native file system such as ext2. This simplifies implementation and allows BlueFS and non-BlueFS files to co-exist on a single local or portable storage device. Thus, a portable storage device with a BlueFS cache may also be used on computers that are not BlueFS clients; however, such computers would only access the non-BlueFS files.

Each object and its attributes are stored in a single container file in the native file system that is named by the object's 96-bit BlueFS id. This implementation sacrifices some efficiency, but has let us quickly support a wide variety of storage media. Data blocks are cached in their entirety, but not all blocks in a file need be cached. Similar to the enode structure, each container file has a header that specifies whether the object attributes are valid, and whether its data is invalid, partially valid, or entirely valid. When only a portion of the data blocks are cached, a bit mask determines which blocks are valid.

To start using a new storage device, a client registers the device with the BlueFS server. The server assigns a unique id that is written to a metadata file on the storage device. During registration, the user also specifies the maximum amount of storage that BlueFS may use on that device. After a device is registered, it may be attached to any BlueFS client.

If the cache on a device becomes full, BlueFS evicts objects using the second-chance clock algorithm, with the additional modification that object metadata is given additional preference to remain in the cache. Wolverine starts the eviction thread whenever a cache becomes 90% full. Since the primary replica of a file always exists on the server, the eviction thread simply deletes container files to create space.

To maintain the LRU approximation, Wolverine ensures that each cache is updated with the latest version of data being read by the user. When data is read from the server or from another device, Wolverine checks the object's enode to determine if the cache already has the latest version of the data. If the cache does not have the data or its status is unknown, Wolverine places a copy of the data on the device's write queue. This ensures that each local and portable device has up to date copies of files in the user's current working set.

4.3.5 Affinity

While LRU replacement is a convenient way to capture a user's current working set on a local or portable device, it is often the case that a user would prefer to always have a certain set of files on a particular device. In BlueFS, device *affinity* provides this functionality.

Affinity specifies that the latest version of an object should always be cached on a particular device. Although affinity is similar in spirit to hoarding [11] in the Coda file system, affinity can support portable devices because it is implemented at device granularity. In contrast, Coda implements hoarding at client granularity, meaning that one cannot hoard files to portable storage or to multiple devices on the same client.

In BlueFS, a command-line tool provides users with the ability to add, display, or remove affinity for files and subtrees. Affinity is implemented as an attribute in the cache container file header. If the affinity bit is set, the object is never evicted. If a file with affinity is invalidated by a modification on another client, Wolverine adds the object to a per-device *refetch list*. Every five minutes, a Wolverine thread scans the refetch lists of all attached devices and requests missing data from the server. If the set of files with affinity to a device is greater than the size of the cache, the user must remove affinity from some files before the thread can refetch all files.

Affinity for subtrees is supported with a *sticky affinity* attribute that is associated with directories. Whenever an object is added as a child of a directory with sticky affinity to a device, the new object is given affinity to that device. If the new object is a directory, it also receives sticky affinity. For example, if a user specifies that a PowerPoint directory has sticky affinity to a USB stick, new files created in that directory will also have affinity to the USB stick.

4.3.6 Cache consistency

Cache consistency in BlueFS builds upon two pieces of prior work: Coda's use of optimistic replica control [9] and AFS's use of callbacks for cache coherence [8]. BlueFS adds two important modifications. First, BlueFS maintains callbacks on a per-device basis, rather than on a per-client basis. Second, the BlueFS server queues invalidation messages when a device is disconnected. These modifications let BlueFS efficiently support portable media and clients that frequently hibernate to save energy.

Optimistic replica control provides greater availability and improves performance by allowing clients to read or write data without obtaining leases or locks. When two clients concurrently modify a file, the write/write conflict is detected at the server and flagged for manual resolution. Like Coda, BlueFS stores a version number in the metadata of each object. Each operation that modifies an object also increments its version number. Before committing an update operation, the server checks that the new version number of each modified operation is exactly one greater than the previous version number. If this check fails, two clients have made conflicting modifications to the object. The user must resolve such conflicts by selecting a version to keep. In the absence of disconnection, such conflicts occur only when two different clients update the same file within 30 seconds of each other (i.e. the amount of time an update can reside in the server write queue). Prior experience in other distributed file systems [9] has shown that such conflicts are rare.

Callbacks are a mechanism in which the server remembers that a client has cached an object and notifies the client when the object is modified by another client. In contrast to AFS and Coda, BlueFS maintains callbacks on a per-device rather than a per-client granularity. Specifically, a BlueFS device may hold a data and/or a metadata callback for each object. When Wolverine adds an operation to a device write queue that caches data, it also adds a callback request on behalf of that device to the server write queue. It uses the enode cache to reduce the number of callback requests. The object enode records the set of devices for which the server is known to hold a callback. When a callback is already known to exist for a device, no request need be sent to the server. Wolverine also sets callbacks for objects stored in the Linux file cache. From the point of view of the server, a client's file cache is simply another device.

Each callback request contains the version number of the cached object. If this does not match the version number of the primary replica, the server issues an immediate invalidation. Otherwise, the server maintains the callback as soft state.

Wolverine optimizes invalidation delivery by notifying the server when storage devices are attached and detached. The server does not send invalidations to devices that are currently attached to the client that made the modification since the client will already have placed the modification on each device's write queue. The server sends invalidations to all other devices that have a callback on the object; invalidations for multiple devices attached to the same client are aggregated.

Upon receipt of an invalidation, the client updates the object enode to indicate that the data and/or metadata are no longer valid. Invalidations for the Linux file cache are passed to the kernel module, which removes the data from the file cache. Other invalidations are placed on device write queues. When the queues are flushed, invalidated objects are deleted. Note that the presence of the invalidation record in a queue's hash table prevents stale data from being read before the queue is flushed.

As described, callbacks provide cache consistency only when the server and the client to which a storage device is attached remain continuously connected. During disconnection, invalidation messages could potentially be lost, so the validity of cached objects becomes unknown. In our initial design, BlueFS performed *full cache revalidation* upon reconnection. During revalidation, BlueFS verifies each cached object by asking the server for the version number and identifier of the last client to make an update. If these values match those of the cached replica, the replica is valid and a new callback is set. Otherwise, the replica is deleted.

However, due to the growing size of mobile storage devices, full cache validation is expensive in terms of both time and energy. Unfortunately, mobile storage devices are frequently disconnected from the server. First, portable devices are naturally disconnected when they are detached from one computer and reattached to another. Second, handheld and other mobile computers frequently hibernate or enter other power saving modes in which the network interface is disabled. Thus, the devices specifically targeted by BlueFS frequently disconnect and would often require full cache revalidation given our initial design.

We therefore modified our design to have the server queue invalidation messages for later delivery when a device holding a callback is disconnected. When a portable device is next attached to a BlueFS client, the server forwards all invalidations queued during disconnection to the new client. Similarly, when a client hibernates to save power, the server queues invalidations for that client's file cache and all its attached devices. This lets handheld clients nap for short periods without missing invalidations. With these modifications, BlueFS avoids full cache revalidations in most cases. A revalidation is required only when soft state is lost due to client or server crash, or when a portable storage device is not cleanly detached as described in the next section.

From the server's point of view, client disconnection is equivalent to hibernation. When Wolverine detects that the server is unreachable, it writes operations that are flushed from the server write queue to a file on a user-specified local (non-portable) storage device. Upon reconnection, it reads the file and sends these operations to the server before transmitting new updates.

4.3.7 Adding and removing storage devices

Once a portable storage device is registered with the server, it may be dynamically attached and detached from any BlueFS client.

When a portable device is detached, Wolverine:

- sends a detach message to the server. The server starts queuing callbacks at this point.
- stops placing new operations on the device's write queue. This is done atomically with processing of the detach message.
- retrieves and stores the current version of any objects on the device's affinity-driven refetch list.
- flushes any operations remaining on the write queue to the device
- flushes the server write queue.
- writes to the device the state of the enode cache for that device.
- writes a clean shutdown record to that device.

A clean detach guarantees that the latest version of all files with affinity to the device are stored on that device. BlueFS also supports an option in which the detach does not refetch files from the server, but instead records the list of files to be refetched in a file on the device. This option makes detach faster but these files are not available if the device is next attached to a disconnected client.

When a portable device is attached, Wolverine:

- checks for a clean shutdown record. If the record is found, it is deleted. If it is not found, full cache revalidation is performed.
- sends an attach message to the server.
- deletes any objects invalidated by queued callbacks contained in the server's response to the attach message.
- warms the enode cache with the device's previously saved state. New enodes are created if the cache is not full; otherwise, only enodes in the cache are updated with the device state.
- reschedules any refetch requests that were saved during detach.

If a device was not cleanly detached, it must be fully revalidated on reattachment. Currently, revalidation must also be performed when devices are detached from a disconnected client.

4.3.8 Fault tolerance

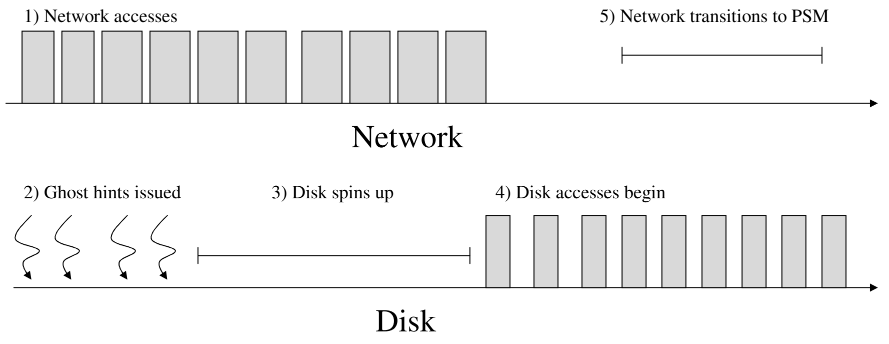
The BlueFS server writes modifications to a write-ahead log before replying to RPCs. Since modifications are safe only after the server replies, BlueFS clients can lose approximately 30 seconds worth of updates if they crash (i.e. the set of updates that resides in the server write queue). In the common case, no data is lost when a local or portable storage device on the client suffers a catastrophic failure (since the primary replica of each object exists on the server). However, modifications made during disconnected operation are lost if the device holding the persistent head of the server write queue fails. Although BlueFS does not currently guard against catastrophic server failure, well-known mechanisms such as server or storage replication could provide this functionality.

4.3.9 Integration with power management

BlueFS works closely with device power management to minimize energy usage. It discloses hints about its I/O activity using self-tuning power management (STPM). Since STPM has been described elsewhere [1], we present only a brief description here.

BlueFS issues a STPM hint every time it performs an RPC. The hint contains the size of the RPC and discloses whether the RPC represents background or foreground activity. The STPM module uses this information to decide when to enable and disable power-saving modes. This decision is based upon a cost/benefit analysis that considers the observed distribution of RPCs, the base power of the mobile computer, and the user's current relative priority for performance and energy conservation. For example, if BlueFS performs three RPCs in close succession, STPM might anticipate that several more RPCs will soon occur and disable power management to improve performance.

If an adaptive system such as BlueFS adopts a purely *reactive* strategy that considers only one access at a time, it will often make incorrect decisions about which device to access. As an example, consider the case where BlueFS is reading many small files stored on the server and on a disk in standby mode. The best strategy is to fetch the files from disk, since the transition cost of spinning up the disk will be amortized across a large number of accesses. However, a system that considers only one access at a time will never spin up the disk since the transition cost always outweighs the benefit that will be



BlueFS masks the latency of disk power management by initially fetching data from the server. Ghosts hints are issued to the disk, which transitions to active mode. After the transition, BlueFS fetches the remaining data from disk. Eventually, the idle network device transitions to PSM to save power.

Figure 2. Masking the latency of disk power management

realized by a single file. Dynamic power management cannot help here: the disk will never spin up because the OS sees no disk accesses.

BlueFS solves this problem using *ghost hints* [2] that disclose to the OS the opportunity cost that is lost when a device is in a power-saving mode. When Wolverine predicts the time and energy cost of accessing each device in its current power mode, it also predicts the cost that would have resulted from that device being in its ideal mode. If the device that has the lowest current cost differs from the device that has the lowest ideal cost, Wolverine issues a ghost hint to the device with lowest ideal cost. The ghost hint discloses the time and energy wasted by that device being in the wrong power mode. For example, when the disk STPM module receives a set of ghost hints whose lost opportunity cost exceeds the cost of the disk transition, it spins up the disk. Thus, ghost hints enable BlueFS to *proactively* switch to the best device during periods of high load.

We have designed BlueFS to mask the performance impact of device power management. Consider the case shown in Figure 2 where BlueFS is fetching a large file with the disk initially in standby mode to conserve power. Using other file systems, fetching the first block incurs a large delay as the disk spins up (800 ms for the Hitachi microdrive [7] and one second for the IBM T20 laptop drive). In contrast, BlueFS hides this delay by fetching blocks over the network. For each block that it fetches from the network, BlueFS issues a ghost hint that discloses the opportunity cost of the disk being in standby. While the disk spins up in response to the ghost hints, BlueFS continues to fetch blocks from the server. After the transition completes, BlueFS fetches remaining blocks from disk. Eventually, the network device enters a power-saving mode since it is now idle. No ghost hints are issued to the network device because the disk offers both the lowest current and ideal cost.

Our belief is that many users avoid disk power management because of lengthy spin up delays. If BlueFS hides these delays, then users might use more aggressive power management policies. However, the potential energy savings are extremely difficult to quantify; if such savings were to occur, they would supplement those we report in Section 5.5.

4.4 BlueFS server

Since the focus of BlueFS is on the mobile client, our file server implementation is fairly standard. We expect a single BlueFS server to handle roughly the same number of clients handled by a current AFS or NFS server. The BlueFS server stores the primary replica of every object. When a client modifies an object, it sends the modification to the server. The server provides atomicity and durability for file system operations through write-ahead logging and serializability by locking the BlueFS identifier namespace. The largest difference between the BlueFS server and those of other file systems is that the BlueFS server is aware of individual storage devices.

5 Evaluation

Our evaluation answers five questions:

- Do the new features of BlueFS hurt performance during traditional file system benchmarks?
- How well does BlueFS mask the access delays caused by disk power management?
- How well does BlueFS support portable storage?
- How much does BlueFS reduce energy usage?
- Can BlueFS adapt to storage devices with heterogeneous characteristics?

5.1 Experimental setup

We used two client platforms: an IBM T20 laptop and an HP iPAQ 3870 handheld. The laptop has a 700 MHz Pentium III processor, 128 MB of DRAM and a 10 GB hard drive. The handheld has a 206 MHz StrongArm processor, 64 MB of DRAM and 32 MB of flash. The laptop runs a Linux 2.4.28-30 kernel and the handheld a Linux 2.4.18-rmk3 kernel. Unless otherwise noted, both clients use a 11 Mb/s Cisco 350 802.11b PCMCIA adapter to communicate with the server. The portable device in our experiments is a 1 GB Hitachi microdrive [7].

The server is a Dell Precision 350 desktop with a 3.06 GHz Pentium 4 processor, 1 GB DRAM, and 120 GB disk, running a Linux 2.4.18-19.8.0 kernel. The client and server communicate using a Cisco 350 802.11b base station. When we insert delays, we route packets through an identical Dell desktop running the NISTnet [4] network emulator.

We measure operation times using the `gettimeofday` system call. Energy usage is measured by attaching the iPAQ to an Agilent 34401A digital multimeter. We remove all batteries from the handheld and sample power drawn through the external power supply approximately 50 times per second. We calculate system power by multiplying each current sample by the mean voltage drawn by the mobile computer — separate voltage samples are not necessary since the variation in voltage drawn through the external power supply is very small. We calculate total energy usage by multiplying the average power drawn during benchmark execution by the time needed to complete the benchmark. The base power of the iPAQ with network in PSM and disk in standby is 1.19 Watts.

We limit maximum write queue memory usage for BlueFS to 50 MB on the laptop and 15 MB on the handheld. Unless otherwise noted, we assign an equal priority to energy conservation and performance.

5.2 Modified Andrew benchmark

We begin our evaluation with the traditional modified Andrew benchmark. While this benchmark does not have a workload typical of how we expect BlueFS to be used, its widespread adoption in the file system community makes it a useful data point. We compare three file systems: NFS version 3, operating in asynchronous mode; Coda version 6.0.3 running in both write connected mode, in which modifications are reflected synchronously to the server, and write disconnected mode, in which modifications are reflected asynchronously; and BlueFS. We measure the time needed by each file system to untar the Apache 2.0.48 source tree, run configure, make the executable, and delete all source and object files. In total, the benchmark generates 161 MB of data.

Latency (ms)	NFS	Coda		BlueFS
		Wr. Conn.	Wr. Disc.	
0	879 (3)	702 (3)	601 (15)	552 (11)
30	5779 (56)	1681 (4)	637 (10)	535 (14)

This figure shows the number of seconds needed to untar, configure, make, and delete the Apache 2.0.48 source tree. Each value shows the mean of 5 trials with standard deviation given in parentheses.

Figure 3. Modified Andrew benchmark

Figure 3 shows the result of running this benchmark on the IBM T20 laptop. With no network latency between client and server, BlueFS executes the benchmark 59% faster than NFS and 9% faster than Coda in write disconnected mode. With a 30 ms latency, BlueFS is over 10 times faster than NFS and 19% faster than Coda.

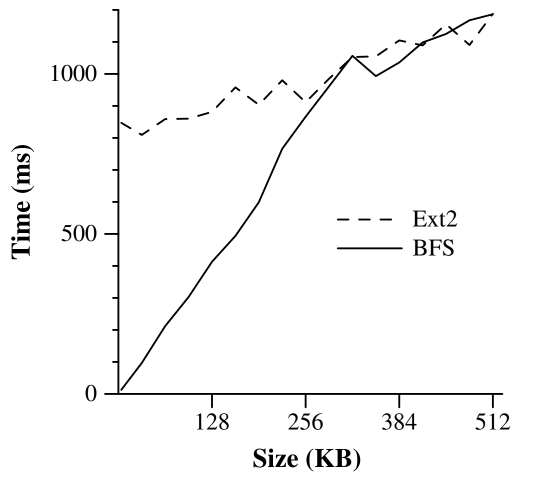
BlueFS and Coda both outperform NFS because they cache data on the laptop hard drive. In contrast, NFS must refetch data from the server during benchmark execution — this is especially costly when latency is high. The difference in Coda's performance between write connected and write disconnected mode shows the benefit of writing modifications asynchronously to the server. Given this performance difference, we operate Coda in write disconnected mode in subsequent experiments.

Originally, we had expected BlueFS to exhibit performance similar to that of Coda in write disconnected mode. One possible reason for the performance advantage seen by BlueFS is its use of TCP rather than Coda's custom transport protocol. Another possible reason is Coda's use of RVM. However, we note that Coda's resilience to client crash in write disconnected mode is similar to that of BlueFS — up to 30 seconds of data can be lost.

5.3 Masking the latency of power management

We next examined how well BlueFS hides the access delays caused by disk power management. We first measured the time used by the Linux ext2 file system to read files from the Hitachi 1 GB microdrive when the drive is in standby mode. As shown by the dashed line in Figure 4, ext2 reads are delayed for approximately 800 ms while the drive spins up to service the first request. Note that distributed file systems such as Coda pay the same performance penalty as ext2 because they use a static cache hierarchy that always fetches data from disk.

We next measured the time to read the same files using BlueFS. BlueFS masks disk spin-up delay by reading the first few blocks of the file from the server. For small files, this leads to a substantial performance improvement; BlueFS reads a 4 KB file in 13 ms, whereas ext2 requires over 800 ms. This gap may be larger for other hard drives — our laptop hard drive spin-up delay is greater than 1 second.



This figure shows the time used by ext2 and BlueFS to read 4–512 KB files. Files are stored on a portable Hitachi 1 GB microdrive and on the BlueFS server. At the beginning of each trial, the network is in CAM and the microdrive is in standby mode. No latency is inserted between the BlueFS client and server. Each value is the mean of five trials.

Figure 4. Masking power management latency

The performance benefit of BlueFS decreases with file size. For large files, BlueFS ghost hints spin up the disk. During the transition, BlueFS fetches blocks from the network. Although BlueFS improves performance by fetching the first part of each file from the network, this benefit is offset by the lack of readahead in our current implementation. The break-even file size, at which BlueFS and ext2 have equivalent performance, is 256 KB. However, *first-byte latency* is an important metric for applications such as grep and Mozilla that can begin useful processing before all bytes have been read. Since BlueFS delivers the first block of large files to the application 800 ms faster than ext2, many applications will be able to begin processing data sooner.

5.4 Support for portable storage

We measured how well BlueFS supports portable storage using a benchmark devised by Tolia et al. [20]. The benchmark replays a set of Coda traces collected at Carnegie Mellon University using Mummert's DFSTrace tool [11]. The traces, whose characteristics are shown in Figure 5, capture file system operations performed over several hours on four different computers. Due to Coda's open-close semantics, the traces do not capture individual read and write operations. We therefore modified DFSTrace to assume that files are read in their entirety on open. Further, files that are modified are assumed to be written in their entirety on close.

Each file system trace is accompanied by a snapshot of the file system at the start of trace collection. At the beginning of each experiment, the snapshot data exists on the file server and the portable microdrive, but not on the local disk of the client. We also generate a lookaside index of the snapshot on the microdrive. Before running the benchmark, we flush the Linux file cache on both the client and server.

Trace	Number of Ops.	Length (Hours)	Update Ops.	Working Set (MB)
purcell	87739	27.66	6%	252
messiaen	44027	21.27	2%	227
robin	37504	15.46	7%	85
berlioz	17917	7.85	8%	57

This figure shows the file system traces used in our evaluation. Update operations are those that modify data. The working set is the total size of the files accessed during a trace.

Figure 5. File traces used in evaluation

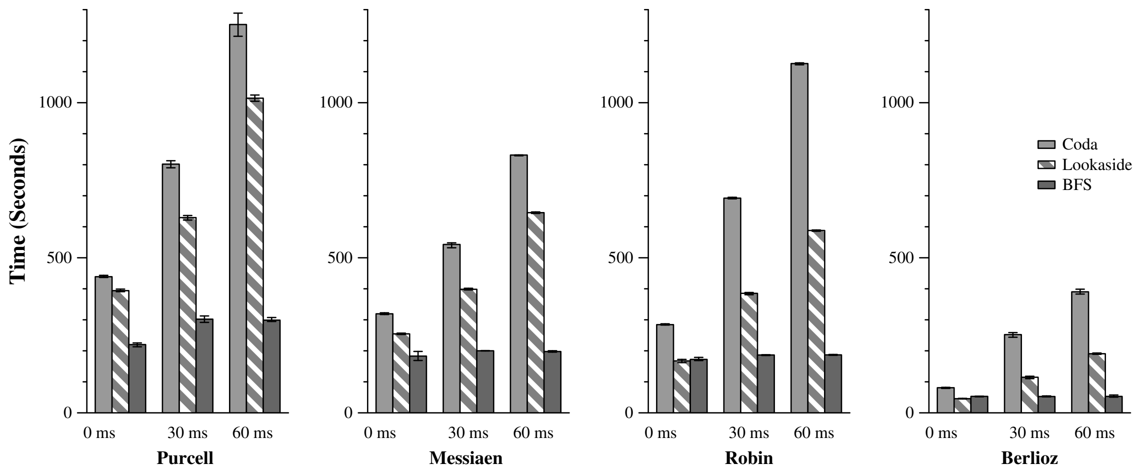
Tolia's benchmark replays each trace as fast as possible and measures the time to complete all operations. We therefore set BlueFS's performance/energy knob to maximum performance. We ran the benchmark on the IBM T20 laptop, using a Lucent 2 Mb/s wireless network card to limit network bandwidth. We also use NISTnet to vary network latency from 0 to 60 ms.

Figure 6 compares the time to replay the traces without portable storage using Coda, with portable storage using Coda's lookaside caching, and with portable storage using BlueFS. At 0 ms latency, BlueFS reduces benchmark execution time 28–44% for the Purcell and Messiaen traces compared to Coda lookaside caching. Although lookaside caching pays little penalty for revalidating objects with the server at this latency, BlueFS achieves a substantial benefit by fetching small files from the server rather than the microdrive. For the Robin and Berlioz traces, BlueFS and lookaside caching perform similarly with no latency. For the four traces at 0 ms latency, BlueFS reduces replay time 34–50% compared to Coda without portable storage.

As latency increases, the performance advantage of BlueFS grows. Lookaside caching requires one server RPC per object in order to retrieve the attributes and SHA-1 hash — BlueFS avoids this round-trip by using device callbacks. At a latency of 30 ms, BlueFS fetches all data from portable storage — thus, it is not affected by further increases in network latency. At this latency, BlueFS is at least twice as fast as Coda with lookaside caching for the four traces.

At 60 ms latency, BlueFS executes the benchmark 5–7 times faster than Coda without lookaside caching and over 3 times faster than Coda with lookaside caching. Over a higher-bandwidth, 11 Mb/s network connection, we saw roughly similar results — BlueFS executed the benchmark approximately 3 times faster than Coda with lookaside caching at a network latency of 60 ms.

Finally, it is important to note that BlueFS updates the microdrive as objects are modified during the trace. At



This figure compares the performance benefit achieved by BlueFS and Coda through the use of portable storage. Each graph shows the time to replay a file system trace with data stored on a local 1 GB microdrive and on a file server with 0, 30, and 60 ms of network latency. Each value is the mean of 3 trials — the error bars show the highest and lowest result.

Figure 6. Benefit of portable storage

the end of trace replay, the microdrive holds a complete version of the modified file tree. Since Coda lookaside caching provides read-only access to portable storage, many cached objects are invalid at the end of trace replay. In order to bring the cache up to date, the user must manually synchronize modified files and generate a new lookaside index. For large caches, index generation is time-consuming (e.g. 3 minutes for the Purcell tree), because it requires a full cache scan.

5.5 Energy-efficiency

We next measured the energy-efficiency of BlueFS by running the Purcell trace on the iPAQ handheld using the microdrive as local storage. In this experiment, it is important to correctly capture the interaction of file system activity and device power management by replicating the interarrival times of file system operations. Thus, between each request, replay pauses for the amount of time recorded in the original trace. Since running the entire trace would take too long using this methodology, we replay only the first 10,000 operations — this corresponds to 42:36 minutes of activity.

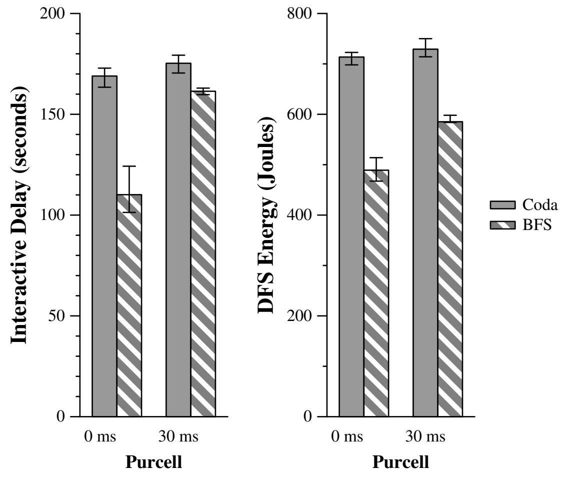
In our first experiment, shown in Figure 7, all files accessed by the trace are initially cached on the microdrive. The left graph compares the interactive delay added to trace replay by BlueFS and Coda — this shows the time that the user waits for file operations to complete. The right graph shows the amount of additional energy used by the iPAQ to execute file system operations (i.e. it excludes energy expended during think time). Coda uses default power management (PSM for the network and the microdrive's ABLE power manager). Note that power management is essential to achieve reasonable battery lifetime; Coda runs faster without power management, but the iPAQ expends over 3 times more power.

With 0 ms latency, BlueFS reduces interactive delay by 35% and file system energy usage by 31%. Since all files are on disk, Coda only uses the network to asynchronously reintegrate modifications to the server. BlueFS, however, achieves substantial performance and energy benefit by fetching small objects from the server and by using the network when the disk is in standby.

With 30 ms network latency, the microdrive always offers better performance than the server *when it is active*. In the absence of power management, Coda's static hierarchy would always be correct. However, because BlueFS hides performance delays caused by the drive entering standby mode, it reduces interactive delay by 8% compared to Coda. BlueFS also uses 20% less energy. In addition to the energy reduction that comes from its performance gains, BlueFS saves considerable energy by aggregating disk and network accesses.

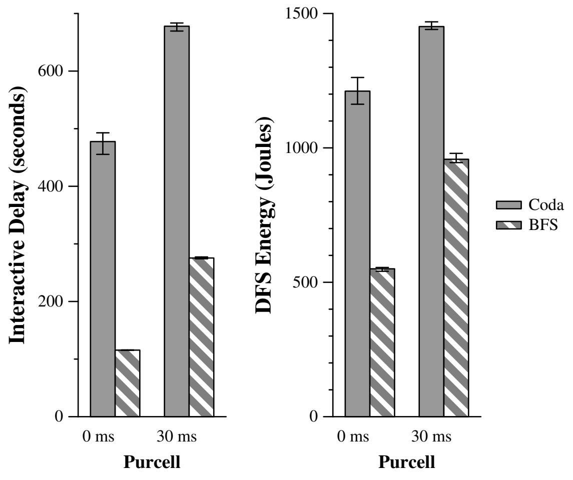
In Figure 8, we show results when only half of the files accessed by the trace are initially cached on the microdrive. The set of cached files was randomly chosen and is the same in all experiments.

BlueFS excels in this scenario. Compared to Coda, BlueFS reduces interactive delay by 76% and file system energy usage by 55% with no network latency. With 30 ms latency, BlueFS reduces interactive delay by 59% and file system energy usage by 34%. When some files are not cached, file accesses often go to different devices. While one device is being accessed, the others are idle. During long idle periods, these other devices enter power saving modes, creating considerable variability in access times. Thus, this scenario precisely matches the type of dynamic environment that BlueFS is designed to handle.



The left graph compares the interactive delay caused by Coda and BlueFS during the execution of the first 10,000 operations in the Purcell trace on an iPAQ handheld. The right graph shows the energy used by each file system. All data is initially cached on a local 1 GB microdrive. Each value is the mean of three trials — the error bars show the highest and lowest result.

Figure 7. Benefit of BlueFS with warm cache



The left graph compares the interactive delay caused by Coda and BlueFS during the execution of the first 10,000 operations in the Purcell trace on an iPAQ handheld. The right graph shows the energy used by each file system. Half of the data is initially cached on a local 1 GB microdrive. Each value is the mean of three trials — the error bars show the highest and lowest result.

Figure 8. Benefit of BlueFS with 50% warm cache

Of course, file system energy usage is only a portion of the total energy expended by the mobile computer. To calculate the effect of using BlueFS on battery lifetime, one must first determine the power used by non-file-system activities. In Figure 8, for example, if one assumes that the iPAQ constantly draws its base power (1.19 Watts) during user think time, use of BlueFS extends battery lifetime by 18% with a 0 ms delay and by 12% with a 30 ms delay. However, the extension in battery lifetime would be much greater if the iPAQ were to hibernate during trace periods with no activity (because average power usage due to non-file-system activity would decrease). Conversely, application activity may increase average power usage and reduce the energy benefit of BlueFS.

5.6 Exploiting heterogeneous storage

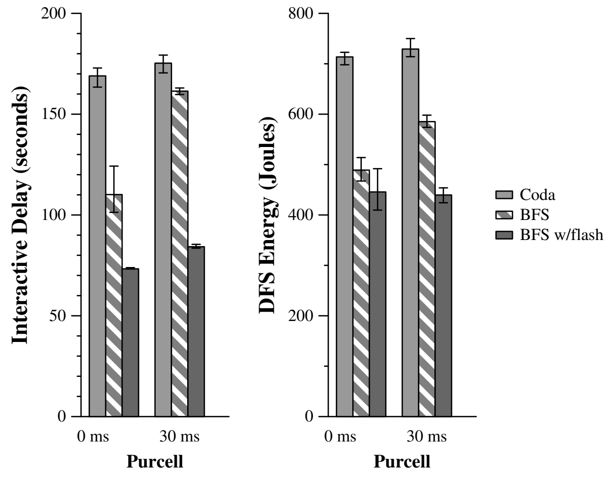
In our final experiment, we explored how well BlueFS utilizes multiple storage devices with heterogeneous characteristics. The iPAQ handheld contains a small amount (32 MB) of NOR flash that the familiar Linux distribution uses as the root partition. Reading a 4 KB block from flash is inexpensive: it takes less than 1 ms and uses negligible energy. However, flash writes are very costly — our measurements show that the iPAQ flash has substantially lower write throughput and higher latency than the microdrive. One reason for the low throughput is that modifying a block of flash requires a time-consuming erase operation before the block can be overwritten. Another reason is that the iPAQ uses the jffs2 file system, which performs writes synchronously. Finally, jffs2 is log-structured, and garbage collection uses substantial time and energy [25].

Despite these obstacles, we decided to see if BlueFS would benefit by using the iPAQ's flash. We could allocate only 16 MB of flash for BlueFS — much less than the working set of the Purcell trace. When we untar the initial tree for the trace on the iPAQ, BlueFS caches all data on the microdrive but only a small portion on flash. At the end of the untar, roughly 10,000 attributes and the last several hundred files accessed are on flash.

Our initial results were disappointing. After investigation, we found the culprit: flash read operations block for up to several hundred milliseconds while one or more flash blocks are erased to make room for new data. Based on this observation, we modified the BlueFS latency predictor to correctly anticipate reduced performance during flash queue flushes. We also increased the maximum delay of the flash write queue to 300 seconds to minimize the number of flushes that occur. We found it interesting that write aggregation proves useful even for a device with no power mode transitions.

With these modifications, BlueFS makes excellent use of the iPAQ flash, as shown in Figure 9. With no network latency, BlueFS with flash reduces interactive delay by 33% compared to our previous results, which used only the microdrive. Use of flash also reduces BlueFS energy usage by 9% on average, despite writing data to an additional device.

With 30 ms latency, flash reduces interactive delay by 48% and BlueFS energy usage by 25%. Observation of inidvidual file operations shows that BlueFS makes the best use of each device: data is read from flash when



The left graph compares the interactive delay caused by Coda and BlueFS during the execution of the first 10,000 operations of the Purcell trace on an iPAQ handheld. The right graph shows the energy used by each file system. All data is initially cached on the microdrive. In addition, 16 MB of data is cached on local flash. Each value is the mean of three trials — the error bars show the highest and lowest result.

Figure 9. Benefit of heterogeneous storage

available, large files are read from the microdrive to utilize its superior bandwidth, and the network is used to read small files and the first block of large files when the disk is spun down to save power.

6 Conclusion

Compared to previous distributed file systems, BlueFS provides three substantial benefits: it reduces client energy usage, seamlessly integrates portable storage, and adapts to the variable access delays inherent in pervasive computing environments. Rather than retrofit these features into an existing file system, we have taken a clean-sheet design approach. The benefits of designing a new system are most apparent in the portable storage experiments: BlueFS substantially outperforms an implementation that modifies an existing file system.

In the future, we plan to broaden the set of caching policies supported on individual devices. For example, on a device that is frequently shared with friends, a user may want to store only files that have affinity to that device. Alternatively, a user might wish to encrypt data on portable devices that could be easily stolen. We believe that BlueFS provides an excellent platform on which to explore this topic, as well as other mobile storage issues.

Acknowledgments

We would like to thank Landon Cox, Niraj Tolia, the anonymous reviewers, and our shepherd, Greg Ganger, for their suggestions that improved the quality of this paper. We would also like to thank Manish Anand for his help with STPM. This research was supported by the National Science Foundation under awards CCR-0306251 and CNS-0346686, and also by an equipment grant from Intel Corporation. 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, Intel, the University of Michigan, or the U.S. government.

References

[1] M. Anand, E. B. Nightingale, and J. Flinn. Self-tuning wireless network power management. In *Proceedings of the 9th Annual Conference on Mobile Computing and Networking*, pages 176–189, San Diego, CA, September 2003.

[2] M. Anand, E. B. Nightingale, and J. Flinn. Ghosts in the machine: Interfaces for better power management. In *Proceedings of the 2nd Annual Conference on Mobile Computing Systems, Applications and Services*, pages 23–35, Boston, MA, June 2004.

[3] T. E. Anderson, M. D. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless network file systems. In *Proceedings of the 15th ACM Symp. on Op. Syst. Principles*, Copper Mountain, CO, Dec. 1995.

[4] M. Carson. *Adaptation and Protocol Testing thorugh Network Emulation*. NIST, http://snad.ncsl.nist.gov/itg/nistnet/slides/index.htm.

[5] F. Douglis, P. Krishnan, and B. Marsh. Thwarting the power-hungry disk. In *Proceedings of 1994 Winter USENIX Conference*, pages 292–306, San Francisco, CA, January 1994.

[6] T. Heath, E. Pinheiro, and R. Bianchini. Application-supported device management for energy and performance. In *Proceedings of the 2002 Workshop on Power-Aware Computer Systems*, pages 114–123, February 2002.

[7] Hitachi Global Storage Technologies. *Hitachi Microdrive Hard Disk Drive Specifications*, January 2003.

[8] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[9] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1), February 1992.

[10] T. L. Martin. *Balancing Batteries, Power, and Performance: System Issues in CPU Speed-Setting for Mobile Computing*. PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, 1999.

[11] L. Mummert, M. Ebling, and M. Satyanarayanan. Exploiting weak connectivity in mobile file access. In *Proceedings of the 15th ACM Symp. on Op. Syst. Principles*, Copper Mountain, CO, Dec. 1995.

[12] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 174–187, Banff, Canada, October 2001.

[13] Network Working Group. *NFS: Network File System protocol specification*, March 1989. RFC 1094.

[14] J. M. Paluska, D. Saff, T. Yeh, and K. Chen. Footloose: A case for physical eventual consistency and selective conflict resolution. In *Proceedings of the 5th IEEE Workshop on Mobile Computing Systems and Applications*, Monterey, CA, Oct. 2003.

[15] A. E. Papathanasiou and M. L. Scott. Energy efficiency through burstiness. In *Proceedings of the 5th IEEE Workshop on Mobile Computing Systems and Applications*, pages 444–53, Monterey, CA, October 2003.

[16] S. Schlosser, J. Griffin, D. Nagle, and G. Ganger. Designing computer systems with MEMS-based storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 1–12, Cambridge, MA, November 2000.

[17] S. Sobti, N. Garg, C. Zhang, X. Yu, A. Krishnamurthy, and R. Wang. PersonalRAID: Mobile storage for distributed and disconnected computers. In *Proceedings of the 1st Conference on File and Storage Technologies, Monterey, California*, pages 159–174, Jan. 2002.

[18] S. Sobti, N. Garg, F. Zheng, J. Lai, Y. Shao, C. Zhang, E. Ziskind, A. Krishnamurthy, and R. Y. Wang. Segank: A distributed mobile storage system. In *Proceedings of the 3rd Annual USENIX Conference on File and Storage Technologies*, San Francisco, CA, March/April 2004.

[19] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 172–182, Copper Mountain, CO, 1995.

[20] N. Tolia, J. Harkes, M. Kozuch, and M. Satyanarayanan. Integrating portable and distributed storage. In *Proceedings of the 3rd Annual USENIX Conference on File and Storage Technologies*, San Francisco, CA, March/April 2004.

[21] N. Tolia, M. Kozuch, M. Satyanarayanan, B. Karp, T. Bressoud, and A. Perrig. Opportunistic use of content addressable storage for distributed file systems. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 127–140, May 2003.

[22] A. Tridgell and P. Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, Department of Computer Science, The Australian National University, Canberra, Australia, 1996.

[23] A.-I. A. Wang, P. Reiher, G. J. Popek, and G. H. Kuenning. Conquest: Better performance through a disk/persistent-RAM hybrid file system. In *Proceedings of the 2002 USENIX Annual Technical Conference*, Monterey, CA, June 2002.

[24] A. Weissel, B. Beutel, and F. Bellosa. Cooperative I/O: A novel I/O semanatics for energy-aware applications. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 117–129, Boston, MA, December 2002.

[25] D. Woodhouse. Jffs: The journalling flash file system. In *Ottawa Linux Symposium*. RedHat Inc., 2001.

[26] J. Zedlewski, S. Sobti, N. Garg, F. Zheng, A. Krishnamurthy, and R. Wang. Modeling hard-disk power consumption. In *Proceedings of the 2nd USENIX Conference on File and Storage Technology*, pages 217–230, San Francisco, CA, March/April 2003.