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

Conquest is a disk/persistent-RAM hybrid file system that is incrementally deployable and realizes most of the benefits of cheaply abundant persistent RAM. Conquest consists of two specialized and simplified data paths for in-core and on-disk storage and outperforms popular disk-based file systems by 43% to 97%.

1 Introduction

The declining cost of persistent RAM (e.g., battery-backed RAM) makes its use as persistent storage feasible in ordinary computers. Desktops will soon be able to have 4 to 10 GB of persistent RAM, enough for most file-system services, except high-capacity storage.

The *Conquest* file system is designed to make good use of both persistent-RAM-based storage and of disks. Unlike existing RAM file systems, *Conquest*'s storage capacity is not limited by the size of persistent RAM, and *Conquest* can incrementally assume more responsibility for in-core storage as memory prices decline. Unlike caching, which treats main memory as a scarce resource, *Conquest* anticipates the abundance of cheap persistent RAM and uses memory as the final storage destination. *Conquest* uses disks to store only the data well suited for disk characteristics, which also allows simpler disk management. Unlike HeRMES [5], 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. Unlike ad hoc approaches that would provide only partial solutions, *Conquest* retains the semantics of a single file system (e.g., name space, hard links, etc.) while providing significant performance gains.

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

2 *Conquest* Design

The *Conquest* file system consists of two specialized and simplified data paths for in-core and on-disk storage. *Conquest* assumes a single-user desktop environment with 1 to 4 GB of persistent RAM, which is affordable today.

2.1 File System Design

Conquest stores small files, metadata, executables, and shared libraries in persistent RAM; disks hold only the data content of remaining large files. An in-core file is stored contiguously (in the virtual sense) in persistent RAM, accessed by a single pointer to the data and a file offset. When possible, a file is resized by remapping rather than copying.

Disks store only the data content of large files, reducing management overhead. *Conquest* maintains a per-large-file segment table in persistent RAM. On-disk allocation is contiguous whenever possible.

For each directory, *Conquest* maintains a dynamically allocated hash table of metadata entries, keyed by file names. Multiple names (potentially under different directories) can hash to the same entry, providing hard links.

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

Paging and swapping are disabled for *Conquest* memory, but enabled for the non-*Conquest* memory region.

2.2 Major Design Considerations

Media usage strategy: Recent studies [3, 8] confirm the often-repeated observations [7]: (1) Most files are small; (2) Most accesses are to small files; and (3) Most storage is consumed by large files, which are accessed sequentially most of the time.

Therefore, we use a simple threshold to choose which files to store on disk instead of complex data placement algorithms (e.g., LRU-style migration of unused files to disk). The data content of files above the threshold (currently 1 MB) are stored on disk. Smaller files, metadata, executables, and libraries are stored in RAM. The current arbitrarily chosen threshold keeps 99% of all files in RAM. Future research will determine the optimum threshold and examine alternative approaches.

The threshold simplifies the code without wasting memory, since small files do not consume a large amount of total space. Accesses to small files and metadata incur no data duplication or disk-related overhead. For the large-file-only disk storage, we use a larger access granularity to reduce the seek-time overhead. Because most accesses to large files are sequential, we relax many historical disk design constraints, such as complex layout heuristics that reduce fragmentation or average seek times.

Metadata representation: *Conquest* does not use the *v*-node data structure provided by VFS to store metadata, because the *v*-node is designed to accommodate widely varying file systems. *Conquest* does not need many *v*-node mechanisms such as metadata caching. *Conquest*'s file metadata consists of only the fields (53 bytes) required for POSIX conformance.

The large-file data blocks are currently stored on disk 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 [9], we keep metadata in-core, and existing file blocks are updated in-place instead of appending data-block versions to the log's end. Therefore, *Conquest* does not consume contiguous regions of disk space as fast as LFS, and demands no continuous background disk cleaning. We plan to apply approaches from video-on-demand servers and traditional file systems research for the final layout.

Memory management: Although it reuses the existing memory manager's code, *Conquest* governs its memory region with its own instance of the manager, whose data resides persistently inside *Conquest*'s dedicated physical address space. All references within this memory manager are inside the *Conquest* region, so we can save and restore the manager's runtime states directly in-core without serialization and deserialization.

Conquest avoids memory fragmentation by using existing Linux memory-manager mechanisms. For sub-block allocations, the slab allocator compacts small memory requests according to object types and sizes [1]. For block-level allocations, memory mapping assures virtual contiguity without external fragmentation.

Reliability: Disk storage is considered less vulnerable to software failure corruption because it is less likely to perform illegal operations through its rigid interface. However, [6] has shown that the risk of data corruption due to kernel failures is low. Assuming one system crash every two months, one would lose in-memory data about once a decade.

We currently rely on atomic pointer commits. In the event of crashes, the system integrity remains intact, at the cost of potential memory leaks (which can be cleaned by fsck) for in-transit memory allocations. We also plan to use approaches similar to Rio [2], which allows volatile memory to be used as a persistent store with little overhead. Conventional techniques of access control, system backup, and fsck also apply.

64-bit Architecture: Our current implementation on a 32-bit machine demonstrates that 64-bit addressing implications are largely orthogonal to *Conquest*, although a wide address space does offer the opportunity for future extensions.

3 *Conquest* Performance

The *Conquest* prototype is a POSIX-compliant loadable kernel module under Linux 2.4.2, supporting both in-core and on-disk storage. Experiments were conducted on a Dell PowerEdge 4400, with 1 GHz 32-bit Xeon Pentium and 2 GB of memory. The machine uses a 73.4 GB Seagate disk (ST173404LC) connected through a SCSI interface.

We compared *Conquest* with *ext2 (0.5b)*, *reiserfs (3.6.25)*, *SGI XFS (1.0)*, and *ramfs* by Transmeta. Unless specified, file systems were configured with default options. *Reiserfs* was configured with the *notail* option, and *SGI XFS* was mounted with eight 32-MB buffers for logging.

Note that *ramfs* cannot be used as persistent storage because it stores data and metadata in temporary caches under VFS, which cannot survive reboots. We compared *Conquest* to *ramfs* because *ramfs* approximates the achievable bound for file system performance within VFS legacy constraints,.

Sprite LFS microbenchmarks: The small-file benchmark consists of creating, reading, and unlinking 10,000 1-KB files in three phases [9]. *Conquest* shows 5% and 13% slower transaction rates in file creation and deletion than *ramfs* because *Conquest* has not yet been tuned. However, *Conquest* has a 15% faster read transaction rate than *ramfs*. *Conquest* is faster because the critical path to the in-core data path contains no generic disk-related code built into VFS, such as checking for cache status. Compared to disk-based file systems, *Conquest* is at least 50% faster for creation and deletion, and 19% faster for read.

We have altered the original large-file benchmark to perform sequential and random access operations on a set of large files, rather than a single file. For ten 1-MB (*Conquest* in-core) files, *Conquest* demonstrates an 8 to 16% bandwidth improvement over *ramfs* in reads and up to 8% in writes. Compared to disk-based file systems, *Conquest* demonstrates at least 800% improvement in sequential writes, 2800% in random writes, and 8 to 16% in reads.

For 40 100-MB (*Conquest* on-disk) files, the working set size exceeds the memory size, so *ramfs* is omitted from our comparison. Compared to disk-based file systems, *Conquest* shows 4 to 8% faster disk accesses.

PostMark macrobenchmark: The PostMark benchmark models the workload of Internet service providers by simulating a combination of electronic mail, netnews, and web-based commerce transactions [4].

We ran experiments with a size range of 512 bytes to 16 KB. Each experiment performs 200,000 transactions with equally probable creates and deletes, and reads four times more probable than appends. The transaction block size is 512 bytes. We varied the total number of files from 5,000 to 30,000. We configured 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.

Unoptimized *Conquest* performs 1 to 2% better than *ramfs*. It outperformed disk-based file systems by 24% to 350% as the number of files increased from 5,000 to 30,000.

Modified PostMark macrobenchmark: To exercise both the in-core and the on-disk components of *Conquest*, we modified the PostMark benchmark. We generated a percentage of files in a large-file category, with file sizes uniformly distributed between 2 MB and 5 MB, the total number of files fixed at 10,000, and the percentage of large files varying from 0.0 to 10.0 (0 GB to 3.5 GB). Again, since the working set exceeds the storage capacity of *ramfs*, we omit *ramfs* from our results.

Without any disk traffic (0% large files), *Conquest* is 1200% faster than *SGI XFS* in transaction rate, 510% faster than *reiserfs*, and 29% faster than *ext2*. As more large-file traffic is injected, all file systems slow down, but we see a relatively constant performance ratio between *Conquest* and disk-based file systems. *Conquest* is 43% faster than *SGI XFS*, 76% faster than *ext2*, and 97% faster than *reiserfs*.

4 Conclusion

Conquest is an operational file system that integrates persistent RAM with disk storage to provide significantly improved performance compared to other approaches. In general use, we anticipate a 43% to 97% speedup over popular disk-based file systems.

Removing the disk-based assumptions integrated into operating systems was difficult, but necessary for *Conquest* to achieve its goals. Obvious ad hoc approaches fail to provide a complete solution and perform worse than *Conquest* due to the high cost of using the buffer cache and disk-specific code.

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

5 References

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

[2] 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*, 1996.

[3] 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.

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

[5] Miller E, Brandt S, Long D. HeRMES: High-Performance Reliable MRAM-Enabled Storage. *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, May 2001.

[6] 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*, 1996.

[7] 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.

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

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