Chapter 1

Introduction

For over 25 years, disk has been the dominant storage medium for most file systems. Although disk technology is advancing at a rapid rate, notably in storage density, the mechanical latency of disk has improved little compared to other electronic components such as memory and CPU. As a consequence, file system designers have devoted significant code complexity in modern file systems to hide this mechanical latency.

The *Conquest* disk/persistent-RAM hybrid file system addresses the performance and complexity problems introduced by disk. The key observation is that the cost of persistent RAM (e.g., battery-backed DRAM) is rapidly declining, and the conventional assumption of RAM as a scarce resource is becoming less true for average users. In a search for a simpler design with faster performance, this dissertation examines the new notion of RAM as an inexpensive resource and investigates its implications on file-system architecture.

Compared to other leading disk-based file systems, *Conquest* shows a 20% smaller code base in terms of semicolon counts. For memory-resident workloads, *Conquest* can improve the performance by 24% to 1900%. For workloads that exercise both memory and disk, *Conquest* can improve the performance by 43% to 96%.

1.1 The Disk Performance Problem

Disk has been a major system-wide performance bottleneck. Figure 1.1 shows the relative performance improvement over years for CPU, memory and disk. The graph shows that first, CPU and memory are still improving at 50% per year according to Moore's law, while disk is improving only at 15% per year. Second, the performance gap between disk and CPU has been widening from five orders of magnitude to six orders of magnitude within the past ten years.

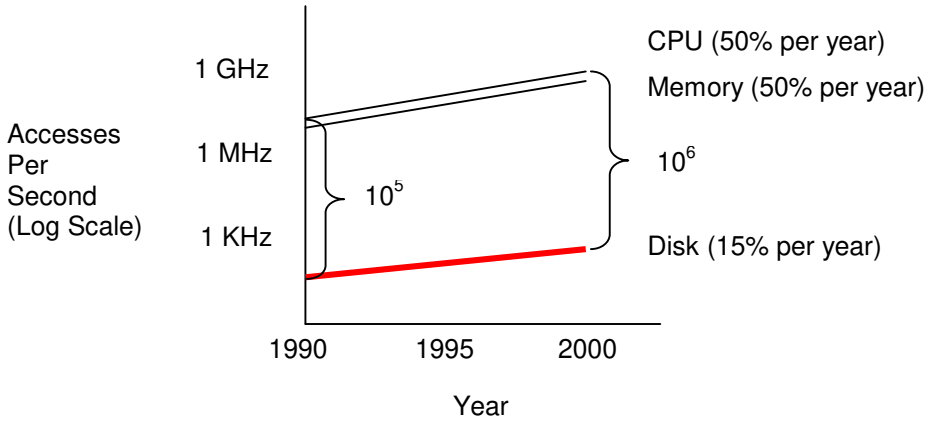


Figure 1.1: Performance improvement trends for CPU, memory, and disk.

On a human scale, if a CPU access had taken one second in the year 1990, a disk access would have taken about six days. If a CPU access had taken one second in the year 2000, a disk access would have taken about three months. This widening performance gap also means that increasing numbers of CPU cycles are required to be executed in parallel with disk to mask the disk latency completely. Otherwise, the CPU will become idle waiting for results from disk accesses. For over 25 years, researchers

have built complex optimizations to exploit these potentially idle cycles for better perceived disk performance.

1.2 Disk-Related Complexity

Disk optimizations are based on the observation that the relative slowness of disk is mostly attributable to the presence of mechanical components. Each disk access involves *seek time* (moving the disk arm to the correct disk track), *rotational latency* (waiting for the target data to be rotated under the disk head), and *transfer time* (transferring the data).

For example, to minimize the disk arm movements, researchers have tried to cluster related information in *cylinder groups*, or adjacent areas on disk, hoping that related data are likely to be accessed together. To minimize the rotational delay for reads, a piece of data can be replicated on one or multiple disks, so the expected rotational delay to access a piece of data is reduced by a factor proportional to the number of replicas. To minimize transfer time, a piece of data can be sliced and stored on multiple disks, so it can be transferred in parallel. To avoid accessing disk altogether, a file system can dedicate a portion of memory to caching recently used data, predicting that those data are more likely to be reused in the near future.

Disk-related optimizations are also present throughout all levels of storage services. Figure 1.2 depicts the storage service layers commonly found in modern operating systems. The device driver layer handles vendor-specific characteristics of disk drives. A modern device driver handles the specific physical geometry of a hard drive, performs variable rate transfers at different concentric disk zones, schedules disk arm movements, maps bad blocks to alternative blocks, and so on. The file system layer

provides a filing service to end-users, but this layer is also involved in exploiting the user access patterns and tuning the disk layout to minimize disk head movements and maximize bandwidth. The virtual file system (VFS) layer allows different file systems to interoperate on a single machine in a transparent way, and the VFS abstracts common file system functionalities such as access control, and disk optimizations such as I/O buffering. The virtual memory (VM) layer provides the illusion of a RAM capacity larger than the physical RAM, and the mechanism involves migrating data between memory and disk in the background. This use of the disk competes with the file system's use, and disk optimizations must bear their complexity needs in mind. The granularity of virtual memory management is based on the block nature of the disk device. Also, the algorithms governing the virtual memory are designed to avoid disk accesses.

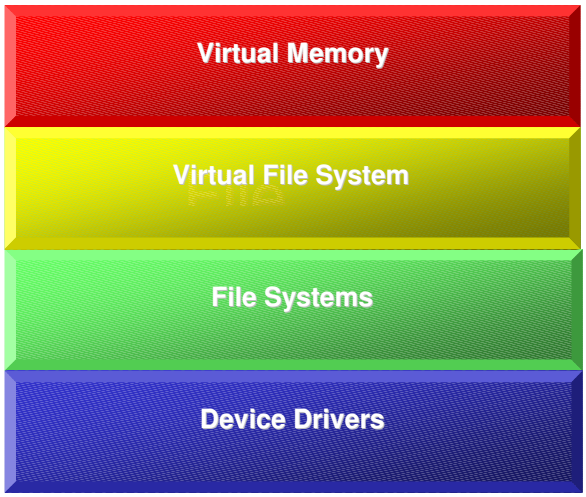


Figure 1.2: A common organization of storage service layers.

Unfortunately, disk-related complexity is still proliferating, and the storage substrate has become one of the largest and most complex components in modern

operating systems. The legacy assumption of disk storage has spread from the level of device driver and logical file systems to the level of memory management. In addition, the behaviors and interactions among various disk-related optimizations are becoming increasingly difficult to understand. In some cases, even applications are aware of rotational and seek behaviors of disk when designing their data structures.

1.3 The Emergence of Persistent RAM

Researchers have been seeking alternative storage technologies to overcome the deficiencies of disks for many years [Baker et al., 1992; Douglis et al., 1994]. Recently, persistent RAM technologies have emerged as a good candidate.

Unlike disk storage, persistent RAM can achieve fast and uniform access times under arbitrary access patterns. Unlike bubble memory, persistent RAM has reached the commercial stage, where sales revenues can further research momentum. In recent years, persistent RAM has entered a phase of high growth [Peripheral Research 1998; Toigo 1998], resulting in drastic price reductions and its ubiquity in personal digital assistants (PDAs), MP3 players, PCMCIA devices, digital cameras, and various small mobile computing devices [Toigo 1998].

Most commercial persistent RAM comes in two different physical flavors—flash memory and battery-backed DRAM (BB-DRAM). Both forms of persistent RAM can achieve two to six orders of magnitude faster access times than disks; however, each has its own characteristics. The major advantage of flash RAM is its ability to retain information with very low power. However, flash memory suffers from a number of drawbacks: (1) each memory location of a flash RAM is limited in terms of the number

of times it can be written and erased; (2) the erasure time is in the range of seconds [Cáceres et al., 1993]; and (3) the density of flash memory storage is low compared to DRAM, and its physical size imposes limitations for deployment on general-purpose machines. BB-DRAM can operate at the speed of DRAM for all operations, but it requires a constant supply of power (e.g., uninterruptible power supply or UPS) for persistent storage. A promising future technology is magnetic RAM (MRAM) [Boeve et al., 1999], which promises the density of DRAM, the speed of static RAM, and the nonvolatility of flash. MRAM is expected to become available in the market within a few years [Butner 2001].

One major concern in using RAM-based storage is economics. Figure 1.3 shows that the cost of persistent RAM is unlikely to match disks in the short run; however, the figure reveals several encouraging pieces of information.

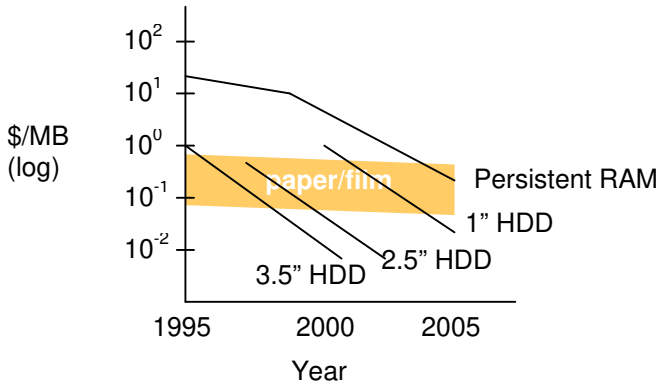


Figure 1.3: Average and projected price of storage. The shaded area shows the cost of paper and film storage as a comparison [Grochowski 2002].

First, this figure presents the price trend for paper and film as a base comparison. The cost region for paper and film has historically been a critical barrier to cross for a

storage media to become economically sensible and competitive. Recent introductory prices of disks with various geometries reflect the importance of this barrier for penetrating the market at a rapid rate. Currently, persistent RAM is crossing this barrier and is likely to become a cost-effective storage option by this measure in 2003. Second, the price decline rate of persistent RAM has accelerated since 1998 due to the boom of digital photography and PDAs. The rapid growth of market demand and the resulting economy of scale enable persistent-RAM manufacturers to profit with lower prices. Third, by 2005, high-end desktop machines will likely to be equipped with 4 to 10 Gbytes of persistent RAM for storage [Grochowski 2002].

1.4 *Conquest* Approach

Conquest is a disk/persistent-RAM hybrid file system that delivers all file system services from persistent RAM, with the single exception of providing high-capacity storage, which is offered by traditional disks. In essence, *Conquest* provides two specialized and simplified data paths to persistent-RAM and disk storage: (1) *Conquest* stores all small files and metadata (e.g., directories and file attributes); and (2) disks hold only the data content of remaining large files (with their metadata stored in persistent RAM). By doing so, *Conquest* achieves simpler data paths and superior performance compared to leading disk-based file systems.

In terms of system simplifications, the memory data path of *Conquest* is devoid of disk-related complexities. By matching the physical characteristics of disks with user access patterns, the disks need to handle only access patterns that are suitable for them,

leading to a simplified disk data path as well. As a side benefit, simplifying data paths also leads to shorter critical paths and more maintainable and reliable code.

For performance, the *Conquest* design performs all file system services and management on memory-resident data structures, thereby minimizing disk accesses. Tailoring file-system data structures and management to the physical characteristics of memory significantly improves performance compared to disk-based designs. As a result of *Conquest*'s simpler design, traversing the memory data path incurs no disk-related overhead, and the disk data path consists of only the processing costs for handling access patterns that are suitable for disk.

Another benefit of *Conquest* is its ability to provide a smooth and cost-effective transition from disk-based to persistent-RAM-based storage. Unlike other memory file systems [McKusick et al., 1990; Douglis et al., 1994; Wu and Zwaenepoel 1994], *Conquest* provides an incremental solution that allows persistent RAM to assume more file system responsibility as memory prices decline. Thus, *Conquest* can achieve most of the benefits of persistent RAM without the full cost of RAM-only solutions.

1.5 Roadmap

The reminder of this dissertation presents the design and implementation of *Conquest*, a disk/persistent-RAM hybrid file system. Chapter 2 first compares *Conquest* to disk caching and other popular alternatives at the conceptual level. Chapter 3 describes the technical design of *Conquest*, followed by a discussion of major design decisions, such as the delegation strategy of storage media (Section 3.2). The internal representation of *Conquest* data structures is discussed in Chapter 4, the support for *Conquest* content to

survive reboots while retaining legacy semantics of volatile memory allocations is described in Chapter 5, and the handling of data integrity, reliability and resource depletion is presented in Chapter 6. Chapter 7 demonstrates the performance and architectural improvements of *Conquest* over leading disk and memory approaches. Chapter 8 examines some major performance anomalies observed during the benchmarking process of *Conquest*. Chapter 9 relates *Conquest* to various disk optimization techniques, applications of persistent RAM, and systems that use similar design philosophies. Chapter 10 presents future improvements for *Conquest* and research directions suggested by the *Conquest* experience. Chapter 11 summarizes this dissertation and recapitulates contributions and lessons learned from designing and implementing *Conquest*.

Chapter 2

Alternatives to *Conquest*

The abundance of inexpensive RAM is a substantial departure from the conventional assumption that RAM is a scarce resource. Therefore, simple solutions derived from the old view of memory as a scarce resource fail to take complete advantage of new possibilities. In many cases, extensions to these simple methods can give results similar to the *Conquest* approach, but these extensions add so much complexity that they are no longer attractive alternatives.

This chapter discusses these approaches and their limitations. Some do not provide the expected performance gains, while others do not provide a complete solution to the problem of storing arbitrary amounts of data persistently, reliably, and conveniently. Rather than adding the complications necessary to fix these approaches, a better approach is to begin the design with a clean slate.

2.1 Caching

The most popular and effective approach to improving disk access speed is the disk buffer cache, which fulfills disk requests from memory whenever possible. Disk caching is based on a fundamental observation: the majority of disk accesses land on a relatively small fraction of disk blocks; therefore, a relatively small amount of memory is able to cache enough information to satisfy the majority of disk requests. At the implementation level, disk caching consists of policies to replace memory content when the buffer cache

is full, to commit written memory data to trade off between performance and data integrity, and to prefetch disk content. For instance, reading subsequent disk blocks into memory along with a requested block can speed up sequential accesses. Postponing the commit of changed data to disk can reduce the disk traffic significantly. The tradeoff is the potential loss of modified data stored in memory due to failure before it is committed to disk. Replacing least frequently used (LFU) memory blocks with newly read blocks from disk can keep the disk cache populated with frequently referenced disk content.

The existing disk caching approach is an attractive alternative to *Conquest*, since the cache tends to populate itself with frequently referenced files rather than wasting space on files that may have been untouched for lengthy periods. Also, adding more memory to existing machines requires no changes to the underlying operating system. Perhaps, with a few changes to the cache replacement policy, the existing disk buffer cache can behave just like *Conquest*, leaving only the data content of large files on disk.

However, disk caching has several drawbacks. Roselli et al., [2000] showed that caching often experiences diminishing benefits. Even very large caches have limited effectiveness in reducing read misses. They also found that for measured instructional, research, Web, and NT workloads, caches could experience miss rates as high as 10%, which is enough to reduce performance significantly.

In addition, disk-caching policies are speculative in nature, and they can potentially degrade file system performance when encountering access patterns that can fill disk caches with content that is unlikely to be reused in the near future. This effect of *cache pollution* occurs quite frequently when reading large files, buffering large numbers

of asynchronous writes, performing daily backups, searching the system, and maintaining the file system.

This problem led to complex solutions proposed at various levels of the computing stack to reduce the cache-miss latency. From the application level, programmers can disclose future disk access behaviors of an application, so that the system is better informed when deciding what and when to cache and prefetch [Patterson et al., 1995; Steere 1997]. At a lower level, a compiler can insert disk-block prefetching controls automatically into the source code by exploiting the information available during the process of compilation [Mowry et al., 1996]. At the system level, the kernel can monitor the relationships among file references and prefetch files automatically based on the observed reference patterns [Griffioen and Appleton 1995; Kroeger and Long 1996]. Both application- and compiler-level approaches require the source code to be available for modifications, and this constraint is limiting for applications that are only available in the form of binaries. The system-level monitoring and prefetching approaches are often resource-intensive and difficult to scale as the number of files increases.

Caches also require maintaining data consistency between memory and disk. A classic example is committing metadata. Since the correct functioning of a file system relies heavily on the integrity of metadata, many file system designs (VMS, MS-DOS, and most variants of UNIX file systems) choose the conservative approach of committing modified metadata from memory to disk synchronously [Ritchie and Thompson 1978; McKusick et al., 1984; Duncan 1986; McCoy 1990]. The resulting file systems have to commit metadata updates at disk speed rather than at speeds of memory and processor

[Ousterhout 1990; McVoy and Kleiman 1991; Seltzer et al., 1993]. The performance degradation is particularly visible for accessing small files where the overhead of updating metadata is significant, and the semantics of a write-back cache is effectively degraded into a write-through cache. Asynchronous solutions exist, but at the cost of significantly more code complexity. Two common approaches are journaling [Sweeney et al., 1996; Namesys 2002] and soft updates [Ganger et al., 2000; Seltzer et al., 2000]. The journaling approach records metadata updates in a memory-resident log, and the log is committed in a larger granularity to achieve high performance. The log is written to disk before the actual commitment of data, so that after a crash, replaying the log can bring a system back to a consistent state. The soft update approach maintains the dependency of metadata updates in the memory, so that modified blocks that contain metadata are written to disk in their required order without using synchronous disk commits.

Conquest differs fundamentally from disk caching in many ways: First, *Conquest* does not treat memory as a scarce resource. Instead, *Conquest* anticipates the abundance of cheap persistent RAM and uses the disk to store the data well suited to disk characteristics. Second, *Conquest* uses memory as the final storage location for all essential metadata and most files. Accessing memory content under *Conquest* involves no caching-related prefetching, eviction, and replacement mechanisms. Under popular benchmarks, merely adding more physical memory to existing disk caching results in a 43% to 96% slower performance compared to *Conquest*. Third, although *Conquest* may store many infrequently accessed small files in memory, the memory consumption for

those small files is not as significant in the new memory-rich environment. The disk component is also freed from complex optimizations for speeding up accesses to small files. Fourth, by permanently storing small files and metadata in memory, the cache pollution caused by large files has little effect on the access speed to *Conquest* memory content. Finally, *Conquest* can afford to use synchronous, in-place, metadata commit due to the memory speed, thereby avoiding many performance problems and code complexity.

2.2 RAM Drives and RAM File Systems

RAM drives and RAM file systems are two other popular approaches to avoid disk accesses. RAM drives and RAM file systems are sometimes used interchangeably, but they are accomplished through different means. *RAM drives* are accessed through the device interface. A RAM device is dedicated to handle a memory region preallocated to emulate a hard disk. To set up a RAM drive, the RAM device is first formatted into a particular file system just like a disk device, and the RAM device is mounted as a normal file system. RAM drives are attractive in the sense that creating RAM drives cleanly follows existing file system semantics and interfaces. However, the RAM drive is subject to all disk-related management—such as caching. Therefore, a piece of data can be both stored in the RAM drive and cached in the disk buffer cache, doubling the memory consumption and halving the potential access bandwidth.

A *RAM file system* can be implemented as a dynamically mountable kernel module, without the need for an associated device. RAM file systems under Linux and BSD [McKusick et al., 1990] are built on the top of various VFS temporary caches, so

memory consumed by the RAM file system can be dynamically allocated as needed. By saving both data and metadata directly into various caches under VFS, RAM file systems avoid the duplicate memory consumption of RAM drives, and RAM file systems are expected to perform at the speed of disk caching. In practice, RAM file systems tend to perform much faster than disk caching due to the lack of metadata commits to disk (Section 7.2.2). RAM file systems are commonly used as temporary file systems, and they demonstrate significant performance improvement for applications that generate many short-lived files (e.g., compilers).

At a brief glance, RAM drives or RAM file systems appear to be alternatives ready to replace disks with the arrival of inexpensive persistent RAM. However, on a closer look, neither RAM drives nor RAM file systems provide persistence of data across reboots. Although persistent RAM provides nonvolatility of memory content, persistence also requires a protocol for storing and retrieving the information from the persistent medium, so that data can survive reboots. Both the file system and the memory manager need to know how to resurrect information from the storage medium.

For RAM drives, although a file system protocol exists for storing and retrieving the in-memory information, there is no protocol to recover the associated memory states. Given that the existing memory manager makes no distinctions between persistent and temporary states, isolating these persistent memory states can be nontrivial. For RAM file systems, since the memory manager is unaware of data content stored under various VFS caches, neither the file system nor the memory states survive reboots without significant modifications to the existing memory manager.

Both RAM drives and RAM file systems also incur unnecessary disk-related overhead. On RAM drives, existing file systems, tuned for disk, are installed on the emulated drive without regard for the absence of the mechanical limitations of disks. For example, access to RAM drives is done in blocks, and the file system will still waste effort attempting to place files in "cylinder groups" even though cylinders and block boundaries no longer exist.

Although RAM file systems have eliminated some disk-related complexities, many RAM file systems rely on VFS and its generic storage access routines; many built-in mechanisms such as readahead and buffer-cache reflect assumptions that the underlying storage medium is slower than memory.

In addition, both RAM drives and RAM file systems limit the size of the files they can store to the size of main memory. These restrictions have limited the use of RAM drives and RAM file systems to caching and temporary file systems. To move to a general-purpose persistent-RAM file system, we need a substantially new design.

2.3 Customized Memory Filing Services Within Applications

A number of storage-intensive applications (e.g., databases) are designed with their own memory management and customized filing services in memory to avoid accessing disks. By accessing files within the same process address space, this approach can avoid the performance penalties of kernel crossings and system calls, in addition to expensive disk accesses. Also, since this approach involves no changes to the underlying kernel, modified applications are more portable to various operating system platforms.

There are two major drawbacks to this approach. First, application designers need to duplicate the effort of constructing filing and memory services already offered at the operating system level, not to mention the possible redundant efforts among similar applications. The required additional complexity and development costs to those applications can be nontrivial. Second, this approach involves modifying individual applications. A kernel-level approach can better deliver performance improvements transparently to legacy applications and to applications whose source code is not available.

2.4 Disk Emulators

To speed up deployment that is backward compatible, some manufacturers advocate the use of RAM-based disk emulators to avoid kernel modifications [ZDNet 1999]. These emulators generally plug into a standard SCSI or similar I/O port, and look exactly like a disk drive to the CPU. Although they provide a convenient solution to those who need an instant speedup, and they do not suffer the persistence problem of RAM drives, they again are an interim solution that does not address the underlying problem and does not take advantage of the unique benefits of RAM. In addition, standard I/O interfaces force the emulators to operate through inadequate access methods and low-bandwidth cables, greatly limiting the utility of this option as something other than a stopgap measure. As demonstrated in his study, Riedel [1998] observed as much as 55% overheads for accessing 8-Kbyte blocks sequentially from disk through the SCSI interface. Given that disk emulators are much faster than disk, the relative overheads imposed by the existing I/O interface will be significantly greater.

2.5 Ad Hoc Approaches

There are also less structured approaches to use existing tools to exploit the abundance of RAM. Ad hoc approaches are attractive because they can often avoid modifying the operating system. Also, the design, development, and deployment cycle for ad hoc solutions can be significantly shorter than an approach that involves complete redesign. For example, one could achieve persistence by manually transferring files into a RAM file system at boot time and preserving them again before shutdown. However, this method would require an end user to identify the set of files that are active and small enough to fit into the memory. The user also needs to be aware of whether doing so can yield enough benefit for the particular set of files. In addition, this approach would drastically limit the total file system size.

Another option is to manage RAM space by using a background daemon to stage files to a disk partition. However, this approach would require significant additional complexity to maintain a single name space (as does *Conquest*), and to preserve the semantics of symbolic and hard links when moving files between storage media. Also, since RAM and disk storage are two separate devices, the design must handle the semantics where only one of the devices is mounted. A simple approach is to manage both RAM and disk file systems with the semantics of a single file system; however, a single caching policy specified at the mount time cannot satisfy both the RAM and disk file systems, since caching contributes little toward accelerating RAM accesses and wastes memory resources. Separate caching policies demand that a file be able to change caching status dynamically. As details of an ad hoc approach grow, the resulting

complexity is likely to match or exceed that of *Conquest*, without achieving *Conquest* performance.

Chapter 3

Conquest File System Design

Conquest's design assumes the popular single-user desktop hardware environment enhanced with 1 to 4 Gbytes of persistent RAM (very affordable today). As of October 2002, we can add 2 Gbytes of battery-backed RAM to our desktop computers and deploy *Conquest* for under $200 [Price Watch 2002]. Extending the *Conquest* design to other environments, such as laptops and distributed systems, will be work for the future. This chapter first presents the design overview of *Conquest* (Section 3.1), followed by a discussion of various major design decisions (Section 3.2 and Chapters 4 to 6).

3.1 *Conquest* File System Design Overview

Conquest stores all small files and metadata in persistent RAM; disks hold only the data content of remaining large files. Section 3.2 will further discuss this storage delegation strategy.

An in-memory file is logically stored contiguously in persistent RAM. Disks store the data content of large files with coarse granularity, thereby reducing management overhead. For each large file, *Conquest* maintains a segment table in persistent RAM that tracks segments of data on disk. On-disk allocation is done contiguously whenever possible, and data are stored in temporal order similar to a log-structured file system (LFS) [Rosenblum and Ousterhout 1991]. Unlike LFS, *Conquest* commits large-file updates in-place as opposed to appending to the sequential log on-disk. Therefore,

Conquest does not consume contiguous disk storage nearly as fast as LFS and does not need to continuously perform disk cleaning in the background.

For each directory, *Conquest* maintains a dynamically allocated hash table of file metadata entries, with conflicts resolved by double hashing (Section 4.2.3). Hard links are supported by hashing multiple names (potentially under different directories) to the same file metadata entry.

The RAM storage allocation uses existing mechanisms in the memory manager to avoid duplicate functionality. For example, metadata allocation and memory allocation share similar requirements such as: (1) keeping track of unused resources, (2) avoiding duplicate allocation of resources, and (3) fast access to the allocated resource, given a reference to the resource. The *Conquest* storage manager can completely reuse the underlying implementation of the existing memory manager to perform metadata management.

Although it reuses the code of the existing memory manager, *Conquest* has its own dedicated instances of the manager, each governing its own memory region and residing persistently inside *Conquest*. Paging and swapping are disabled for *Conquest* memory, but enabled for the non-*Conquest* memory region for backward compatibility.

Unlike caching, RAM drives, and RAM file systems, *Conquest* memory is the final storage destination for many files and all metadata. A storage request can traverse the critical path of *Conquest*'s main store without such disk-related complexity as data duplication, migration, translation, synchronization, and associated management. Unlike

RAM drives and RAM file systems, *Conquest* provides persistence and storage capacity beyond the size limitations of the physical main store.

Since *Conquest* follows the VFS interface, *Conquest* has not changed the model of access controls, the memory protection mechanisms, or the resulting reliability model. However, since metadata commits are at memory speed, *Conquest* reduces the window of opportunity for the file system to become inconsistent. Updates to data structures are ordered in such a way that an interrupted transaction degenerates into the case of a memory leak, which can be garbage collected periodically. In the rare event of failures, *Conquest* falls back on its built-in checkpointing and daily backups to roll back to the last known file system state that is consistent.

3.2 Strategy for Delegating Storage Media

The decision on how to delegate the use of memory and disk is fundamental to the *Conquest* design, and this decision has contributed most of the performance and architectural simplification of *Conquest*. *Conquest*'s strategy to use memory and disk is based on a variety of studies on user access patterns and file size distributions. Sections 3.2.1 and 3.2.2 survey existing studies within the past two decades. Section 3.3.3 identifies some major trends from those studies that are exploited by *Conquest*. Sections 3.2.4 and 3.2.5 describe *Conquest*'s approach to using memory and disk based on knowledge of user workloads and file size distributions.

3.2.1 Early Studies on User Access Patterns and File Size Distributions

Given the wealth of publications on user workloads in the past two decades, it is surprising that many early discoveries and resulting wisdom remain relevant and still influence modern file system designs.

In one of the early empirical studies, Ousterhout et al., [1985] presented a trace-driven analysis of the UNIX 4.2 BSD file system. This BSD study collected about 3 million records from three servers (VAX-11/780s) at UC Berkeley, capturing workloads for program development, administration, and computer-aided design tools. From their traces, they concluded that individual users make only occasional (though bursty) use of the file system. Files are usually open only a short time, and they tend to be read or written sequentially in their entirety. Non-sequential access is rare. Most of the files that are accessed are short, and most new files have short lifetimes of only a few minutes.

Partially due to the shift from time-shared machines to personal workstations, Baker et al., [1991] chose to repeat the BSD study on 40 10-MIPS workstations running the Sprite operating system. They traced eight 24-hour periods, capturing workloads similar to the BSD study: program development, debugging, electronic mail, document production, and simulation. They observed that the per-user file throughput has become burstier and has increased by a factor of 20 since the BSD study. In many ways, results from this Sprite study agree with the BSD study: The vast majority of file access are to small files; however, large files have become an order of magnitude larger, so that they account for an increasing fraction of bytes transferred. File accesses remain largely

sequential, files are typically open for only a fraction of a second, and file lifetimes are short.

Bozman et al., [1991] conducted a similar study but in commercial environments. They collected traces from two separate IBM sites with the following workloads: program development and testing, preparing papers and publications, and sending and receiving electronic mail. They too reported that file system activity was bursty and dominated by reading. Most files are read and written sequentially. Finally, most read and write activity is for small amounts of data, but access to large files accounts for most of the bytes transferred.

Irlam [1993] performed a file size survey by requesting that the Usenet community run a simple shell script to collect statistical summary on various aspects of file systems. The survey consisted of 12 million files on 1050 file systems. He discovered that the file size is heavily skewed toward small files, but with a very long tail. The average file size is 22 Kbytes. About 10% of files consume 90% of disk space, and 90% of files take up only 10% of space.

3.2.2 Recent Studies on User Access Patterns and File Size Distributions

Studies on user workloads and file size distributions are constantly being updated, and recent studies are starting to reveal the long-term effects of changing hardware and environments on user workloads and file size distributions.

In its early stage, the *Conquest* design involved analyzing a web server trace collected in January 1998 at UC Berkeley's Computer Science Division [UC Berkeley 1998]. The trace contains about 5 million accesses generated from 150,000 sites to

70,000 files. A statistical summary of the trace shows that over 90% of accesses are to files with sizes smaller than 17 Kbytes (totaling 160 Mbytes, or 8.2% of total file size being referenced during the trace). Among all 249 referenced files that are larger than 1 Mbyte, over 90% of bandwidth is consumed by accessing files that have sequential-access semantics (Table 3.1).

Table 3.1

FILE BANDWIDTH DISTRIBUTION FOR FILES LARGER THAN 1 MBYTE.

File Types	Percentage of Bandwidth Consumption
PS (postscript printing file)	40%
AU (multimedia file)	15%
MOV (multimedia file)	8%
GZ (archive file)	8%
Z (archive file)	7%
HQX (printing or data file)	6%
AVI (multimedia file)	5%
MPG (multimedia file)	4%
PDF (printing or data file)	4%
PPT (application data file)	3%
U (multimedia file)	1%

The specific file types can be grouped into printing-related files (49%), multimedia files (33%), archive files (15%), and application data files (3%). Printing-related files are fed to printers sequentially in their entirety. Archive files are often created for compression, storing older information, and for ease of data transfer and organization. In both cases, the access patterns are largely sequential. Typically, multimedia files are temporally structured due to the common access patterns. For both video and audio streams, an end-user is very likely to view those files in a sequential order. The internal access granularities are large because video and audio outputs are inherently data-intensive to achieve an acceptable quality for human perceptions.

Douceur and Bolosky [1999] analyzed a snapshot of about 10,000 file systems from about 5,000 Windows personal computers at Microsoft. From the 140 million files captured in the snapshot, they found that the mean file size is about 64 Kbytes to 128 Kbytes, which is about three to six times larger than the earlier reported size under UNIX [Irlam 1993]. Larger files tend to be composed of blocks sized in powers of two. File-name extensions are strongly correlated with file sizes. On average, file systems are only half full.

Vogels [1999] collected three sets of two-week traces from 45 hosts running Windows NT. Compared to the Sprite study, they found that files are opened for increasingly shorter periods: most files are open for less than 10 milliseconds compared to 250 milliseconds in the Sprite study; most accessed files are short in length (less than 26 Kbytes); most access to files is sequential, but there is a clear shift toward random access when compared to Sprite; the size of large files has increased by an order of magnitude, and access to these files accounts for the majority of the transferred bytes; most newly created files are short-lived.

Roselli et al., [2000] conducted a trace study for a number of different environments, specifically, UNIX and Windows NT, client and server, instructional, research, and production. They reported that applications are accessing larger files than previously accessed. The size of large files has increased ten-fold since the Sprite study, but the percentage of large files has not increased (except for the NT workload). Files are either read-mostly or write-mostly, and larger files are more likely to be read than

written. However, compared to the Sprite study, larger files are more likely to be accessed randomly than they were in the past.

Evans and Kuenning [2002] gathered and analyzed 22 snapshots of file systems at Harvey Mudd College. Their snapshots included a variety of computing environments, including from Windows, Linux, and MacOS X. They reported that multimedia files have changed the size distributions found in previous years, and large files start to form a separate "bump" on the file size distribution.

3.2.3 Overall Trends of User Access Patterns and File Size Distributions

From various studies within the past two decades, some important trends emerge:

- Most files are small, and they consume a small fraction of total disk storage.
- Most accesses are to small files.
- Most accesses are sequential. This trend is shifting toward random accesses for two plausible reasons. First, the proliferation of dynamic libraries and data files that contain hyperlinks (e.g., Web pages and Microsoft documents) encourages direct associative accesses to various locations within files without the sequential access semantics; second, the existing definition of a random access may be misleading. Section 3.2.5 will reexamine this definition.

- Most storage is consumed by large files, which are read most of the time. Large files are also becoming larger over time as the popularity of multimedia files grows.
- For the first time, disk storage capacity is outgrowing its consumption rate as observed by Vogels [1999] and Santry et al., [1999]. This trend is a positive indication that memory storage capacity will likely outgrow the needs of an average user in the future.

Although one could imagine many complex data-placement algorithms (including a LRU-style migration of unused files to the disk), *Conquest* has taken advantage of the above characteristics by using a simple size threshold to choose which files are candidates for disk storage. Only the data content of files above the threshold (currently 1 Mbyte) is stored on disk. Smaller files and the metadata (for both large and small files) are stored in RAM. The current choice of threshold works well for various benchmark workloads (Section 7.2). Also, according to various UNIX file size studies [Irlam 1993; Roselli et al., 2000], this threshold will keep 99% of all files in RAM. By enlarging this threshold, *Conquest* can incrementally use more RAM storage as the price of RAM declines. The current threshold was chosen somewhat arbitrarily, and future research will examine its appropriateness.

The decision to use a threshold simplifies the code, yet does not waste an unreasonable amount of memory since small files do not consume a large amount of total space. An additional advantage of the size-based threshold is that all on-disk files are large, which allows us to achieve significant simplifications in disk management. For

example, we can avoid adding complexity to handle fragmentation with "large" and "small" disk blocks, as in FFS [McKusick et al., 1984]. Since we assume cheap and abundant RAM, the advantages of using a threshold far outweigh the small amount of space lost by storing rarely used small files in RAM.

The media delegation strategy for *Conquest* is not favorable for random seeks within large files; random seeks on disk are two orders of magnitude slower than sequential disk accesses. *Conquest* is not optimized for random seeks of large files that are frequently observed in database applications. *Conquest* covers common loads such as sequential accesses to large multimedia, archive, and compressed data objects. *Conquest*'s strategy also covers associative accesses among the small data objects (e.g., hypermedia and dynamic linked libraries).

3.2.4 Files Stored in Persistent RAM

Small files and metadata benefit the most from being stored in persistent RAM, given that they are more susceptible to disk seek time and rotational latency. Although persistent RAM can already achieve two to four orders of magnitude faster access time than disk at the block level, given per-byte granularity, the time to access a single byte in persistent RAM can actually be up to six orders of magnitude faster [Micron 2002].

Metadata no longer have dual representations, one in memory (runtime format) and one on disk (storage format). The removal of the disk representation also removes the complex synchronous or asynchronous mechanisms needed to propagate the metadata changes to disk [McKusick et al., 1984; Ganger et al., 2000; Seltzer et al., 2000; McKusick 2002], and avoids translation between the memory and disk representations.

Figure 3.1 illustrates how our use of persistent RAM results in a simpler data path. For conventional disk-based file systems, a storage request has to begin with I/O buffer management (e.g., LRU caching). If a request is not fulfilled by the I/O buffer cache, the request needs to go through persistence support (e.g., translating metadata between the memory and disk representations) and disk management (e.g., disk layout management). For *Conquest*, memory accesses bypass I/O buffer management and go only through the persistence support, which requires less processing since it stores only the memory representation of the metadata. In addition, updates to memory content are fast, synchronous, and in place.

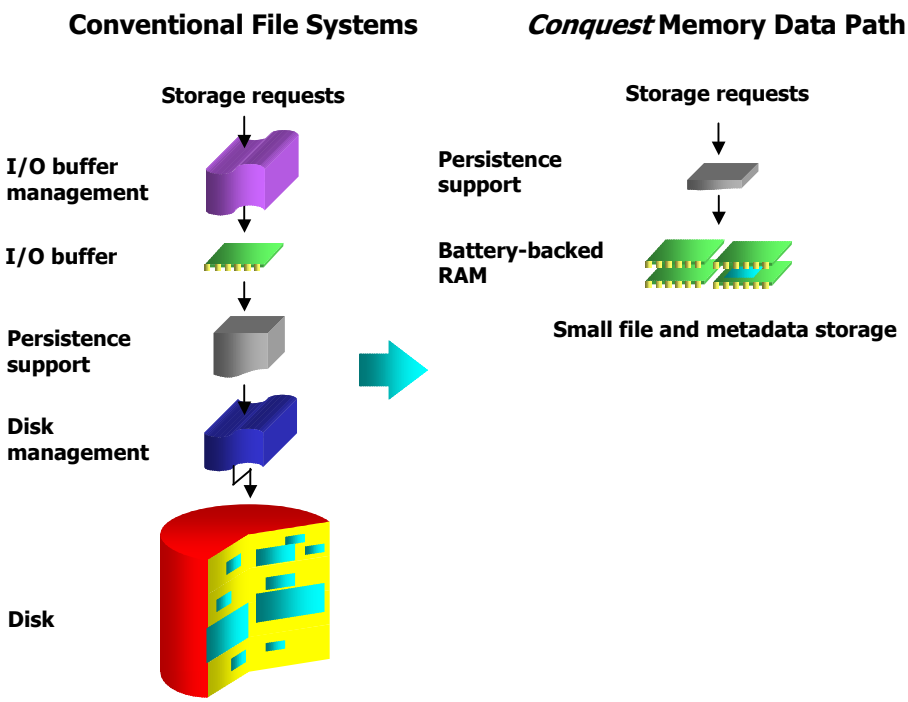


Figure 3.1: Comparison between the data path of conventional file systems and the memory data path of *Conquest*.

3.2.5 Large-File-Only Disk Storage

Historically, the handling of small files has been one major source of file system design complexity. Since small files are accessed frequently and have relatively high mechanical overheads, designers have explored various techniques to speed up small-file accesses. For example, the content of small files can be stored in the metadata directly, or a directory structure can be mapped into a balanced tree stored on disk to ensure a minimum number of indirections (or disk accesses) before locating a small file [Namesys 2002]. Methods to reduce the seek time and rotational latency [McKusick et al., 1984] are other attempts to speed up small-file accesses.

Optimal disk-access granularities tend to be large and fixed, so storing small files on disk leads to excessive internal fragmentation, a significant source of storage overhead. If small files must be stored on disk, a common solution is to introduce sub-granularities and extra management code [McKusick et al., 1984], which causes higher overhead and results in lower effective disk bandwidth.

Large-file-only disk storage can avoid all these small-file-related complexities, and management overhead can be reduced with coarser access granularity. Sequential-access-mostly large files exhibit well-defined read-ahead semantics. Large files are also read-mostly and incur little synchronization-related overhead. Combined with large data transfers and the lack of disk arm movements, disks can deliver near raw bandwidth when accessing such files.

Figure 3.2 compares the data path of conventional file systems with the disk data path of *Conquest*. Under *Conquest*, the persistence support is not in the disk critical path

because all metadata are stored and managed inside the memory. *Conquest* I/O buffer management and disk management are simpler because the disk storage under *Conquest* needs to handle only large files that are accessed sequentially most of the time.

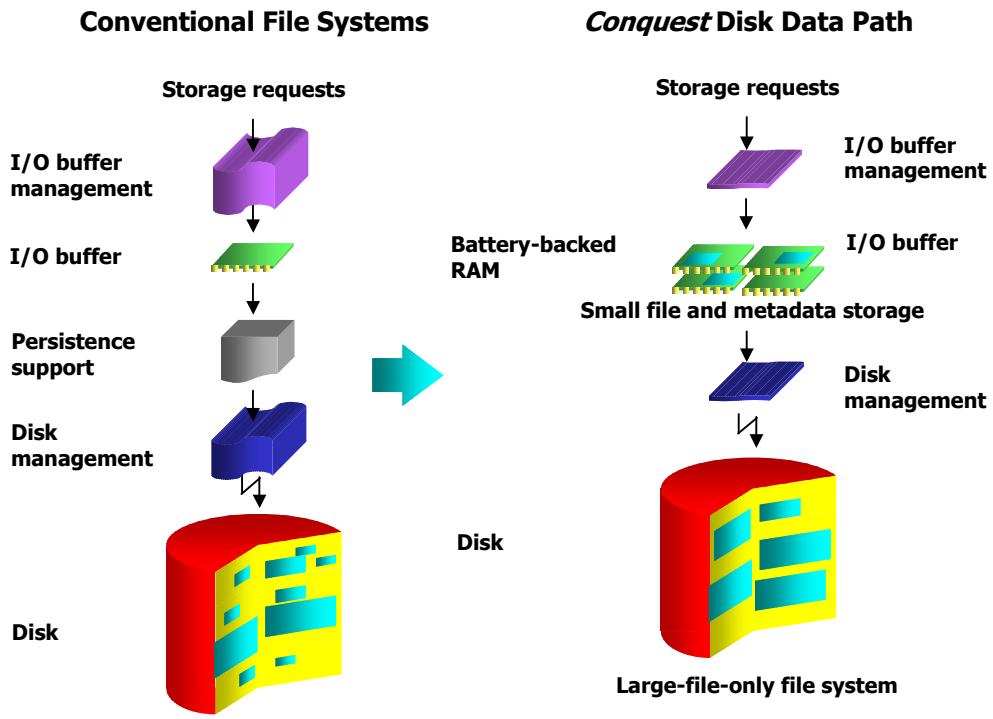


Figure 3.2: Comparison between the storage data path for conventional file systems and the disk data path of *Conquest*.

For randomly accessed large files, the commonly used term—*random*—deserves some reexamination. In the literature, an access is commonly defined as random if it is not sequential [Baker et al., 1991; Vogels 1999; Roselli et al., 2000]. This definition of random access may be misleading. Consider the MP3 format. The title of a file is stored at the end of the file, and is usually accessed when the file is opened. But most other references to the file are sequential from the beginning. For video files, there are a relatively small number of scene changes that an end user is likely to access, and within

each scene, video frames are likely to be viewed again in a sequential order. For such files the common notion of random access can more accurately be termed *near sequential*. With this observation, the metadata representation for large files can be greatly simplified, as described in the next chapter.

Chapter 4

Metadata Representation

The handling of file system metadata is critical, since this information is in the path of all file accesses. This chapter first describes how metadata are typically represented in legacy UNIX file systems (Section 4.1). The remaining sections detail how *Conquest* simplifies the existing metadata representation (Section 4.2), performs storage allocation (Section 4.2.2), and represents directories (Section 4.2.3).

4.1 UNIX Metadata Representation

At a high level, a *logical file* is commonly presented to users as data with associated file attributes (e.g., timestamp) or *metadata*. A *logical directory* is a compilation of files and directories, and directories are typically organized in a hierarchical manner (i.e., a file or a directory belongs to one parent directory). Directories are a form of *metadata* for a file system, since they are not usually the integral part of data that an end user intends to store. In particular, a file can be transferred from one file system to another with a completely different metadata representation for files and directories.

At a lower level, a *physical file* needs to consider the block access granularity of the low-level disk, and the file content is divided and stored in data blocks. As a result, metadata need to keep track of the disk data block locations. A *physical directory* needs to keep track of the files and directories residing underneath it, so an end user can use a

name to lookup a file or a directory within a directory. Sections 4.1.1 and 4.1.2 will describe how file metadata and directories are represented under UNIX.

4.1.1 Metadata Representation of a UNIX File

The metadata for a UNIX file is represented with the legacy *i*-node data structure, and its design reflects the deep-root assumption of disk storage [Thompson 1978]. Although file systems have evolved through many generations, the original *i*-node design has changed little in the past 30 years. Figure 4.1 shows the *i*-node design under the popular *ext2* file system [Card et al., 1994].

The *i*-node under *ext2* contains 13 pointers used to track locations of data blocks on disk. The first 10 pointers point to the first 10 data blocks. After consuming all 10 pointers, the 11th pointer points to a *single indirect block*, which contains pointers to data blocks. The 12th pointer points to a *double indirect block*, which contains pointers to single indirect blocks. The 13th pointer points to a *triple indirect block*, which contains pointers to double indirect blocks.

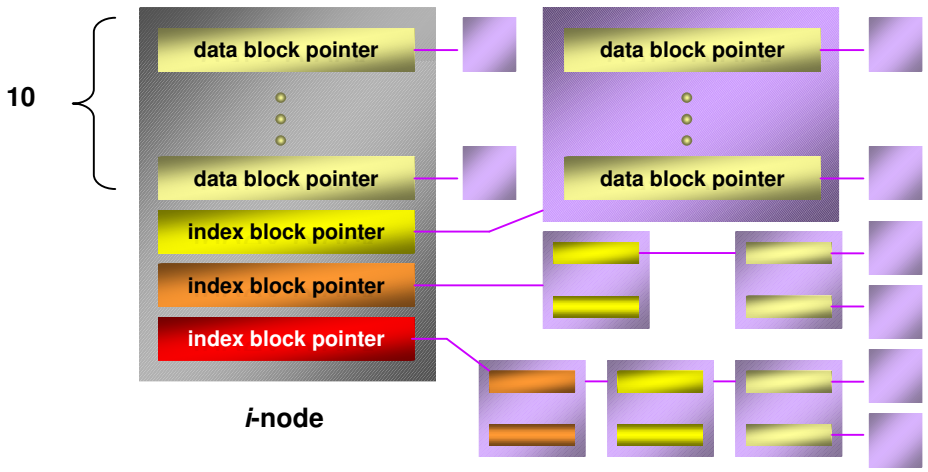


Figure 4.1: The *i*-node design under the *ext2* file system.

This design has a number of advantages. Frequently accessed small files can have fast direct access to data blocks, while infrequently accessed large files use the slower mechanism of traversing the nested indirect blocks. Block-based allocation avoids the need to manage *external fragmentation*, which occurs when pieces of unused storage are not contiguous or big enough for allocation. For block-based allocation, each unused piece of storage is always in blocks and is always allocatable. A single representation of files and directories simplifies the file system building blocks.

However, this design is limiting in several ways. First, a more complex design is needed to provide both fast accesses for frequently accessed small files and indexing for large files. The resulting skewed tree illustrates how optimizations for small file accesses complicate the data path for accessing large files. Second, although block-based allocation prevents external fragmentation, this design still needs to manage *internal fragmentation*, which occurs when an allocated piece of storage is not fully used. However, with inexpensive modern disk storage, the purpose of managing internal fragmentation is not so much to reduce the wasted storage, but rather to improve the disk access bandwidth. Accessing loosely packed blocks of data from small files can reduce disk bandwidth significantly, since disk transfers are at block granularity even for a partially used block. Third, a data structure with full support for random accesses imposes unnecessary overhead and complexity for the common case of sequential accesses for large files. Fourth, the access time of a single byte depends on the byte position in a file. Accessing the end of a large file would become inherently slower than

accessing the beginning of the same file due to the skewed-tree design. Finally, the total number of pointers provided by this data structure limits the largest file size.

4.1.2 Metadata Representation of a UNIX Directory

The original UNIX directories are represented as normal files, so the data blocks of a directory contain a list of *directory entries* (names and *i*-node identification numbers) for files and directories residing underneath. Under *ext2*, directory entries are stored on directory blocks in a variable length format, so that files with shorter names consume less storage. For simplicity, the size of an *ext2* directory shrinks only when the entire directory is removed. If a file is removed, a directory entry is simply merged with the previous entry by increasing the length of the previous entry. The observation is that the majority of removal operations involve the removal of an entire directory, and storage optimizations are not necessary.

The major advantage of this design is the reuse of file abstraction to manipulate directories. However, the frequent operation of file lookup (e.g., `ls` and `dir`) within a directory generally requires linear searches.

4.2 *Conquest* Metadata Representation

Conquest's metadata representation removes the nested indirect blocks from the commonly used *i*-node design. For in-core files, data blocks for a given file are accessed through a uniform, single-level dynamically allocated index array, in which each pointer points to a data block to achieve logical contiguity. For on-disk files, contiguous segments of disk data blocks for a given file are stored in a dynamically allocated, doubly

linked list in persistent RAM, so that maximum file size is limited only by the physical storage. The worst-case scenario of performing random accesses to on-disk files results in sequential searches in this in-core data structure. Directories are represented in hash tables, so the frequent operation of file lookup can be performed in *O(1)* time.

4.2.1 *Conquest i-Node*

Conquest does not use the *v*-node data structure provided by VFS to store metadata, because the *v*-node is designed to accommodate different file systems with a wide variety of attributes. Also, *Conquest* does not need many of the mechanisms involved in manipulating *v*-nodes, such as metadata caching. *Conquest's* file metadata consists of only the fields (53 bytes) needed to conform to POSIX specifications.

```
typedef struct conquest_inode_tag {
  unsigned int                magic;       /* 4 bytes */
  struct conquest_inode_tag   *id;         /* 4 bytes */
  unsigned                    reserved:4;  /* 1 byte */
  unsigned                    flags:4;
  nlink_t                     nlink;       /* 2 bytes */
  dev_t                       rdev;        /* 2 bytes */
  umode_t                     mode;        /* 4 bytes */
  unsigned int                blocks;      /* 4 bytes */
  uid_t                       uid;         /* 2 bytes */
  gid_t                       gid;         /* 2 bytes */
  size_t                      size;        /* 8 bytes */
  time_t                      atime;       /* 4 bytes */
  time_t                      mtime;       /* 4 bytes */
  time_t                      ctime;       /* 4 bytes */
  void *                      data;        /* 4 bytes */
  atomic_t                    count;       /* 4 bytes */
} conquest_inode;
```

Figure 4.2: *Conquest i*-node.

Figure 4.2 shows the fields in a *Conquest* *i*-node. The `magic` number (or a unique identification number) allows *Conquest* to verify an *i*-node pointer before deferencing, thereby reducing the chances of making unintentional modifications. The metadata ID is the physical address of the metadata, and the purpose is to avoid the implementation of metadata management completely. When allocating metadata, the existing memory manager is invoked to allocate a memory region with the size of a file's metadata. Since no two *Conquest* *i*-nodes can have the same physical address, using the physical address as the metadata ID assures unique IDs. In addition, with a given ID, the corresponding metadata can be quickly located. However, for this metadata numbering scheme to maintain valid associations between physical addresses and metadata IDs across reboots, the underlying memory manager needs to be aware that certain allocations can persist beyond reboots (Chapter 5). In addition, allocating small pieces of memory such as metadata raises the need to manage internal memory fragmentation (Chapter 5).

The `flags` indicate whether a file is located in memory, located on disk, in the process of being deleted, or a hard link. A *Conquest* *i*-node also keeps track of the number of hard links to a file, device information, file type (e.g., file, directory, or symbolic link), file size, data block count, access control information, and various timestamps. The data field refers to either the single-level index array or the doubly linked list, depending on the storage medium for the file. For directories, the data field can also refer to other data structures (described in Section 4.2.3). A counter-based lock is used to prevent premature deallocation of the *i*-node, which occurs when a deletion operation runs concurrently with other file operations.

4.2.2 Storage Allocation

For in-core write requests where the total allocation is unknown in advance, *Conquest* allocates data blocks incrementally. The current implementation does not return unused memory in the last block of a file. A possible future optimization is to add automatic truncation. *Conquest* also supports "holes" within a file, so parts of a file can remain unallocated if those parts are never written. This usage pattern is common in applications such as compilers, where a program would divide a file into several regions and randomly write to various file positions.

Disk storage is allocated contiguously whenever possible, in temporal or LFS order [Rosenblum and Ousterhout 1991]. Large-file data blocks are written sequentially to disk as the write requests arrive, without regard to file membership. The choice of this temporal order is simple and effective, since temporal locality seems to correlate with spatial locality in many workloads. Unlike LFS, *Conquest* stores metadata in-core, and existing disk data blocks are updated largely in-place as opposed to being appended to the end of the *disk log*. In addition, deleted data blocks are reclaimed immediately as opposed to being reclaimed in bulk. Therefore, *Conquest* does not consume contiguous regions of disk storage nearly as fast as LFS, and demands no continuous background disk cleaning.

Although *Conquest* has a linear search structure for disk storage, its simplicity and in-core speed outweigh its algorithmic inefficiency, as demonstrated in its performance evaluation (Chapter 7). In the worst case of severe disk fragmentation, *Conquest* will encounter a linear slowdown in traversing the link list. However, given

that *Conquest* has coarse disk allocation granularity, the segment list is likely to be short. Also, since the search is in-core but access is limited by disk bandwidth, random accesses to large files should degrade little compared to the corresponding performance of disk-based file systems.

The eventual design of the *Conquest* disk layout is to apply existing approaches from both video-on-demand (VoD) servers and traditional file systems research. For example, given its sequential-access nature, a large media file can be striped across concentric disk zones, so disk scanning can serve concurrent accesses more effectively [Chen and Thapar 1997]. Frequently accessed large files can be stored completely near outer zones on disk platters for higher disk bandwidth. Spatial and temporal ordering can be applied within each disk zone, at the granularity of an enlarged disk block.

Another example is to use track-aligned extents to align allocated large-file data segments at disk track boundaries [Schindler et al., 2002]. Empirical measurements of this approach have reported improved disk access efficiency up to 50% for mid-sized requests (100 to 500 Kbytes), which can be used as a practical granularity for allocating large files.

With a variety of options available, the presumption is that after enlarging the disk access granularity for large file accesses, disk transfer time will dominate access times. Since most large files are accessed sequentially, I/O buffering and simple predictive prefetching methods should still be able to deliver good read bandwidth.

4.2.3 Directory Representation

The data structure design for directories needs to meet three requirements: (1) fast sequential retrieval of directory entries for common traversal operations (e.g., `ls` and `dir`), (2) fast random lookup (e.g., locating a file), and (3) management for hard links. To meet these requirements, *Conquest*'s directory structure is built with a hierarchy of dynamically allocated hash tables using file names as keys. A *path* (e.g., `/usr/bin`) is resolved by recursively hashing each name component of the path at each level of the hash table. A collision is resolved by circularly searching the table for an empty hash entry at an interval defined by another hash function (double hashing). The second hash function uses a lookup table to generate prime search intervals.

As a simple example, Figure 4.3 shows how double hashing works under a 4-bit hash key scenario. Initially, the hashing of `file1` results in the key `1001`. After the key is masked with the size of the table `0011`, `file1` is hashed to the entry `01`. Then, if `file2` is hashed to the key `1101`, `file2` will encounter a collision with the `01` entry occupied by `file1`. To resolve the collision, the key of `file2` is used as a lookup index into an array of prime numbers. If the resulting prime number is `0010`, the prime number is added repeatedly to the key of `file2` (`1101`) with the table size masking, until an empty entry is found (if available).

From a storage perspective, the use of prime numbers to search empty table entries assures that the entire table is fully hashed before expansion, resulting in less memory waste. However, from a performance perspective, insertions of the last few

entries to the hash table before an expansion are likely to degenerate into linear searches. A simple remedy is to expand the table before it is full.

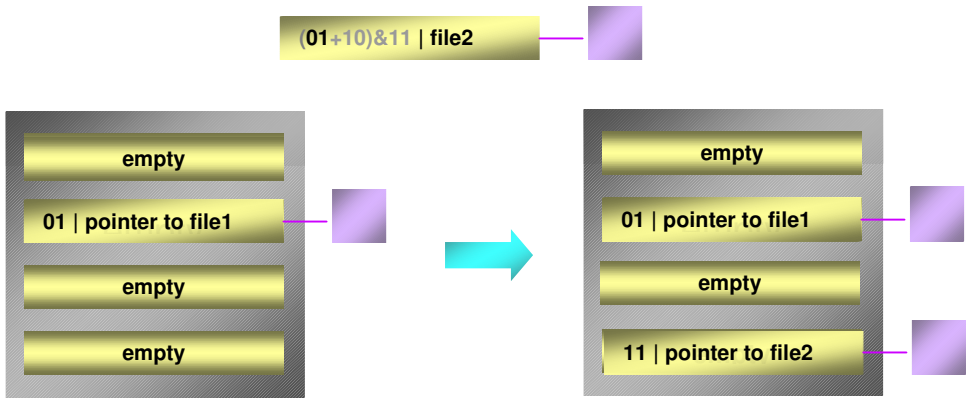


Figure 4.3: A doubly hashed table. Collisions are resolved through double hashing. The `file2` hash key collides with the `file1`; therefore, a prime number (stored in a pregenerated array) is repeatedly added to the key of `file2` (modulo the table size) to find an empty entry in the table.

Deletions of doubly hashed entries also need special handling. For example, after deletion of `file1`, the `01` entry is marked as `DELETED` as opposed to `EMPTY`, so the lookup of `file2` can continue after encountering a `DELETED` entry and stop at either finding the `file2` entry or an `EMPTY` entry.

The current implementation of *Conquest* hash functions is specific to 32-bit architectures, but the conceptual design is applicable to 64-bit architectures. Figure 4.4 shows the primary hash function in C code. The hash function combines the Buz hashing [Uzgalis and Tong 1995] and the hash function used for metadata caching under Linux 2.4.2. The hash function computes the hash table index by processing a file name in the reverse order. Similar to the Buz hashing, each character in the file name is first mapped to a pre-computed 32-bit random number stored in an array, so that the introduction of

each character can affect all bits of the resulting hash key. *Conquest* currently uses `lrand48()` to pre-generate these pseudo-random numbers. Similar to the hash function used under Linux, the *Conquest* hash function `XOR`s upper and middle bits with lower bits to try to increase the randomness of lower bits. In the case of collisions, the search interval for a given file name is defined by hashing the original hash key to a pre-generated prime number.

```
inline unsigned
int hash(const unsigned char *file_name,
         unsigned char name_len) {
       int i;
       unsigned int hash_key = 0;

       for (i = name_len - 1; i >= 0; i--) {
             hash_key ^= rand48[file_name[i]];
             hash_key = hash_key ^ (hash_key >> 10)
                   ^ (hash_key >> 20);
       }
}
```

Figure 4.4: The primary hash function for the doubly hashed tables under *Conquest*.

Compared to *ext2*'s approach, hashing simplifies management of directories that live in multiple (possibly indirect) blocks. Hashing still allows hard links by allowing multiple names to hash to the same file metadata entry.

However, the use of dynamically allocated, doubly hashed tables needs to address the legacy semantics of directories being represented as normal files. For example, for each opened file, the owner process typically maintains a pointer to the current file position, which needs to be adjusted dynamically as a hash table (directory) resizes and rehashes its hash contents (directory entries). The situation becomes particularly problematic with the `readdir()` file system call which, according to the POSIX

specification, should return the next directory entry on each call and walk through the entire directory after being called enough times. When a directory is either growing or shrinking, `readdir()` may return duplicate entries or skip entries because a typical hash table resize operation randomly rehashes and reorders existing hash entries. This scenario occurs quite frequently during recursive deletions, which could cause deletions to fail to remove all files. Although *Conquest* can provide an auxiliary sorted list for directory walks and handling `readdir()`, it currently just disables shrinking for hash tables to handle the common case of recursive deletions. So far, growing directories have not caused noticeable problems. In the future, *Conquest* will use an alternative directory representation that can handle dynamic resize operations.

One promising candidate is variants of *extensible hash tables* [Fagin et al., 1979]. Unlike most hash tables, extensible hash tables use the upper bits of the hash table indices for hashing. As an extensible hash table grows or shrinks, a bit mask is adjusted dynamically to increase and decrease the width of hash table indices (Figure 4.5). One nice property of extensible hashing is that the ordering of hashing entries is preserved across table resizes, which is crucial for scenarios such as recursive deletions.

Interestingly, this data structure, invented back in the 70s, is well suited for the modern RAM-rich environment. The discovery of an old simple solution for a new environment demonstrates the value of rethinking and redesigning systems when encountering drastic changes in legacy assumptions.

The implementation of extensible hashing also requires overcoming a number of minor engineering nuisances. For example, popular hash functions tend to randomize

lower bits, and some hash functions do not consider the sign bit and upper bits. The dynamic resizing of the hash table also means that current file positions must be interpreted in the context of the bit mask for the current size of the hash table. Collision handling also involves many design choices. For example, with each hash bin containing a single data entry, an extensible hash table can grow very quickly due to the birthday paradox [Cormen et al., 1990]. With each hash bin containing multiple data entries, the implementation and fault recovery mechanism can grow in complexity rather quickly. Finding the balance between memory consumption and implementation complexity is essential for *Conquest* to move toward the extensible hashing solution.

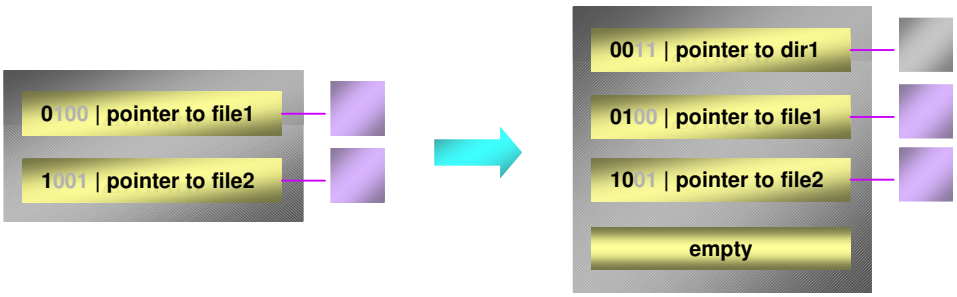


Figure 4.5: Expansion of a variant of the extensible hash table. Extensible hashing uses upper bits of hash keys to perform hashing. As the hash table expands, additional upper bits are used to rehash table entries. Note that the ordering of hash entries is unchanged across the resize operation.

Chapter 5

Conquest Persistence Support

Since the proper operation of *Conquest* metadata allocation depends on a persistent association between the metadata ID and the physical memory address, *Conquest* needs additional mechanisms for the underlying memory manager to retain persistent states across reboots. Otherwise, at boot times, an operating system will reinitialize the memory manager and erase all information pertaining to prior allocations of *Conquest* metadata. While retaining information in persistent RAM seems simple, the design for *Conquest* persistence also needs to retain the legacy semantics of booting, where the content of volatile memory (or the content not pertaining to *Conquest*) needs to be properly reset. This chapter first describes the existing memory manager under Linux 2.4.2 (Section 5.1) and explores the design space for *Conquest* persistence support (Section 5.2). This chapter then describes the *Conquest* design (Section 5.3) and details its implementation (Section 5.4).

5.1 Memory Manager Under Linux 2.4.2

The memory manager under Linux 2.4.2 is structured into three layers (Figure 5.1). The bottom layer (closest to the memory hardware) is the page allocator, which tracks allocation and memory attributes at the granularity of individual pages. The page allocator is relatively simple, and locating an unallocated memory region is largely based

on two levels of linear searches through allocation bitmaps. However, as the memory size increases, a linear-search-based page allocator becomes prohibitive to scale.

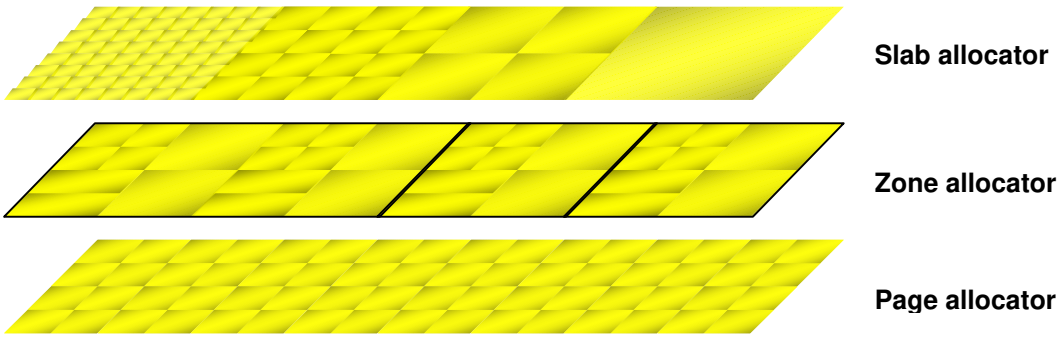


Figure 5.1: Linux memory manager. A memory allocation has to traverse through the slab allocator, the zone allocator, and the page allocator.

The layer above the page allocator is the zone allocator, which allocates memory zones of potentially different sizes for responsibilities such as direct memory access, high memory, and I/O buffering. Most zones are specified to share responsibilities to balance memory allocations, thereby avoiding the heavy cost of performing exhaustive searches as the memory within a zone is nearly depleted. Within each zone, the zone allocator uses buddy allocation [Peterson and Norman 1977] to accelerate allocation of memory pages. In brief, buddy allocation predefines allocation granularities in power-of-two pages, so memory regions can split and merge quickly to fulfill memory requests. One natural problem arising from the buddy allocation scheme is internal fragmentation within allocated pages, which leads to the need for a higher-level slab allocator [Bonwick 1994].

The slab allocator provides an efficient means of allocating memory at sub-page granularities and reducing internal memory fragmentation. The basic observation is that

common data structures are typically small compared to the memory page size, and they are likely to be allocated frequently. Therefore, the slab allocator allocates one page at a time, which is filled with objects of a single type that are initialized in bulk. By doing so, the slab allocator can significantly reduce the overhead of allocation, deallocation, and initialization. For example, when *Conquest* allocates a *Conquest i*-node for the first time, the slab allocator will allocate one page that is to be filled with initialized *Conquest i*-nodes, so the next allocation will not incur the cost of allocating the page and initializing the *i*-node. Also, since a single type of data structures is likely to be referenced together due to locality, compacting data structures of a similar type into pages can improve the memory behavior of a program, in the sense that temporally related references are directed to a small number of memory pages.

5.2 Design Space for *Conquest*'s Persistence Support

With the existing memory manager architecture, preserving persistent states across reboots is difficult for three reasons: (1) *Conquest* persistence support needs to preserve the mapping among three layers of memory management. For example, pages allocated for the slab allocator may belong to different instances of a buddy allocator under different zones. (2) Existing memory manager layers have no notion of persistence; therefore, temporary and persistent allocations are intermingled within three layers of complex data structures. (3) States within the buddy and slab allocators have causal dependencies on the request ordering of memory allocation. It is difficult to bypass the standard memory deallocation routines to remove the internal memory states for temporary allocations directly, without also disturbing the states for persistent allocations.

A number of alternative designs were explored before reaching the current design of *Conquest* persistence support. At one extreme of the design space, all states within the existing memory manager could be first treated as volatile states that are erased at boot times. Persistent states of the memory manager can then be reconstructed through serialization and deserialization, similar to the disk approach. One major advantage of this design is that the serialized image (along with memory and disk data content) can be transferred to another machine and can be resurrected by *Conquest*, potentially running under other versions of Linux (or under different OS platforms). However, this design reintroduces the disk-related mechanisms of serialization and deserialization, which are particularly complex for the existing pointer-rich layers of Linux's memory manager.

An alternative would be to log or summarize memory requests in an efficient manner, so that the log could be replayed to regenerate only the states for persistently allocated memory regions. However, logging each memory allocation and deallocation request would be storage-intensive, considering the extreme high frequencies of memory allocation and deallocation requests. In addition, both serialization and logging impose relatively high complexity at the kernel level, while contributing little to the core functionality of file system services.

At the other extreme of the design space, all states within the memory manager are first treated as persistent states. Unwanted temporary states are then selectively erased through standard memory deallocation routines at either shutdown or boot time. By doing so, states within the Linux memory manager can achieve persistence across reboots without constructing complex mechanisms for serialization and logging. Also, this

approach does not violate the existing interface for memory allocation and deallocation. However, this approach has a few severe drawbacks. First, the performance penalty of going through the memory deallocation routines is relatively high compared to directly zeroing out regions of memory. Second, since the mechanisms involved in discerning temporary from persistent states are inherently kernel-specific, content stored under *Conquest* becomes dependent on the specifics of a kernel implementation. Third, the semantics of handling memory manager states are poorly defined. In particular, this selective deletion of memory states still requires the handling of persistent states even when *Conquest* is not mounted, so that a subsequent mount of *Conquest* can operate correctly. Imposing *Conquest*-related overhead to boot sessions that do not use *Conquest* is highly undesirable.

5.3 The Design of *Conquest*'s Persistence Support

Between the two extremes of the above design alternatives, *Conquest* adapts an approach that can both zero out unwanted memory regions directly and reconstruct persistent states without complex mechanisms for serialization and logging. To be specific, *Conquest* dynamically allocates memory zones, with each containing its own instantiation of the Linux memory manager (Figure 5.2). Swapping and paging are disabled for *Conquest* zones to reduce their associated overheads. However, swapping and paging are enabled for non-*Conquest* zones for backward compatibility (i.e., for memory-intensive applications).

By instantiating from the existing Linux code base, *Conquest* can reuse and share existing memory management code without building additional components. The reuse

of the existing memory manager also implies compliance with the existing memory interface and leverage of all existing memory services. *Conquest* memory regions can be allocated through the same `kmalloc()` calls with additional *Conquest* flags. Memory fragmentation management is handled by the existing slab allocator design.

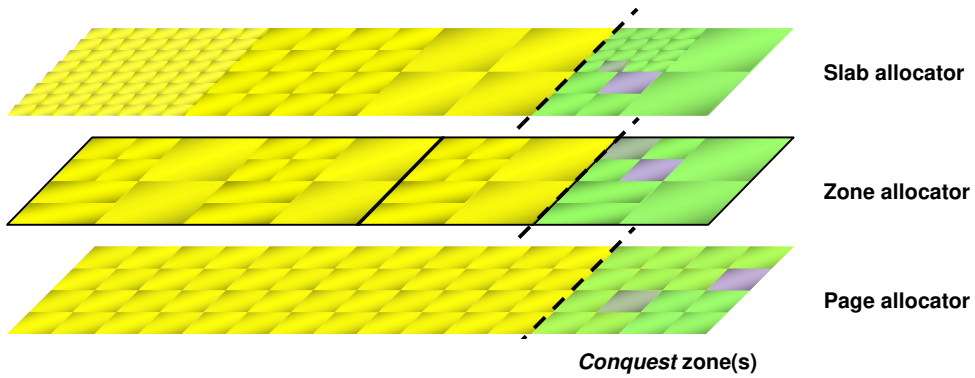


Figure 5.2: *Conquest* memory manager. *Conquest* dynamically allocates memory zones, and each zone contains its own instantiation of the Linux manager. Since all pointers within a *Conquest* memory manager use physical addresses, and they point only to memory addresses within the governed zone, *Conquest* memory managers can survive reboots without serializing those pointers.

By making an instantiated memory manager reside within the *Conquest* memory zone that it governs, all pointers within those manager layers can be preserved across reboots due to the following invariants: (1) Those pointers use physical memory addresses that are unchanged across power cycles, and (2) those pointers only point to physical addresses within the governed memory zone. Both invariants can be trivially satisfied and verified. The resulting encapsulation avoids the complex serialization and logging code, and the runtime data structures of the *Conquest* memory manager can survive across reboots. The zone-based isolation also simplifies the semantics when

Conquest is not mounted. A system can use the remaining memory resources to operate without *Conquest*-related overhead.

Like most file systems, the *Conquest* design needs to rely on the specifics of a kernel version to achieve competitive performance. Fortunately, since many system components rely on the memory management substrate, the code base for memory management of common operating systems is relatively stable and not as frequently changed as other kernel components.

Another drawback of *Conquest* is that a single storage medium can no longer be pulled from a computer to transfer data physically. BB-DRAM, in particular, is difficult to remove physically without shutting down a system. However, this constraint can be overcome with the construction of a simple tool to dump the memory image of *Conquest* zones to a disk before the disk is physically transferred.

One additional concern with the *Conquest* approach is that the existing memory manager layers may not match *Conquest*'s usage patterns well. For example, if all in-core files are incrementally allocated at the page granularity, the buddy allocator will have no chance to allocate multiple pages. Therefore, the buddy allocator imposes unnecessary overhead and data-path complexity, since it functions essentially like a page allocator. Also, the slab allocator might not anticipate the high number of small objects with skewed file-size distributions; allocator components that are currently based on linear searches may need redesign for better scaling. One possibility would be to custom-build a memory manager for *Conquest*, but the new memory manager will likely duplicate the existing memory manager in a number of functionalities.

According to the benchmark numbers in Chapter 7, the existing memory manager does not seem to pose performance limitations with 2 Gbytes of physical RAM. Those performance limitations may become more visible as the size of memory grows by an order of magnitude.

5.4 The Implementation of *Conquest*'s Persistence Support

The main challenge to implementing *Conquest*'s persistence support was to instantiate the Linux memory manager by reusing available memory interfaces and code constructs. The Linux memory manager is instantiated from the bottom up. The instantiation process starts with a *bootmem allocator* that is statically allocated by the Linux kernel. The allocator resides in a designated range of memory addresses. The bootmem allocator allows the instantiation of a slightly more complex page allocator, which is the construction block for the zone allocator and instances of buddy allocators. At the top, the slab allocator requests individual pages or multiple pages from buddy allocators residing in different zones.

To allow *Conquest* to instantiate various allocators within a *Conquest* zone, one could use the same bottom-up ordering of instantiation. However, given that high-level memory services are available at the time of *Conquest* instantiation, *Conquest* can use those services to perform instantiations top down, thereby avoiding various ad hoc bootstrapping mechanisms, such as constructing the bootmem allocator.

5.4.1 *Conquest* Instantiation of Slab Allocator

A *Conquest* zone instantiates a slab allocator by using the existing slab interfaces. The instantiation process involves cloning the slab data structure, which is a two-level tree (Figure 5.3). The first level is a list of slab types, categorized by either the size or the type of memory allocation requests. The second level is the actual list of memory objects being allocated. Each level has its own corresponding constructor and destructor calls. With the option of specifying the memory zone to request memory pages, the *Conquest* implementation mainly involves invoking existing slab interface calls. Additional code deals with keeping track of the root node of *Conquest*'s two-level tree.

Conquest instantiates the slab allocator by first cloning the two-level tree structure under the original two-level tree. The cloning of the first-level tree is achieved by constructing an object list that is the same type as the original slab at the second-level of the original tree. The second-level tree of *Conquest* is created by invoking the original second-level constructor with slab-type objects belonging to *Conquest*.

The first-level list of *Conquest* is initialized with the original first-level constructor. *Conquest* does not reuse entries from the original first-level list because the slab allocator creates a page of objects at a time, and potentially two slab types from different zones can falsely share a page. Should a slab type belonging to *Conquest* be stored on the same page allocated from a non-*Conquest* zone, *Conquest*'s slab type will be erased after the next reboot. Should a slab type not belonging to *Conquest* be stored on a page allocated from the *Conquest* zone, the temporarily allocated slab type may

persist across reboots even though its object list may have been erased, resulting in inconsistent states in the memory manager.

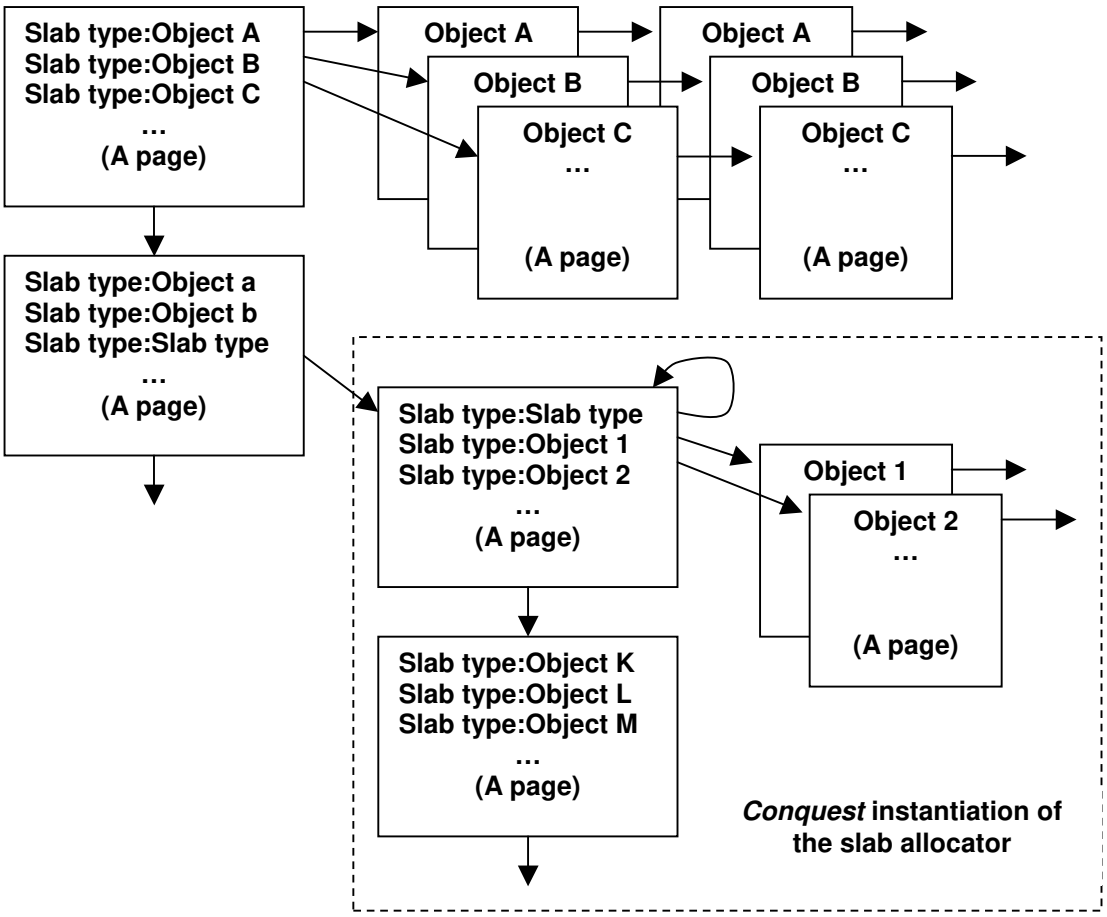


Figure 5.3: Linux and *Conquest* instantiation of the slab allocator.

A clean creation of the object list assures that no allocated objects under *Conquest* can falsely share a page with other allocated objects not under *Conquest*. Also, with this type of implementation, the *Conquest* portion of the slab allocator can survive reboot independent of the remaining data structures of the slab allocator. In addition, *Conquest* can completely reuse the initialization code for the first-level list.

Conquest also needs to track the memory being used by the allocator itself. If it fails to do so, the instantiated allocator could potentially return its own memory as an unused region to a memory request, thereby putting the instantiated allocator at risk for an overwrite. To accomplish this bookkeeping, the first entry of the object list points to itself, so that the first object list of the *Conquest* slab allocator is the slab-type list itself.

5.4.2 *Conquest* Instantiation of Zone, Buddy, and Page Allocator

A buddy allocator constructs lists of power-of-two memory pages on top of the page allocator. Figure 5.4 shows the `zone` data structure, which keeps track of clean unallocated pages and the states of a buddy allocator (the free memory area regions that belong to certain size categories). The linked list of pages is constructed from an array of the `page` data structure (under the page allocator) that keeps track of states and attributes for individual pages.

Conquest's instantiation of various allocators is greatly simplified by observing certain properties of these allocators. First, each zone is defined by a contiguous physical address space, which translates into a contiguous range of pages under the page allocator. Second, with proper specification at zone creation time, pointers involved in the various allocators will always point to the predefined range of memory address space belonging to a zone. Third, the physical address of a zone remains static across reboots. By knowing the above conditions, all pointers involved in zone, buddy, and page management can survive across reboots.

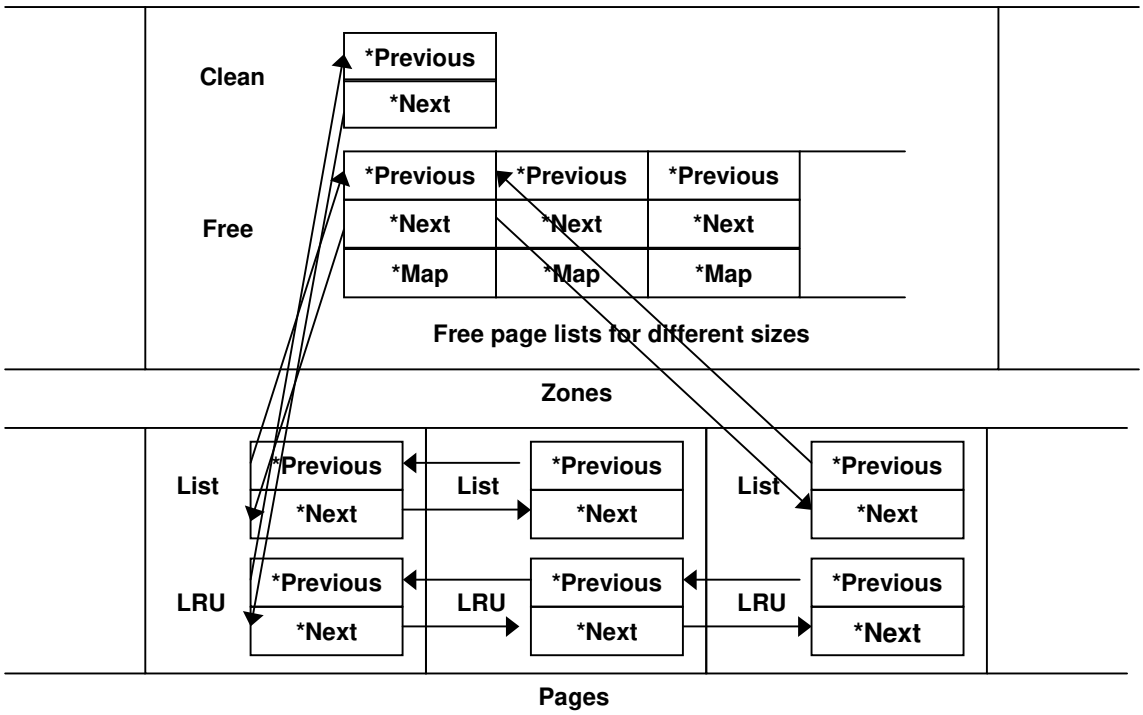


Figure 5.4: Data structures for zone, buddy, and page allocators.

5.4.3 Persistence Emulation

Since the standard PC BIOS zeros out memory at the kernel boot time, *Conquest* needs a simple mechanism for dumping and restoring *Conquest* memory zones to emulate the semantics of persistent RAM. Currently, the deployment of *Conquest* requires two disk partitions: one for the disk storage, and another for dumping and storing the memory image. Dumping of memory content occurs at shutdown time, and restoration occurs at boot time. The use of timestamps at the beginning and end of the dump and the restoration can ensure completion of these processes. The use of `magic` numbers and proper flags can prevent the *Conquest* file system from being formatted accidentally.

Conquest currently does not cover all cases of failures, such as failing in the middle of the memory dumping and restoration processes; however, solutions for these

cases are proposed in Section 6.1.3. Note that the code pertaining to the solutions can be removed once *Conquest* is deployed on a machine with a *Conquest*-aware BIOS. In future work, *Conquest* can leverage the on-going LinuxBIOS project [Minnich et al., 2000], which is an open-source endeavor that allows researchers to modify BIOS for research purposes (such as *Conquest*).

Chapter 6

Conquest Deployment Considerations

Although *Conquest*'s design overcomes the cost constraint of persistent RAM, the practical deployment of *Conquest* also relies on the proper handling of storage reliability and memory depletion. In addition, the *Conquest* implementation needs to minimize changes to the existing kernel structure, so that resulting code is more maintainable as the underlying kernel evolves. This chapter first compares *Conquest* to disk-based file systems in terms of reliability models, and then describes techniques used by *Conquest* to provide reliability (Section 6.1). Section 6.2 explores the design space for handling memory depletion. Section 6.3 reports the implementation status of *Conquest*.

6.1 Reliability

Storing data in-memory inevitably raises concerns of reliability and data integrity. At the conceptual level, disk storage is often less vulnerable to corruption by software failures because it is less likely to perform illegal operations through the rigid disk interface, unless memory-mapped. Main memory has a very simple interface of load and store, which allows a greater risk of corruption. A single wild kernel pointer could destroy many important files. However, a closer examination reveals that memory can be reliable enough as a persistent storage medium in the sense that the use of common disk reliability techniques (e.g., backup) can provide a similar level of protection against data loss and failures.

6.1.1 *Conquest* vs. Disk-Based File Systems

Although conventional file systems use disk as the persistent storage, a significant portion of memory is still used for caching data and metadata. The integrity of the memory content of disk-based file systems is protected at multiple fronts: the rigid interface of the VFS, the built-in access control mechanisms within the VFS, and the underlying memory protection. For example, a misbehaving application owned by `root` can trespass the VFS interface and the access control mechanisms, but will still be trapped when making an illegal access to a memory block. From this perspective, *Conquest* offers the same reliability model as disk-based file systems, since *Conquest* does not alter any VFS reliability and memory protection mechanisms mentioned to ensure the integrity of memory content.

At the kernel level, operating system crashes raise another threat to reliability. A study at the University of Michigan examined 650 induced operating system crashes, ranging from bit errors in the kernel stack to deleting branch instructions to C-level allocation management errors. From this study, researchers discovered that 1.1% of crashes corrupted the data on disk, compared to 1.5% for memory corruptions. Assuming one system crash every two months, one can expect to lose in-memory data (to the extent allowed by the memory protection mechanisms described above) about once a decade [Ng et al., 1996]. These numbers also indicate that memory can serve as a reasonably reliable storage.

At the hardware level, modern disks have a mean time between failures (MTBF) of 1 million hours [Seagate 2002]. Two hardware components, the RAM and the battery

backup system, cause *Conquest*'s MTBF to be different from that of a disk. Currently, *Conquest* uses a UPS as the battery backup. The MTBF of a modern UPS is lower than that of disks, but is still around 100,000 hours [Gibson and Patterson 1993; Power Conversion Systems 2000]. The MTBF of the RAM is comparable to disk [Micron 1997]. However, the MTBF of *Conquest* is dominated by the characteristics of the complete computer system; modern machines again have an MTBF of over 100,000 hours. Thus, it can be seen that, at most, *Conquest* should lose data due to hardware failures only once every few years. For common users, this level of reliability is well within the acceptable range when standard backup procedures are used.

6.1.2 Pointer-Switch Commits

In addition to a low memory corruption rate, *Conquest* also relies on a number of known techniques to enhance reliability. For example, assuming that a failure cannot occur in the middle of a pointer assignment, *Conquest* can use the pointer assignment as an atomic operation to commit data updates or newly allocated objects.

Figure 6.1 shows how a pointer assignment can be used as a commit operation. To assign a newly allocated `new_object` to a `pointer`, the `pointer` assignment needs to occur after the `new_object` is fully initialized and ready to commit. The `old_object` being pointed by `temp_pointer` is deallocated after the *pointer-switch* commit. Should a failure occur between the allocation of the `new_object` and the `pointer` assignment, the pointer will still point to the `old_object`. The `new_object` becomes a memory leak, which can be garbage collected, and the integrity of the `old_object` being pointed to by the `pointer` remains intact. Should

a failure occur between the `pointer` assignment and the deallocation of the `old_object` pointed by the `temp_pointer`, the `pointer` already points to the `new_object` (committed). The `old_object` becomes a memory leak, which again can be garbage collected.

```
pointer = old_object; /* initial value */

temp_pointer = pointer;
new_object = malloc(sizeof(object));
initialize(new_object);
pointer = new_object;
free(temp_pointer);
```

Figure 6.1: An example of using a pointer assignment as a commit operation. In the case of a failure, this sequence of updates to the pointer results in a memory leak, which can be garbage collected.

The pointer-switch commit is particularly useful in a hierarchical organization of data structures, where a single pointer-switch at a high level of the hierarchy can commit many updates to the low level of the hierarchy being switched. Given that *Conquest*'s metadata is organized largely in hierarchy, the pointer-switch commit allows *Conquest* to transition from one consistent file system state to another consistent file system state.

Therefore, *Conquest* can implement synchronous metadata updates to assure the integrity of critical file system services, such as directories. In addition, `magic` numbers in critical data structures can detect corruptions before pointers are dereferenced or modified.

6.1.3 Recovery from Catastrophic Crashes

In the rare event of a catastrophic crash (e.g., both the power source and the UPS fail), *Conquest* currently relies on daily backups to recover a recently checkpointed file system image. For the purpose of emulating persistent RAM, *Conquest* also has a built-in mechanism to checkpoint the memory content to disk at shutdown times. However, this shutdown-time checkpointing is insufficient to perform on-line checkpointing. With on-line periodic memory dumping, infrequently checkpointed metadata can still be incorrectly associated to data content of large files, which is written to disk more frequently through the I/O buffering mechanism. As future work, the current *Conquest* checkpointing mechanism can be extended so that updates to large files are directed to alternative locations on disk after a memory checkpoint. By doing so, the metadata in memory will point to the more current data content of large files, while the recently checkpointed metadata will point to a corresponding version of data content for large files.

This approach suggests that at the time of checkpointing, *Conquest* needs to flush all dirty data content of large files from I/O buffers to disk. *Conquest* also needs to handle the case where a failure may occur in the middle of the checkpointing process. The simplest approach is to recover from the latest system backup. A more integrated solution is to avoid overwriting the previously captured file system image on disk, so that *Conquest* can recover from an older image if needed.

6.2 Memory Depletion

Memory depletion occurs when allocation for persistent memory storage fails. The equivalent event for the conventional file system is disk depletion. Memory depletion

can cause programs or operating systems to fail, and graceful recoveries are not always possible. In conventional systems, memory depletion is commonly handled by the virtual memory, so processes that are not currently running are temporarily swapped out to disk to free up memory. Disk depletion is commonly handled by reserving extra disk storage, so the processes of reclaiming or rearranging disk storage can still create temporary files.

Under *Conquest*, the design space for handling memory depletion ranges from sophisticated, fine-grained, speculative data migration (e.g., LRU management), to simply complaining to the user (as in PDAs). Fine-grained, speculative data migration is appealing in the sense that the memory can provide the illusion of an infinite storage, of a size practically bounded by the disk size. For example, the LRU caching keeps a brief history of memory references at the page level, so memory pages that are not recently used are replaced by newly read content from disk. However, from the *Conquest* perspective, memory is abundant; the fine-grained, speculative migration of individual memory blocks imposes unnecessarily high system complexity and overhead, as demonstrated in the performance study of *Conquest* (Chapter 7).

A coarser-grained approach allows *Conquest* to adjust the large-file threshold dynamically, so smaller files can migrate to disk when the memory is nearly depleted. Although this approach allows *Conquest* to operate in a more memory-constrained environment (e.g., laptops), a dynamic threshold can complicate *Conquest* in two ways. First, large-file-only disk storage is potentially threatened with smaller-than-expected files and associated performance degradation. In particular, bulk migration of data based on file sizes may capture neither spatial nor temporal localities. One remedy is to design

a new disk layout, so that different disk zones store files accessed or created at similar time frames improve temporal locality. Nevertheless, this design would reintroduce disk-related complexity that conflicts with the original goal of *Conquest*—achieving significantly better performance with reduced data-path complexity. Second, changing the large-file threshold can potentially lead to a bulk migration of files. While the intended effect of bulk migration is to free up memory space, this migration may need to be scheduled offline and designed as an infrequent or incremental event to avoid visible performance degradation by end-users.

Currently, *Conquest* uses the simplest strategy of just declaring the depletion of memory. This approach creates a firm boundary to prevent the disk code from handling small files. *Conquest*'s design also reflects the disk depletion case, where freeing up storage requires an end user to archive infrequently used data content (usually at the granularity of directories) to removable media.

With an abundance of memory, memory depletion can be handled in a similar way, except that the same archiving operation has different semantics under *Conquest*. After archiving a large-enough directory (which contains small-file data content greater than 1 Mbyte), the newly created archived file is automatically transferred from memory to disk due to the file-size threshold, thereby freeing up memory storage. (Of course, *Conquest* needs to reserve just enough memory to handle the scenario of memory depletion.) As future work, one could automate this migration process at the user level, with the management at the coarse granularity of directories, as opposed to memory

blocks. Also, an attempt to reference a migrated directory would cause the entire tree to be restored.

6.3 *Conquest* Implementation Status

Conquest is fully operational as a loadable kernel module under Linux 2.4.2, with minor modifications to the Linux memory manager to support persistence. The current implementation follows the VFS API, but *Conquest* needs to override generic file access routines at times to provide both memory and on-disk accesses.

For example, inside the read routine, *Conquest* assumes that accessing memory is the common case, and accessing disk is forwarded through a secondary data path. The memory data path no longer contains code for checking the status of the buffer cache, faulting and prefetching pages from disk, flushing dirty pages to disk to make space, performing garbage collection, and so on. The disk data path no longer contains mechanisms for on-disk metadata chasing or various internal fragmentation and seek-time optimizations for small files.

Conquest does not disable caching of metadata under the VFS due to difficulties in removing the deep-rooted assumption of high-latency storage and the pervasive use of data and metadata caching. As a result, routines within the VFS often use the data structures of caching as the standard internal representation for lookups and function parameters. Tailoring a memory data path through the VFS is likely to involve redesigning the data representation, the data paths, and the interfaces of internal calls within the VFS. *Conquest* currently passes its metadata structure through VFS calls such

as `mknod`, `unlink`, and `lookup`. However, the VFS was altered so that the *Conquest* metadata, in particular the file system root, are not destroyed at `umount` times.

Conquest is POSIX-compliant and supports both memory and on-disk storage. *Conquest* currently uses a 1-Mbyte static dividing line to separate small files from large files (Chapter 3). Large files are stored on disk in 4-Kbyte blocks to reuse the existing paging and protection code without alterations. An optimization would be to enlarge the block size to 64 Kbytes or 256 Kbytes for better performance.

Chapter 7

Conquest Evaluation

The evaluation of *Conquest* is divided into two components: (1) architectural simplification, and (2) performance improvement. Section 7.1 will present a high-level feature count to illustrate how *Conquest* data paths are simplified compared to the conventional data path of disk-based file systems. Section 7.2 will then compare the performance of *Conquest* with leading disk-based file systems under popular benchmarks. Section 7.3 summarizes the evaluation results of *Conquest*.

7.1 Architectural Simplification

As a simple approximation, we assume that the complexity of software of the same type is proportional to the size of the program. The size of a program is commonly summarized as the number of semicolons and lines of commented source code. C*onquest* is at least 20% smaller than the popular *ext2* file system in terms of semicolons. However, line-counting metrics have their limitations. For example, by no means are 100 lines of high-level function calls comparable to 100 lines of raw algorithmic implementation. Instead of exploring alternative and more complex metrics such as decision tree analysis and counting the number of branch instructions, we will illustrate the data path simplification of *Conquest* qualitatively at the feature level.

7.1.1 Data Path for Conventional File Systems

Figure 7.1 shows the data path for conventional disk-based file systems. A typical storage request has to first go through the I/O buffer management, which includes mechanisms to allocate, deallocate, and garbage-collect I/O buffers. I/O buffer management also abstracts away common file system functions such as speculative memory-management logic (e.g., predictive readahead, write behind, and cache replacement) and the caching of data and metadata.

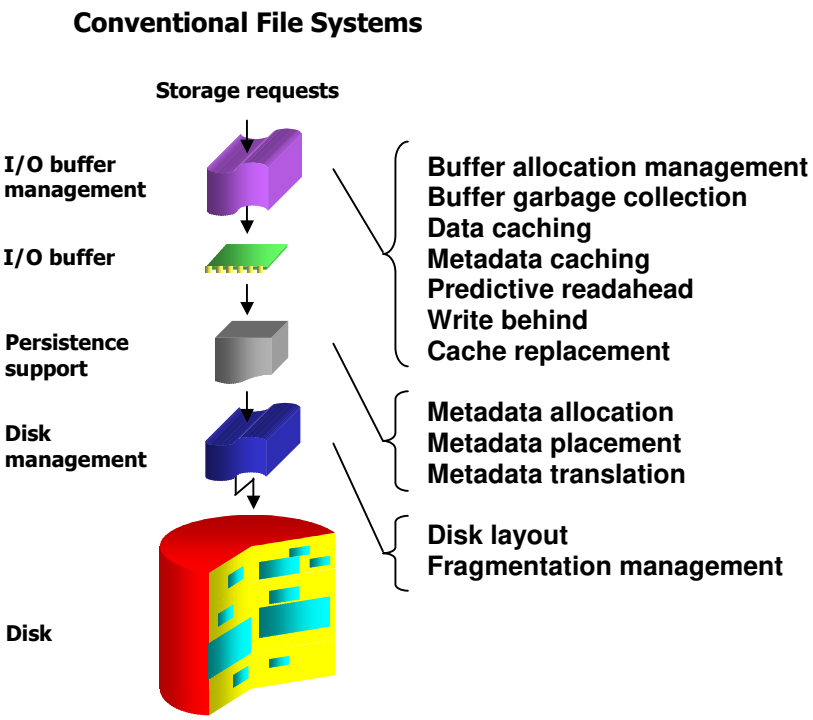


Figure 7.1: The conventional data path for disk-based file systems.

If the I/O buffer in the physical memory cannot fulfill the storage request, a file system has to track down the requested content by going through the persistence support component, which keeps track of the metadata and the data locations on disk. For a

typical disk-based file system, the persistence support needs to handle the allocation, deallocation, and placement of metadata on disk. The persistence support also needs to translate the metadata between the runtime memory format and the on-disk, block-oriented, serialized format. Finally, a storage request has to locate the data by following the disk layout and consulting with the fragmentation manager to see if the data content is stored in a sub-block.

7.1.2 *Conquest* Memory Data Path

Figure 7.2 shows the memory data path of *Conquest*. A typical access to the *Conquest* memory storage bypasses all complexity of I/O buffer management and disk management.

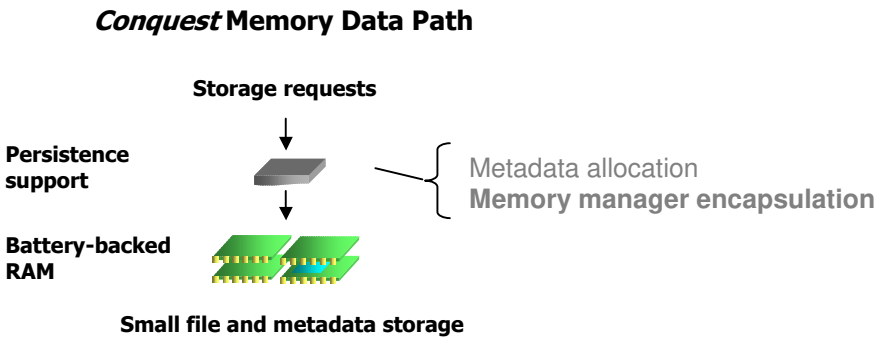


Figure 7.2: The *Conquest* memory data path. *Conquest* has removed the I/O buffer and disk management. The persistence support under *Conquest* consists of a simplified metadata allocation component (not in bold type font) and mechanisms to encapsulate memory manager.

For the persistence support, reusing the underlying Linux memory manager reduces the metadata allocation service under *Conquest* to only metadata initialization routines, invoked in a tight loop when the Linux memory manager allocates metadata entries to fill up one page at a time. This reuse also avoids implementing the metadata

placement completely. By storing metadata in its runtime form, *Conquest* additionally avoids translating the metadata between the memory and disk storage formats. However, *Conquest* requires a new component to instantiate and to encapsulate its memory manager within various *Conquest* zones. The implementation reuses existing memory manager constructs whenever possible, adding about 500 lines of commented, high-level, kernel function calls to the Linux memory manager (which consists of approximately 14,000 lines of commented C code).

7.1.3 *Conquest* Disk Data Path

Figure 7.3 shows the disk data path of *Conquest*. Since *Conquest* has not altered the VFS API for the reasons mentioned in Section 6.3, a typical disk request still goes through the I/O buffer management code under the VFS, with metadata caching still in place. However, given that speculative memory-management logic is controlled at the file-system level, *Conquest* can exploit the access characteristics of sequentially accessed large files and reduce the complexity for predictive readahead, write behind, and cache replacement policies. Since *Conquest*'s metadata are stored in memory, the disk data path can bypass the persistence support. In addition, *Conquest* stores only data blocks of large files on disk. Disk management does not need to handle fragmentation management, or to perform complex disk layout optimizations for small files.

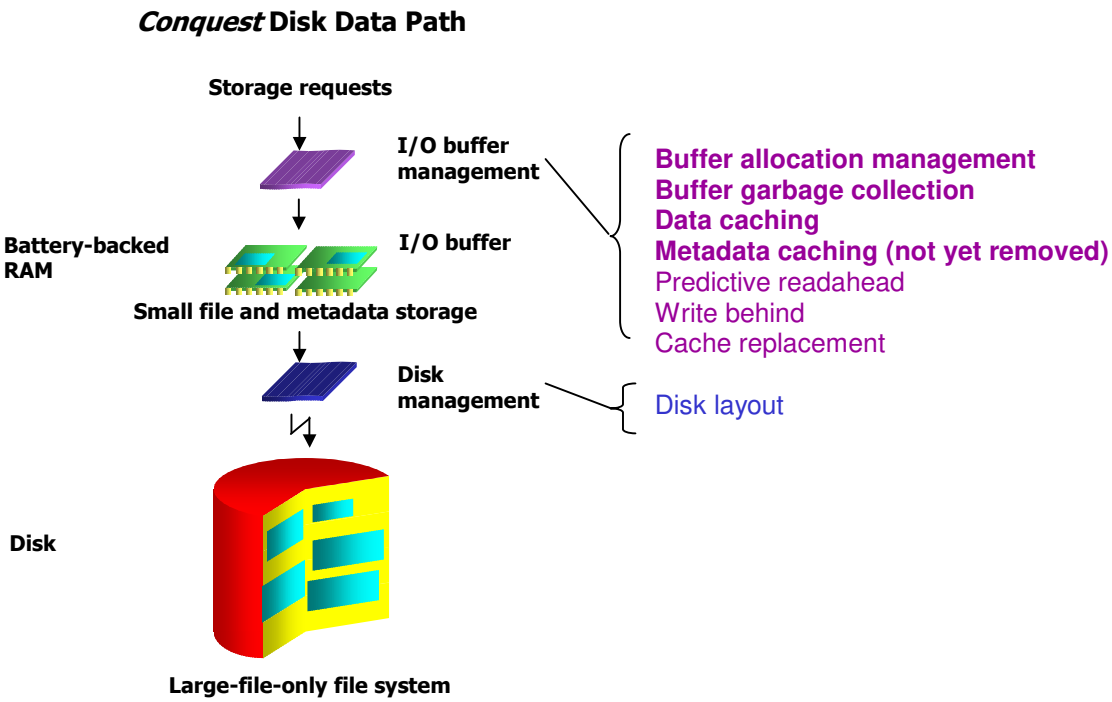


Figure 7.3: The *Conquest* disk data path. *Conquest* has removed persistence support and disk fragmentation management. *Conquest* has also simplified many disk-related components (not in bold type font).

7.1.4 Summary of Architectural Simplification by *Conquest*

As a brief recap of architectural simplification, the memory data path of *Conquest* consists of no disk-related optimizations, and accessing memory content under *Conquest* does not need to go through the I/O buffer management. The disk data path of *Conquest* consists of simplified I/O buffer management, and the data path avoids the metadata management and disk fragmentation management.

7.2 Performance Evaluation

We compared the performance of *Conquest* with that of both disk-based and memory-based file systems: *ext*2 [Card et al., 1994], *reiserfs* [Namesys 2002], *SGI XFS* [Sweeney

et al., 1996], and *ramfs* by Transmeta [Shankland 2001]. The choice of disk-based file systems (*ext2*, *reiserfs*, and *SGI XFS*) is largely based on their common use for various performance comparisons. Since both *Conquest* and *ramfs* are under the VFS API and various OS legacy constraints, the use of *ramfs* aimed at approximating the practical achievable bound for *Conquest* performance. Note that *ramfs* uses the page cache and *v*-nodes of the VFS to store the file system content and metadata directly, and provides no means of achieving data persistence after a system reboot.

Table 7.1 describes the experimental platform. Note that with 2 Gbytes of physical RAM, all disk-based file systems use caching extensively, and the performance numbers presented reflect how well various file systems can exploit memory hardware. In the experiments, all file systems have the same amount of memory available as *Conquest*.

Table 7.1

EXPERIMENTAL PLATFORM.

	Experimental platform
Manufacturer model	Dell PowerEdge 4400
Processor	1 GHz 32-bit Xeon Pentium
Processor bus	133 MHz
Memory	4x512 MB, Micron MT18LSDT6472G, SYNCH, 133 MHz, CL3, ECC
L2 cache	256 KB Advanced
Disk	73.4 GB, 10,000 RPM, Seagate ST173404LC
Disk partition for testing	6.1 GB partition starting at cylinder 7197 (8783 cylinders total)
I/O adaptor	Adaptec AIC-7899 Ultra 160/m SCSI host Adaptor, BIOS v25306
UPS	APC Smart-UPS 700
OS	Linux 2.4.2

Table 7.2

FILE SYSTEM SETTINGS.

	File system settings
Conquest	creation: default; mount: default
ext2fs (0.5b)	creation: default; mount: default
tramsmeta ramfs	creation: default; mount: default
reiserfs (3.6.25)	creation: default; mount: -o notail
SGI XFS (1.0)	creation: -l size=32768b mount: -o logbufs=8, logbsize32768

Table 7.2 lists various file system settings. *Reiserfs* and *SGI XFS* were not created and mounted with default settings because the default settings for those two file systems assume memory is a scarce resource. Section 7.2.1.1 will detail those non-default options.

Although the most widely used benchmark in file system literature is the Andrew File System Benchmark [Howard et al., 1988], the benchmark no longer stresses modern file systems because its data set is too small. The chosen benchmarks for this study are the Sprite LFS microbenchmarks [Rosenblum and Ousterhout 1991] with modifications to one of the benchmark programs (Section 7.2.1.2), The PostMark macrobenchmark[1] [Katcher 1997], and a modified version of PostMark macrobenchmark that will be described in Section 7.2.3. All results are presented at a 90% confidence level. The details of individual benchmark experiments will be discussed in corresponding subsections.

[1] As downloaded, Postmark v1.5 reported times only to a 1-second resolution. The benchmark was altered to report timing data at the resolution of the system clock.

7.2.1 Sprite LFS Microbenchmarks

The Sprite LFS microbenchmarks measure the latency and throughput of various file operations, and the benchmark consists of two separate suites for small and large files.

7.2.1.1 Small-File Benchmark

The small-file benchmark measures the latency of file operations. Each run consists of creating, reading, and unlinking 10,000 1-Kbyte files, in three separate phases. Since the total file size is only 10 Mbytes, which fits comfortably within the 2-Gbyte RAM, the machine being measured was rebooted only when switching file systems. For each file system, the performance numbers were collected over six runs, but averaged over only the last five runs to avoid the warm-up effect. Figure 7.4 summarizes the results.

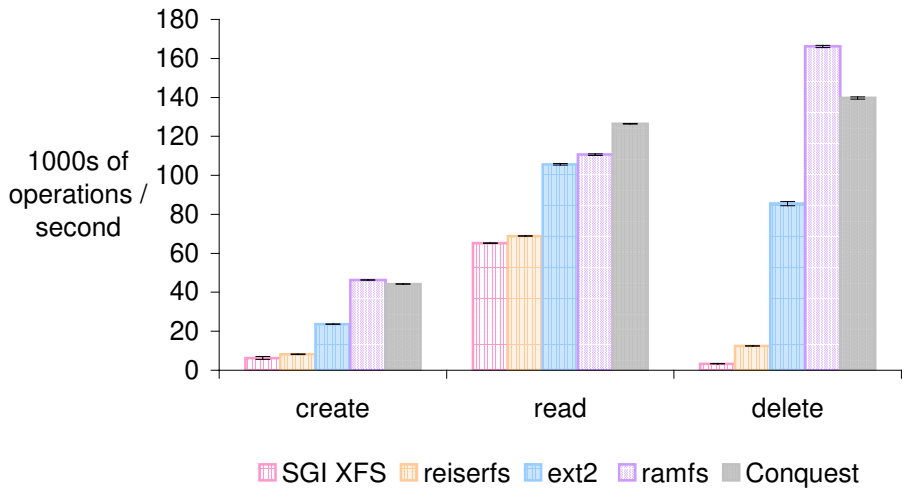


Figure 7.4: Transaction rate for the different phases of the Sprite LFS small-file benchmark, run over *SGI XFS*, *reiserfs*, *ext2*, *ramfs,* and *Conquest.* Each run of the benchmark creates, reads, and unlinks 10,000 1-Kbyte files in separate phases. Each data point is averaged over five runs. In this and subsequent figures, the 90% confidence bars are nearly invisible due to the narrow confidence intervals.

Conquest vs. *ramfs*

Compared to *ramfs*, *Conquest* incurs 5% overhead in file creation and 13% for deletion. This discrepancy is caused by *Conquest*'s need to maintain its own metadata and hashing data structures to support persistence, which is not provided by the *ramfs*. Also, *Conquest* has not removed or disabled VFS caching for metadata for reasons mentioned in Section 6.3; therefore, at times, VFS needs to go through an extra level of indirection to access *Conquest* metadata, while the metadata of *ramfs* are stored in the VFS cache directly.

Nevertheless, *Conquest* has demonstrated a 15% faster read transaction rate than *ramfs*, even though *ramfs* was considered to be a practical bound for the memory performance of file systems. *Conquest* is able to improve the read performance because the critical path to the memory data contains no generic disk-related code, such as readahead and checking for cache status.

Conquest vs. Disk-Based File Systems

Compared to *ext2*, *Conquest* demonstrates a 50% speed improvement for creation and deletion, mostly attributable to the lack of committing metadata synchronously to disk. Like *ramfs*, *ext2* uses the generic disk access routines provided by VFS, and *Conquest* is 19% faster in read performance than cached *ext2*.

The performance of *SGI XFS* and *reiserfs* is slower than *ext2* because of journaling overheads and their memory behaviors. *Reiserfs* achieved even poorer performance with its original default settings. Interestingly, *reiserfs* performs better with the *notail* option, which disables certain disk optimizations for small files. One example

of the disk optimizations being disabled is not to use a balanced tree to store the content of small files and the fractional blocks at the end of large files. While the intent of using this balanced tree is to save extra disk accesses, its overhead outweighs the benefits when a sufficient amount of memory is available to buffer the disk.

As for *SGI XFS*, its original default settings also produced poorer performance, since journaling consumes the log buffer quite rapidly. With a larger buffer size for logging, *SGI XFS* performance improved. The numbers for both *reiserfs* and *SGI XFS* suggest that the overhead of journaling is very high.

7.2.1.2 Modified Large-File Benchmark

Each run of the original large-file benchmark writes a large file sequentially, reads from it sequentially, and then writes a new large file randomly, reads it randomly, and finally reads it sequentially. Data was `fsync`ed to disk at the end of each write phase. The final read phase was originally designed to measure the sequential read performance after random write requests were sequentially appended to the log in a log-structured file system.

The original benchmark was intended to measure disk performance. When directly applied to measuring the memory performance of file systems, the benchmark revealed a number of anomalies due to effects of memory and L2 caching. Therefore, the benchmark was modified for the purpose of *Conquest* measurement, so that each phase of the benchmark operates on a set of files before executing the next phase. In addition, all random accesses are aligned to reflect common application usage patterns (i.e., compilers). Detailed reasons for these modifications are explained in Chapter 8.

To test the effect of the large-file threshold, we conducted two experiments. For the first experiment, each phase of benchmark operated on eleven 1-Mbyte files stored in memory, but results were averaged over only the numbers collected from the last ten files to avoid warm-up effects at the beginning of each phase in the benchmark. To reset the memory states, the machine was rebooted when switching file systems.

For the second experiment, the file size was increased to 1.01 Mbytes to see the performance difference once files were switched from being stored in memory to stored on-disk. (Recall that *Conquest*, as tested, had a hard limit of 1 Mbyte to separate large files stored on disk and small files stored in RAM.) For large on-disk files, each phase of the benchmark was performed on forty 100-Mbyte files.

The 1-Mbyte Large-File Benchmark

The 1-Mbyte large-file benchmark measures the throughput of *Conquest*'s memory files (Figure 7.5a). Compared to *ramfs*, *Conquest* achieves 8% higher bandwidth in both random and sequential writes and 15% to 17% higher bandwidth in both random and sequential reads. Compared to disk-based file systems, *Conquest* demonstrates a 1900% speed improvement in sequential writes over *ext2*, 15% in sequential reads, 6700% in random writes, and 18% in random reads. *SGI XFS* and *reiserfs* perform either comparably to or slower than *ext2*.

These numbers lead to several interesting observations. First, once the disk content is cached, the read performance differs little between existing disk-based file systems and *ramfs*. Since the read performance for 1-Mbyte files is dominated by transferring bytes between the VFS data cache and user-level buffers, *Conquest*'s faster

read performance is mostly attributable to bypassing the VFS I/O buffer management, as opposed to having the in-memory metadata management.

It is also interesting to note that random memory writes and reads are faster than corresponding sequential accesses. The cause is cache hits: for 1-Mbyte memory accesses with a 256-Kbyte L2 cache size, random accesses could have up to a 25% chance of reusing the L2 cache content, disregarding the additional uses of L2 caching by the operating system and the microbenchmark itself. However, sequential accesses are guaranteed to miss in such a small cache.

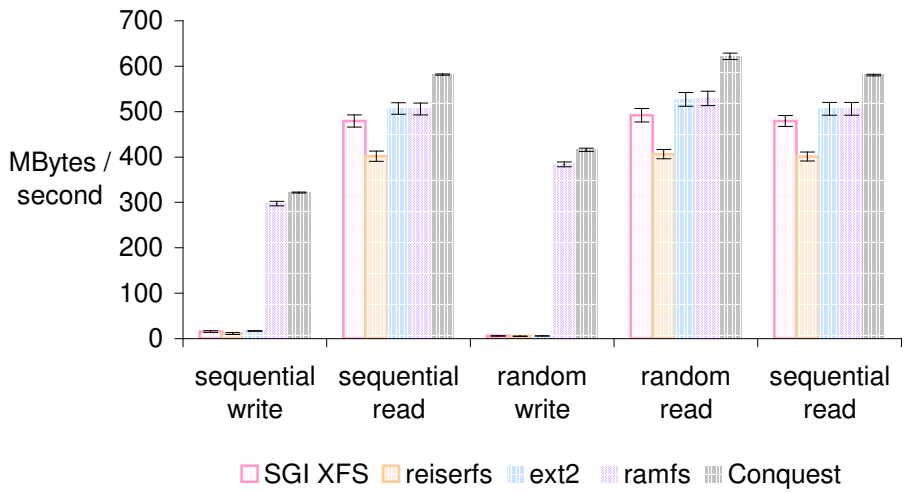
Another intriguing observation is that the performance difference between random and sequential writes is larger than the performance difference for corresponding reads. The cause can be traced to the management semantics of L2 caching. Chapter 8 is dedicated to exploring the memory behavior of the Sprite LFS large-file benchmark and providing detailed explanations of subtleties in these benchmark numbers.

The 1.01-Mbyte Large-File Benchmark

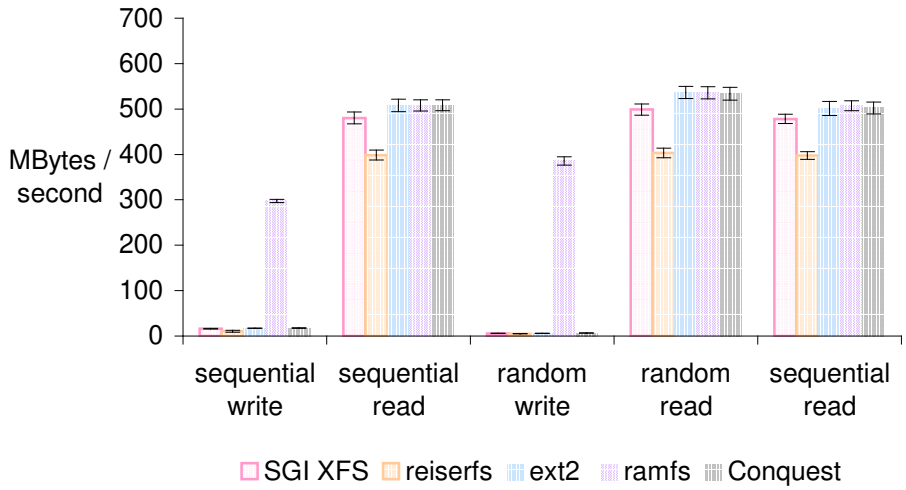
The 1.01-Mbyte large-file benchmark shows the performance effects of switching a file from being stored in memory to stored on-disk under *Conquest* (Figure 7.5b). For large files, since the overheads of metadata management and fragmentation management are negligible compared to the time to transfer bytes, *Conquest* benefits mostly in architectural simplification, only matching the performance of cached *ext2* for accessing large files. However, *Conquest* does show that the use of disjoint data paths for memory and disk imposes little or no extra overhead for disk accesses.

The 100-Mbyte Large-File Benchmark

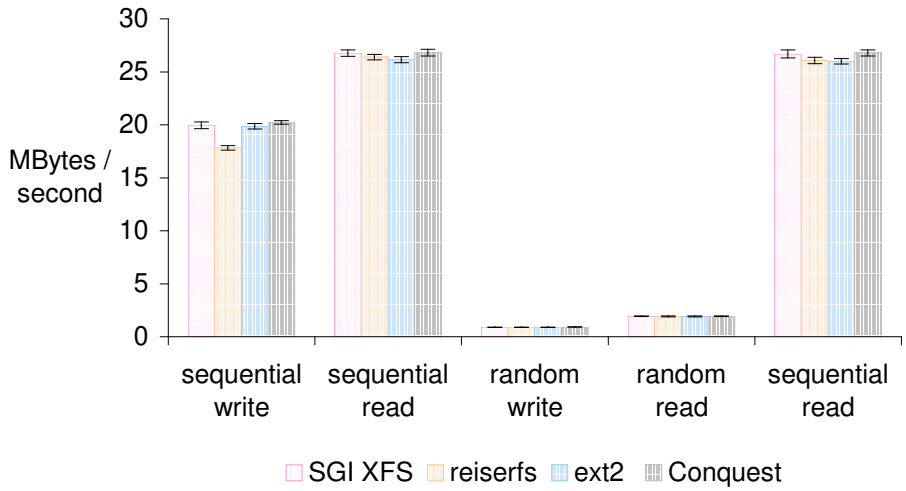
The 100-Mbyte large-file benchmark measures the throughput of *Conquest* on-disk files (Figure 7.5c). This experiment only compared *Conquest* against disk-based file systems because the total size exercised by the benchmark exceeds the capacity of *ramfs*. All file systems demonstrate a similar performance. However, *Conquest* currently uses 4 Kbytes as the access and management granularity for large files, which can be enlarged to 64 Kbytes or 256 Kbytes and aligned to track boundaries for further performance improvements [Schindler et al., 2002].



(a) Modified Sprite LFS large-file benchmark for ten 1-Mbyte files (stored in memory under *Conquest*).



(b) Modified Sprite LFS large-file benchmark for ten 1.01Mbyte files (stored on-disk under *Conquest*).



(c) Modified Sprite LFS large-file benchmark for forty 100-Mbyte files (stored on-disk under *Conquest*).

Figure 7.5: Bandwidth for the different phases of the modified Sprite LFS large-file benchmarks, run over *SGI XFS*, *reiserfs*, *ext2*, *ramfs, and Conquest.* These three experiments compare the large-file performance of memory and on-disk files under *Conquest*.

7.2.2 PostMark Macrobenchmark

The PostMark benchmark was designed to model the workload seen by Internet service providers [Katcher 1997]. Specifically, the workload simulates a combination of electronic mail, netnews, and web-based commerce transactions.

PostMark creates a set of files whose sizes are chosen at random and are uniformly distributed over a file size range. The files are then subjected to transactions consisting of a pairing of file creation or deletion with file read or append. Each pair of transactions is chosen randomly, with a bias controlled by parameter settings. A deletion operation removes a file from the active set. A read operation reads a randomly selected file in its entirety. An append operation opens a random file, seeks to the end of the file, and writes a random amount of data, without exceeding the maximum file size.

Early PostMark experiments used a configuration of 10,000 files with a size range of 512 bytes to 16 Kbytes. One run of this configuration performs 200,000 transactions with equal probability of creates and deletes, and a four times higher probability of performing reads than appends. The transaction block size is 512 bytes. However, since this workload is far smaller than the workload observed at any ISP today, we conducted experiments varying the total number of files from 5,000 to 30,000 to see the effects of scaling.

Another adjustment to the default settings was the assumption of a single flat directory. Since it is unusual to store 5,000 to 30,000 files in a single-level directory, the experiments were performed with PostMark configurated to use one subdirectory level to

distribute files uniformly, with the number of directories equal to the square root of the number of files. This setting ensures that each level has the same directory fanout.

Since all files within the specified size range will be stored in memory under *Conquest*, this benchmark did not exercise the disk aspect of the *Conquest* file system. Also, since this configuration specifies an average file set of only 250 Mbytes, which fits in 2 Gbytes of memory, this benchmark compared the memory performance of *Conquest* against the performance of existing cache and I/O buffering mechanisms under a realistic mix of file operations.

The measurement machine was rebooted when switching file systems. A shell first repeated six runs of the modified PostMark with the 5,000-file configuration, with the numbers collected from the first run dropped to reduce warm-up effects. Then, the script removed those 5,000 files, proceeded to the configuration of 10,000 files, and so on.

Figure 7.6 compares the transaction rates of *Conquest* with various file systems as the number of files varies from 5,000 to 30,000. First, the performance of *Conquest* differs little from *ramfs* performance. *Conquest*'s performance is satisfactory at this point, given that we still have room to reduce costs further (e.g., disabling the metadata caching in the VFS). *Conquest* outperforms *ext2* significantly; the performance gap widens from 24% to 350% as the number of files increases. *SGI XFS* and *reiserfs* perform much slower than *ext2* due to journaling overheads.

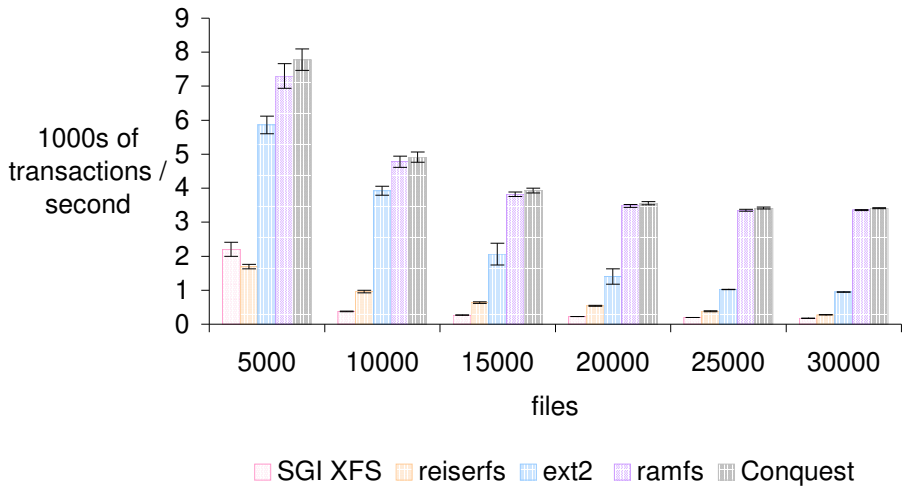


Figure 7.6: PostMark transaction rate for *SGI XFS*, *reiserfs*, *ext2, ramfs*, and *Conquest*, varying from 5,000 to 30,000 files. The results are averaged over five runs.

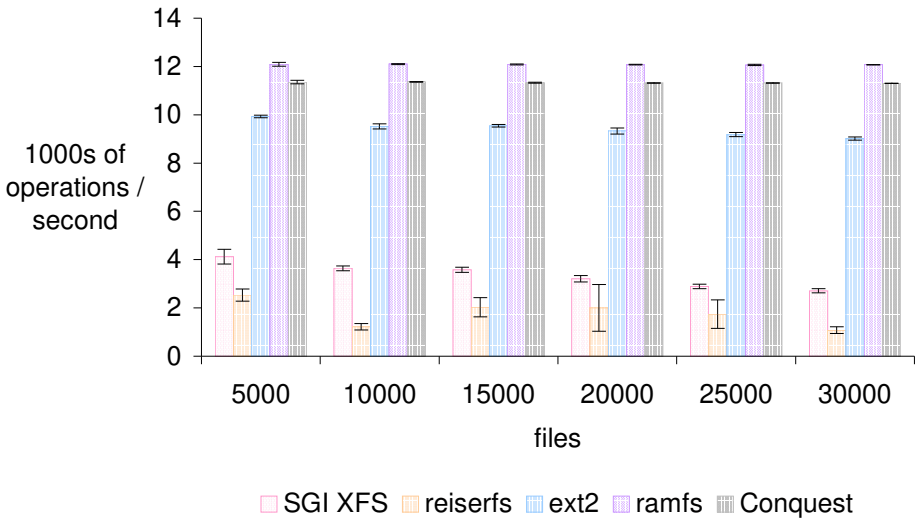
Conquest's ability to outperform disk caching in this memory-only workload leads to two conclusions. First, while a memory-only workload may seem to be an unfair comparison for disk-based file systems, the trend of increasing memory size actually makes disk caching the common workload. Second, since file system designers do not emphasize the memory performance aspect of a file system as much as disk performance, disk-caching designs fail to fully take advantage of memory bandwidth. The overhead due to caching-related management can lead to overall performance that is several times worse than *Conquest*.

Conquest outperforms disk caching due to two aspects of performance improvement: (1) simplified metadata manipulation, and (2) shorter data path. To see the contribution of each, Figures 7.7 and 7.8 present the performance numbers for only the file-creation and file-deletion operations. These figures show the performance of the

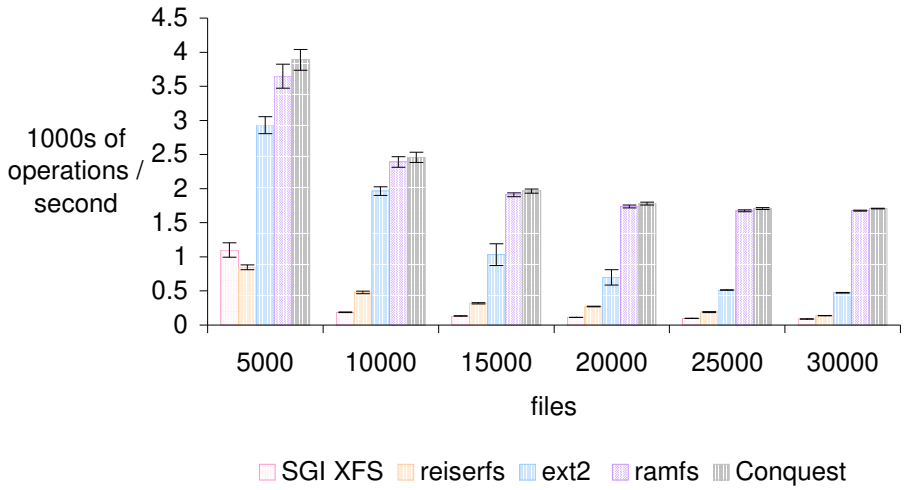
tested file systems, both when each operation is performed without the interference of other operations, and when the tested operation is mixed with other operations.

As one can see, when measured without other types of operations, file-creation and file-deletion rates show little degradation for all systems as the number of files increases (Figures 7.7a and 7.8a). When mixed with other types of file transactions (Figures 7.7b and 7.8b), file-creation and file-deletion rates degrade drastically for all file systems tested; the conclusion of this subsection offers some plausible explanations. Also, since the performance trends for mixed file creation and file deletion (Figures 7.7b and 7.8b) are similar to the trends for the throughput (Figure 7.9) and transaction rate (Figure 7.6), one can see that the effect of *Conquest*'s simplification of data paths (or read and write operations) has contributed more to outperforming disk caching than has in-memory metadata handling. This finding is surprising, since PostMark is known to be a metadata-intensive workload, but the read and write throughput dominates the performance behaviors.

For the pure file-creation and file-deletion numbers (Figures 7.7a and 7.8a), *Conquest* is about 6% to 9% slower than *ramfs* because *Conquest* has not disabled the VFS metadata caching due to reasons mentioned in Section 6.3. However, when mixed with other types of file transactions, *Conquest* can create and delete files at rates comparable to *ramfs* since PostMark numbers are largely dominated by the performance of reads and writes (Figures 7.7b and 7.8b).

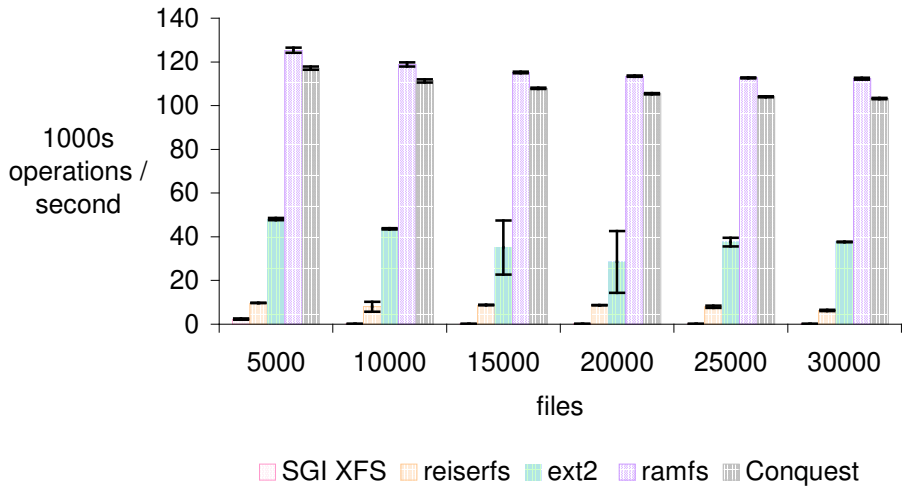


(a) PostMark file-creation rate, without the interference of other operations.

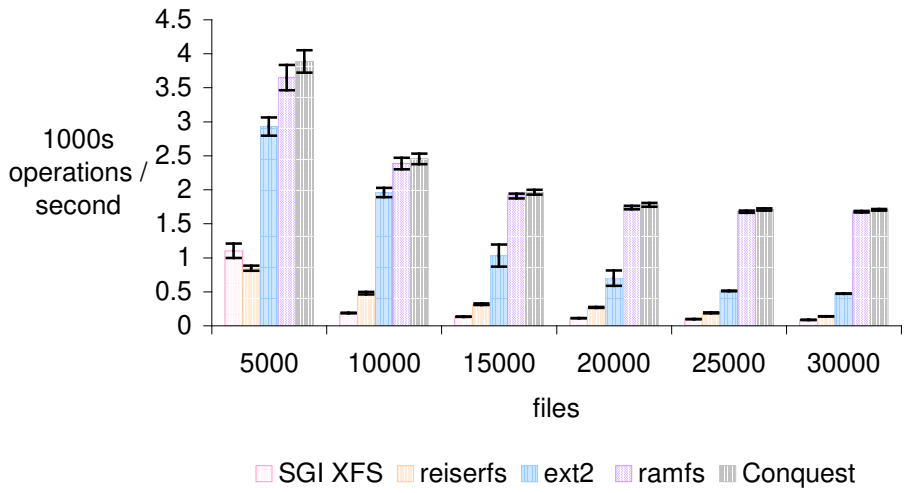


(b) PostMark file-creation rate, mixed with other operations.

Figure 7.7: PostMark file-creation performance for *SGI XFS*, *reiserfs*, *ext2*, *ramfs*, and *Conquest*, varying from 5,000 to 30,000 files. The results are averaged over five runs.



(a) PostMark file-deletion rate, without the interference of other operations.



(b) PostMark file-deletion rate, mixed with other operations.

Figure 7.8: PostMark file-deletion performance for *SGI XFS*, *reiserfs*, *ext2*, *ramfs*, and *Conquest*, varying from 5,000 to 30,000 files. The results are averaged over five runs.

Compared to *ext2*, *Conquest* performs at a 26% faster creation rate (Figure 7.7a), lower than the 50% faster rate in the Sprite LFS benchmark (Figure 7.4). *Ext2* has a better creation rate under PostMark than *Conquest* because the files being created are larger. The write buffer used by *ext2* narrows the performance difference of file creation

when compared to *Conquest*. For pure deletions, *Conquest* performs about twice as fast as *ext2*, which is consistent with the microbenchmark findings.

Similar to the comparison between *Conquest* and *ramfs*, it is interesting to see that *SGI XFS* has a faster file-creation rate than *reiserfs* without mixed traffic, but a slower rate with mixed traffic. This result demonstrates that optimizing individual operations in isolation does not necessarily yield better performance when they are mixed, especially when other operations dominate the performance characteristics.

The confidence intervals tend to be larger with fewer files (Figure 7.7b and Figure 7.8b) due to the lower accuracy when averaging over shorter elapsed times. Also, the confidence intervals are large at times for disk-based file systems. For *SGI XFS*, file creations may interact with the journaling mechanisms that use the disk heavily (Figure 7.7a). For *ext2*, file deletion interacts with recent asynchronously buffered writes.

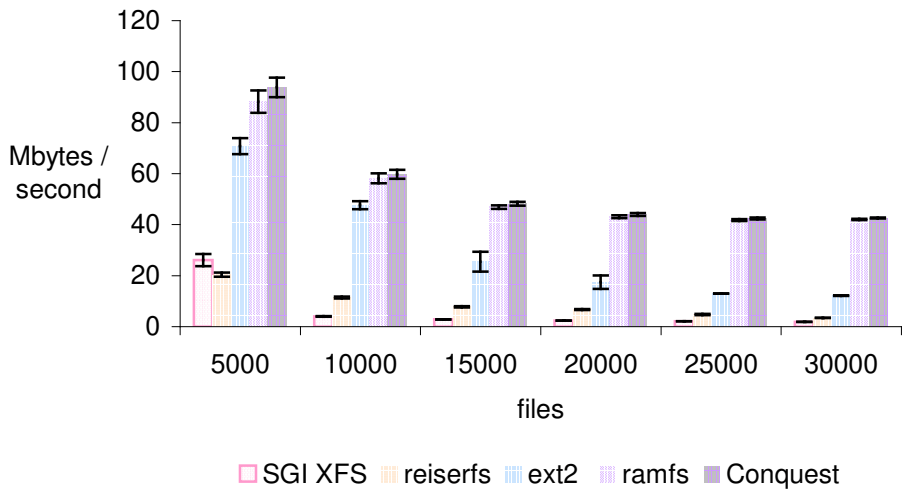


Figure 7.9: PostMark throughput for *SGI XFS*, *reiserfs*, *ext2*, *ramfs*, and *Conquest*, varying from 5,000 to 30,000 files. The results are averaged over five runs.

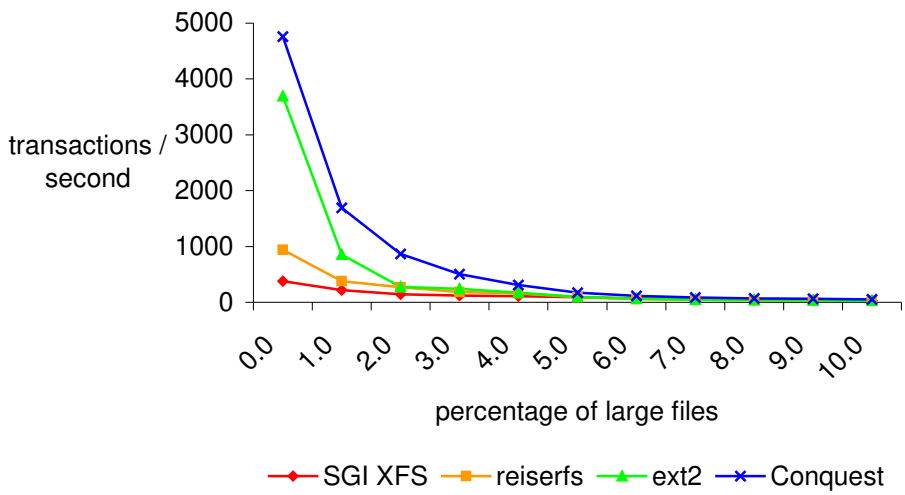
In terms of the PostMark throughput numbers (Figure 7.9), although the overall performance trend matches the trend for the transaction rate (Figure 7.6), Figure 7.9 shows a significant throughput loss when compared to the microbenchmark numbers (Figure 7.5a). This discrepancy may be caused by a combination of factors, including the mixture of creation and deletion operations, a larger number of files and total file size, and the interaction of the benchmark footprint with the L2 caching management. Since this performance loss is not specific to *Conquest*, a detailed investigation will be conducted in future work.

7.2.3 Modified Postmark Benchmark

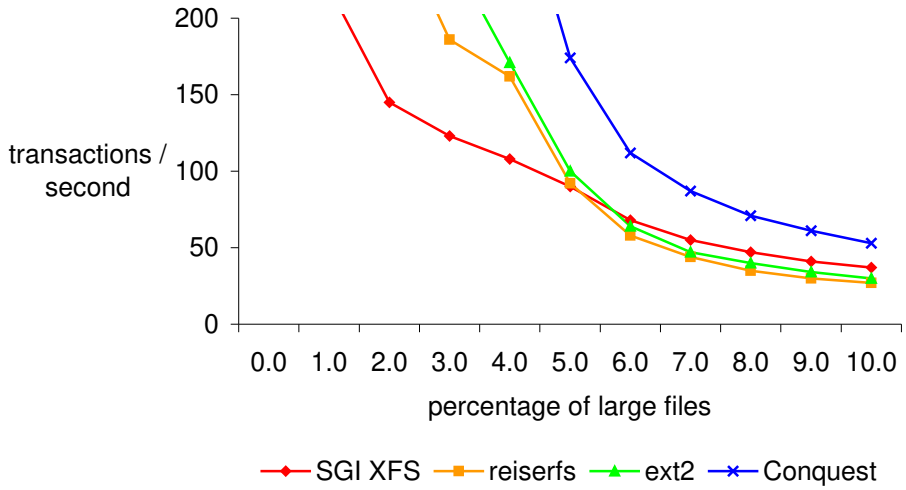
The PostMark benchmark was also modified to exercise both the memory and disk components of *Conquest*. The modified PostMark benchmark generates a percentage of files in a large-file category, with file sizes uniformly distributed between 2 Mbytes and 5 Mbytes. The mean of this size range is twice the mean video request size observed in a proxy traffic workload study [Mahanti et al., 2000], giving a more conservative picture of *Conquest*'s performance when both the memory and disk component are exercised. This setting is reflective of the growing number of multimedia files and their emerging file size distribution [Evans and Kuenning 2002]. The setting is also anticipatory given that large files grow over time, as indicated by various studies [Douceur and Bolosky 1999; Vogels 1999; Roselli et al., 2000; Evans and Kuenning 2002]. The remaining files were uniformly distributed between 512 bytes to 16 Kbytes. The total number of files was fixed at 10,000, and the percentage of large files varied from 0.0 to 10.0 (0 Gbytes to 3.5

Gbytes). Since the file set exceeds the storage capacity of *ramfs*, *ramfs* could not be included in the results.

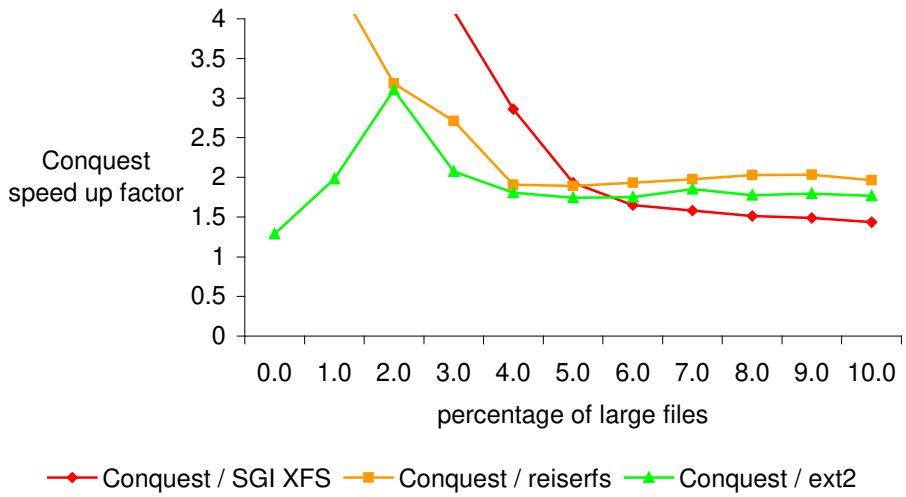
Figure 7.10 compares the transaction rate of *SGI XFS*, *reiserfs*, *ext2*, and *Conquest*. Figure 7.10a shows how the measured transaction rates of the four file systems vary as the percentage of large files increases. Because the scale of this graph obscures important detail at the right-hand side, Figure 7.10b zooms into the graph with an expanded vertical scale. Finally, Figure 7.10c shows the performance ratio of *Conquest* compared to other disk-based file systems.



(a) The full-scale graph.



(b) The zoomed graph.



(c) *Conquest* speedup curves for the full graph.

Figure 7.10: Modified PostMark transaction rate for *SGI XFS*, *reiserfs*, *ext2*, and *Conquest*, with varying percentages of large (on-disk *Conquest*) files ranging from 0.0 to 10.0 percent.

Conquest demonstrates 29% to 310% faster transfer rates than *ext2* (Figure 7.10c). The shape of the *Conquest* speedup curve over *ext2* reflects the rapid degradation of *ext2* performance with the injection of disk traffic. As more disk traffic is injected, we start to see a relatively steady performance ratio. At steady state, *Conquest* shows a 77%

faster transaction rate than *ext2*, 46% faster rate than *SGI XFS*, and 96% faster rate than *reiserfs*.

Both *SGI XFS* and *reiserfs* show significantly slower memory performance (left side of Figure 7.10a). However, as the file set exceeds the memory size, *SGI XFS* starts to outperform *ext2* and *reiserfs* (Figure 7.10c). Clearly, different file systems are optimized for different workloads.

7.3 Summary

Conquest demonstrates that it is feasible to simplify an existing system and to achieve better performance at the same time. For architectural simplification, the memory data path of *Conquest* contains no disk-related optimizations, and the disk data path avoids persistence management and fragmentation management and simplifies I/O buffer management. The code base of *Conquest* is at least 20% smaller than leading disk-based file systems.

Compared to disk caching, *Conquest* demonstrates about 50% faster speed in metadata handling and 19% faster speed in reads under the Sprite LFS small-file microbenchmark. Compared to *ramfs*, *Conquest* demonstrates 8% to 17% faster reads and writes under the Sprite LFS modified large-file microbenchmark.

With a memory-only PostMark workload, *Conquest* matches the performance of *ramfs* and outperforms disk caching by 24% to 350% in transaction rates. With a modified PostMark that exercises both its memory and disk components, *Conquest* still outperforms disk caching by 43% to 96%.

Conquest shows that existing disk-based file systems do not take full advantage of memory hardware, resulting in disk caching performance that is several times slower than *Conquest*. In a memory-rich computing era, designing a file system that treats memory performance as the common case for accesses can result in both architectural simplification and performance improvement.

Chapter 8

Effects of Memory and L2 Caching on File System Microbenchmarks

The performance of modern file systems is heavily dependent on multi-level caching, and minor tuning of caching mechanisms can have huge effects on the final results. In traditional benchmarking environments where disk is the primary storage, the major caching effect is of disk blocks at the memory level. However, as the memory size increases, more programs execute entirely from memory, with no disk activity. The amount of memory available has made caching of disk contents vastly more effective. However, one result of this change is that the effects of L2 caching are becoming increasing visible in the microbenchmarks and in real programs.

Currently, popular file system microbenchmarks are designed to measure the file system's disk performance. Exercising the disk with a sizable working set is still the dominant practice. A common assumption was that memory performance numbers could always be obtained from these microbenchmarks by simply shrinking the size of the working-set below the size of the physical memory. However, experiences from measuring *Conquest* reveal that microbenchmarks also need to consider L2 caching effects to provide true insight into file system performance.

Through the process of examining performance anomalies encountered when comparing memory-based and disk-based file systems, we demonstrate how the L2

caching policy, memory page alignment, the state of the L2 cache, and the sizes of the file system footprints can significantly affect file system microbenchmark numbers in non-intuitive ways. To the extent that file systems are designed or chosen for use based on the results of such benchmarks, these effects can be misleading. In some cases, these effects cause a 20% or greater deviation in the key metrics produced by these benchmarks.

8.1 Microbenchmark Anomalies

During the performance measurement phase of *Conquest*, many early microbenchmark numbers seemed inexplicable. Figure 8.1 shows the performance numbers from the original Sprite LFS microbenchmark with one 1-Mbyte file. These results raise questions about two major anomalies:

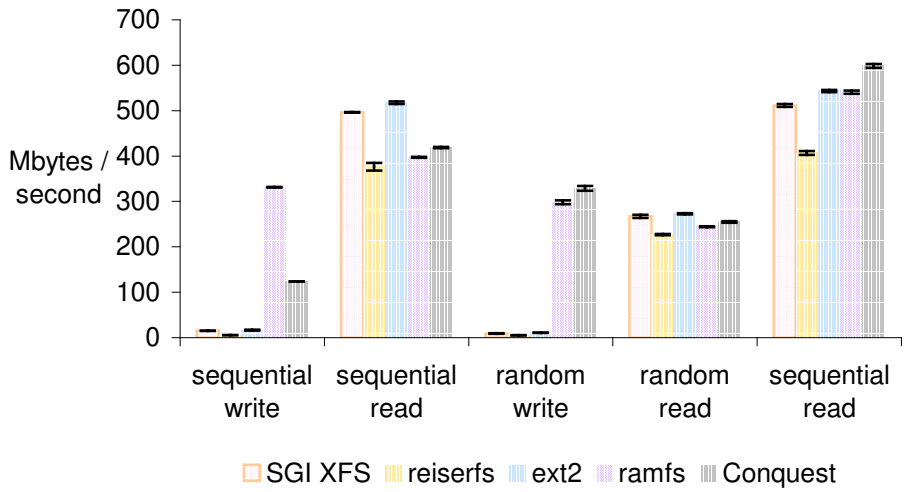


Figure 8.1: Original Sprite LFS large-file microbenchmark for one 1-Mbyte file.

1. ***Why are memory-based file systems slower than some of the disk-based file systems in sequential read performance?*** In particular, why are *Conquest* and *ramfs* considerably slower than *ext2* and *SGI XFS* in terms of the bandwidth for the first sequential read? This question is puzzling because the code bases for *Conquest* and *ramfs* are significantly smaller and simpler than those of *ext2* and *SGI XFS*. Smaller and simpler code should intuitively translate into better performance, and it does for many phases of the benchmark. In addition, both *ramfs* and *ext2* use the same generic code provided by the VFS to perform reads. The same data path should yield similar performance behaviors.
2. ***Why are random memory reads considerably slower than sequential memory reads?*** Conceptually, the access times for any random memory location should be relatively uniform, unless optimizations accelerate the sequential accesses. However, since data is moved in 4-Kbyte blocks, most bytes should be transferred sequentially.

For ease of presentation and discussion, the remaining chapter will investigate these two questions and their derivatives by comparing only *ramfs* and *ext2*. Measurement numbers for *Conquest*, *reiserfs*, and *SGI XFS* are omitted because the *ramfs* and *ext2* numbers capture the relevant differences between disk-based and memory-based

file systems. The other file systems demonstrate such sufficient similarities in performance characteristics that they reveal little new information.

8.2 Why is *ramfs* Slower Than *ext2*?

As shown in Figure 8.1, *ramfs* is slower than *ext2* during the first sequential read phase of the LFS benchmark. This section describes a series of hypotheses and the experiments that we performed to trace the cause of this memory-based file system running slower than the disk-based one.

8.2.1 Sequential Read Differences Between *ext2* and *ramfs*

Since both *ramfs* and *ext2* share the same generic code under the VFS, one would expect both file systems to achieve similar performance, but perhaps they reach the steady-state peak at different speeds. Therefore, an experiment was conducted to perform three sequential reads after a sequential write of a 1-Mbyte file. An `fsync` call was invoked at the end of the write phase.

Table 8.1

TEST ON REPEATED SEQUENTIAL READS.

	ext2	*ramfs*
sequential write	14 (±0.29) Mbytes/second	390 (±0.80) Mbytes/second
sequential read	520 (± 1.4) Mbytes/second	400 (±0.65) Mbytes/second
sequential read	530 (± 0.65) Mbytes/second	520 (±2.5) Mbytes/second
sequential read	530 (±2.6) Mbytes/second	530 (±0.73) Mbytes/second

Table 8.1 shows that a warm-up effect for sequential reads is evident for both file systems. In the steady state, *ramfs* achieves the same performance as *ext2*. This

somewhat confirms the intuition that the shared code base should be able to produce a similar performance, but there is still a significant warm-up effect to explain.

Why does ext2 warm up faster than ramfs? There are several hypotheses: (1) The default file system on the test machine is *ext2*; therefore, path resolutions (for various files and shared libraries) that involve *ext2* may cause contention for cache lines with *ramfs* during benchmarking. In a personal communication, Linus Torvalds [2001] also speculated that (2) *ramfs* may have cache layout problems, and that (3) *ramfs* may have a different warm-up behavior for sequential reads than *ext2* because of different data cache buffer use.

We attempted to examine various hypotheses using the Linux kernel profiling facility, but it yielded inconclusive results. Detailed profiling significantly distorted the collected numbers at the microsecond scale. The next experiment involved only sequential reads—namely, one sequential read on a pre-existing 1-Mbyte file, followed by another sequential read.

Table 8.2

TEST ON THE WARM-UP BEHAVIORS FOR SEQUENTIAL READS.

	ext2	*ramfs*
sequential read	480 (±0.98) Mbytes/second	480 (±1.3) Mbytes/second
sequential read	520 (± 2.7) Mbytes/second	520 (±0.0) Mbytes/second

Table 8.2 shows that both *ramfs* and *ext2* now have similar numbers for two consecutive sequential reads, suggesting that the three proposed hypotheses are unlikely candidates to explain the fast warm-up effects of *ext2*. Since both sets of numbers are statistically the same, even if external factors, such as the choice of default file systems,

the effects are likely to be trivial. If *ramfs* has cache layout problems, its performance here should be noticeably slower than *ext2*. Also, both file systems warm up at similar rates, so different ways of using the memory data cache do little to explain the anomaly.

8.2.2 Were the Initial States of the Microbenchmark Bad?

As a sanity check, we took a number of measurements to ensure the same initial states for various experiments. Benchmarking machines were rebooted after each experimental run. However, the booting process might still lead to different caching states. Therefore, the original Sprite LFS large-file microbenchmark was modified to perform a sequential read on a pre-existing 1-Mbyte file followed by a sequential write (again, with `fsync`) and a sequential read on the same file. This precaution should ensure that all experiments started with the same state after performing the first sequential read.

Table 8.3

INITIAL CONDITION TEST.

	ext2	*ramfs*
sequential read	480 (±1.5) Mbytes/second	480 (±1.4) Mbytes/second
sequential write	14 (± 0.47) Mbytes/second	400 (±1.2) Mbytes/second
sequential read	520 (± 0.73) Mbytes/second	400 (±1.3) Mbytes/second

Table 8.3 shows that both *ext2* and *ramfs* have similar performance for the first sequential read, confirming that both sets of experiments have similar initial states. Therefore, the cause of the anomaly is likely to be the sequential write. However, ***why would the same sequential write operation cause the subsequent sequential read to speed up by 40 Mbytes/second for ext2, while slowing it down by 80 Mbytes/second for***

ramfs? Is it possible that a sequential write somehow warms up the L2 cache locality for *ext2* and destroys the locality for *ramfs* for the subsequent sequential read?

8.2.3 Does L2 Caching Buffer Management Cause the Anomaly?

Linux's memory manager provides high-speed memory allocation by writing back dirty memory pages to disk speculatively in several ways. Notably, if the memory is being allocated at a rapid rate, the memory write-back will be performed at a higher rate to anticipate near-term allocations. Also, whenever the allocated memory exceeds a certain threshold, the memory manager will also start writing back dirty pages to avoid an exhaustive search for allocations.

L2 caching tries to handle bursty allocations in a different manner. The Pentium® III has a streaming I/O option to bypassing L2 caching for large sequential accesses [Keshava and Penkovski 1999]. Should the high I/O bandwidth of *ramfs* trigger the streaming I/O mechanism, it would leave the L2 cache with little file content available for reuse during the subsequent read operation.

Therefore, we designed an experiment to test the effect of write request rates on the triggering of any streaming I/O mechanism. For the sequential write, a `usleep` call was inserted to sleep one microsecond within each iteration of write call, so the write request rate was significantly slowed down to a similar speed for both *ramfs* and *ext2*. The `fsync` call is still invoked at the end of the write phase. If either the cache buffer management or the streaming I/O mechanism caused this performance anomaly, we should then see sequential read results significantly different from the last row of Table 8.3.

Table 8.4

L2 CACHING POLICY TEST.

	ext2	*ramfs*
sequential write (with `usleep`)	200 (±0.32) Kbytes/second	200 (±0.40) Kbytes/second
sequential read	510 (± 1.9) Mbytes/second	410 (±5.5) Mbytes/second

Unfortunately, Table 8.4 shows that the rates of write operations have little effect on the sequential read bandwidth, disproving hypotheses involving special sequential hardware logic as the explanation for the anomaly.

8.2.4 Do Random Writes Also Cause the Anomaly?

Results obtained from the slow sequential write experiment raised the suspicion that the sequential nature of a write may contribute little to this anomaly. Therefore, an experiment was conducted to test a random write (with `fsync`) followed by a sequential read.

Table 8.5

TEST ON EFFECTS OF RANDOM WRITES.

	ext2	*ramfs*
random write	9.2 (±0.27) Mbytes/second	280 (±2.3) Mbytes/second
sequential read	510 (± 1.4) Mbytes/second	400 (±1.4) Mbytes/second

Table 8.5 shows that a random write operation can also cause abnormal behavior in the subsequent sequential read. Therefore, any write access can cause a subsequent read to achieve higher bandwidth for *ext2* and lower bandwidth for *ramfs*, pointing toward a completely different hypothesis.

8.2.5 Forming a New Hypothesis

One puzzling aspect of this performance anomaly was the inverse association between the size of the file system memory footprint and the performance. From the code base, the cache footprint of *ramfs* was significantly smaller than *ext2*. But, ***why would a smaller cache footprint result in worse performance?***

Given that the Sprite LFS large-file microbenchmark measures the bandwidth of read and write operations, the entire benchmark should be data-intensive, and the timing should be strongly correlated to the amount of data being transferred into and out of the L2 cache. Since microbenchmarking experiments revealed only the black-box behavior of the system, a simple and naïve model of the L2 cache was used to visualize the underlying mechanisms within the black box.

Suppose an L2 cache has four cache lines, and the first two phases of the microbenchmark operate on a file with a size of 16 file segments, with each segment occupying one cache line. For the scenario of a large cache footprint, the file system and the benchmark being run have an active footprint of three out of four cache lines (Figure 8.2). This simple model assumes that the model starts with only the active cache footprint in the cache. The model also assumes that the effects of cache collisions are minimal, and that cache collisions will never evict the cache lines containing the active footprint.

During the sequential write phase, the first write can directly modify the clean cache line, and all subsequent writes require purging the dirty cache content before modification. During the sequential read phase, the first read requires purging the dirty

cache content, because the previous write operation leaves the end of the file in the last cache line, and the read cannot reuse the content. However, all subsequent reads can simply overwrite the previously read-in cache line from memory, since those lines are not modified during the read operation, and thus require no eviction for committing changes.

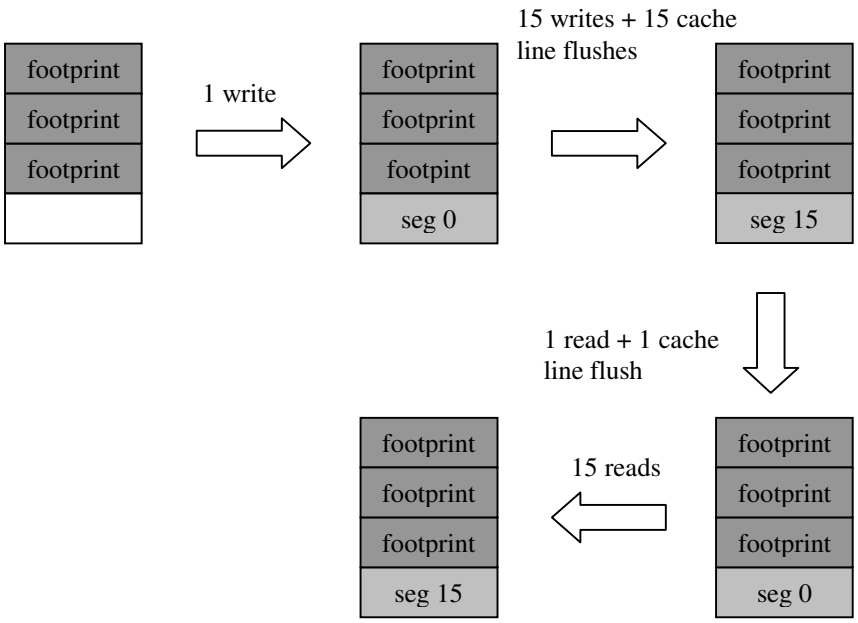


Figure 8.2: The model for a large cache footprint. The first two phases of the Sprite LFS large-file benchmark are analyzed with a simple L2 cache with four cache lines. The active footprint of the file system and benchmark being tested consists of three out of four cache lines, and the file being tested has a size of 16 segments, with each segment occupying one cache line.

For the scenario of a small cache footprint, the active footprint is decreased to one cache line (Figure 8.3). During the sequential write phase, the first three writes can directly modify the clean cache lines, and 13 subsequent writes require purging dirty cache content. During the sequential read phase, the first three reads need to purge the dirty cache content, since the cache stores the last three cache lines of the file being

benchmarked, and the read starts from the beginning. The other 13 reads can simply overwrite the previously read-in cache lines from memory.

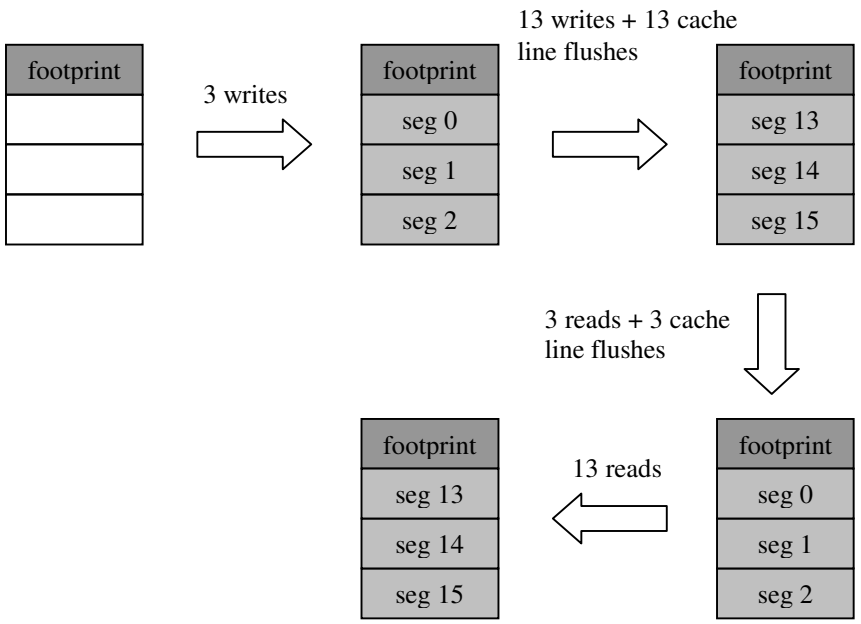


Figure 8.3: The model for a small cache footprint. The first two phases of the Sprite LFS large-file benchmark are analyzed with a simple L2 cache with four cache lines. The active footprint of the file system and benchmark being tested consists of one cache line, and the file being tested has a size of 16 segments, with each segment occupying one cache line.

Table 8.6

SUMMARY OF THE FIRST TWO PHASES OF THE SIMPLE CACHE MODEL

	Large cache footprint	*Small cache footprint*
sequential write	16 writes, 15 flushes	16 writes, 13 flushes
sequential read	16 reads, 1 flushes	16 reads, 3 flushes

Table 8.6 summarizes the differences between the two scenarios. Although simple, this model shows that it is possible for a smaller cache footprint to result in a slower sequential read after a sequential write. A smaller cache footprint leaves more room to cache dirty data during the sequential write phase, and the sequential read phase

needs to finish committing previous dirty cache content before reading new information into cache. In essence, the file system with the larger footprint has already paid nearly the complete price for cache flushes during its write phase, while the file system with the smaller footprint has used the cache to defer some of the costs until later.

This hypothesis also explains why the anomaly of the sequential read also occurs after a random write, since a smaller cache footprint leaves more room to cache dirty data for any write pattern.

8.2.6 Verifying the New Hypothesis

The new hypothesis was verified through low-level instrumentation, high-level microbenchmarking, and reproducing the same abnormal behavior at the user level.

8.2.6.1 Performance-Counter Instrumentation

Our first measurement was of the amount of data being transferred into and out of the L2 cache. The instrumentation tool used was Rabbit [Heller 2000], a performance counters library for Intel processors running Linux. The package allowed monitoring a few registers with little or no disturbance to the overall performance numbers, even for microbenchmarks. The instrumentation monitored CPU registers that contain information regarding the movement of data for the L2 cache. Collected numbers include the total number of events and elapsed times for computing the volume of data being moved.

At the beginning of a sequential read after a write operation, *ramfs* has to move 230 Kbytes of modified data out of the 256-Kbyte L2 cache, while *ext2* moves only 66

Kbytes of modified data from the L2 cache. Given that a write operation precedes the read operation, the moved bytes are likely to represent the recently written part of the 1-Mbyte file. Also, given that *ramfs* has a smaller cache footprint than *ext2*, the L2 cache can hold more dirty data when running *ramfs* than *ext2*. Since both file systems need to read in the 1 Mbyte of data after moving the dirty data out of the cache, the discrepancy in data volume being moved explains why *ext2* can achieve higher read bandwidth than *ramfs*.

8.2.6.2 Microbenchmark Behaviors

At the microbenchmark level, our first experiment toward proving this new hypothesis consisted of a few sequential write operations (with `fsync`). The expectation was that the first sequential write would affect the performance of the second sequential write. Also, starting from the second sequential write, the bandwidth numbers should not fluctuate as much.

Table 8.7

TEST ON REPEATED SEQUENTIAL WRITES.

	ext2	*ramfs*
sequential write	13 (±0.61) Mbytes/second	380 (±1.4) Mbytes/second
sequential write	16 (± 0.32) Mbytes/second	350 (±0.80) Mbytes/second
sequential write	16 (± 0.32) Mbytes/second	350 (±0.65) Mbytes/second

Table 8.7 shows that the second sequential write for *ext2* has increased bandwidth (probably due to the caching of *ext2* file system code), while the second sequential write for *ramfs* has reduced bandwidth (probably because of the need to purge many dirty cache lines from the previous sequential write).

The original plan was to modify *ramfs* to use a larger L2 footprint, but altering the source code would make the test method less portable and repeatable for future investigations. Instead, a 128-Kbyte file was used to see if *ramfs* could outperform *ext2* when both the *ramfs* footprint and the file being tested can fit into the 256-Kbyte L2 cache. (Note that since *ext2* flushed 66 Kbytes of data in the previous experiment, the expectation was that the 128-Kbyte file and the *ext2* footprint would not fit into the 256-Kbyte L2 cache.) The experiment consisted of a sequential write (with `fsync`), followed by two sequential reads.

Table 8.8 shows that when *ramfs* can read recently written content completely from the L2 cache, it can significantly outperform *ext2*.

Table 8.8

L2 CACHE TEST

	ext2	*ramfs*
sequential write	4.9 (±0.43) Mbytes/second	600 (±0.61) Mbytes/second
sequential read	440 (± 5.7) Mbytes/second	810 (±25) Mbytes/second
sequential read	1200 (± 35) Mbytes/second	1400 (±52) Mbytes/second

8.2.6.3 Testing a User-Level Program

To extend the applicability of the findings on microbenchmark anomalies, we created a trivial array-indexing program to reproduce the write-back effect of L2 caching. The user program walks through an integer array of 1 Mbyte, assigning each array entry a different integer, followed by two loops that retrieve the value of each array entry to a variable. Although they were designed to exercise the same behavior as the benchmark, these types of operations are reasonably realistic in various applications.

Table 8.9 shows an anomaly similar to that in the microbenchmark, verifying that the effects of L2 caching are easily visible in places other than simple file system microbenchmarks.

Table 8.9

A USER PROGRAM TEST.

	User program
sequential write	220 (±0.32) Mbytes/second
sequential read	150 (± 31) Mbytes/second
sequential read	200 (± 1.3) Mbytes/second

However, these numbers have peculiarities of their own. These bandwidth numbers appear to be low compared to Table 8.1, and the sequential-write bandwidth is slightly higher than the sequential-read bandwidth. Since investigation of these peculiarities digresses from the purpose of this dissertation, it is left as future work. However, this user program does show the difficulty of designing microbenchmarks that actually match real program behavior.

8.2.6.4 Lessons Learned

The numbers obtained from these three levels of experiments demonstrate several lessons:

- Intuitively, a smaller cache footprint should yield faster performance, but under certain workloads, a smaller footprint can also increase the effects of a write-back cache policy, causing the performance of a subsequent phase of a benchmark to swing in either direction.

- The effects of memory and L2 caching are easily visible at the application level, beyond file system microbenchmarks.
- To measure the memory performance of file systems with microbenchmarks, we have to consider the L2 cache states at each phase of measurement.

Incorporating the states of the L2 cache into the design of microbenchmarks can introduce additional complexity, since the cache size, cache line eviction policy, and n-way associativity are different from machine to machine. Also, microbenchmarks need to consider whether the dependencies between adjacent file system operations are intended and reflective of actual access patterns.

The Sprite LFS large-file microbenchmark was modified to incorporate these lessons: Each phase operates on 11 files in a fixed order, and the numbers are averaged across the last ten files. The numbers collected from the first file are dropped to reduce the warm-up effect of L2 caching on final results and to collect the steady-state numbers for each phase of the microbenchmark.

Figure 8.4 presents the results of this newer version of the microbenchmark for *ext2* and *ramfs*. The anomaly of *ramfs* being slower than *ext2* has disappeared. Both *ext2* and *ramfs* performed equally well on reads since the file content was now read from the memory buffer. However, another anomaly still exists: namely, sequential memory reads are much faster than random memory reads.

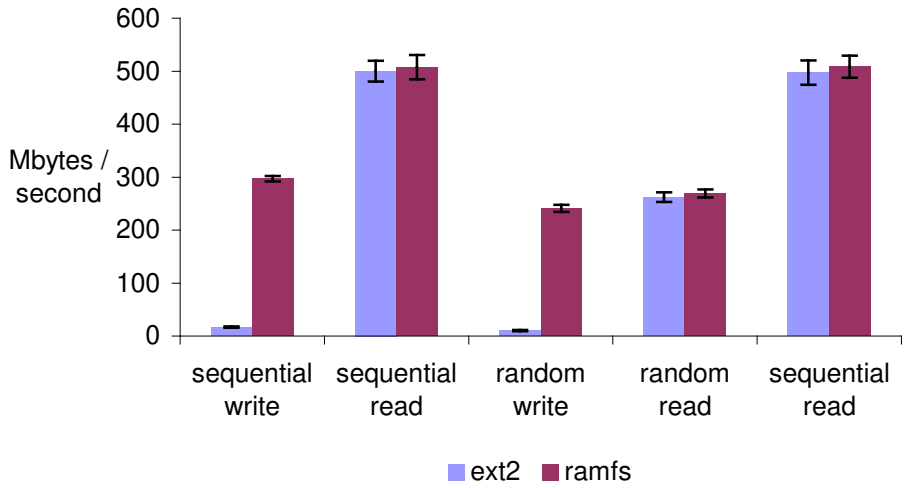


Figure 8.4: Sprite LFS large-file microbenchmark, with file size equal to 1 Mbyte, and each phase repeated ten times.

8.3 Why are Random Memory Accesses Slower Than Sequential Accesses?

The performance of random memory accesses could differ from that of sequential memory accesses for a few reasons. First, sequential accesses might be accelerated by either hardware or software sequential prefetching logic. Second, each random file access involves an extra seek system call, incurring extra kernel-crossing overhead. Third, the memory hardware might have a faster sequential access mode. Since the upper bits of memory addresses usually stay the same for sequential accesses, memory accesses might not need to wait for the upper bits of a memory address to stabilize before accessing the memory content.

Unfortunately, none of these three hypotheses plausibly explain the performance gap encountered. First, since both random and sequential accesses are performed at the granularity of 4-Kbyte blocks in the benchmark, most bytes are transferred sequentially,

leaving a minimal effect on random and sequential read performance due to sequential logic or memory hardware.

In addition, given that sequential reads achieve about 500 Mbytes/second for a 1-Mbyte file, the elapsed time to perform a sequential read is about 2 milliseconds; a random read is 4 milliseconds. Unless the overhead of locating a random memory block or making an extra system call can cost 2 milliseconds, this hypothesis cannot explain why random memory reads are so slow.

Therefore, we once again resorted to the use of the Rabbit performance-counter library to monitor CPU registers during the microbenchmark. Special attention was paid to corresponding *ext2* and *ramfs* numbers that differed by more than a factor of two. The numbers revealed that random memory accesses incurred many misaligned memory accesses, while sequential memory accesses incurred none.

An inspection of the microbenchmark source code revealed that the random memory accesses were indeed unaligned.[2] Therefore, such accesses usually required fetching data from two memory words, while the same-sized access in the sequential mode fetched data from only one word. Thus, each memory reference by the random read phase made two memory accesses most of the time. This effect caused the random memory-read bandwidth to be half of the sequential read bandwidth. Many applications that perform so-called *random* file reads (e.g., linkers) do so on word-aligned or page-aligned boundaries, so the benchmark is not necessarily indicative of real-world performance.

With knowledge of this anomaly, the microbenchmark was modified to perform page-aligned random reads. Figure 8.5 shows a newer version of the data for the Sprite LFS large-file microbenchmark with aligned random memory reads.

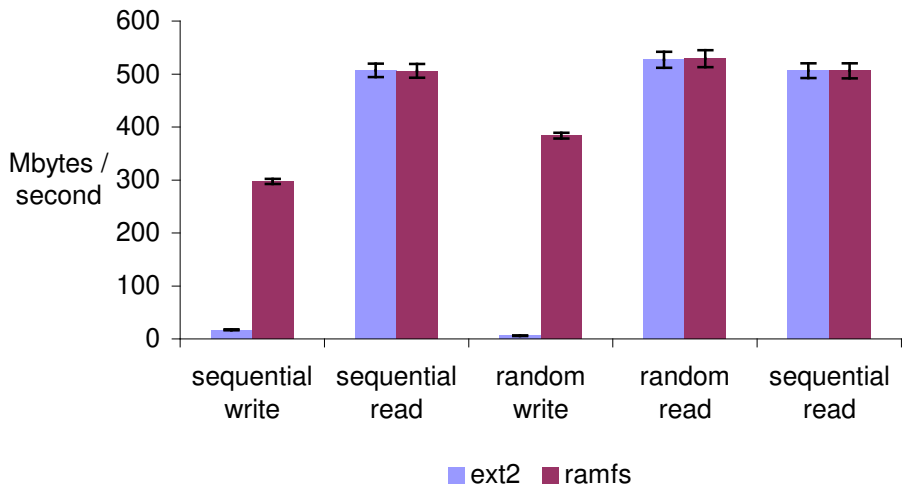


Figure 8.5: Sprite LFS large-file microbenchmark, with file size equal to 1 Mbyte, each phase repeated ten times, and memory-aligned random memory reads.

However, Figure 8.5 shows yet more performance glitches to explain. ***Why are random memory accesses now faster than sequential memory accesses?*** Also, why is the performance difference between random and sequential memory writes larger than that between corresponding reads?

8.4 Why are Random Memory Accesses Faster Than Sequential Accesses?

The performance differences between sequential and random memory performance can be explained through probability analysis. Caching works because of the possibility of

[2] This dissertation used a version of the Sprite LFS benchmark downloaded in the spring of 2001. A later version of the benchmark performs random accesses on page-aligned boundaries, so apparently this benchmark problem was independently discovered and fixed in the interim.

reusing cache content. When a file is larger than the cache size, sequential accesses provide no opportunity for cache reuse before a given line is replaced, while random accesses will sometimes hit in the cache.

The performance difference between a random and a sequential write is greater than the performance gap for corresponding reads because of the relative cost of cache misses. When reading a 1-Mbyte file from memory through a 256-Kbyte L2 cache, initial read misses might involve some purging of dirty L2 cache content to memory. Once the L2 cache is filled with read-only content, subsequent read misses will likely involve reading the file content into the cache directly, with few dirty cache lines to evict. One the other hand, a write miss (occurring when accessing a 1-Mbyte file from memory through a 256-Kbyte L2 cache) is very likely to involve evicting a dirty cache line, reading in the target cache content, and modifying the cache line. For a write operation, since the disadvantage of a cache miss is greater, the relative advantage of a hit is also greater. Therefore, the performance difference between random and sequential writes is greater than that of reads.

A microbenchmark that operated on 128-Kbyte files was used to verify the above hypothesis. One would expect no performance differences among file operations for the following reasons: (1) both sequential or random write are less likely to flush the dirty L2 cache content because the file size is small compared to the size of the L2 cache; (2) since an `fsync` call is invoked at the end of each file close, and dirty L2 cache content is flushed, the subsequent write operation does not need to commit the dirty data for the previous write; and (3) without the need to flush existing dirty L2 cache content, the cost

of write misses should be similar to the cost of read misses, assuming that data migration is the dominating performance cost.

With the expectation of uniform performance across all file operations, Figure 8.6 once again shows surprising results: random accesses under *ramfs* are 20% slower than sequential accesses.

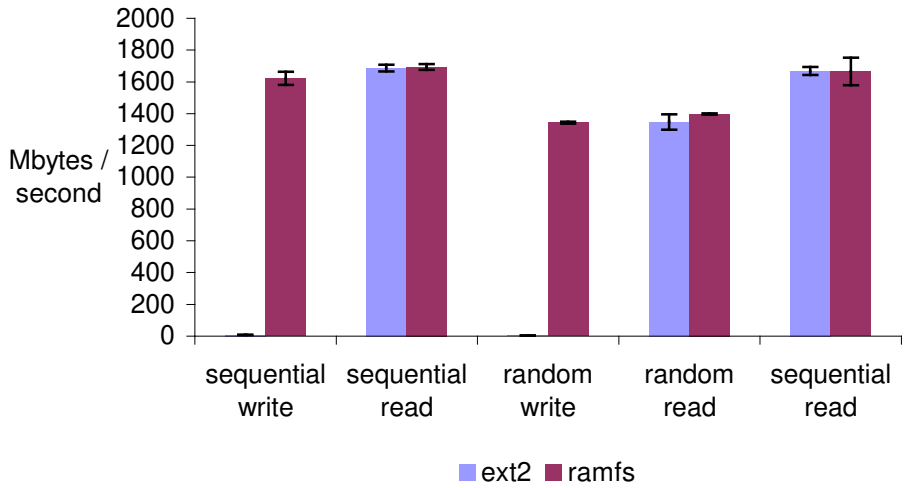


Figure 8.6: Sprite LFS large-file microbenchmark, with file size equal to 128 Kbytes, each phase repeated ten times, and memory-aligned random memory reads.

Based on the differences between achieved bandwidths, the computed difference in elapsed time is about 150 microseconds. The extra `lseek` system call and the random number generation call may account for this timing difference. To test this explanation, for each sequential operation an iteration of a file access was padded with dummy `lseek` calls and dummy random number generation calls.

Figure 8.7 finally shows the anticipated results, with uniform memory access times for all file accesses under *ramfs*. However, this experiment points out that with the speed of the L2 cache on modern machines, the overhead of a system call and crossing

the kernel boundary can potentially degrade file system throughput by 20%. This effect shows that performance numbers are very sensitive to minute details in the design of microbenchmarks and experiments. The effect should thus be considered in the design of data-intensive applications.

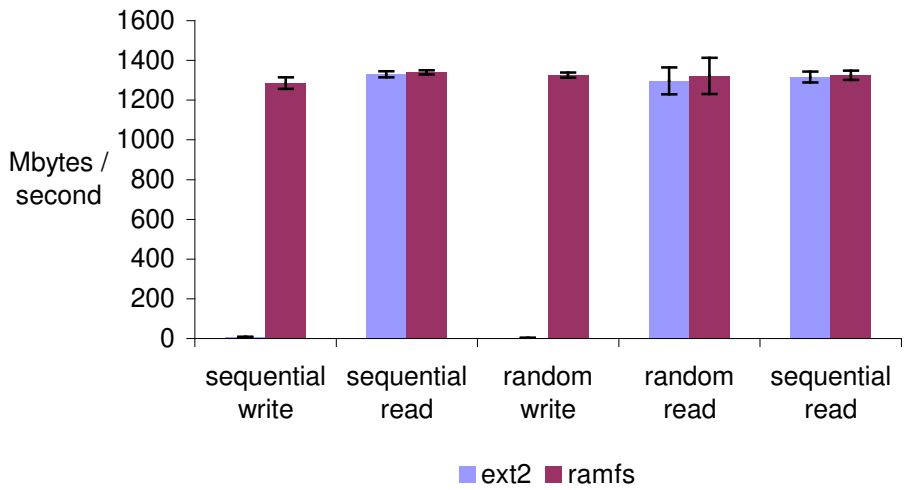


Figure 8.7: Sprite LFS large-file microbenchmark, with file size equal to 128 Kbytes, each phase repeated ten times, and memory-aligned random memory reads. Sequential accesses are augmented with dummy `lseek` and random number generation calls.

8.5 Wider Implications of Memory and L2 Caching Behaviors

The discovery of memory and L2 caching effects on benchmarking demonstrates one of the many discrepancies between memory-rich and memory-scarce computing environments, and these effects may have far-reaching implications ranging from low-level hardware to high-level application designs. At the hardware level, designers should consider the semantic implications of performance behaviors for individual operations as well as improving the performance for groups of operations. Otherwise, the aggregate performance behavior across various software and hardware layers can be very difficult

to interpret. File system designers also need to become increasingly aware of the common case of memory accesses and focus more on the memory aspect of file system design. Performance abnormities that could previously be disguised by slow disk should now be properly identified and handled under memory-only workloads. At the highest level, since many applications show phases of behavior similar to these benchmarks, designers can take advantage of such knowledge to improve the performance behavior and predictability of these applications.

Chapter 9

Related Work

Caching in volatile RAM has inspired *Conquest* to a large extent, and the relative strengths and weaknesses of caching have already been discussed in Section 2.1. Various disk-related optimization techniques have also provided guidance in designing *Conquest* (Section 9.1).

The idea of combining memory-based and disk-based storage service can be traced back to main-memory databases (Section 9.2). Researchers then began to use persistent RAM in the file systems arena (Section 9.3). PDA operating systems represent a major step, providing both memory and file system services in persistent RAM (Section 9.4); however, *Conquest* assumes an abundance of persistent RAM, which is scarce on those handheld devices. The design philosophy of *Conquest* is also confirmed by existing systems that share similar characteristics (Sections 9.5 and 9.6).

9.1 Disk Optimization Techniques

Disk optimization approaches directly address various components of the disk access time equation: seek time, rotational latency, and transfer time.

9.1.1 Seek-Time Reduction

The Berkeley fast file system (FFS) adopted the concept of spatial locality, which collocates related files in adjacent access regions on the disk [McKusick et al., 1984]. At the time of development, FFS accomplished a factor of 4 to 15 performance improvement

over the original UNIX file system, and was CPU-bound. But with modern CPU-speed, where I/Os are disk-bound, FFS can achieve less than 10% of available raw disk bandwidth.

A log-structured file system reduces disk seeks by appending all new updates to disk in a sequential structure called the *log* [Rosenblum and Ousterhout 1991]. The assumptions are that a large enough memory for caching essentially eliminates most of the read traffic, and thus disk access is dominated by writes. In addition, since the majority of read accesses are to recently written data, appending write accesses in a sequential fashion should minimize disk seeks and translate into good performance. Although LFS can achieve an order-of-magnitude speedup, it requires cost and complexity in recreating contiguous empty space. A heavily loaded system may not have enough spare CPU cycles to perform the disk reorganization [Matthews et al., 1997]. In addition, a recent study of user workloads discovered that although a small cache can sharply reduce the disk read traffic, disk traffic is not dominated by writes when large caches are deployed [Roselli et al., 2000]. Although results of Roselli et al., are dependent on the workloads and the write delay, the effectiveness of LFS disk layout under daily workload needs further investigation.

Another way to reduce seek time is through buffered writes, which allow operating systems to reorder disk accesses to minimize seek distances [Silberschatz et al., 1992]. The tradeoff is the risk of data loss in the event of system failures. For critical data and metadata, immediate updates are often required.

9.1.2 Rotational Latency Reduction

Data duplication can reduce the rotational latency for read accesses. By duplicating data on separate disks, the rotational delay for read accesses is reduced by a factor proportional to the number of replicas. However, duplication comes with the complexity of maintaining replicated data in consistent states, and if used aggressively, this method greatly reduces the storage capacity of the disks [Bitton and Gray 1988].

9.1.3 Transfer-Time Reduction

The transfer time of a disk can be improved through parallelism. A redundant array of inexpensive disks with level-5 replication (RAID5) allows a data object to be stored piece-wise across disks [Patterson et al., 1988]. When accessed, all disks can transfer the data in parallel. However, as the number of disks increases, the chance of having disk failures also increases. Reliability is thus integrated as part of the RAID5 design to store enough redundant information to protect against a single disk failiure.

Another way to improve disk transfer time is to tailor the storage system to a narrow range of applications. The video-on-demand (VoD) file system is one popular example [Haskin and Schmuck 1996; Bolosky et al., 1997]. VoD systems typically achieve high transfer rates by using RAID. Also, they use large-grained data structures to avoid processing and storage overhead.

9.1.4 Disk Optimization Techniques and *Conquest*

Disk optimizations have grown over the years merely to accommodate the widening performance gap between the CPU and disk. However, as the number of optimizations

grows, it becomes more difficult to identify the effectiveness of individual optimizations. The disk data path for *Conquest* is currently not optimized due to the lack of such knowledge. Future studies are required to identify a minimum subset of optimizations to handle large-file disk storage. Nevertheless, *Conquest*'s memory data path is freed from the complexity of disk-related optimizations.

9.2 Main-Memory Databases

The database community has a long-established history of main-memory database systems (MMDBs). An early survey paper reveals key architectural implications of abundant RAM [Garcia-Molina and Salem 1992]. One file-system-related observation is that since memory is much faster than disk, each transaction is completed within a shorter time; therefore, the probability of locking contention is smaller. A larger locking granularity in a MMDB can reduce the locking overhead [Lehman and Carey 1987] and the complexity of the system. A reduced probability of waiting for locks also translates into fewer context switches and resulting cache flushes, further improving the overall system performance.

The memory data path of *Conquest* is currently not optimized in terms of locking granularity, cache layout, and data structure. However, *Conquest* has already experienced better performance than existing disk caching. Further tuning of *Conquest*'s performance will be future work.

Garcia-Molina and Salem also discussed several early main-memory databases. For example, IMS/VS Fast Path [Gawlick and Kinkade 1985] delivers frequently used database items from an MMDB and infrequently used items from a disk-resident database

(DRDB). The two databases are designed with separate access mechanisms. Instead of making duplicate copies of data as in multi-level caching, IMS/VS Fast Path occasionally migrates data from one persistent medium to another based on access patterns. Similar to *Conquest*, IMS/VS Fast Path relies on battery backing, frequent system backup, and uninterruptible power supplies for reliability. MARS [Eich 1987], HALO [Garcia-Molina and Salem 1987], TPK [Li and Naughton 1988], and other early MMDBs rely on data mirroring, background logging, or dual processing to achieve reliability.

The cost of main memory may be the primary concern that prevents operating systems from adopting similar solutions for practical use, and *Conquest* offers a transition for delivering file system services from main memory in a practical and cost-effective way.

9.3 File-System Applications of Persistent RAM

One early proposed use of persistent RAM was to hold write buffers [Baker et al., 1992]. Since the data would be buffered in persistent memory, the interval between synchronizations to the disk could be lengthened. Although disk activities would be reduced significantly, this approach does not eliminate the problems of data duplication, migration, synchronization, and fresh loading of buffer when the disk content is first accessed.

A plethora of flash-memory-based file systems has emerged to replace disks on small mobile-computing devices [Wu and Zwaenepoel 1994; Kawaguichi et al., 1995; Niijima 1995; Torelli 1995]. The low-power requirements of these devices make flash memory an attractive choice; however, flash memory has unpleasant characteristics of its

own, such as limited numbers of erase-write cycles and slow (second-range) time for storage reclamation. These characteristics cause performance problems and introduce new kinds of file-system complexity in the code written to handle them.

Quantum has proposed using battery-backed DRAM for logging, metadata updates, transactions, and caching for disk arrays, database, and multimedia servers [Quantum 2000a; Quantum 2000b; Quantum 2000c]. They do not plan to use BB-DRAM as the primary storage device, however. Instead, they access the BB-DRAM devices as if they were mechanical disks via the SCSI interface, with the same software infrastructures to handle disks. From a marketing point of view, Quantum has shortened the development process by making BB-DRAM plug-and-play, but from the engineering point of view, those BB-DRAM devices are far from achieving their full performance potential.

The Rio file cache [Chen et al., 1996; Ng and Chen 2001] combines UPS, volatile memory, and a modified write-back scheme to achieve the reliability of a write-through file cache and the performance of a pure write-back file cache (with no reliability-induced writes to disk). The resiliency offered by Rio complements *Conquest* well. While *Conquest* uses main store as the final storage destination, Rio's BIOS *safe sync* mechanism provides high assurance for a system to: (1) transfer control to the `sync` routine during a crash, and (2) write the data content of memory to disk. Rio's reliability mechanisms try to avoid any dependency on underlying kernel mechanisms to minimize the effect of kernel crashes on the proper operations of Rio. For example, Rio removes dependencies on the virtual memory system by switching the processor to use physical

addresses. Also, Rio removes dependencies on the kernel device drivers by using the BIOS interface to disk.

The HeRMES project [Miller et al., 2001] takes advantage of a form of persistent RAM under development—Magnetic RAM (MRAM) [Boeve et al., 1999]. HeRMES uses MRAM primarily to store the file metadata to reduce a large component of existing disk traffic, and also to buffer writes to lengthen the time frame for committing modified data. HeRMES also assumes that persistent RAM will remain a relatively scarce resource for the foreseeable future, especially for large file systems.

9.4 PDA Operating Systems

The leading PDAs on the market are the PalmPilot and Windows CE devices. Both systems deliver memory and file system services via BB-DRAM, but their designs are more concerned with fitting the operating systems into the memory-constrained environment than with exploiting an assumed abundance of persistent RAM.

To use limited BB-DRAM on handhelds, both the PalmPilot and Windows CE make many design simplifications. PalmPilot has designed a commodity with a few essential core services. For Palm OS 5 [Palm 2000], the execution environment is devoid of the concept of processes. A single program executes at a time. To ease memory management, Palm OS hardwires partitions of memory space for purposes such as low memory globals, dynamic heap, and storage heap. The low memory is used for various OS subsystems. The dynamic heap stores global data for the operating system, TCP/IP stack, library data, buffers, application global variables, application data, application stack, and dynamically allocated program memory. The storage heap is for persistent

storage. The data storage granularity is a contiguous *chunk* of memory of 1 to 64 Kbytes, managed by an internal database engine. For large files, an alternative file API can support files of arbitrary sizes, and file-based operations are buffered. The reliability model of PalmPilot mainly relies on battery backing and data synchronization to a desktop machine.

Unlike Palm OS, Windows CE tries to miniaturize the full operating system environment to the scale of a PDA, with full support for multiple processes and threads for execution. Windows CE 3.0 [Microsoft 2002a; Microsoft 2002b] uses a variety of techniques to simplify memory management and to reduce memory overhead. For example, Windows CE does not use the current-directory concept. Instead, all references to an object are given with the full path. File operations retain data-duplication semantics, with memory-mapping mechanisms available to avoid copying. Windows CE can mount external file systems, but they are limited to the size of the memory on the handheld device, and the maximum file size is limited to 32 Mbytes.

Neither PDA design is suitable for general deployment on desktop computers. The PalmPilot lacks a full-featured execution model, and efficient methods for accessing large data objects are limited. Windows CE is not designed for desktop-scale deployment, and many management functions are simplified by requiring the end users to specify them explicitly (e.g., the boundary between the program memory and the permanent storage).

9.5 IBM AS/400

IBM AS/400 servers provide the appearance of storing all files in memory from the user's point of view. This uniform view of storage access is accomplished by the extensive use of virtual memory. The AS/400 design is an example of how *Conquest* can enable a different file system API. However, underneath the hood of AS/400, the conventional role of memory as the cache for disk content still applies, and disks are still the persistent storage medium for files [IBM 2002].

9.6 Slice

Conquest's approach of separating data paths based on the file sizes and metadata can also be found in distributed storage systems. The Slice file service [Anderson et al., 2000] is an example. Each client's request stream is partitioned into three functional request classes: (1) high-volume I/O to large files, (2) I/O on small files, and (3) operations on the name space or file attributes. Based on the request type and arguments, a front-end μproxy switches to redirect requests to a selected server responsible for handling a given class of requests. Directory servers provide naming services through distributed hashing and load balancing. Small-file servers are specialized for fragmentation management so they can provide both efficient storage and high bandwidth. Bulk I/O operations route directly to an array of storage nodes, which provide block-level access to raw storage objects.

9.7 Summary

Conquest's design is unique in a number of ways:

- Unlike disk caching and various applications of persistent RAM, *Conquest* assumes an abundance of persistent RAM to deliver all file system functions with the single exception of storing large files.
- *Conquest* is not another technique to optimize disk accesses, since *Conquest* tries to remove all disk-related complexity from the critical path of accessing memory content.
- *Conquest* aims to provide a practical and inexpensive solution to the transition from disk-based to memory-based file systems at the scale of the desktop machine; this design has distinct characteristics from PDAs, databases, and distributed systems.
- Instead of providing the *illusion* of everything being stored in memory (e.g., virtual memory and IBM AS/400), *Conquest* stores small file and metadata in memory persistently.

Chapter 10

Future Work

Conquest is operational, but its performance and usability can continue to be improved. A few improvement areas have already been discussed: optimizing large-file-only disk storage (Section 4.2.2), overcoming the limitations of the BIOS for software distribution (Section 5.4.3), adding recovery mechanisms from catastrophic crashes (Section 6.1.3), and designing mechanisms for adjusting file size thresholds dynamically (Section 6.2).

The design and the implementation of *Conquest* have also revealed many future research directions for the memory-rich environment (Section 10.1). In addition, the current version of *Conquest* can be enhanced in performance (Section 10.2) and extended for network accesses (Section 10.3).

10.1 Research Directions

The experience gained in designing *Conquest* suggests future research in many areas: distributed storage systems (Sections 10.1.1 and 10.1.2), the computing paradigm (Section 10.1.3), application programming interface design for file and memory systems (Section 10.1.4), and performance measurement (Section 10.1.5).

10.1.1 Distributed and Automated Specialization of Storage Functions

The design philosophy of matching usage patterns with physical characteristics can be extended to the distributed domain. Unlike the popular assumption of homogeneous hardware, this extension assumes an environment of heterogeneous hardware with

different available resources and computing power. Since different machines excel in delivering different storage requests, machines could specialize and coordinate in such a way that the overall system performance is improved.

Slice [Anderson et al., 2000] is an example of such specialization based on file sizes and metadata (Section 9.6). Other approaches to specialization are also possible. For example, large files can be further separated into streaming-based and non-streaming-based files. For *streaming*-based files, a part of a file can be processed while waiting for other parts to become available. Therefore, a file system only needs to make parts of files available in a timely fashion, with relaxed disk layout and I/O buffering constraints. For *non-streaming*-based files, the whole file needs to be available before further processing. Therefore, a file system needs to minimize the elapsed time needed to access files *in toto*.

Preferably, such a distributed system could self-characterize its performance, self-organize functions among systems, and self-evolve as user access patterns change over time.

10.1.2 *Conquest* and Replication

Conquest's separation of metadata management from data is not new. This style of management is commonly seen in databases, where a single data file is accessed through multiple index files. However, the separation of management offers *Conquest* additional possibilities for providing replication services. For example, if metadata can be associated with data with different fidelities, a *Conquest* file system could be replicated across hardware with substantially different capabilities. On a PDA that is equipped with only persistent RAM, *Conquest* could provide a smaller yet semantically identical data

set (e.g., via lossy compression, context abstraction) that can fulfill both the size threshold and the memory storage constraints. On a desktop, *Conquest* could provide the full-fidelity version of data from disk storage.

This *hierarchical* model of replication differs from Odyssey [Noble and Satyanarayanan 1999], which aims to provide access to remote data with different fidelities, depending on the network conditions. The hierarchical replication also differs from the Roma Personal Metadata Service [Swierk et al., 2000], which relies on a centralized metadata service to coordinate replicas and does not handle varying fidelities of data.

Research challenges to designing this hierarchical replication involve defining the semantic equivalence of data among platforms. This definition is particularly tricky for updates to data replicas with different fidelity. In addition, as file sizes change, data may need to change storage medium and migrate from time to time. The overall reliability model also needs to accommodate this distributed environment with heterogeneous hardware.

10.1.3 State-Rich Computing

With the advent of network and distributed computing, an increasing number of systems advocate a stateless design for ease of failure recovery. However, this trend is orthogonal to the usefulness of stateful computing. Stateful computing is prevalent in other areas, such as the extensive use of caching to improve performance. *Conquest*'s direct storage of metadata in runtime form takes a further step and advocates state-rich computing. In

addition to caching data, caching runtime data structures can further improve system performance.

One concrete use of stateful computing is to dedicate an area of memory for scratch purposes, similar to `/tmp`. Storing runtime data structures in this area can greatly accelerate application startup times if the dominating overhead is resurrecting and reconstructing data structures from the prior runs. In addition, programs can be configured so that all internal data structures are presented as files to facilitate debugging. This feature generalizes the `/proc` concept and makes a service similar to `/proc` available to all applications. The main research challenges to building this feature involve designing the programming interface and the access control mechanisms.

10.1.4 API Expansion

For persistent RAM, the semantics of common file operations—`open`, `close`, `read`, and `write`—are no longer adequate because once a file is opened, its entire content is immediately accessible without further reads or seeks. Memory-mapped semantics are more appropriate in this context. However, memory-mapping semantics are awkward in situations like file creation, where the total amount of data is not known in advance. The most suitable form of API is not obvious at this point, but this API will affect the implementation of many I/O libraries, such as the I/O stream library in C++.

While accessing runtime data structures in the kernel using the physical addressing mode is trivial, making persistent memory available to application developers is another API design area. One possibility is to provide a memory-naming service so applications do not need to track memory addresses across invocations. The immediate

challenges are finding a protection and access control model that does not conflict with the existing protection mechanisms at the memory-mapping granularity. A complementary service to memory-naming is future support for unnamed files, which are useful for applications that need to generate temporary files with unique names. Relevant research questions include the structure of the memory name space and its relation to the storage name space.

10.1.5 VFS Interface Expansion

The VFS interface needs to become aware of the persistent RAM medium. In particular, VFS should provide the option to disable data and metadata caching for memory-speed storage media. The VFS implementation under Linux 2.4.2 does not support this option.

10.1.6 Performance Measurement

The *Conquest* design has aggressively removed many disk-related complexities from the memory critical path without questioning exactly how much each disk optimization adversely affects system performance. One area of research is to break down these performance costs, so that designers can improve memory performance for disk-based file systems. Experiences from the software engineering field have shown us that frequently 10% of code is often responsible for 90% of the running time. The analogous postulate would be that 10% of optimization is responsible for 90% of performance gains. Should the postulate be proven correct, the remaining performance benefit may not justify the additional 90% of complexity, which shall be subject to removal.

A more general research area is how ineffective optimizations can be identified in an automated manner.

During the measurement phase of *Conquest*, the examination of the Sprite LFS microbenchmark uncovered a number of potential problems with applying existing disk-based file system benchmarks to measure the memory performance of file systems. One natural extension is to analyze additional popular benchmarks to identify similar problems. A more far-reaching question is how file system designers and benchmark makers can take account of those anomalies and design systems and benchmarks accordingly.

As CPU speeds continue to increase, file system performance is becoming more sensitive to the underlying hardware than ever before. Various asynchronous mechanisms expose additional unanticipated dependencies among high-level operations and file system performance numbers. One naive solution is to alter the `fsync` system call to also flush L1 and L2 cache content. Doing so would speed up the read phase of the Sprite LFS microbenchmark, but not speed up the total execution time of the benchmark suite. Whether doing the flush depends on the goals of benchmark measurement. The time may be coming for designers of file systems and benchmarks to rethink the effects of the L2 cache.

10.2 Performance Improvement

Although *Conquest* has already demonstrated significant performance improvements over disk caching and disk-based file systems, numerous approaches to further improving *Conquest* exist.

10.2.1 Removal of Virtual Memory and Metadata Caching Overheads

Conquest still contains a number of complex mechanisms that impose unnecessary overheads—notably virtual memory and VFS metadata caching.

The major function of VM is to provide the illusion of having an ample amount of memory. Since *Conquest* assumes the emergence of a memory-rich era, many aspects of VM services can be either simplified or removed.

VM consists of several major functions—demand paging, swapping, protection, and memory mapping. Removing demand paging is problematic for legacy applications. For example, a legacy application can memory-map files into the address space of an application and demand-page the file as needed. NFS also relies on demand paging to memory map parts of remote files or executables across a network. However, in an environment with an abundance of memory and a high-speed network backbone, the NFS style of remote access can impose unacceptably high overhead. Instead of maintaining backward compatibility, a complete rethinking and redesign of the storage API and remote access methods might yield simpler solutions that could significantly outperform the current ones.

As the size of memory increases, the swapping code would become less likely to be exercised and could be removed. Memory mapping is an important service for programs to run in an address space with virtual contiguity, so this service will probably remain. The security and protection offered by the virtual memory subsystem can also remain an independent service. However, both memory mapping and protection

functions should be enhanced to accommodate more flexible management granularity, since persistent RAM is no longer a low-latency, block-oriented device.

At the VFS front, the default path assumes high-latency storage, making it hard for *Conquest* to disable metadata caching (Section 6.3). Modifications to the VFS internal routines and core data structures would be necessary to make memory access be the default data path and to allow the option to disable metadata caching.

10.2.2 Kernel-Level Performance Tuning

The removal of the high-latency storage assumption might affect existing kernel design and performance in terms of locking, scheduling, and storage access.

10.2.2.1 Locking Granularity

As a study by the database community points out, faster speed for storage access results in a lower probability of lock contention and higher overhead. Also, larger locking granularity is more desirable for overhead reduction [Lehman and Carey 1987]. However, faster speed for storage access also makes finer locking granularity more affordable. Instead of locking at a directory level, individual entries of the directory can be locked to provide better concurrency. In addition, since the performance landscape will be changed by the introduction of persistent-RAM storage, new performance characteristics may lead to new bottlenecks and different places for lock contentions. Further studies are needed to investigate these topics.

10.2.2.2 Kernel Task Scheduler

The effective functioning of conventional kernel task schedulers currently relies on a proper mixture of CPU- and I/O-bound jobs. However, under *Conquest*, this scheduling policy may no longer yield good performance, since I/O jobs can complete at memory speed. The removal of I/O-bound jobs may either lead to simpler schedulers, or to equally complex ones that divide jobs according to different characteristics than are used today.

10.2.2.3 Kernel-Level Storage Access

Traditionally, operating system developers have avoided using the file system to store any kernel-related data because of the high cost of disk accesses. However, *Conquest* now provides high-speed accesses to persistent storage at the kernel level, and the availability of this feature could simplify message logging and configuration management significantly.

For security and diagnostic purposes, a kernel often produces text messages that are recorded to a persistent storage medium. In most existing systems, this recording is accomplished using a complex mechanism involving a special kernel logging buffer and a daemon that periodically copies the contents of the buffer to the file system. If the kernel has direct and fast access to in-memory persistent storage, the specialized mechanisms can be replaced by a simple file-system write.

A modern kernel also requires numerous configuration parameters to operate correctly. For example, network communication depends on a routing table to identify

the next hop or the gateway for forwarding a particular message. In current systems, these parameters are loaded into kernel memory at boot time, usually from a file, and usually via a mechanism that is specialized to the individual parameters under consideration. If the system administrator needs to modify those parameters, he or she must first edit the file and then make the changes visible to the kernel by running a special program that will load the modifications into the kernel tables. Under *Conquest*, however, the file would be stored in a location that is directly accessible by the kernel. There would be no need to load parameters at boot time, and any changes made by the system administrator would be visible without extra effort. This approach would simplify administration and reduce the number of special-purpose mechanisms built into the kernel.

10.2.3 Executables and Shared Libraries

Conquest does not give special treatment to executables and shared libraries at this time. However, if both are stored in memory, the cost of starting up a new program will be reduced by not faulting pages into memory during execution. Since shared libraries are modular extensions of executables, storing libraries in-core should yield similar benefits.

10.2.4 L2-Caching-Related Optimizations

The write-back effects of L2 caching may lead to the conclusion that the costs of writes are being shifted around. However, write-back effects could also be exploited for performance gains with proper ordering of reads and writes. For instance, instead of accessing a large file in phases (as shown in the Sprite LFS large-file microbenchmark),

multiple phases could iterate through chunks from a large file that are each small enough to fit in the L2 cache. In addition, such exploitation could be automated at the compiler level, so software developers would not be burdened with details of underlying hardware.

A more general question is how to take advantage of the low-level dependencies among various file operations to improve high-level performance.

In addition, at the speed of the L2 cache, the overhead of kernel crossing for system calls starts to become significant. Multiple system calls can be bundled together as a way to amortize the cost of crossing the kernel boundary [Purohit et al., 2002]. Also, similar to out-of-order execution at the hardware level, multiple system calls might be executed out-of-order for better performance.

10.3 Network Extensions of *Conquest*

One natural extension of *Conquest* is to go beyond the machine boundaries. Although NFS [Walsh et al., 1985] offers remote data accesses with a simple crash recovery scheme, *Conquest* will lose much of its benefits if it uses NFS to access remote *Conquest* files. In particular, given that NFS is stateless, each NFS request must describe the operation completely. For example, since the NFS server does not keep track of open files, a `read` request from a client needs to `open` a file at the server side, `read` the file content, and `close` the file. Also, NFS assumes high-latency storage media and uses many disk-related optimizations in the VFS.

The network extension of *Conquest*, on the other hand, will have many opportunities for simplification and optimization. For instance, *Conquest* can exploit 64-

bit addressing [Chase et al., 1992] so pointers are meaningful across machine boundaries, thereby avoiding serialization and deserialization when transferring data structures. Also, if the server side is dedicated only to remote accesses, the data content of files can be stored in pre-packetized format, ready for raw network transmission.

Chapter 11

Conclusions

11.1 Summary of the Problem

Modern file systems are designed with the assumption that disk will be used; this assumption is problematic in two ways. First, disk performance is six orders of magnitude slower than CPU, and disk bottleneck is becoming worse. Second, disk-related optimizations have increased over the years in an attempt to mask this widening performance gap, resulting in a proliferation of disk-related complexity in operating system code. Popular solutions, such as disk caching and RAM file systems, either fail to take full advantage of memory speed or fail to provide the flexibility of storing arbitrary amounts of data persistently, reliably, and conveniently.

11.2 The *Conquest* Solution

The design of *Conquest* is based on the observation that memory is becoming an inexpensive resource, and is able to deliver all file system services with the single exception of providing large storage capacity. Instead of using memory to buffer and speed up disk accesses, *Conquest* uses memory with battery backup as a persistent storage, and provides two separate data paths to memory and disk storage. By specializing the data path, the memory data path is devoid of disk-related complexity. The disk data path consists of only optimizations for the specialized usage. *Conquest*'s design solves the two major problems faced by disk-based file systems: (1) making

memory the first-class storage medium provides memory-speed access of memory content without disk-related overhead; (2) specialized use of memory and disk reduces the complexity of both data paths.

11.3 Summary of the *Conquest* Design

Based on studies of common workloads, *Conquest* divides files into large and small files based on a static threshold (1 Mbyte). *Conquest* stores small files and metadata in RAM and only the data content of large files on disk. Storing small files and metadata in RAM gives the most performance benefits by avoiding the relatively high overhead of disk seeks. Storing metadata in RAM also allows in-place updates and avoids many synchronous disk-related mechanisms propagating metadata changes to disk. Since large files are accessed sequentially most of the time, the disk is able to deliver data blocks of large files at near-raw bandwidth.

For metadata representation, the use of memory allows *Conquest* to use hash tables for looking up directories and keeping track of data blocks. Each directory under *Conquest* maintains a dynamically allocated hash table of file metadata entries, and a file path is resolved recursively at each level of the hash table. The data content of small files is stored in RAM and accessed through a uniform single-level indirect index. The data content of large files is stored on disk in large contiguous segments, and the metadata of large files are represented as linked lists of segments.

Conquest manages metadata by reusing and instantiating the underlying Linux memory managers. In particular, *Conquest* reuses the high-level slab allocator to avoid memory fragmentation management. It reuses buddy allocators to provide quick

allocations of power-of-two sizes, and also reuses the low-level page allocator to provide attributes for memory pages. The use of dedicated *Conquest* zones and physical addressing allows the encapsulation of *Conquest* pointers and all states of *Conquest* memory managers. Paging and swapping are disabled in *Conquest* memory regions but enabled in non*Conquest* regions for backward compatibility.

Conquest relies on pointer-switch commits to avoid the cost of two-phase commits. In the worst case, failures result in memory leaks that can be garbage-collected. *Conquest* relies on built-in mechanisms and daily backups to rollback to a prior memory image.

Similar to disk-based file systems, *Conquest* currently considers memory depletion as storage exhaustion. However, users can simply migrate memory content to disk by archiving unwanted large directories into a file that exceeds the size threshold, and the archived file is automatically moved from memory to disk.

Currently, *Conquest* still has a number of challenges to overcome before full deployment: optimizing large-file-only disk storage, overcoming the limitations of the BIOS, adding improved recovery mechanisms for catastrophic crashes, and designing mechanisms for adjusting file size thresholds dynamically.

Our evaluation of *Conquest* shows that it is feasible to simplify an existing file system and to achieve significant performance gains at the same time. With a memory-only workload, *Conquest* matches the performance of *ramfs* and outperforms disk caching in transaction rates by 24% to 350%. With a workload that exercises memory and disk components, *Conquest* still outperforms disk caching by 43% to 96%.

11.4 Lessons Learned

The experience of designing and implementing *Conquest* offers several major lessons:

- ***The handling of disk characteristics permeates file system design even at levels above the device layer.*** For example, the default VFS routines contain readahead and buffer-cache mechanisms that add high and unnecessary overheads to low-latency main store. Because of the need to bypass these mechanisms, building *Conquest* was much more difficult than we initially expected. For example, certain internal storage routines anticipate data structures associated with disk handling. Reusing these routines either involves constructing memory-specific access routines from scratch, or finding ways to invoke these routines with memory-based data structures.
- ***File systems that are optimized for disk are not suitable for an environment where memory is abundant.*** For example, *ext2*, *reiserfs*, and *SGI XFS* do not exploit the speed of RAM as well as anticipated. Disk-related optimizations impose high overheads on in-memory accesses.
- ***Matching the physical characteristics of media to storage objects provides opportunities for faster performance and considerable simplification for each medium-specific data path.*** *Conquest* applies this principle of specialization: leaving only the data content of large

files on disk leads to simpler and cleaner management for both memory and disk storage. This observation may seem obvious, but good results are not achieved automatically. For example, should the L2 cache footprint of two specialized data paths exceed the size of a single generic data path, the resulting performance can go in either direction, depending on the size of the physical cache.

- ***Access to cached data in traditional file systems incurs performance costs due to commingled disk-related code.*** Removing disk-related complexity for in-core storage under *Conquest* therefore yields unexpected benefits even for cache accesses. In particular, one surprising result was *Conquest*'s ability to outperform *ramfs* by 8% to 17% in bandwidth under microbenchmarks, despite the fact that storage data paths in *ramfs* are already heavily optimized.
- ***It is much more difficult to use RAM to improve disk performance than it might appear at first.*** Simple approaches such as increasing the buffer-cache size or installing simple RAM-disk drivers do not generate a full-featured, high-performance solution.

The measurement phase of *Conquest* was also instructive:

- ***The effects of memory and L2 caching have become highly visible in memory-only workloads.*** For example, the L2 write-back effect in the original Sprite LFS large-file microbenchmark introduced 20%

differences in performance numbers when comparing *ramfs* to cached *ext2*. As modern workloads become increasingly memory-intensive, application writers and system designers need to become increasingly aware of L2 caching effects.

- ***Disk-based benchmarks are not suitable to measure memory performance of file systems.*** Although the computing environment is moving toward a memory-rich environment where more programs execute entirely from memory, existing file-system benchmarks still emphasize measuring disk performance. Non-cache-aware benchmark designs can easily overlook subtle memory behaviors that can significantly alter the outcomes of the benchmark. Our investigation of the Sprite LFS large-file microbenchmark alone has identified a number of such effects: the write-back policy of L2 caching, memory access alignment, effects of L2 cache footprint on the relative speeds of random and sequential accesses, and the overhead of system call crossing.

The overall lesson that can be drawn is that seemingly simple changes to existing file system designs can have much more far-reaching effects than first anticipated. The modifications may be more difficult than expected, but the benefits can also be far greater. Also, moving from a disk-based to a memory-based design is more than a mere shift of storage medium. The new system characteristics have far-reaching implications for benchmarking techniques, benchmark designs, file system structure, and applications.

11.5 Contributions

The research behind this dissertation has made the following unique contributions:

- The construction and daily use of *Conquest* as the development platform demonstrate the feasibility of designing inexpensive, fast, and deployable disk/persistent-RAM hybrid file systems. With a cost below $200, one can purchase a UPS and enough RAM to deploy *Conquest*.
- *Conquest* illustrates that speculative disk optimizations (e.g., caching) harm the common case of accessing memory content. By providing a dedicated memory data path that is devoid of disk-related complexity, even an untuned implementation of *Conquest* can outperform *ramfs* under certain memory-only workloads.
- The success of *Conquest* in achieving better performance with simpler data paths shows that performance does not preclude simplicity. The use of two specialized and simplified data paths under *Conquest* can outperform a generalized complex data path in conventional file systems.
- The investigation of microbenchmark anomalies pinpoints memory and cache-related problems with modern disk-based benchmarks. Without this understanding, one might falsely conclude that disk-based file systems can outperform memory-based file systems (Chapter 8).

Overall, the experience of developing *Conquest* contributed early insights into the design, implementation, and measurement of disk/persistent-RAM hybrid file systems.

11.6 Final Comments

This dissertation presents *Conquest*, a fully operational file system that integrates persistent RAM with disk storage to provide significantly improved performance compared to other approaches, such as RAM disks or enlarged buffer caches. With the involvement of both memory and disk components, *Conquest* demonstrates a 43% to 96% speedup compared to popular disk-based file systems.

During the development of *Conquest*, we discovered a number of unexpected results. Obvious approaches failed to provide a complete solution, but existing file systems performed more poorly than *Conquest* due to the unexpectedly high cost of going through the buffer cache and disk-specific code. We found that it was very difficult to remove the disk-based assumptions integrated into operating systems, a task that was necessary to allow *Conquest* to achieve its goals. In addition, we discovered that applying existing disk-based benchmarks to measure the memory performance of file systems could lead to counter-intuitive results. A smaller L2 cache footprint can actually lead to worse performance numbers due to write-back policies.

The benefits of *Conquest* arose from rethinking basic file system design assumptions. *Conquest* explores the implications of a memory-rich environment, challenges the commonly perceived performance bound of LRU disk caching, and questions layer-based optimizations by proposing separate cut-through data paths. By

revisiting individual decision points in a new context and designing from the ground up, we evolved *Conquest* into a simple architecture that surpasses the performance accomplished by many layers of legacy optimizations.

As modern hardware pushes toward faster performance and cheaper production costs, we will inevitably confront many difficult design choices—between incremental solutions and complete redesigns, performance optimizations and design simplicity, generalized solutions and case-specific approaches, transparent benefits and user-specified options, and so on. By applying *Conquest*'s rethinking process and simple handling of common cases, many design decisions will be simplified as well.

- Incremental performance improvement may not justify additional complexity.
- A solution that provides benefits transparently can avoid inventing an API, which legacy applications must adopt to gain the benefits of the new solution.
- A solution that handles simple common cases can be sufficient to avoid the complexity and overhead of a generalized solution.
- A complete redesign due to a new set of assumptions can remove many noncontributing legacy optimizations while retaining backward compatibility.

Conquest's success demonstrates how this process of rethinking underlying assumptions can lead to significant performance benefits and architectural

simplifications. This success suggests that radical changes in the hardware, applications, and user expectations of the past decade should also lead us to reflect on future file system design and other aspects of operating system design.

Trademarks

UNIX is a registered trademark of the Open Group. LINUX is a registered trademark of Linus Torvalds. BSD is a registered trademark of UUNET Technologies, Inc. MAC OS is a registered trademark of Apple Computer, Inc. PALM OS is a registered trademark of Palm Computing, Inc. MS-DOS, WINDOWS 95, WINDOWS 98, WINDOWS 2000, WINDOWS NT, and WINDOWS CE are registered trademarks of Microsoft Corporation. VMS is a registered trademark of Digital Equipment Corporation.

XFS is a registered trademark of Silicon Graphics, Inc. NFS is a registered trademark of Sun Microsystems, Inc. POSTSCRIPT is a registered trademark of Adobe Systems Inc. POWERPOINT is a registered trademark of Microsoft Corporation.

PENTIUM and XEON are registered trademarks of Intel Corporation. IBM and AS/400 are registered trademarks of International Business Machines Corporation. DELL and POWEREDGE are registered trademarks of Dell Computer Corporation. ADAPTEC is a registered trademark of Adaptec, Inc. APC is a registered trademark of American Power Conversion Corporation.

References

[Anderson et al., 2000] Anderson D, Chase J, Vahdat A. Interposed Request Routing for Scalable Network Storage. *Proceeding of the 4th Symposium on Operating System Design and Implementation (OSDI)*, October 2000.

[Baker et al., 1991] Baker MG, Hartman JH, Kupfer MD, Shirriff KW, Ousterhout JK. Measurements of a Distributed File System. *Proceedings of the 13th Symposium on Operating Systems Principles*, December 1991.

[Baker et al., 1992] Baker M, Asami S, Deprit E, Ousterhout J, Seltzer M. Non-Volatile Memory for Fast, Reliable File Systems. *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[Bitton and Gray 1988] Bitton D, Gray J. Disk Shadowing. *Proceedings of Very Large Database Conference XIV*, pp. 331-338, 1988.

[Boeve et al., 1999] Boeve H, Bruynseraede C, Das J, Dessein K, Borghs G, De Boeck J, Sousa R, Melo L, Freitas P. Technology Assessment for the Implementation of Magnetoresistive Elements with Semiconductor Components in Magnetic Random Access Memory (MRAM) Architectures. *IEEE Transactions on Magnetics*, 35(5), pp. 2820-2825, 1999.

[Bolosky et al., 1997] Bolosky WJ, Fitzgerald RP, Douceur JR. Distributed Schedule Management in the Tiger Video Fileserver. *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Saint-Malo, October 1997.

[Bonwick 1994] Bonwick J. The Slab Allocator: An Object-Caching Kernel Memory Allocator. *Proceedings of USENIX Summer 1994 Technical Conference*, June 1994.

[Bozman et al., 1991] Bozman GP, Ghannad HH, Weinberger ED. A Trace-Driven Study of CMS File References. *IBM Journal of Research and Development*, 35(5-6) pp. 815-828, September-November 1991.

[Butner 2001] Butner R. Computing Unplugged: Magnetic RAM cures your computer of short-term memory loss. *IBM Think Research*. http://www.research.ibm.com/thinkresearch/pages/2001/20010202_mram.shtml, 2001.

[Cáceres et al., 1993] Cáceres R, Douglis F, Li K, Marsh B. Operating System Implications of Solid-State Mobile Computers. Technical Report MITL-TR-56-93, Matsushita Information Technology Laboratory, US, May 1993.

[Card et al., 1994] Card R, Ts'o T, Tweedie S. Design and Implementation of the Second Extended Filesystem. *Proceedings of the First Dutch International Symposium on Linux*, ISBN 90-367-0385-9, December 1994.

[Chase et al., 1992] Chase J, Levy H, Lazowska E, Baker-Harvey M. Opal: A Single Address Space System for 64-Bit Architectures. *Proceedings of IEEE Workshop on Workstation Operating Systems*, April 1992.

[Chen et al., 1996] Chen PM, Ng WT, Chandra S, Aycock C, Rajamani G, Lowell D. The Rio File Cache: Surviving Operating System Crashes. *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[Chen and Thapar 1997] Chen S, Thapar M. A Novel Video Layout Strategy for Near-Video-on-Demand Servers. Technical Report HPL-97-52, 1997.

[Cormen et al., 1990] Cormen TH, Leiserson CE, Rivest RL. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1990.

[Douceur and Bolosky 1999] Douceur JR, Bolosky WJ. A Large-Scale Study of File-System Contents. *Proceedings of the ACM Sigmetrics '99 International Conference on Measurement and Modeling of Computer Systems*, Atlanta, GA, USA, May 1999.

[Douglis et al., 1994] Douglis F, Cáceres R, Kaashoek F, Li K, Marsh B, Tauber JA. Storage Alternatives for Mobile Computers. *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, November 1994.

[Duncan 1986] Duncan R. *Advanced MSDOS Programming*. Microsoft Press, Redmond, Washington, 1986.

[Eich 1987] Eich MH. A Classification and Comparison of Main Memory Database Recovery Techniques. *Proceedings of International Conference on Data Engineering*, February, 1987.

[Evans and Kuenning 2002] Evans KM, Kuenning GK. A Study of Irregularities in File-Size Distributions. *Proceedings of 2002 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS '02)*, San Diego, July 2002.

[Fagin et al., 1979] Fagin R, Nievergelt J, Pippenger N, Strong HR. Extensible Hashing—A Fast Access Method for Dynamic Files, *ACM Transactions on Database Systems*, 4(3) pp. 315-344, 1979.

[Ganger et al., 2000] Ganger GR, McKusick MK, Soules CAN, Patt YN. Soft Updates: A Solution to the Metadata Update Problem in File Systems. *ACM Transactions on Computer Systems*, 18(2) pp. 127-153, May 2000.

[Garcia-Molina and Salem 1987] Garcia-Molina H, Salem K. High Performance Transaction Processing with Memory Resident Data. *Proceedings of International Workshop on High Performance Transaction Systems*, December 1987.

[Garcia-Molina and Salem 1992] Garcia-Molina H, Salem K. Main Memory Database Systems: An Overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6) pp. 509-516, December 1992.

[Gawlick and Kinkade 1985] Gawlick D, Kinkade D. Varieties of Concurrency Control in MIS/VS Fast Path. *IEEE Database Engineering*, 8(2), pp. 3-10, 1985.

[Gibson and Patterson 1993] Gibson GA, Patterson DA. Designing Disk Arrays for High Data Reliability. *Journal of Parallel and Distributed Computing*, 1993.

[Griffioen and Appleton 1995] Griffioen J, Appleton R. Performance Measurements of Automatic Prefetching. *Proceedings of the ISCA International Conference on Parallel and Distributed Computing Systems*, September 1995.

[Grochowski 2002] Grochowski E. Storage Price Projections. http://www.almaden.ibm.com/sst/html/leadership/g05.htm, 2002.

[Haskin and Schmuck 1996] Haskin R, Schmuck F. The Tiger Shark File System. *Proceedings of IEEE 1996 Spring COMPCON*, Santa Clara, CA, February 1996.

[Heller 2000] Heller DE. Rabbit, A Performance Counters Library for Intel Processors and Linux, Iowa State University, 2000.

[Howard et al., 1988] Howard J, Kazar M, Menees S, Nichols D, Satyanarayanan M, Sidebotham R, West M. Scale and Performance in a Distributed File System, *ACM Transactions on Computer Systems*, 6(1), pp. 51-81, February 1988.

[IBM 2002] IBM iSeries Storage Overview. http://www-1.ibm.com/servers/eserver/iseries/hardware/storage/overview.html, 2002.

[Irlam 1993] Irlam G. UNIX File Size Survey—1993, http://www.base.com/gordoni/ufs93.html, 1993.

[Katcher 1997] Katcher J. PostMark: A New File System Benchmark. Technical Report TR3022. Network Appliance Inc., October 1997.

[Kawaguichi et al., 1995] Kawaguichi A, Nishioka S, Motoda H. A Flash-Memory-Based File System. *Proceedings of USENIX 1995 Technical Conference on UNIX and Advanced Computing Systems*, Louisiana, January 1995.

[Keshava and Penkovski 1999] Keshava J, Penkovski V. Pentium® III Processor Implementation Tradeoffs. *Intel Technology Journal*, 1999.

[Kroeger and Long 1996] Kroeger KM, Long DDE. Predicting File System Actions from Prior Events. *Proceedings of the USENIX 1996 Annual Technical Conference*, January 1996.

[Kuenning 1995] Kuenning GK. Precise Interactive Measurement of Operating Systems Kernels. *Software Practice and Experience*, 25(1), pp. 1-22, 1995.

[Lehman and Carey 1987] Lehman TJ, Carey MJ. A Recovery Algorithm for a High-Performance Memory-Resident Database System. *Proceedings of ACM SIGMOD Conference*, 1987.

[Li and Naughton 1988] Li K, Naughton JF. Multiprocessor Main Memory Transaction Processing. *Proceedings of International Symposium on Databases in Parallel and Distributed Systems*, December 1988.

[Mahanti et al., 2000] Mahanti A, Williamson C, Eager D. Traffic Analysis of a Web Proxy Caching Hierarchy. *IEEE Network Magazine: Special Issue on Web Performance*, May/June 2000.

[Matthews et al., 1997] Matthews JN, Roselli D, Costello AM, Wang RY, Anderson TE. Improving the Performance of Log-Structured File Systems with Adaptive Methods. *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pp. 238-251, October, 1997.

[McCoy 1990] McCoy K. *VMS File System Internals*. Digital Press, Newton, Massachusetts, 1990.

[McKusick et al., 1984] McKusick MK, Joy WN, Leffler SJ, Fabry RS. A Fast File System for UNIX. *ACM Transactions on Computer Systems,* 2(3), pp. 181-197, 1984.

[McKusick et al., 1990] McKusick MK, Karels MJ, Bostic K. A Pageable Memory Based Filesystem. *Proceeding of USENIX Conference*, June 1990.

[McKusick 2002] McKusick MK. Running "fsck" in the Background. *Proceedings of BSDCon 2002*, February 2002.

[McVoy and Kleiman 1991] McVoy L, Kleiman S. Extent-Like Performance from a Unix File System. *Proceedings on 1991 Winter USENIX Conference*, USENIX Association, Berkeley, California, pp. 137-144, 1991.

[Micron 1997] Module Mean Time Between Failures (MTBF). Technical Note TN-04-45. http://download.micron.com/pdf/technotes/DT45.pdf (go to micron.com, and search for MTBF), 1997.

[Micron 2002] Micron DRAM Product Information. http://www.micron.com (under DRAM and data sheets), 2002.

[Microsoft 2002a] Microsoft Windows CE 3.0: Files, Databases, and Persistent Storage. *MSDN Online Library*. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncenet/html/systemmemorymgmtwince.asp, April 2002.

[Microsoft 2002b] Microsoft Windows CE 3.0: Windows CE Memory Architecture. *MSD Online Library*. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wcefiles/htm/_wcesdk_using_the_file_system.asp, March 2002.

[Miller et al., 2001] Miller EL, Brandt SA, Long DDE. HerMES: High-Performance Reliable MRAM-Enabled Storage. *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems*, May 2001.

[Minnich et al., 2000] Minnich R, Hendricks J, Webster D. The Linux BIOS. *Proceedings of the Fourth Annual Linux Showcase and Conference*, Atlanta, GA, October 2000.

[Mowry et al., 1996] Mowry TC, Demke AK, Krieger O. Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications. *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, 1996.

[Namesys 2002] Namesys. http://www.namesys.com, 2002.

[Ng et al., 1996] Ng NT, Aycock CM, Rajamani G, Chen PM. Comparing Disk and Memory's Resistance to Operating System Crashes. *Proceedings of the 1996 International Symposium on Software Reliability Engineering*, October 1996.

[Ng and Chen 2001] Ng NT, Chen PM. The Design and Verification of the Rio File Cache. *IEEE Transactions on Computers*, 50(4), April 2001.

[Niijima 1995] Niijima H. Design of a Solid-State File Using Flash EEPROM. *IBM Journal of Research and Development*. 39(5), September 1995.

[Noble and Satyanarayanan 1999] Noble BD and Satyanarayanan M. *Mobile Networks and Applications 4*, pp. 245-254, 1999.

[Ousterhout et al., 1985] Ousterhout JK, Da Costa H, Harrison D, Kunze A, Kupfer M, Thompson JG. A Trace Driven Analysis of the UNIX 4.2 BSD File Systems. *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pp. 15-24, 1985.

[Ousterhout 1990] Ousterhout JK. Why Aren't Operating Systems Getting Faster as Fast as Hardware? *Proceedings of the Summer USENIX Conference*, USENIX Association, Berkeley, California, pp. 247-256, 1990.

[Palm 2000] Palm OS Memory Architecture—Take Two: 3.0 and Beyond. http://oasis.palm.com/dev/kb/papers/1145.cfm#header, 2000.

[Patterson et al., 1995] Patterson RH, Gibson GA, Ginting E, Stodolsky D, Zelenka J. Informed Prefetching and Caching. *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pp. 79-95, December 1995.

[Patterson et al., 1998] Patterson DA, Gibson G, Katz RH. A Case for Redundant Arrays of Inexpensive Disks (RAID). *Proceedings of the 1988 ACM Conference on Management of Data (SIGMOD)*, pp. 109-116, June 1988.

[Peripheral Research 1998] Solid State Storage Markets and Technology at Record Levels. http://www.businesswire.com/webbox/bw.020298/593940.htm, 1998.

[Peterson and Norman 1977] Peterson JL, Norman TA. Buddy Systems. *Communications of the ACM*, 20(6) pp. 421-431, 1997.

[Price Watch 2002] Price Watch. New Computer Components. http://www.pricewatch.com, 2002.

[Power Conversion Systems 2000] Power Conversion Systems. http://www.schaeferpower.com/sminvter.htm (Google keywords: schaeferpower, UPS, MIL-HDBK-217), 2000.

[Purohit et al., 2002] Purohit Amit, Spadevecchia J, Write C, Zadok E. Work-in-Progress Report: Improving Application Performance Through System Call Composition. Presented at the 5th Symposium on Operating Systems Design and Implementation, December 2002.

[Quantum 2000a] Achieving Real-Time Multimedia Performance with Multistream Solid-State Disk, http://www.arrownacp.com/arrowess/whitep/ssdvideo.html. January 2000.

[Quantum 2000b] Quantum Solid-State Disk Drives Boost WebServer and Net Performance, http://www.arrownacp.com/arrowess/whitep/ssdweb.html, January 2000.

[Quantum 2000c] Solid-State Disk Turns RAID Systems into Hyperarrays White Paper. http://www.arrownacp.com/arrowess/whitep/ssdraid.html, January 2000.

[Riedel 1998] Riedel E. A Performance Study of Sequential I/O on Windows NT 4. *Proceedings of the 2nd USENIX Windows NT Symposium*, Seattle, August 1998.

[Ritchie and Thompson 1978] Ritchie D, Thompson K. *The Unix Time-Sharing System. Bell System Technical Journal*, 57(6), July-August, pp. 1905-1930, 1978.

[Roselli et al., 2000] Roselli D, Lorch JR, Anderson TE. A Comparison of File System Workloads. *Proceedings of the 2000 USENIX Annual Technical Conference*, June 2000.

[Rosenblum and Ousterhout 1991] Rosenblum M, Ousterhout J. The Design and Implementation of a Log-Structured File System. *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, October 1991.

[Santry et al., 1999] Santry DJ, Feeley MJ, Hutchinson NC, Veitch AC. Elephant: The File System that Never Forgets. *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems (HOTOS)*, March 1999.

[Schindler et al., 2002] Schindler J, Griffin JL, Lumb CR, Ganger GR. Track-Aligned Extents: Matching Access Patterns to Disk Drive Characteristics. *Proceedings of the File and Storage Technologies (FAST) Conference*, January 2002.

[Seagate 2002] Barracuda Technical Specifications. http://www.seagate.com (click on find, barracuda, and technical specifications), 2002.

[Seltzer et al., 1993] Seltzer M, Bostic K, McKusic M, Staelin C. An Implementation of a Log-Structured File System for UNIX. *Proceedings of the Winter USENIX Conference, USENIX Association*, Berkeley, California, pp. 201-220, 1993.

[Seltzer et al., 2000] Seltzer MI, Ganger GR, McKusick MK, Smith KA, Soules CAN, Stein CA. Journaling Versus Soft Updates: Asynchronous Meta-Data Protection in File Systems. *Proceedings of 2000 USENIX Annual Technical Conference*, June 2000.

[Shankland 2001] Shankland S. Transmeta Taking Linux Gadgets Mobile. *CNET News.com*, http://news.com.com/2100-1001-254020.html?legacy=cnet, March 13, 2001.

[Silberschatz et al., 1992] Silberschatz A, Peterson J, Galvin P. *Operating System Concepts*, Third Edition, Addison Wesley, New York, 1992.

[Steere 1997] Steere DC. Exploiting the Non-Determinism and Asynchrony of Set Iterators to Reduce Aggregate File I/O Latency. *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, December 1997.

[Sweeney et al., 1996] Sweeney A, Doucette D, Hu W, Anderson C, Nishimoto M, Peck G. Scalability in the XFS File System. *Proceedings of the USENIX 1996 Annual Technical Conference*, January 1996.

[Swierk et al., 2000] Swierk E, Kiciman E, Laviano V, Baker M. The Roma Personal Metadata Service, *Proceedings of the Third IEEE Workshop on Mobile Computing Systems and Applications*, December 2000.

[Thompson 1978] Thompson K. UNIX Implementation. *Bell System Technical Journal*, 57(6), July-August 1978.

[Toigo 1998] Toigo JW. Solid State Disk: A Business Case Develops. *Enterprise Systems Journal*, 13(8), August 1998.

[Torelli 1995] Torelli P. The Microsoft Flash File System. *Dr. Dobb's Journal*, pp. 63-70, February 1995.

[Torvalds 2001] Torvalds L. Why is *ramfs* slower than *ext2*? *Personal communication*, September 2001.

[UC Berkeley 1998] HTTP Monthly Log Files, http://www.cs.berkeley.edu/logs/http, 1998.

[Uzgalis and Tong 1995] Uzgalis RC, Tong MCF. Hashing Myths, submitted to *Communications of the ACM*, 1995.

[Vogels 1999] Vogels W. File System Usage in Windows NT 4.0. *Proceedings of 17th Symposium on Operating Systems Principles (SOSP'99)*, December 1999.

[Walsh et al., 1985] Walsh D, Lyon B, Sager G, Chang JM, Goldberg D, Kleiman S, Lyon T, Sandberg R, Weiss P. Overview of the Sun Network File System. *Proceedings of the Winter USENIX Conference*, 1985.

[Wang et al., 2001] Wang AIA, Kuenning GH, Reiher P, Popek GJ. Position Summary: The Conquest File System—Life After Disks. *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS VIII)*, Schloss Elmau, Germany, May 2001.

[Wang et al., 2002a] Wang AIA, Kuenning GH, Reiher P, Popek GJ. *Work-in-Progress Report: Conquest: Better Performance Through a Disk/Persistent-RAM Hybrid File System. On-Line Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST)*, Monterey, January 2002.

[Wang et al., 2002b] Wang AIA, Kuenning GH, Reiher P, Popek GJ. *Conquest*: Better Performance Through a Disk/Persistent-RAM Hybrid File System. *Proceedings of the 2002 USENIX Annual Technical Conference*, Monterey, June 2002.

[Wu and Zwaenepoel 1994] Wu M, Zwaenepoel W, eNVy: A Non-Volatile, Main Memory Storage System. *Proceedings of the 6th Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[ZDNet 1999] ZDNet. Quantum Rushmore Solid-State Disk. http://www.zdnet.com/sp/stories/issue/0,4537,396322,00.html (Google keywords: zdnet solid state disk), 1999.