USENIX Association

Proceedings of the 2002 USENIX Annual Technical Conference

Monterey, California, USA
June 10-15, 2002

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Conquest: Better Performance Through A Disk/Persistent-RAM Hybrid File System

An-I A. Wang, Peter Reiher, and Gerald J. Popek✧
Computer Science Department
University of California, Los Angeles
{awang, reiher, popek}@fmg.cs.ucla.edu

Geoffrey H. Kuenning
Computer Science Department
Harvey Mudd College
geoff@cs.hmc.edu

Abstract

The rapidly declining cost of persistent RAM technologies prompts the question of when, not whether, such memory will become the preferred storage medium for many computers. Conquest is a file system that provides a transition from disk to persistent RAM as the primary storage medium. Conquest provides two specialized and simplified data paths to in-core and on-disk storage, and Conquest realizes most of the benefits of persistent RAM at a fractional cost of a RAM-only solution. As of October 2001, Conquest can be used effectively for a hardware cost of under $200.

We compare Conquest's performance to ext2, reiserfs, SGI XFS, and ramfs, using popular benchmarks. Our measurements show that Conquest incurs little overhead compared to ramfs. Compared to the disk-based file systems, Conquest achieves 24% to 1900% faster memory performance, and 43% to 96% faster performance when exercising both memory and disk.

1 Introduction

For over 25 years, long-term storage has been dominated by rotating magnetic media. At the beginning of the disk era, tapes were still widely used for online storage; today, they are almost exclusively used for backup despite still being cheaper than disks. The reasons are both price threshold and performance: although disks are more expensive, they are cheap enough for common use, and their performance is vastly superior.

Today, the rapidly dropping price of RAM suggests that a similar transition may soon take place, with all-electronic technologies gradually replacing disk storage. This transition is already happening in portable devices such as cameras, PDAs, and MP3 players. Because rotational delays are not relevant to persistent RAM storage, it is appropriate to consider whether existing file system designs are suitable in this new environment.

The *Conquest* file system is designed to address these questions and to smooth the transition from disk-based to persistent-RAM-based storage. Unlike other memory file systems [21, 10, 43], *Conquest* provides an incremental solution that assumes more file system responsibility in-core as memory prices decline. Unlike HeRMES [25], which deploys a relatively modest amount of persistent RAM to alleviate disk traffic, *Conquest* assumes an abundance of RAM to perform most file system functions. In essence, *Conquest* provides two specialized and simplified data paths to in-core and on-disk storage. *Conquest* achieves most of the benefits of persistent RAM without the full cost of RAM-only solutions. As persistent RAM becomes cheaply abundant, *Conquest* can realize more additional benefits incrementally.

2 Alternatives to *Conquest*

Given the promise of using increasingly cheap memory to improve file systems performance, it would be desirable to do so as simply as possible. However, the obvious simple methods for gaining such benefits fail to take complete advantage of the new possibilities. In many cases, extensions to the simple methods can give results similar to our approach, but to make these extensions, so much complexity must be added that they are no longer attractive alternatives to the *Conquest* approach.

In this section, we will discuss the limitations of these alternatives. 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, it is better to start the design with a clean slate.

✧ Gerald Popek is also associated with United On-Line.

2.1 Caching

One alternative to a hybrid RAM-based file system like *Conquest* is instead to take advantage of the existing file buffer cache. Given that a computer has an ample amount of RAM, why not just allocate that RAM to a buffer cache, rather than dedicating it to a file storage system? This approach seems especially appropriate because the buffer cache tends to populate itself with the most frequently referenced files, rather than wasting space on files that have been untouched for lengthy periods.

However, using the buffer cache has several drawbacks. Roselli et al., [34] showed that caching often experiences diminishing marginal returns as the size of cache grows larger. They also found that caches could experience miss rates as high as 10% for some workloads, which is enough to reduce performance significantly.

Another challenge is handling cache pollution, which can have a variety of causes—reading large files, buffering asynchronous writes, daily backups, global searches, disk maintenance utilities, etc. This problem led to remedies such as LFU buffer replacement for large files or attempts to reduce cache-miss latency by modifying compilers [39], placing the burden on the programmer [31], or constructing user behavior-analysis mechanisms within the kernel [15, 19].

Caches also make it difficult to maintain data consistency between memory and disk. A classic example is metadata commits, which are synchronous in most file systems. Asynchronous solutions do exist, but at the cost of code complexity [12, 38].

Moving data between disk and memory can involve remarkably complex management. For example, moving file data from disk to memory involves locating the metadata, scheduling the metadata transfer to memory, translating the metadata into runtime form, locating data and perhaps additional metadata, scheduling the data transfer, and reading the next data block ahead of time.

Conquest fundamentally differs from caching by not treating memory as a scarce resource. Instead, *Conquest* anticipates the abundance of cheap persistent RAM and uses disk to store the data well suited to disk characteristics. We can then achieve simpler disk optimizations by narrowing the range of access patterns and characteristics anticipated by the file system.

2.2 RAM Drives and RAM File Systems

Many computer scientists are so used to disk storage that we sometimes forget that persistence is not automatic. In addition to the storage medium, persistence also requires a protocol for storing and retrieving the information from the persistent medium, so that a file system can survive reboots. While persistent RAM provides nonvolatility of memory content, the file system and the memory manager also need to know how to take advantage of the storage medium.

Most RAM disk drivers operate by emulating a physical disk drive. Although there is a file system protocol for storing and retrieving the in-memory information, there is no protocol to recover the associated memory states. Given that the existing memory manager is not aware of RAM drives, isolating these memory states for persistence can be nontrivial.

RAM file systems under Linux and BSD [21] use the IO caching infrastructure provided by VFS to store both metadata and data in various temporary caches directly. Since the memory manager is unaware of RAM file systems, 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. For 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 disks to caching and temporary file systems. To move to a general-purpose persistent-RAM file system, we need a substantially new design.

2.3 Disk Emulators

Some manufacturers advocate RAM-based disk emulators for specialty applications [44]. These emulators generally plug into a standard SCSI or similar IO 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 disks, 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 IO interfaces force the emulators to operate through inadequate access methods and low-bandwidth cables, greatly limiting the utility of this option [33] as something other than a stopgap measure.

2.4 Ad Hoc Approaches

There are also a number of less structured approaches to using existing tools to exploit the abundance of RAM. For example, one could achieve persistence by manually transferring files into *ramfs* at boot time and preserving them again before shutdown. However, this method would drastically limit the total file system size.

Another option is to attempt to manage RAM space by using a background daemon to stage files to a disk partition. Although this could be made to work, it would require significant additional complexity to maintain the single name space provided by *Conquest* and to preserve the semantics of symbolic and hard links when moving files between storage media.

3 *Conquest* File System Design

Our initial design assumes the popular single-user desktop environment with 1 to 4 GB of persistent RAM, which is affordable today. As of October 2001, we can add 2 GB of battery-backed RAM to our desktop computers and deploy *Conquest* for under $200 [32]. Extending our design to other environments will be future work.

We will first present the design of *Conquest*, followed by a discussion of major design decisions.

3.1 File System Design

In our current design, *Conquest* stores all small files, metadata, executables, and shared libraries in persistent RAM; disks hold only the data content of remaining large files. We will discuss this media usage strategy further in Section 3.2.

An in-core file is stored logically contiguously in persistent RAM. Disks store only 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. On-disk allocation is done contiguously whenever possible in temporal order, similar to LFS [35] but without the need to perform continuous disk cleaning in the background.

For each directory, *Conquest* maintains a variant of an extensible hash table for its file metadata entries, with file names as keys. Hard links are supported by allowing multiple names (potentially under different directories) to hash to the same file metadata entry.

RAM storage allocation uses existing mechanisms in the memory manager when possible to avoid duplicate functionality. For example, the storage manager is relieved of maintaining a metadata allocation table and a free list by using the memory address of the file metadata as its unique ID.

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

Unlike caching, RAM drives, and RAM file systems, *Conquest* memory is the final storage destination for many files and all metadata. We can access the critical path of *Conquest*'s main store without disk-related complexity in data duplication, migration, translation, synchronization, and associated management. Unlike RAM drives and RAM file systems, *Conquest* provides persistence and storage capacity beyond the size limitation of the physical main store.

3.2 Media-Usage Strategy

The first major design decision of *Conquest* is the choice of which data to place on disk, and the answer depends on the characteristics of popular workloads. Recent studies [9, 34, 42] independently confirm the often-repeated observations [30]:

- Most files are small.
- Most accesses are to small files.
- Most storage is consumed by large files, which are, most of the time, accessed sequentially.

Although one could imagine many complex data-placement algorithms (including LRU-style migration of unused files to the disk), we have taken advantage of the above characteristics by using a simple threshold to choose which files are candidates for disk storage. Only the data content of files above the threshold (currently 1 MB) are stored on disk. Smaller files, as well as metadata, executables, and libraries, are stored in RAM. The current choice of threshold works well, leaving 99% of all files in RAM in our tests. 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 layout. For example, we can avoid adding complexity to handle fragmentation with "large" and "small" disk blocks, as in FFS [20]. 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 files in RAM.

3.2.1 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-related overheads. Since persistent RAM access granularity is byte-oriented rather than block-oriented, a single-byte access can be six orders of magnitude faster than accessing disk [23].

Metadata no longer have dual representations, one in memory and one on disk. The removal of the disk representation also removes the complex synchronous or asynchronous mechanisms needed to propagate the metadata changes to disk [20, 12, 38], and avoids translation between the memory and disk representations.

At this time, *Conquest* does not give special treatment to executables and shared libraries by forcing them into memory, but we anticipate benefits from doing so. In-place execution will reduce startup costs and the time involved in faulting pages into memory during execution. Since shared libraries are modular extensions of executables, we intend to store them in-core as well.[1]

3.2.2 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 a small transfer size makes mechanical overheads significant, designers employ 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 on disk to ensure a minimum number of indirections before locating a small file [26]. Methods to reduce the seek time and rotational latency [20] are other attempts to speed up small-file accesses.

Small files introduce significant storage overhead because optimal disk-access granularities tend to be large and fixed, causing excessive internal fragmentation. Although reducing the access granularity necessitates higher overhead and lower disk bandwidth, the common remedy is nevertheless to introduce sub-granularities and extra management code to handle small files.

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.

[1] Shared libraries can be trivially identified through magic numbers and existing naming and placement conventions.

3.3 Metadata Representation

How file system metadata is handled is critical, since this information is in the path of all file accesses. Below, we outline how *Conquest* optimizes behavior by its choices of metadata representation.

3.3.1 In-Core File Metadata

One major simplification of our metadata representation is the removal of nested indirect blocks from the commonly used *i*-node design. *Conquest* stores small files, metadata, executables, and shared libraries in persistent RAM, via uniform, single-level, dynamically allocated index blocks, so in-core data blocks are virtually contiguous.

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 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.

To avoid file metadata management, we use the memory addresses of the *Conquest* file metadata as unique IDs. By leveraging the existing memory management code, this approach ensures unique file metadata IDs, no duplicate allocation, and fast retrieval of the file metadata. The downside of this decision is that we may need to modify the memory manager to anticipate that certain allocations will be relatively permanent.

For small 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, though we plan to add automatic truncation as a future optimization. *Conquest* also supports "holes" within a file, since they are commonly seen during compilation and other activities.

3.3.2 Directory Metadata

We used a variant of extensible hashing [11] for our directory representation. The directory structure is built with a hierarchy of hash tables, using file names as keys. Collisions are resolved by splitting (or doubling) hash indices and unmasking an additional hash bit for each key. A path is resolved by recursively hashing each name component of the path at each level of the hash table.

Compared to *ext2*'s approach, hashing removes the need to compact directories that live in multiple (possibly indirect) blocks. Also, the use of hashing easily supports hard links by allowing multiple names to hash to the same file metadata entry. In addition, extendible hashing preserves the ordering of hashed items when changing the table size, and this property allows `readdir()` to walk through a directory correctly while resizing a hash table (e.g., recursive deletions).

One concern with using extensible hashing is the wasted indices due to collisions and subsequent splitting of hash indices. However, we found that alternative compact hashing schemes would consume similar amount of space to preserve ordering during a resize operation.

3.3.3 Large-File Metadata

For the data content of large files on disk, *Conquest* currently maintains a dynamically allocated, doubly linked list of segments to keep track of disk storage locations. Disk storage is allocated contiguously whenever possible, in temporal, or LFS, order [35].

Although we have a linear search structure, its simplicity and in-core speed outweigh its algorithmic inefficiency, as we will demonstrate in the performance evaluation (Section 5). In the worst case of severe disk fragmentation, we will encounter a linear slowdown in traversing the metadata. However, given that we have coarse disk-management granularity, the segment list is likely to be short. Also, since the search is in-core but access is limited by disk bandwidth, we expect little performance degradation for random accesses to large files.

Currently, we store the large-file data blocks sequentially as the write requests arrive, without regard to file membership. We chose this temporal order only for simplicity in the initial implementation. Unlike LFS, we keep metadata in-core, and existing file blocks are updated in-place as opposed to appending various versions of data blocks to the end of the log. Therefore, *Conquest* does not consume contiguous regions of disk space nearly as fast as LFS, and demands no continuous background disk cleaning.

Still, our eventual goal is to apply existing approaches from both video-on-demand (VoD) servers and traditional file systems research to design the final layout. For example, given its sequential-access nature, a large media file can be striped across disk zones, so disk scanning can serve concurrent accesses more effectively [8]. Frequently accessed large files can be stored completely near outer zones for higher disk bandwidth. Spatial and temporal ordering can be applied within each disk zone, at the granularity of an enlarged disk block.

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, IO buffering and simple predictive prefetching methods should still be able to deliver good read bandwidth.

3.4 Memory Management

Although it reuses the code of the existing memory manager, *Conquest* has its own dedicated instances of the manager, residing completely in *Conquest*, with each governing its own memory region. Since all references within a *Conquest* memory manager are encapsulated within its governed region, and each region has its own dedicated physical address space, we can save and restore the runtime states of a *Conquest* memory manager directly in-core without serialization and deserialization.

Conquest avoids memory fragmentation by using existing mechanisms built in various layers of the memory managers under Linux. For sub-block allocations, the slab allocator compacts small memory requests according to object types and sizes [4]. For block-level allocations, memory mapping assures virtual contiguity without external fragmentation.

In the case of in-core storage depletion, we have several options. The simplest handling is to declare the resource depleted, which is our current approach (the same as is used for PDAs). However, under *Conquest*, this option implies that storage capacity is now limited by both memory and disk capacities. Dynamically adjusting the in-core storage threshold is another possibility, but changing the threshold can potentially lead to a massive migration of files. Our disk storage is potentially threatened with smaller-than-expected files and associated performance degradation.

3.5 Reliability

Storing data in-core inevitably raises the question 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, which allows a greater risk of corruption. A single wild kernel pointer could easily destroy many important files. However, a study conducted at the University of Michigan has shown that the risk of data corruption due to kernel failures is less than one might expect. Assuming one system crash every two months, one can expect to lose in-memory data about once a decade [27].

Another threat to the reliability of an in-memory file system is the hardware itself. Modern disks have a mean time between failures (MTBF) of 1 million hours [37]. Two hardware components, the RAM and the battery backup system, cause *Conquest*'s MTBF to be different from that of a disk. In our prototype, we use 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 [14, 36]. The MTBF of the RAM is comparable to disk [22]; 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 *Conquest* should lose data due to hardware failures at most once every few years. This is well within the range that users find acceptable in combination with standard backup procedures.

At the implementation level, an extension is to use approaches similar to Rio [7], which allows volatile memory to be used as a persistent store with little overhead. For metadata, we rely heavily on atomic pointer commits. In the event of crashes, the system integrity can remain intact, at the cost of potential memory leaks (which can be cleaned by fsck) for in-transit memory allocations.

In addition, we can still apply the conventional techniques of sandboxing, access control, checkpointing, fsck, and object-oriented self-verification. For example, *Conquest* still needs to perform system backups. *Conquest* uses common memory protection mechanisms by having a dedicated memory address space for storage (assuming a 64-bit address space). A periodic fsck is still necessary, but it can run at memory speed. We are also exploring the object-store approach of having a "typed" memory area, so a pointer can be verified to be of a certain type before being accessed.

3.6 64-Bit Addressing

Having a dedicated physical address space in which to run *Conquest* significantly reduces the 32-bit address space and raises the question of 64-bit addressing. However, our current implementation on a 32-bit machine demonstrates that 64-bit addressing implications are largely orthogonal to materializing *Conquest*, although a wide address space does offer many future extensions (i.e., having distributed *Conquest* sharing the same address space, so pointers can be stored directly and transferred across machine boundaries as in [6].)

4 *Conquest* Implementation Status

The *Conquest* prototype is operational as a loadable kernel module under Linux 2.4.2. The current implementation follows the VFS API, but we need to override generic file access routines at times to provide both in-core and on-disk accesses. For example, inside the read routine, we assume that accessing memory is the common case, while providing a forwarding path for disk accesses. The in-core 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 and various internal fragmentation and seek-time optimizations for small files.

Because we found it relatively difficult to alter the VFS to not cache metadata, we needed to pass our metadata structures through VFS calls such as *mknod*, *unlink*, and *lookup*. We altered the VFS, so that the *Conquest* root node and metadata are not destroyed at *umount* times.

We modified the Linux memory manager in several ways. First, we introduced *Conquest* zones. With the flexibility built into the Linux zone allocator, it is feasible to allocate unused *Conquest* memory within a zone to perform other tasks such as IO buffering and program execution. However, we chose to manage memory at the coarser grain of zones, to conserve memory in a simpler way.

The *Conquest* memory manager is instantiated top-down instead of bottom-up, meaning *Conquest* uses high-level slab allocator constructs to instantiate dedicated *Conquest* slab managers, then lower-level zone and page managers. By using high-level constructs, we only need to build an instantiation routine, invoked at file system creation times.

Since *Conquest* managers reside completely in the memory region they govern, runtime states (i.e., pointers) of *Conquest* managers can survive reboots with only code written for reconnecting several data structure entry points back to *Conquest* runtime managers. No pointer translation was required.

Conquest is POSIX-compliant and supports both in-core and on-disk storage. We use a 1-MB static dividing line to separate small files from large files (Section 3.2). Large files are stored on disk in 4-KB blocks, so that we can use the existing paging and protection code without alterations. An optimization is to enlarge the block size to 64 KB or 256 KB for better performance.

5 *Conquest* Performance

We compared *Conquest* with *ext*2 [5], *reiserfs* [26], *SGI XFS* [40], and *ramfs* by Transmeta. We chose *ext2*, *reiserfs*, and *SGI XFS* largely because they are the common basis for various performance comparisons. Note that with 2 Gbytes of physical RAM, these disk-based file systems use caching extensively, and our performance numbers reflect how well these file systems exploit memory hardware. In the experiments, all file systems have the same amount of memory available as *Conquest*.

Ramfs by Transmeta uses the page cache and *v*-nodes to store the file system content and metadata directly, and *ramfs* provides no means of achieving data persistence after a system reboot. Given that both *Conquest* and *ramfs* are under the VFS API and various OS legacy constraints, *ramfs* should approximate the practical achievable bound for *Conquest* performance. Our experimental platform is described in Table 5.1. Various file system settings are listed in Table 5.2.

	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
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 5.1: Experimental platform.

	File system settings
cfs	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 5.2: File system settings.

We used the Sprite LFS microbenchmarks [35]. As for macrobenchmarks, the most widely used in the file system literature is the Andrew File System Benchmark [16]. Unfortunately, this benchmark no longer stresses modern file systems because its data set is too small. Instead, we present results from the PostMark macrobenchmark[2] [18] and our modified PostMark macrobenchmark, which is described in Section 5.3. All results are presented at a 90% confidence level.

5.1 Sprite LFS Microbenchmarks

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

5.1.1 Small-File Benchmark

The small-file benchmark measures the latency of file operations, and consists of creating, reading, and unlinking 10,000 1-KB files, in three separate phases. Figure 5.1 summarizes the results.

***Conquest* vs. *ramfs*:** Compared to *ramfs*, *Conquest* incurs 5% and 13% overheads in file creation and deletion respectively, because *Conquest* maintains its own metadata and hashing data structures to support persistence, which is not provided by *ramfs*. Also, we have not removed or disabled VFS caching for metadata; therefore, VFS needs to go through an extra level of indirection to access *Conquest* metadata at times, while *ramfs* stores its metadata in cache.

Nevertheless, *Conquest* has demonstrated a 15% faster read transaction rate than *ramfs*, even when *ramfs* is performing at near-memory bandwidth. *Conquest* is able to improve this aspect of performance because the critical path to the in-core 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 synchronous metadata manipulations. 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 both journaling overheads and their in-memory behaviors. *Reiserfs* actually achieved poorer performance with its original default settings. Interestingly, *reiserfs* performs better with the *notail* option, which disables certain disk optimizations for small files and the fractional block at the end of large files. While the intent of these disk optimizations is to save extra disk accesses, their overhead outweighs the benefits when there is sufficient memory to buffer disk accesses.

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

As for *SGI XFS*, its original default settings also produced poorer performance, since journaling consumes the log buffer quite rapidly. As we increased the 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.

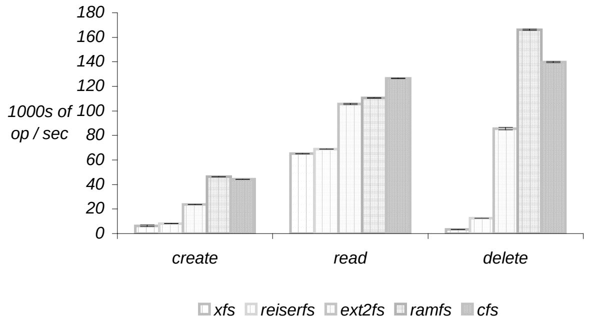


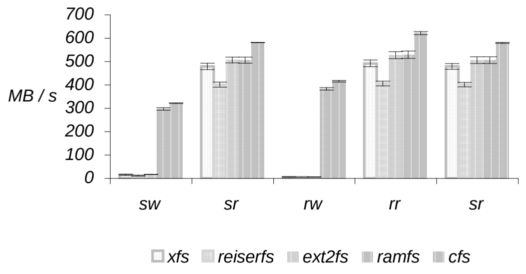
Figure 5.1: Transaction rate for the different phases of the Sprite LFS small-file benchmark, run over *SGI XFS*, *reiserfs*, *ext2*, *ramfs*, and *Conquest*. The benchmark creates, reads, and unlinks 10,000 1-KB files in separate phases. In this and most subsequent figures, the 90% confidence bars are nearly invisible due to the narrow confidence intervals.

5.1.2 Large-File Benchmark

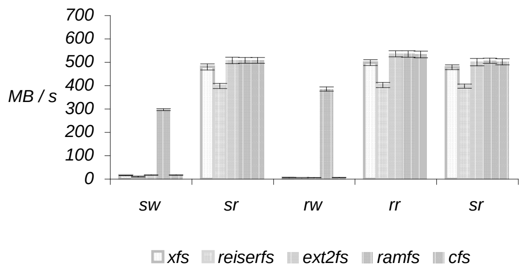
The large-file benchmark writes a large file sequentially (with flushing), reads from it sequentially, and then writes a new large file randomly (with flushing), reads it randomly, and finally reads it sequentially. The final read phase was originally designed to measure sequential read performance after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

For *Conquest* on-disk files, we altered the large-file benchmark to perform each phase of the benchmark on forty 100-MB files before moving to the next phase. Since we have a dividing line between small and large files, we also investigated the sizes of 1 MB and 1.01 MB, with each phase of benchmark performed on ten 1-MB or 1.01-MB files. In addition, we memory-aligned all random accesses to reflect real-world usage patterns.

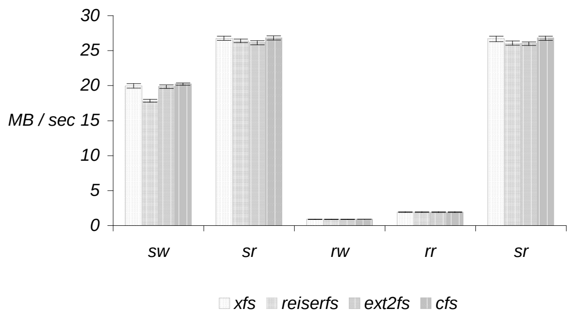
The 1-MB benchmark: The 1-MB large-file benchmark measures the throughput of *Conquest*'s in-core files (Figure 5.2a). Compared to *ramfs*, *Conquest* achieves 8% higher bandwidth in both random and sequential writes and 15% to 17% higher bandwidth in both random and sequential reads. It is interesting to observe that random memory writes and reads are faster than corresponding sequential accesses. This is because of cache hits: for 1-MB memory accesses with a 256-KB L2 cache size, random accesses have a roughly 25% chance of reusing the L2 cache content. We believe that the difference is larger for writes because of a write-back, write-allocate L2 cache design, which incurs additional overhead on sequential writes of large amounts of data.



(a) Sprite LFS large-file benchmark for 1-MB (in-core *Conquest*) files.



(b) Sprite LFS large-file benchmark for 1.01MB (on-disk *Conquest*) files.



(c) Sprite LFS large-file benchmark for 100-MB (on-disk *Conquest*) files.

Figure 5.2: Bandwidth for the different phases (sequential write, sequential read, random write, random read, sequential read) of the Sprite LFS large-file benchmarks, run over *SGI XFS*, *reiserfs*, *ext2*, *ramfs, and Conquest.* These two tests compare the performance of in-core and on-disk files under *Conquest*.

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*.

The 1.01-MB benchmark: The 1.01-MB large-file benchmark shows the performance effects of switching a file from memory to disk under *Conquest* (Figure 5.2b). *Conquest* disk performance matches the performance of cached *ext2* pretty well. In our design, in-core and on-disk data accesses use disjoint data paths wherever possible, so *Conquest* imposes little or no extra overhead for disk accesses.

The 100-MB benchmark: The 100-MB large-file benchmark measures the throughput of *Conquest* on-disk files (Figure 5.2c). We only compared against disk-based file systems because the total size exercised by the benchmark exceeds the capacity of *ramfs*. All file systems demonstrate similar performance. Compared to cached *ext2*, *Conquest* shows only 8% and 4% improvements in sequential and random writes. We expect further performance improvements after enlarging the block size to 64 KB or 256 KB.

5.2 PostMark Macrobenchmark

The PostMark benchmark was designed to model the workload seen by Internet service providers [18]. Specifically, the workload is meant to simulate a combination of electronic mail, netnews, and web-based commerce transactions.

PostMark creates a set of files with random sizes within a set 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 and can be biased via parameter settings. The sizes of these files are chosen at random and are uniformly distributed over the file size range. A deletion operation removes a file from the active set. A read operation reads a randomly selected file in entirety. An append operation opens a random file, seeks to the end of the file, and writes a random amount of data, not exceeding the maximum file size.

We initially ran our experiments using the configuration of 10,000 files with a size range of 512 bytes to 16 KB. 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 varied the total number of files from 5,000 to 30,000 to see the effects of scaling.

Another adjustment of the default setting is the assumption of a single flat directory. Since it is unusual to store 5,000 to 30,000 files in a single directory, we reconfigured PostMark to use one subdirectory level to distribute files uniformly, with the number of directories equal to the square root of the file set size. 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 does not exercise the disk aspect of the *Conquest* file system. Also, since this configuration specifies an average file set of only 250 MB, which fits comfortably in 2 GB of memory, this benchmark compares the memory performance of *Conquest* against the performance of existing cache and IO buffering mechanisms, under a realistic mix of file operations.

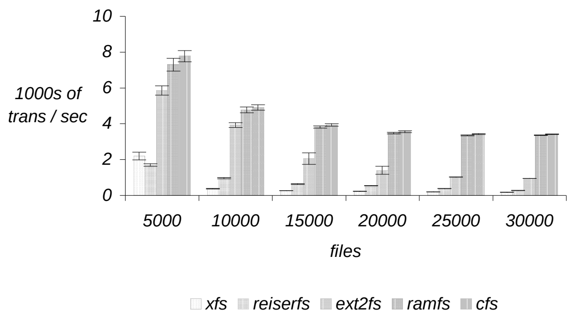


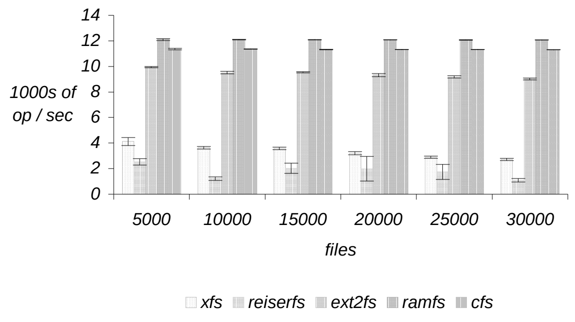
Figure 5.3: PostMark transaction rate for *SGI XFS*, *reiserfs*, *ext2*, *ramfs*, and *Conquest*, varying from 5,000 and 30,000 files. The results are averaged over five runs.

Figure 5.3 compares the transaction rates of *Conquest* with various file systems as the number of files is varied from 5,000 to 30,000. First, the performance of *Conquest* differs little from *ramfs* performance. We feel comfortable with *Conquest*'s performance at this point, given that we still have room to reduce costs for at least sequential writes (enlarging the disk block size). *Conquest* outperforms *ext2* significantly; the performance gap widens from 24% to 350% as the number of files increases. *SGI XFS* and *reiserfs* perform slower than *ext2* due to journaling overheads.

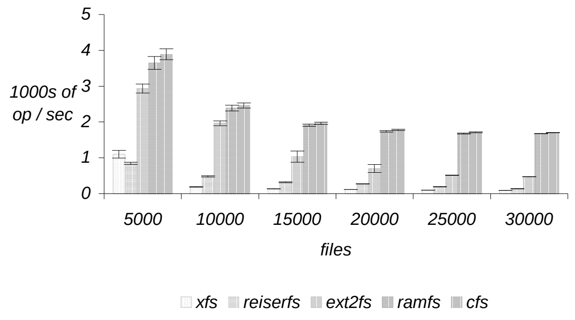
For space reasons, we have omitted other graphs with similar trends—bandwidth, average creation rate, read rate, append rate, and average deletion rate.

Taking a closer look at the file-creation component of the performance numbers, we can see that without interference from other types of file transactions (Figure 5.4a), file creation rates show little degradation for all systems as the number of files increases. When mixed with other types of file transactions (Figure 5.4b), file creation rates degrade drastically.

With only file creations, *Conquest* creates 9% fewer files per second than *ramfs*. However, when creations are mixed with other types of file transactions, *Conquest* creates files at a rate comparable to *ramfs*.



(a) PostMark file creation rate.



(b) PostMark file creation rate, mixed with other types of file transactions.

Figure 5.4: PostMark file creation performance for *SGI XFS*, *reiserfs*, *ext2*, *ramfs*, and *Conquest*, varying from 5,000 and 30,000 files. The results are averaged over five runs.

Compared to *ext2*, *Conquest* performs at a 26% faster creation rate (Figure 5.4a), compared to the 50% faster rate in the LFS Sprite benchmark. *Ext2* has a better creation rate under PostMark because files being created have larger file sizes. The write buffer used by *ext2* narrows the performance difference of file creation when compared to *Conquest*.

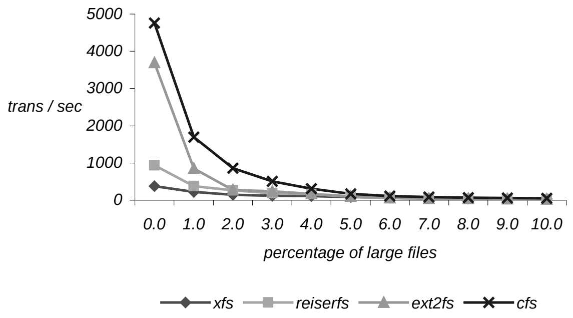
Similar to the comparison between *Conquest* and *ramfs*, it is interesting to see that *SGI XFS* has a faster file creation rate than *reiserfs* without mixed traffic, but a slower rate than *reiserfs* with mixed traffic. This result demonstrates that optimizing individual operations in isolation does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

We have omitted the graphs for file deletion, since they show similar trends.

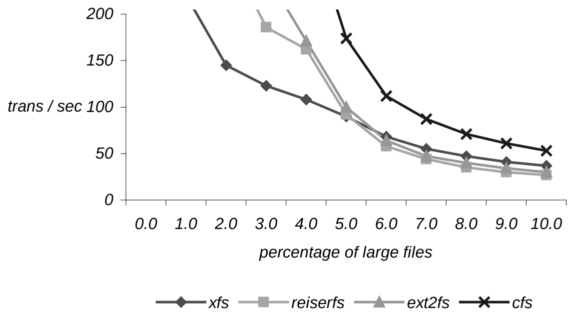
5.3 Modified Postmark Benchmark

To exercise both the memory and disk components of *Conquest*, we modified the Postmark benchmark in the following way. We generated a percentage of files in a large-file category, with file sizes uniformly distributed between 2 MB and 5MB. The remaining files were uniformly distributed between 512 bytes to 16 KB. We fixed the total number of files at 10,000 and varied the percentage of large files from 0.0 to 10.0 (0 GB to 3.5 GB). Since the file set exceeds the storage capacity of *ramfs*, we were forced to omit *ramfs* from our results.

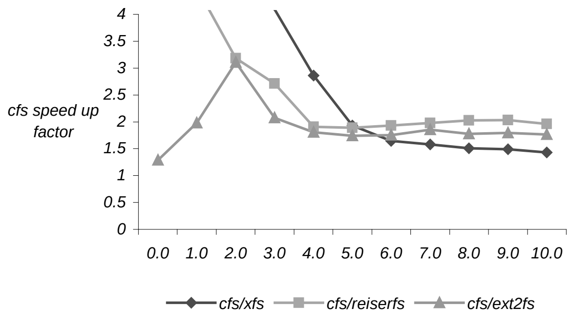
Figure 5.6 compares the transaction rate of *SGI XFS*, *reiserfs*, *ext2*, and *Conquest*. Figure 5.6a shows how the measured transaction rates of the four file systems vary as the percentage of large files increases. Because the scale of this graph obscures important detail at the right-hand side, Figure 5.6b zooms into the graph with an expanded vertical scale. Finally, Figure 5.6c shows the performance ratio of *Conquest* over other disk-based file systems.



(a) The full-scale graph.



(b) The zoomed graph.



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

Figure 5.6: 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 5.6c). 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 75% faster transaction rate than *ext2*..

Both *SGI XFS* and *reiserfs* show significantly slower memory performance (left side of Figure 5.6a). However, as the file set exceeds the memory size, *SGI XFS* starts to outperform *ext2* and *reiserfs* (Figure 5.6c). Clearly, different file systems are optimized for different conditions.

6 Related Work

The database community has a long established history of memory-only systems. An early survey paper reveals key architectural implications of sufficient RAM and identifies several early main memory databases [13]. 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.

In the operating system arena, one early use of persistent RAM was for buffering write requests [2]. Since dirty data were buffered in persistent memory, the interval between synchronizations to the disk could be lengthened.

The Rio file cache [28] combines UPS, volatile memory, and a modified write-back scheme to achieve the reliability of write-through file cache and performance of pure write-back file cache (with no reliability-induced writes to disk). The resiliency offered by Rio complements *Conquest*'s performance well. While *Conquest* uses main store as the final storage destination, Rio's BIOS *safe sync* mechanism provides a high assurance of dumping *Conquest* memory to disk in the event of infrequent failures that require power cycles.

Persistent RAM has been gaining acceptance as the primary storage medium on small mobile computing devices through a plethora of flash-memory-based file systems [29, 41]. Although this departure from disk storage marks a major milestone toward persistent-RAM-based storage, flash memory has some unpleasant characteristics, notably the limited number of erase-write cycles and slow (second-range) time for storage reclamation. These characteristics cause performance problems and introduce a different kind of operating system complexity. Our research currently focuses on the general performance characteristics exemplified by battery-backed DRAM (BB-DRAM).

The leading PDA operating systems, PalmOS and Windows CE, deliver memory and file system services via BB-DRAM, but both systems are more concerned with fitting an operating system into a memory-constrained environment, in contrast to the assumed abundance of persistent RAM under *Conquest*. PalmOS lacks a full-featured execution model, and efficient methods for accessing large data objects are limited [1]. Windows CE is unsuitable for general desktop-scale deployment because it tries to shrink the full operating system environment to the scale of a PDA. Also, the Windows CE architecture inherits many disk-related assumptions [24].

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, conventional roles of memory acting as the cache for disk content still apply, and disks are still the persistent storage medium for files [17].

One form of persistent RAM under development is Magnetic RAM (MRAM) [3]. An ongoing project on MRAM-enabled storage, HeRMES, also takes advantage of persistent RAM technologies [25]. 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.

7 Lessons Learned

Through the design and implementation of *Conquest*, we have learned the following major lessons:

First, the handling of disk characteristics permeates file system design even at levels above the device layer. For example, default VFS routines contain readahead and buffer-cache mechanisms, which add high and unnecessary overheads to low-latency main store. Because we needed to bypass these mechanisms, building *Conquest* was much more difficult than we initially expected. For example, certain downstream storage routines anticipate data structures associated with disk handling. We either need to find ways to reuse these routines with memory data structures, or construct memory-specific access routines from scratch.

Second, file systems that are optimized for disk are not suitable for an environment where memory is abundant. For example, *reiserfs* and *SGI XFS* do not exploit the speed of RAM as well as we anticipated. Disk-related optimizations impose high overheads for in-memory accesses.

Third, 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 results are not automatic. For example, if the cache footprint of two specialized data paths exceeds the size of a single generic data path, the resulting performance can go in either direction, depending on the size of the physical cache.

Fourth, 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, we were surprised to see *Conquest* outperform *ramfs* by 15% in read bandwidth, knowing that storage data paths are already heavily optimized.

Finally, 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 overall lesson that can be drawn is that seemingly simple changes 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.

8 Future Work

Conquest is now operational, but we can further improve its performance and usability in a number of ways. A few previously mentioned areas are designing mechanisms for adjusting file size threshold dynamically (Section 3.4) and finding a better disk layout for large data blocks (Section 3.3.3).

High-speed in-core storage also opens up additional possibilities for operating systems. *Conquest* provides a simple and efficient way for kernel-level code to access a general storage service, which is conventionally either avoided entirely or achieved through the use of more limited buffering mechanisms. One major area of application for this capability would be system monitoring and lightweight logging, but there are numerous other possibilities.

In terms of research, so far we have aggressively removed many disk-related complexities from the in-core critical path without questioning exactly how much each disk optimization adversely affects file system performance. One area of research is to break down these performance costs, so designers can improve the memory performance for disk-based file systems.

Memory under *Conquest* is a shared resource among execution, storage, and buffering for disk access. Finding the "sweet spot" for optimal system performance will require both modeling and empirical investigation. In addition, after reducing the roles of disk storage, *Conquest* exhibits different system-wide performance characteristics, and the implications can be subtle. For example, the conventional wisdom of mixing CPU- and IO-bound jobs may no longer be a suitable scheduling policy. We are currently experimenting with a wider variation of workloads to investigate a fuller range of *Conquest* behavior.

9 Conclusion

We have presented *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, we measure 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 ad hoc approaches not only fail to provide a complete solution, but perform 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.

The benefits of *Conquest* arose from rethinking basic file system design assumptions. This success suggests that the radical changes in hardware, applications, and user expectations of the past decade should also lead us to rethink other aspects of operating system design.

10 Acknowledgements

We would like to thank our shepherd Darrell Anderson and the anonymous reviewers who have offered invaluable suggestions to strengthen this paper. We would also like to thank Michael Gorlick and Richard Guy for reviewing an early presentation of the *Conquest* performance results and offering useful insights. In addition, we want to thank Mark Yarvis, Scott Michel, and Janice Wheeler for commenting on earlier drafts of this paper. This work was supported by the National Science Foundation under Grant No. CCR-0098363.

11 References

[1] 3COM. Palm OS® Programmer's Companion. http://www.palmos.com (under site map and documentation), 2002.

[2] 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.

[3] 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.

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

[5] Card R, Ts'o T, Tweedie S. Design and Implementation of the Second Extended Filesystem. *The HyperNews Linux KHG Discussion*. http://www.linuxdoc.org (search for ext2 Card Tweedie design), 1999.

[6] 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.

[7] 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.

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

[9] 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*, May 1999.

[10] Douglis F, Caceres 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.

[11] Fagin R, Nievergelt J, Pippenger N, Raymond Strong H. Extensible hashing—a fast access method for dynamic files, *ACM Transactions on Database Systems*, 4(3) pp. 315-344, 1979.

[12] 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.

[13] Garcia-Molina H, Salem K. *IEEE Transactions on Knowledge and Data Engineering*, 4(6) pp. 509-516, December 1992.

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

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

[16] 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.

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

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

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

[20] 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.

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

[22] 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.

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

[24] Microsoft. MSDN Online Library, http://msdn.microsoft.com/library (under embedded development and Windows CE), 2002.

[25] 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.

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

[27] Ng WT, 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.

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

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

[30] 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, December 1985.

[31] 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.

[32] Price Watch. New Computer Components. http://www.pricewatch.com, 2001.

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

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

[35] 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.

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

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

[38] 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.

[39] 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.

[40] 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.

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

[42] Vogels W. File System Usage in Windows NT 4.0. *Proceedings of the 17th Symposium on Operating Systems Principles*, December 1999.

[43] 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.

[44] 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.