Performance Analysis and Optimization of the Hurricane File System on the K42 Operating System

by

David Tam

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto.

Abstract

PERFORMANCE ANALYSIS AND OPTIMIZATION OF THE HURRICANE FILE SYSTEM
ON THE K42 OPERATING SYSTEM

David Tam
Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
2003

The performance scalability of the *Hurricane File System* (HFS) is studied under the context of the K42 Operating System. Both systems were designed for scalability on large-scale, shared-memory, non-uniform memory access multiprocessors. However, scalability of HFS was never studied extensively. Microbenchmarks for reading, writing, creating, obtaining file attributes, and name lookup were used to measure scalability. As well, a macrobenchmark in the form of a simulated Web server was used. The unoptimized version of HFS scaled poorly. Optimizations to the meta-data cache in the form of (1) finer grain locks, (2) larger hash tables, (3) modified hash functions, (4) padded hash list headers and cache entries, and (5) a modified block cache free list, resulted in significant scalability improvements.

Acknowledgements

This thesis has been made possible by the generous support of many individuals. I would like to thank my supervisor, Professor Michael Stumm, for his academic guidance; Jonathan Appavoo for his academic advice and technical support; Orran Krieger and the K42 group at the IBM Thomas J. Watson Research Center for their technical support; and Jennie Fan for her emotional support. Finally, I would like to acknowledge my sources of financial support: Professor Michael Stumm, the University of Toronto, IBM, and my parents.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction and Motivation

Computer system performance is one metric of technological advancement. Scientists and engineers are constantly pushing the performance envelope. Due to the efforts of mathematicians, physicists, chemists, electrical engineers, and computer engineers, the "law" of Gordon Moore [46] is maintained. Moore predicted that the number of transistors on an integrated circuit would double every 18 months. Along with this exponential rate of microprocessor improvement, other techniques have been applied to yield further performance improvements in computer systems.

To obtain additional hardware improvements, computer scientists and engineers often use the technique of increasing computer system size. Scaling up the size of a computer system involves increasing the amount of various resources such as the number of microprocessors, the amount of memory, the number of memory banks, the number of disks, and the capacity of inter-connections between these components. Scalable computer systems exist in a spectrum [37, p. 130] ranging from those constructed as a large tightly-coupled monolithic computer, to those constructed out of many inter-connected loosely-coupled commodity computers.

Regardless of the style of computer system, software performance has not kept pace with the underlying hardware performance [52, 3, 41]. Therefore, software scalability is an important aspect to consider when evaluating the performance of large computer systems. It comes down to the question of whether software performance increases proportionately with increases in hardware resources or performance.

One of the fundamental causes of the software scalability problem is the *operating system* (OS). Many Unix-based large computer systems rely on retro-fitted operating systems that were initially designed for small uniprocessor computers [22, p. 4] [18]. While these operating systems extend basic functionality to include the larger set of resources, they are unable to fully exploit the performance potential of these computer systems. The ad hoc approach used to transform a uniprocessor operating system into a large-scale multiprocessor operating system appears to be inadequate. Rather, large-scale multiprocessor operating systems should be specifically and systematically designed for large computer systems [22, p. 7].

The K42 operating system research group at the University of Toronto and the IBM Thomas J. Watson Research Center is currently building this kind of operating system. K42 [78] is a research operating system designed from the ground up for large-scale, shared-memory, *non-uniform memory access* (NUMA) multiprocessor computers. The main goals of K42 include scalability and flexibility.

An important component of the operating system is the file system. A scalable and flexible file system is needed to compliment the scalability and flexibility of K42. A file system is stressed by (1) I/O intensive applications such as Web servers and file servers, (2) many independent applications running concurrently, and (3) demand paging by the memory management system. The file system, in turn, stresses other components of the operating system and exposes bottlenecks throughout. The *Hurricane File System* (HFS) has been chosen as the file system for K42. HFS was specifically designed for large-scale, shared-memory, NUMA multiprocessor computers.

1.1 Thesis Goals

The goal of this thesis is to examine the performance scalability of HFS and implement optimizations where scalability is inadequate. More specifically, we study whether the system can handle proportionately more concurrent requests while maintaining the same throughput per request as the number of processors and disks increase. Under various workloads, we experimentally study the performance of HFS to determine where and why bottlenecks exist. Solutions to these bottlenecks are then developed and implemented, resulting in an optimized and scalable file system.

1.2 Hurricane File System

The Hurricane File System was originally designed and implemented by Orran Krieger [34] at the University of Toronto for the Hector Multiprocessor [83] and Hurricane Operating System [81]. HFS was specifically designed for large-scale, shared-memory, NUMA multiprocessor computers. The main design goals of HFS were flexibility and scalability. Due to the common objectives between HFS and K42, HFS contains a valuable framework to build upon.

HFS differs from other file systems in a number of ways. It is an object-oriented file system that supports many file types and configurations. It does not inherit any architectural components from the prototypical Unix Fast File System [44]. HFS was designed for micro-kernel operating systems and therefore resides in an exclusive virtual address space that is separate from the operating system kernel. It relies on efficient *interprocess communication* (IPC)[1] facilities to interact with various components of the operating system

[1] The terms *interprocess communication* (IPC) and *remote procedure call* (RPC) are equivalent under the context of K42,

kernel. It relies on memory-mapped *input/output* (I/O) capabilities of the memory management system to perform efficient data transfers between disks and requesting applications.

HFS contains the flexibility that is necessary to match the goals of K42, which is why it was selected as the K42 native file system. However, the scalability characteristics of HFS were, to date, never really tested[2]. A secondary motivation is to evaluate the effectiveness and robustness of the original file system design on new operating system and hardware technology, which has evolved significantly since the inception of Hector and Hurricane. That is, we can now determine whether the initial design withstands the test of time and can exploit new technologies, or whether the design is restricted to exploiting only "technology of moment". Since K42 is the University of Toronto's third generation research operating system, many bottlenecks that were present in the previous generations have been eliminated or reduced[3]. K42 is a faster operating system that can stress a file system to an extent not possible in the past. As well, the disparity between processor and memory speeds has become more severe, possibly affecting file system performance. A good design would nullify the disparity.

1.3 K42 Operating System

K42 is a research operating system currently being developed to address performance and scalability issues of system software on large-scale, shared-memory, NUMA multiprocessor computers. The operating system is a product of research at the University of Toronto in collaboration with the K42 group at the IBM Thomas J. Watson Research Center. The core of K42 is based on the University of Toronto's Tornado Operating System [22]. K42 is the university's third generation of research on scalable operating systems. Hurricane OS / Hector Multiprocessor was the first generation and Tornado OS / NUMAchine Multiprocessor [24] was the second generation. K42 runs on 64-bit processors that include the 64-bit PowerPC, MIPS, and AMD x86-64 processors. In terms of complete systems, K42 runs on SimOS machine simulation software [62] (under 64-bit PowerPC and MIPS emulation), the NUMAchine multiprocessor (MIPS), and IBM RS/6000 64-bit PowerPC computers.

K42 uses a micro-kernel architecture rather than the traditional monolithic kernel design. K42 consists of a small exception-handling component that serves as the micro-kernel, a fast IPC mechanism, and servers for all other components of the operating system. These servers exist in separate address spaces and rely upon the fast IPC mechanism for communication with the micro-kernel and other servers. Examples of fundamental servers include the memory management system and processor scheduler. Other servers include the resource

and may be used interchangeably.

[2]As described by Krieger in [34, p. 72], the scalability of HFS was not fully explored under the original Hector / Hurricane environment.

[3]Improvements were achieved mainly between Hurricane and Tornado operating systems. Some of these improvements include IPC performance, the elimination of hierarchical clustering and promotion of clustered objects, and simpler finer grain locking.

manager, pipe server, pty server, name server, shell server, trace daemon, and various file systems. This operating environment, where the file system exists in an exclusive virtual address space, suits the design of HFS naturally.

File system issues have yet to be explored on the K42 operating system and HFS offers a good starting point. Until recently, only a very slow but functional NFS client was available to provide primitive file system capabilities. With the availability of a flexible local file system such as HFS, file system research can begin. With an extremely fast and scalable file system, the interactions between operating system components can be stressed, such as the *virtual file system* (VFS) and memory management system.

1.4 Dissertation Outline

This dissertation describes the architecture of HFS, its integration with K42, the experimental measurements taken, and the optimizations implemented. Using microbenchmarks and macrobenchmarks, we will show that the optimizations we introduced had a significant impact on scalability.

1.4.1 Microbenchmarks and Optimizations

A number of simple microbenchmarks were run to investigate HFS scalability. The goals were to obtain results under controlled conditions, easily identify scalability problems, and make appropriate changes. The experiments were done in a very simple, controlled environment. They stressed the meta-data management component, which is the core of the file system. These experiments were mostly run from within the HFS address space (server-level) rather than in a separate user-level program. This configuration ensured that only HFS was stressed and not other OS facilities such as the IPC mechanism, VFS, and memory management system. Scalability of HFS was initially surprisingly poor but the optimizations resulted in drastic improvements.

1.4.2 Macrobenchmark and Optimizations

The macrobenchmark simulates the workload that a Web server would apply to a file system. The goal was to determine the impact of a more realistic workload on file system scalability. The macrobenchmark was run both from within the file system (server-level) and as a separate user-level program to provide a more realistic workload to the file system and to stress the interaction between the operating system and the file system. Within the operating system, it stresses the VFS and a critical part of the memory management system known as the *file cache manager* (FCM). The optimizations implemented during the microbenchmarks were moderately effective on the macrobenchmark. Additional optimizations were necessary for good scalability.

Chapter 2

Background and Related Work

2.1 Multiprocessor Computers

Multiprocessor computers can be classified under two broad categories: (1) shared-memory multiprocessors, and (2) message-passing multiprocessors. Shared-memory multiprocessors contain hardware mechanisms to allow memory to be transparently shared between processors. Message-passing multiprocessors lack these hardware mechanisms and share data by passing messages. In this thesis, we focus on shared-memory multiprocessors.

Shared-memory multiprocessors contain multiple processors, memory banks, I/O controllers, disks, and an interconnection network between these components. With the help of hardware mechanisms, memory is transparently shared among the processors at the granularity of a cache line size or smaller. On a shared-memory *symmetric multiprocessor* (SMP), all components are equidistant from each other and the interconnection network is often a single bus. Disk locality is not an issue on SMPs since all disks are seen as equidistant to all processors. On shared-memory NUMA multiprocessors, resources are located in small groups, called nodes, that are interconnected in a hierarchical and scalable manner. The interconnection medium can be a combination of various types such as a hierarchical ring network with nodes consisting of SMPs. Memory access speed is non-uniform in these systems since accessing memory on a local node is faster than accessing memory on a remote node, and disk locality on NUMA architectures is an issue as well. In this thesis, we focus on the performance of HFS on bus-based SMPs.

On both types of shared-memory multiprocessors, there are various performance pitfalls to avoid due to the cache-coherence mechanisms and the speed disparities between the processor caches and main memory. When two processors access independent memory locations that reside on the same cache line, false sharing can occur, causing major performance degradation. For example, consider when one processor reads a part of a shared cache line followed by another processor that writes to a different part of the same cache line. The cache-coherence mechanism would invalidate the cache line on the first processor[1], eliminating any chance of cache line reuse on the first processor. False sharing also results in extra traffic on the interconnection medium that can lead to slow-downs in other parts of the system that are not directly involved in the shared access. False sharing can be avoided by ensuring that affected data structures exclusively occupy whole cache lines. This guarantee is implemented by aligning the affected data structures to the starting boundary of cache lines and padding them to the end of their occupied cache line.

True sharing, where two or more processors are accessing the same data, is sometimes avoidable by redesigning the underlying algorithm used in the program. In order to achieve performance scalability, it is critical that HFS avoid true and false cache line sharing as much as possible. Other multiprocessor issues include providing processor affinity on various operations such as interrupt handling and specific disk operations.

2.2 Disks, Disk Controllers, and Device Drivers

Physical disks are directly accessed and controlled by physical disk controllers which in turn are accessed using disk device driver software. These three components provide primitive block-level access to the disks, such as the reading or writing of block "X". At this level, there is no concept of files or directories, only blocks of data. From this simple block-level view, HFS builds a file system, providing the concept of files and directories.

Device drivers are typically part of the operating system kernel address space, since they access hardware and physical memory addresses. For example, a device driver may be instructed to read a specific disk block from a specific physical disk and deposit the data into a specific physical memory address. Despite the inherent safety features behind virtual memory addresses, they are typically not used because they imply a specific processor context and device drivers must operate without regard to context. One reason for context-less operation is that disk controllers typically use *direct memory access* (DMA) to transfer data to or from physical memory addresses. DMA requires hardware support but it frees the processor from the mundane task of memory copying and allows it to perform more important work. Adding additional hardware support for re-establishing context in order to perform virtual-to-physical address translation during the DMA transfer would add to hardware complexity, and so this is rarely supported.

We are not concerned with disk simulation and modelling in this thesis. Optimizing disk layout is not performed. We assume that the disks are either infinitely fast or have a simple fixed latency, allowing us to focus on bottlenecks in the core of the file system. Disk modelling and simulation has been studied by Kotz et al. [33].

[1] On an invalidation-based snoopy bus protocol.

2.3 Operating Systems

Operating systems typically integrate file systems by providing a *virtual file system* (VFS) interface. The operating system uses this interface to create files, read and write files, obtain file attributes, create directories, obtain directory listings, etc., independent of the actual underlying file system being used. Hence, the VFS adds a layer of abstraction and a level of indirection in order to transparently support multiple, different, underlying file systems. To support memory-mapped file I/O, the memory management system must also interface with the file system. The file system must accept physical read and write page requests from the memory management system, and the memory management system must provide an up-call for the file system to invoke on request completions[2]. At the bottom end of the file system, the file system must communicate with the disk device driver to request the reading or writing of a physical disk block.

For operating systems that are capable of paging out portions of the operating system itself, the file system typically must be memory-pinned to prevent it from being paged out. If this precaution is not taken, a paged-out file system would be incapable of paging in data since the operational code would have been paged out of memory, and hence could not be executed.

The virtual file system component of K42 is the main interface between the operating system and the file system. Formally speaking, read and write operations are not specified in the K42 VFS interface. Rather, applications perform read and write operations implicitly using memory-mapped file I/O[3]. The K42 operating system provides a number of useful services that help increase performance of the file system. First, by being designed and tested for performance and scalability, we can be fairly certain that any performance problems are due to HFS and not an inherent bottleneck of the operating system. Second, by never being the bottleneck, we can stress HFS more so than in the past and expose more file system bottlenecks. Third, memory allocation facilities of K42 allow HFS to specify whether memory should be allocated from a local pool or from a global pool, thus facilitating memory locality and reducing false sharing of cache lines. As well, globally allocated memory can be requested in padded form to further prevent false sharing of cache lines. Fourth, the locking facilities provided by K42 are optimized for a multiprocessor environment with the ability to specify exactly the type of lock desired, such as spin locks, spin-block locks, block locks, auto locks, bit locks, etc.

2.4 File Systems

The role of a file system is to manage persistent data through a useful layer of abstraction and act as the middle man between the operating system and the disk device driver. If there was no need for a file system,

[2]Due to these simple requirements, this interface can be independent of the VFS interface and does not need to make use of it.

[3]The memory-mapped file I/O technique will be explained in Section 3.3.1.

the operating system would simply go directly to the device driver to transfer data. However, the device driver provides only primitive access to a physical disk as it can only read or write a block at a specific disk location. The view from the device driver level is that of a very large array of disk blocks. The file system abstracts arrays of disk blocks into files, directories, file attributes, and directory attributes. A file system implements these abstractions using meta-data, which is data that describes the real data, its location, and how to access it. One crucial aspect of file system performance is the speed of accessing and processing the meta-data.

File systems can be classified under two broad categories: (1) distributed file systems, and (2) local file systems. Distributed file systems span multiple computers and use a network to transfer data and co-ordinate state. The client/server model of computing is typically used in this setting. Examples of distributed file systems include NFS [65], AFS [27], CODA [66], Sprite Network File System [47], Microsoft SMB/CIFS [39], and xFS [2].

Local file systems operate on a single computer, which may be a uniprocessor or multiprocessor. Local file systems designed for message-passing multiprocessors do not share memory transparently between processors but use the message-passing model to explicitly communicate and share data at a coarse granularity. Examples of such file systems include the Thinking Machines sfs [38], Galley File System [48], IBM Vesta/PIOFS File System [17], IBM GPFS [67], Intel PFS [79], Intel CFS [10], and PVFS [13].

Local file systems designed for shared-memory multiprocessors share data structures at a very fine level of granularity. Examples of such file systems include SGI XFS [75], IBM JFS [9], Sun UFS [45], HP HFS [11], Linux Ext2 [58], and Microsoft NTFS [63, 64]. The Hurricane File System also falls under this category.

Orthogonally, file systems can also be classified as journaling, meaning they use a transaction log to prevent file system corruption and enable quick file system recovery in the event of a system crash. Journaling file systems include IBM JFS [8], SGI XFS [75], Sun UFS [43], ReiserFS [42], and Linux Ext3 [80, 29].

2.5 Related Work

Three areas of related work are file system design, file system workload studies, and file system benchmarking. Previous file system designs offer useful advice in achieving performance scalability. File system workload studies provide a general idea of realistic workloads and which file system operations are stressed. File system benchmark research provides guidelines in measuring file system performance.

2.5.1 File System Design

Our focus is on local file systems designed for shared-memory multiprocessors. The work most closely related to our work was done by Peacock et al. [55, 56] and LoVerso et al. [40]. They examined shared-memory multiprocessor scalability on the Unix System V Release 4 and OSF/1 UFS file systems, respectively. They examined the underlying infrastructure to determine where and why bottlenecks exist. Peacock et al. implemented optimizations to the file system meta-data caches. A cache organization called *software set-associative cache* (SSAC) was used to relieve significant contention points. SSAC is a compromise between fine grain locking of individual data structures, and coarse grain locking of the entire file system meta-data cache. SSAC gathers data structures together for group locking. LoVerso et al. used time-stamps to improve meta-data operation performance by reducing locking and rescanning requirements of various data structures. These techniques may be useful in HFS.

Multiprocessor capabilities of the Linux file system was explored by Sheth and Gopinath [70]. They distributed various global data structures to local ones and added fine grain locking where appropriate. They were able to reduce locking frequency but mean wait time on the locks remained largely unchanged, indicating that their solution was still incomplete.

A study of file system performance scalability was conducted on the SGI XFS file system by Sweeney et al. [75]. Partitioned data structures allow concurrent access to many data structures with reduced lock contention. B+ trees were used throughout the design to enable performance scalability. Allocating contiguous disk blocks, called extents, helped maintain disk transfer performance. Zero-copy mechanisms are also available to applications, allowing file data to transfer directly between user-level buffers and disk controllers using DMA. Without this mechanism, data must use kernel buffers as an intermediate step, which increases transfer latency, processor utilization, cache pollution, memory contention, and system bus traffic. Asynchronous meta-data updates are supported using asynchronous transaction logging. These and many other features allow XFS to keep up with the underlying hardware bandwidth of up to 60 disks. On the SPEC SFS (LADDIS) benchmark, XFS performed 12% faster than EFS, the previous generation SGI file system. The techniques used in XFS provide performance scalability in terms of the number of disks but not necessarily in terms of the number of processors. Due to the experimental setup, and the nature of the results obtained, the effectiveness of processor scaling is difficult to evaluate. For instance, throughput results are obtained for 1 and 4 threads of execution on an 8 processor SGI Challenge. However, due to aggregate disk bandwidth limitations, it is impossible to determine whether the total throughput of 4 concurrent threads can achieve 4 times the throughput of 1 thread of execution.

File system performance and scalability in Linux 2.4.17 was examined by Bryant et al. [12]. They compared five different file systems: Ext2, Ext3, ReiserFS, XFS, and JFS. A small, medium, and large computer system were used and three benchmarks were run: pgmeter, filemark, and AIM VII. This study constituted a good range of representative system configurations and workloads typically targetted for the Linux operating system. File system scalability results were poor, although XFS scaled well for up to 4 processors but not beyond. A detailed analysis revealing the scalability bottlenecks was not conducted but was a subject of future work. A brief examination of lock times by the authors revealed that the file system can be responsible for causing the kernel to spend up to 88% of the total execution cycles of AIM VII on a busy-wait spin lock known as the "*big kernel lock*" (BKL). This portion varies from 12% on XFS, to 30% on Ext2, to 85% on Ext3, and to 88% on ReiserFS. These values had a direct impact on scalability. From this work, we can see that file systems can have an impact on the operating system and subsequently on scalability. However, more research must be conducted to understand and improve file system performance scalability on shared-memory multiprocessors.

NTFS disk scalability was briefly examined by Riedel et al. [59]. They examined the performance improvements of using up to 4 disks configured for striping and used sequential transfers of 100 MB files as the workload. The results indicate that under NTFS, increasing the number of disks can result in a linear increase in throughput. Larger request sizes resulted in higher throughput and lower processor overhead. Processor scalability was not examined.

Overall, SMP file system scalability has not been extensively studied. Commercial Unix SMP vendors claim performance scalability of their file systems but there is little concrete evidence, such as detailed experimental results, to support the claims.

There have been many performance studies conducted on file systems for message-passing multiprocessors, multicomputers, and clusters such as the performance of the Andrew File System [27], Sprite Network File System [47], Thinking Machines Scalable File System [38], Galley Parallel File System [48, 49, 50], IBM Vesta Parallel File System [17], IBM General Parallel File System [25], IBM Parallel I/O File System [79], Intel Parallel File System [79], Intel Concurrent File System [10], and Parallel Virtual File System [13]. However, there have been few performance studies conducted on parallel file systems on shared-memory multiprocessors. Due to the coarser granularity of sharing in message-passing multiprocessor file systems, these designs do not apply well to tightly-coupled shared-memory multiprocessor file systems such as HFS.

Kotz and Nieuwejaar designed a file system for multiprocessors called Galley [48, 49, 50]. Galley is a file system targetted for scientific workloads, and it has been evaluated on message-passing systems such as the IBM SP2 MPP. Galley is not specifically targetted for shared-memory processors.

Dibble et al. [19] created Bridge as a multi-layered file system designed for parallel processors. It uses conventional local file systems as its base but cordinates them using a Bridge server and Bridge clients to create a single file system image. This multi-layering technique is similar to the configuration of NFS. Since they used conventional local file systems at the client ends, they did not study scalability at the same level of detail as in this thesis. They were interested in a first iteration design that offered basic functionality and showed some performance improvement.

Kotz and Ellis examined caching policies in parallel file systems for multiprocessor scientific workloads [31]. They assumed a fairly rigid configuration of striped data across all disks. They focused on write-only patterns and caching policies. Our focus is on the performance of the deeper, underlying mechanisms and data structures of the file system.

2.5.2 File System Workloads

Selecting an appropriate workload to measure file system performance is a complicated task and has been studied by many researchers. The resulting performance of any experiment is dependent on the workload chosen and the workload may not reflect the ultimate end-use of the computer system.

A study of file system workload was conducted by Ousterhout et al. [53] in 1985 on several time-shared VAX computers that showed a number of important characteristics of file system usage:

1. Many users have mostly low bandwidth file system requirements relative to available file system bandwidth.
2. Most file accesses are sequential (70%).
3. Most data blocks are deleted or replaced shortly after creation (20%-30% within 0.5 seconds, 50% within 5 minutes).
4. There are many more small-file accesses than large-file accesses (80% are less than 10 kbytes).

Since publication, these basic characteristics have been influential to file system design and benchmarking.

Various updated studies were conducted such as by Baker et al. [7] on the Sprite Network File System and by Vogel [82] on a Windows NT workstation environment. Results by Baker et al. reaffirmed most of the initial findings. Two major differences were that average user file throughput was 20 times greater and that the average size of large files have increased by more than an order of magnitude. Results by Vogel showed an additional 3 times increase in average user file throughput and an additional order of magnitude increase in the average size of large files. There was a shift towards random file access. Files were opened for shorter periods, in the order of a magnitude shorter.

A more recent study was conducted by Roselli et al. [61] on a variety of computer systems that serve a variety of purposes. They focused on how disk behavior is affected by workload. Most files have a bimodal access pattern consisting of read-mostly or write-mostly access. The most common file system call is the stat() operation.

A large study on file systems in a large commercial environment was conducted by Douceur and Bolosky [20]. They found that the various attributes of file systems, such as file size, file age, file lifetime, directory size, and directory depth, fit specific probability distributions. File and directory sizes are similar across file systems, however, file life times vary greatly. There is a strong relation between file type and file size. Finally, on average, file systems are half full.

The characteristics of workloads for message-passing multiprocessor file systems running parallel scientific applications was studied by Kotz, Nieuwejaar, Ellis, Purakayastha, and Best in [32, 57, 51]. They found that files are generally larger, with longer life-times, perform mostly writes, and exhibit more intra-job file sharing when compared against general purpose time-sharing, server, and workstation environments. Although large I/O requests are common, small requests are fairly common as well. They believe this is a natural result of parallelization and is an inherent characteristic in most parallel programs. Therefore, file systems for these computers must provide low latency to small requests and high bandwidth to large requests. They concluded that a distributed file system (such as NFS or AFS) would not provide adequate performance since it is designed for completely different workload characteristics.

2.5.3 File System Benchmarking

A classic synthetic benchmark is the Andrew benchmark [27] that was originally designed to measure the performance scalability of the Andrew File System. It claims to be representative of the workload of an average user. However, it is more accurate to classify it as the workload of a typical software developer. The benchmark consists of creating directories that mirror a source tree, copying files from the source tree, statting all files, reading all files, and building the source tree. Developed in 1987 to reflect a workload of five software developers, with approximately 70 files totalling 200 kilobytes, this benchmark environment does not reflect current reality. Consequently, a few researchers have used modified versions of the benchmark with parameters scaled to the current state of technology [60, 14, 28, 68]. Despite modifications, the Andrew benchmark has other limitations. It does not truly stress the I/O subsystem since less than 25% of the time is spent performing I/O [15].

The *Standard Performance Evaluation Corporation* (SPEC) *System File Server* (SFS) 97_R1 V3.0 benchmark [74] is popular among the computer industry. It measures the performance of the file system running as an NFS server. This benchmark is not suitable for our use since it introduces complications and interference from NFS protocol processing and UDP/IP network protocol traffic.

Bonnie, written by Tim Bray in 1990, is a classic microbenchmark that performs sequential reads and writes. It allows for comparison between read and write performance, block access versus character access, and random versus sequential access [15]. Bonnie was designed to reveal bottlenecks in the file system [77]. However, the range of tests appears to be fairly narrow since it only stresses read and write operations of the file system and not operations such as file/directory creation/deletion, path name lookup, and obtaining/modifying file attributes, making it unsuitable for our use.

IOStone [54] simulates the locality found in the BSD file system workload study by Ousterhout et al. [53]. According to Tang [77], the workload does not scale well and is not I/O bound. Parallel file accesses do not occur since only one process is used. Chen and Patterson [15] found that IOStone spends less than 25% of the time doing I/O, making it unsuitable for our use.

SDET [21] simulates a time-sharing system used in a software development environment. It simulates a software developer at a terminal typing and executing shell commands. Each user, simulated by a shell script, has an exclusive home directory and executes shell commands independent of other users. Throughput is measured in terms of scripts per hour. Although it can be used to measure file system scalability, it is not specifically a file system benchmark but more of a general system scalability benchmark. Chen and Patterson [15] have found that SDET spends less than 25% of the time doing I/O, making it unsuitable for our use.

PostMark [30] emphasizes access to many small, short-lived files that all need equally fast access. In particular, it simulates a mail or network news server workload. It uses only 4 simple operations: (1) create file, (2) delete file, (3) read entire file, and (4) write to end of file. A specified number of random operations are executed and statistics are gathered. PostMark uses a single working directory and does not create and exercise a directory hierarchy.

The applicability of benchmark performance to real-world workload performance is a much debated issue. Some researchers claim that previous file system benchmarks are inadequate in reflecting real-world workloads and contain various inadequacies [15, 16, 72, 69]. Chen and Patterson [15, 16] developed a technique of measuring a few basic file system operations and projecting performance based on the characteristics of the proposed real-world workload. They also address the problem of scaling a benchmark appropriately to suit the target platform. Smith [72] developed a benchmarking methodology that predicts file system performance for a specific workload and aids in bottleneck identification. Smith and Seltzer [71] advocate the need to age a file system before taking measurements in order to provide a more realistic file system state. They use a simulated workload to age the file system.

In summary, many researchers are not satisfied with the currently available file system benchmarks. Some are out-dated and no longer applicable. Some do not stress the I/O system adequately because they are not file system bound.

In this thesis, we use custom benchmarks that are designed to stress specific components of the file system. A custom macrobenchmark is used for the sole purpose of verifying that the custom microbenchmark performance results were applicable at some level of generality. We will show that improving the performance scalability of fundamental file system operations leads to scalability of the file system in general.

Chapter 3

HFS Architecture

This chapter describes the various components of HFS and their interactions with the operating system and user-level applications. The organization and operation of the cache systems within HFS are described in detail because they play a crucial role in file system performance scalability, since they are globally accessible to all processors and are the most used components of the file system. Traces of a few fundamental file system operations are described to demonstrate the interaction between the cache system and the on-disk layout of data and meta-data. This chapter will give the reader an understanding of the operation of HFS, the interaction of various components, and the potential scalability bottlenecks.

3.1 HFS Layers

The components of HFS are shown in **Figure 3.1**. Communication between domains (i.e. address spaces such as application, HFS, and kernel) is accomplished using the K42 IPC facility. HFS can be accessed directly using the HFS user-level library, or indirectly through the virtual file system of the operating system kernel using standard I/O system calls. Currently, only the indirect access method has been implemented. Additional details of the HFS architecture can be found in [34].

3.1.1 HFS User-Level Library

The HFS user-level library, also known as the *Alloc Stream Facility* (ASF) [36], is an application level interface to general I/O, which includes HFS, NFS, and other file systems. It is in the form of a user-level library, much like stdio.h in the standard C library. The HFS library has not been ported to K42, but its functionality has been partially fulfilled by the standard Unix/Linux/GNU C library and the K42 user-level library.

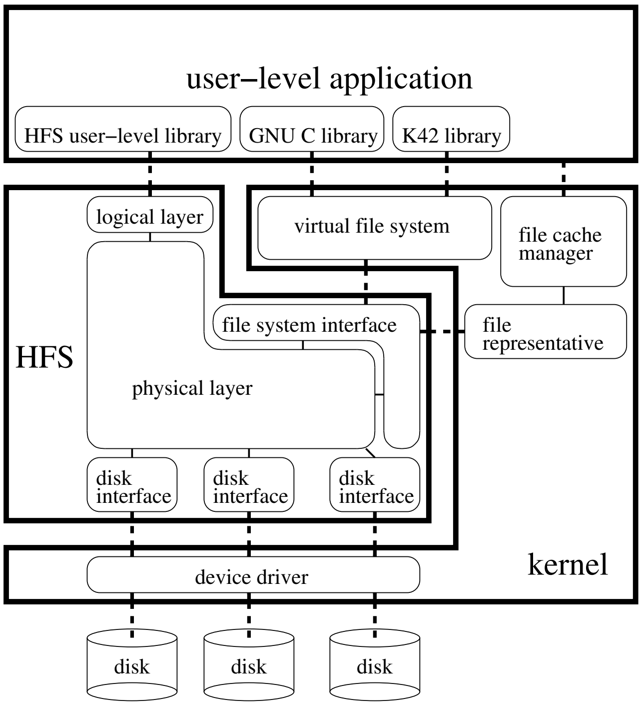

Figure 3.1: HFS layers.

3.1.2 Logical Layer

The logical layer provides naming, authentication, and locking services. Naming services include path name lookup and caching. This layer was not ported since the VFS layer and the name space server components of K42 handle these services. Krieger states that, "We are less concerned about this layer, since it is not heavily involved when reading and writing file data, and hence is not performance critical" [34, p. 82].

3.1.3 Physical Layer

The physical layer maintains the mapping of logical file blocks to physical disk blocks. It handles file structuring, disk block placement, load balancing, locality management, and cylinder clustering. This layer is our area of focus since it contains the performance critical meta-data caches and implements the core functionality of the file system. It will be described in more detail in Section 3.3.2.

3.1.4 File System Interface

The file system interface is similar to the Unix/Linux/GNU virtual file system interface definition. As shown in **Table 3.1**, it contains functions to obtain and change file attributes, open and close files, obtain directory listings, and read and write from and to the file system. The speed and scalability of these function calls determines the overall speed and scalability of the file system.

File System Call	Description
open()	Opens a file.
createFile()	Creates a new file.
deleteFile()	Deletes a file.
mkdir()	Creates a new directory.
rmdir()	Deletes an empty directory.
getAttribute()	Obtains file and directory attributes.
getDents()	Obtains directory listing.
rename()	Renames a file or directory.
link/unlink()	Creates or removes hard links.
truncate()	Truncates or expands a file to a given length.
chmod()	Modifies file and directory access permissions.
chown()	Changes ownership of files and directories.
utime()	Modifies access time of files and directories.
freeFileInfo()	Frees file handle.
ReadBlockPhys()	Reads a file block.
WriteBlockPhys()	Writes a file block.

Table 3.1: File system interface.

3.1.5 Disk Interface

The lowest layer of HFS is the Disk class and this is where accesses to the K42 block device driver are performed. Disk block scheduling for reading and writing of disk blocks is performed in this module and is crucial to raw disk performance since it can minimize disk seek time and rotational delay. Since our scalability focus does not depend upon disk scheduling performance, we have not ported the functionality of disk block scheduling but it will be a subject of future work.

3.2 Disk Layout

Strictly speaking, meta-data is information that describes the actual data stored on disk. It describes the disk layout of the data, creation, access, modification times of the data, access permissions, ownership, size of the data, and other information. Meta-data can also be viewed as the information generated and stored by the file system to keep track of actual data that must be kept persistent for the applications. Data structures used to hold meta-data include the *block index*, *file index*, *indirect file index*, *superblock*, and *block bitmap*.

There are several layers of meta-data that are used to organize disk layout. At the lowest layer, information about the physical layout of a particular file is kept in meta-data called the *block index* of the file, which itself occupies one or more blocks. This per file structure indicates the physical disk block location of each logical block of the file. The block index is equivalent to the inode data structure of the Berkeley *Fast File System* (FFS). However, unlike FFS, the location of the block indices are not fixed at disk formatting time. Rather, the location of all block indices is kept in meta-data called the *file index*. Since the file index may occupy more than one block, an *indirect file index* (occupying less than one block) is used to store the

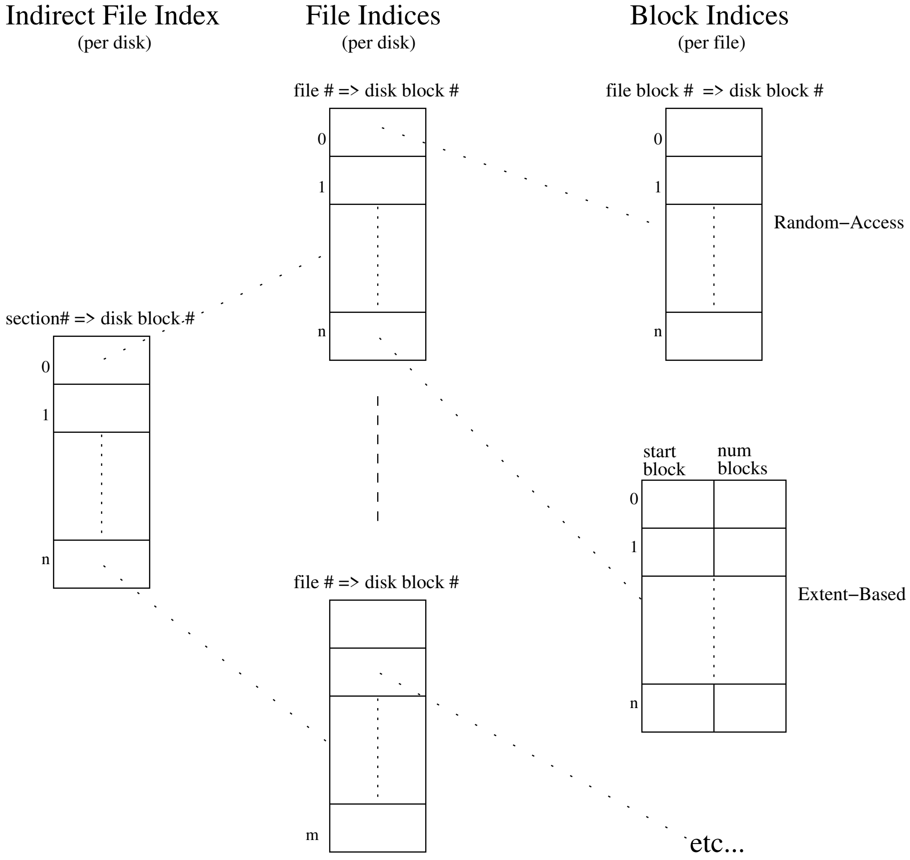

Figure 3.2: Indices.

location of all file index blocks. The indirect file index resides in a meta-data structure called the *superblock*, which occupies exactly one physical disk block at a well-known, fixed location.

The layout of the meta-data on disk is shown in **Figure 3.2**. To obtain the block index of file **#N**, the following calculations are done. Using standard C and C library terminology:

- `Let A = floor(N / #_of_file_index_blocks)`
- `Let B = N % #_of_file_index_blocks`

By looking up the A^{th} entry in the indirect file index, the appropriate file index block is found. By looking up the B^{th} entry in that file index block, the block index of file **#N** is obtained.

Under the "block indices" column of the diagram, two types of block indices are shown for two different types of files, random-access and extent-based. An extent-based file allocates physical disk blocks in contiguous groups called extents. An extent is specified by its starting block and the number of blocks it occupies. HFS supports other file types and these all have their own unique block index structure. Other supported file types, not shown in the figure, include sparse-data, small-data, fixed-sized record-store, striped, replicated, distribution, checkpoint, parity, and application-specific distribution. Details of these file types can be found in [34].

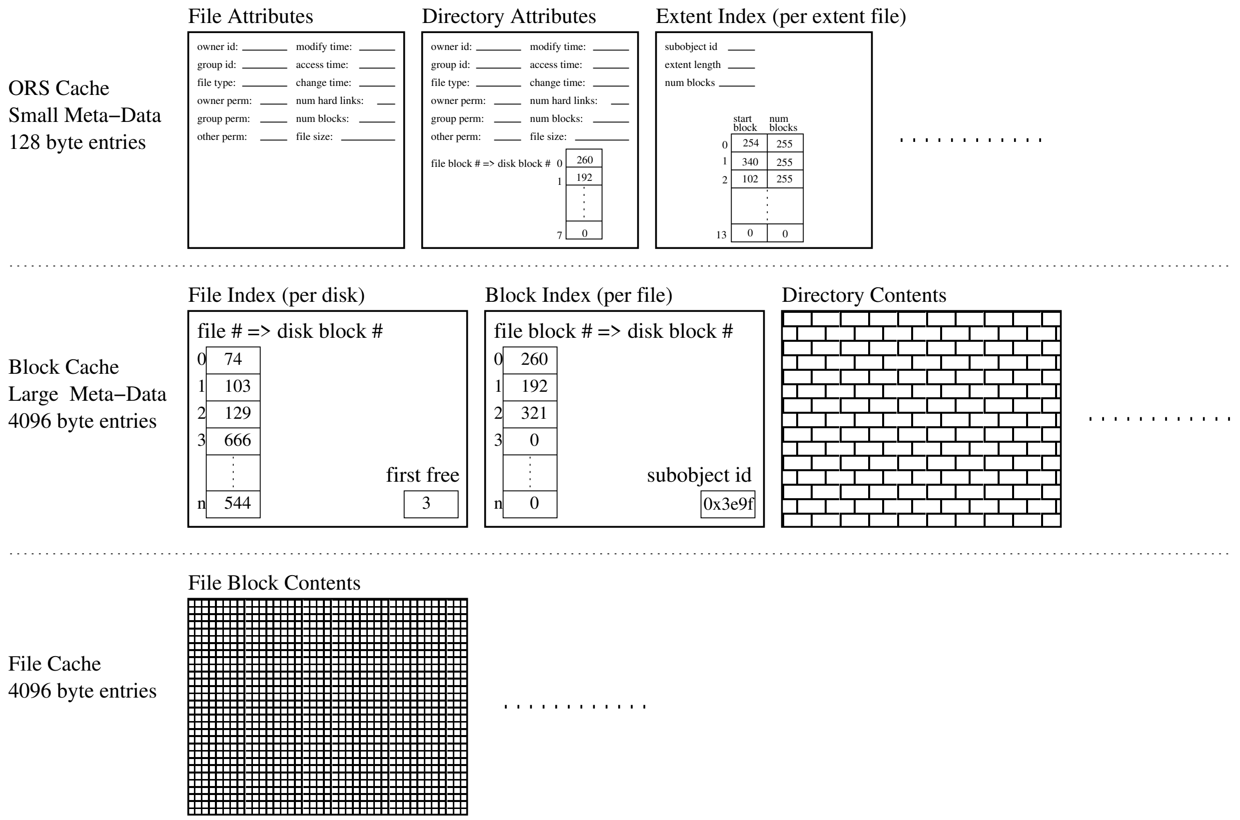

Figure 3.3: Cache types.

HFS uses bitmaps to keep track of which blocks have been allocated and which blocks are free. The block bitmap occupies several physical disk blocks at a well-known, fixed location.

Unlike FFS, visible file attributes of files, such as those returned from a typical Unix stat() file system call, are not stored in the block index of the associated file. Instead, these are stored on disk in one large fixed-size-record file to make more efficient use of disk space.

HFS allows data and meta-data blocks to be written to any disk block. When writing an updated meta-data block to disk, the old disk block location is freed and a new location is determined and written to. The new location can be the block closest to the current disk head location or some other criteria. This is in contrast to the Berkeley *Fast File System* (FFS) and the *Open Software Foundation Unix File System* (OSF/1 UFS), where meta-data blocks, such as inodes (also known as direct blocks), indirect, doubly-indirect, and triply-indirect blocks are fixed once they are allocated and must be updated in place. In particular, inodes are found in fixed, deterministic locations when a disk is formatted and can never change location. On HFS, the block indices have no restrictions. The only fixed blocks are (1) the superblock, which contains the indirect file index, and (2) the bitmap blocks, which indicate which disk blocks are allocated and which blocks are free.

3.3 Caches

Caches temporarily store data and meta-data from disk in memory to enable higher performance since memory accesses are much faster than disk accesses. There are four caches in the file system: (1) the file cache for file data, (2) the *object-related state* (ORS) cache for small meta-data entries, (3) the block cache for large meta-data entries, and (4) the name tree cache for caching path names. The operational characteristics of the four caches are all slightly different to suit the access patterns of the particular cache. The cache entries for the data and meta-data cache systems are shown in **Figure 3.3**, along with examples of cache entry contents.

3.3.1 File Cache

The file cache holds the contents of file blocks. Data caching is an important mechanism in reducing disk traffic, reducing data access latency, and increasing data throughput. The file cache is external to HFS and is managed by K42 *file cache managers* (FCMs), which are part of the K42 memory management system, on a per-file basis. The file cache entries, which are 4096 bytes in size and equal to the K42 memory page size, are, in effect, regular pages of memory and transparently managed in the typical virtual-memory management fashion.

File transfer from disk occurs exactly like demand-paging of application code. In demand-paging, application code is transferred from disk into memory one page at a time as needed during application execution. Initially, the first page of application code is loaded into memory and the application begins execution. When the application attempts to access code that is not yet in memory, a page fault occurs, the application is interrupted, the appropriate page is loaded into memory from disk, and the application is resumed. This process repeats as necessary, resulting in the technique of demand-paging.

HFS file transfers from disk occur in the same manner, allowing the K42 demand-paging mechanism to be reused for this purpose. HFS uses memory-mapped I/O to transfer data between the disk and application. Memory-mapping a file informs the K42 file cache manager that any accesses to a particular memory region should be treated as if the corresponding disk file block was accessed. File blocks are transferred between disk and memory one block at a time and only when needed. When the application attempts to access a part of the file that is not in memory, a page fault occurs, the application is interrupted, the appropriate block is loaded into memory from disk, and the application is resumed. This process repeats as necessary, resulting in the technique of memory-mapped I/O.

Since the file cache manager is involved in memory-mapped I/O and uses the existing demand-paging mechanisms, it transparently handles the caching of file data using the existing page caching mechanism. Therefore, HFS does not handle the caching of file data, but only handles the caching of meta-data. In this thesis, we are not concerned with caching issues of file data.

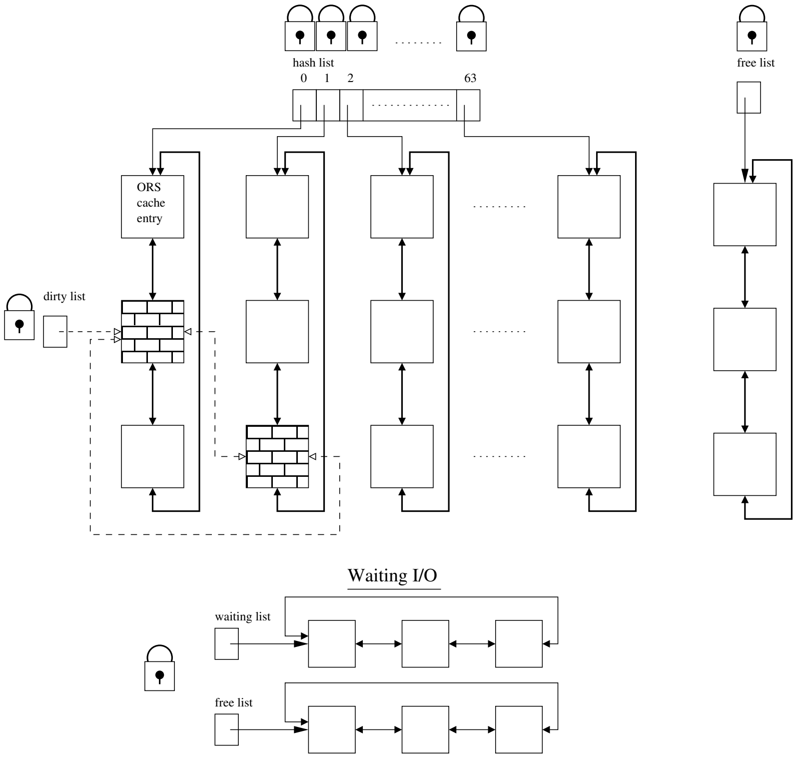

Figure 3.4: ORS cache.

3.3.2 Meta-Data Caches

There are two meta-data caches in HFS: (1) the *object-related state* (ORS) cache, and (2) the block cache. The management of these two caches is the core functionality of the file system. The cache entries and the cache management code are the most used sections of the file system since every file system operation requires a meta-data cache look up. The speed and scalability of these two cache systems essentially determines the speed and scalability of the file system. In this thesis, we primarily focus upon these two components of HFS.

The two caches are located in the physical layer of HFS and are globally accessible to the file system. They contain a number of common lists.

1. hash list : Contains meta-data cache entries.
2. free list : Contains meta-data cache entries available for reuse.
3. dirty list : Contains modified meta-data cache entries.
4. waiting I/O list : Used to queue requests waiting on cache entries that are currently performing disk I/O. Waiting for disk I/O is typically a long process so this list is used to queue and block requests.
5. waiting I/O free list : Contains waiting I/O list entries available for reuse.

The hash list headers (also known as hash table entries) and dirty list headers reside contiguously in

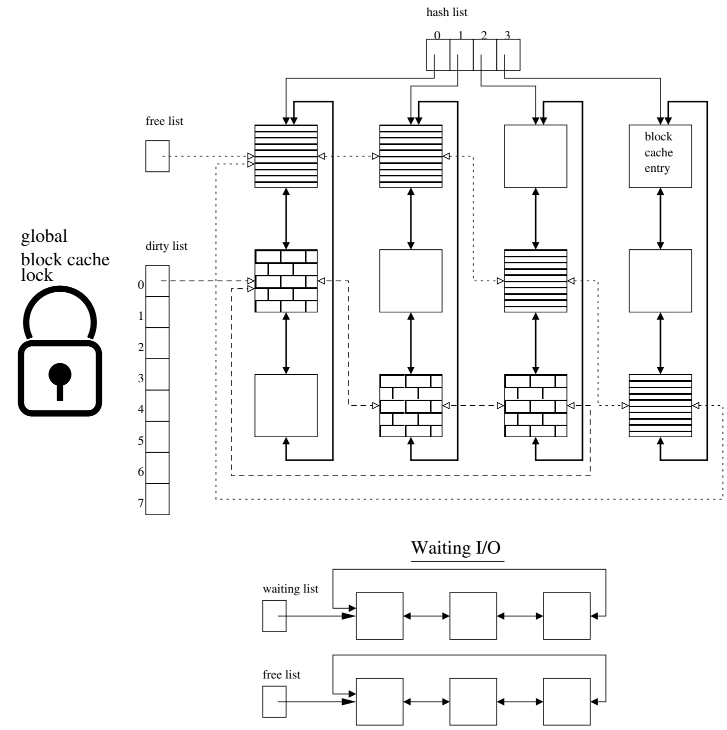

Figure 3.5: Block cache.

memory, similar to the memory layout of an array in the C language[1], and point to the head of each list. All lists are doubly-linked and circular. Each cache entry in the hash list is protected with a simple reader-writer lock[2] implemented using three data structures.

1. A counter indicating the number of reads in progress. The counter is protected by either the global lock (block cache) or the corresponding hash list lock (ORS cache).
2. A busy bit flag indicating that an exclusive write is in progress.
3. A global wait queue (one for the block cache, and one for the ORS cache) for waiting readers and writers.

The first cache is the ORS cache, which contains entries of 128 bytes in size. It is hashed by file token and is mainly used to hold file attributes such as the meta-data information that is obtained from a typical Unix stat() function call. For certain types of files such as extent-based files, it is used to hold the index of extent blocks. It can also be used to hold a shortened version of the directory block index. The structure of the ORS cache system implementation is shown in **Figure 3.4**.

The second cache is the block cache, which contains larger cache entries of 4096 bytes in size[3]. It is used to hold the file index of each disk, block index of each file, directory contents, etc. It is hashed by three values: (1) file token, (2) block number, and (3) type of meta-data block, such as block index, file index,

[1]Due to the allocation algorithm of hash and dirty list headers, these data structures are arranged in memory in a form that is equivalent to a contiguously allocated array.

[2]This reader-writer lock is by no means the most efficient.

[3]In reality, the block cache entry contains a pointer to a 4096 byte memory region to hold the cache contents rather than having it embedded within the cache entry itself.

directory contents, etc. The structure of the block cache system is shown in **Figure 3.5**. As stated by Krieger [34, p. 53], "Having multiple caches, each tuned for a different size, results in better performance than having a single physical cache with multiple element sizes".

In the original HFS design, the ORS cache contained 64 hash lists whereas the block cache contained only 4. The reasons for the difference have not been determined, however the size of the hash lists can be changed easily during the optimization process. The ORS cache contains a single dirty list whereas the block cache contains 8 dirty lists due to flushing order requirements of different types of meta-data.

The free list in both the ORS and block cache indicates which items can be recycled although they are given a second chance to be reused under their original identity should they be needed. However, due to the nature of the content in the two caches, the operation of the free list in the ORS cache is slightly different from the block cache. In the ORS cache, entries are placed on the free list only after the represented file is invalidated. In the block cache, entries are placed on the free list quite liberally. As long as the entry is not being used, meaning that it does not have a reader or writer lock set, and the contents are unaltered, it is placed onto the free list. Consequently, the free list is utilized much more frequently in the block cache. In the ORS cache, the entries cannot be on the free list and the hash list simultaneously and hence the free list exists as a separate list. In the block cache, entries can exist on both the free list and a hash list simultaneously and hence the free list is threaded throughout the cache. This policy in the block cache leads to a natural recycling process of cache entries.

The locking infrastructure is different as well. In the block cache, there is a single global lock protecting all data structures. On the other hand, the ORS cache uses a number of locks, each protecting a specific group of data structures. For instance, there is a lock for each hash list header, for the dirty list header, and free list header. The difference in locking infrastructure is due to the nature of access and purpose of the two different meta-data caches. The ORS cache contains file attributes that must be consulted (and possibly modified) frequently. Access latency is very important, which may be the reason for the finer-grain locks. On the other hand, access latency to the block cache contents may not be as critical so the simpler design of a single global lock may be sufficient.

There are trade-offs between the two designs. Under a single global locking scheme, the number of locks that need to be acquired for any operation on the target data structures is small, whereas under a finer grain locking scheme, several locks need to be acquired for an operation. The disadvantage of the global locking scheme is that it may lead to high contention on the single lock if the lock is held for long periods of time and there are many concurrent threads that are competing for the single global lock. Finer grain locks can reduce this contention but at the cost of higher overhead.

Meta-data cache flushing policies have not been decided and may have a significant impact on performance. For our experiments, cache flushing is triggered explicitly when desired.

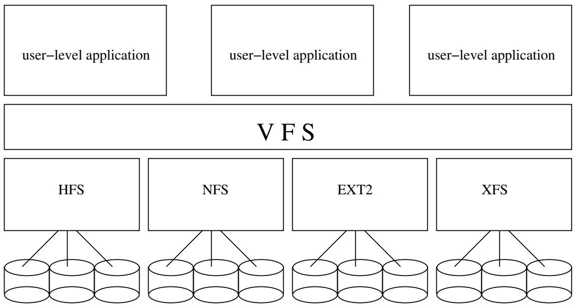

Figure 3.6: VFS.

3.3.3 Name Tree Cache

The HFS name tree cache[4], which is located in the logical layer, was not ported since a separate, file system independent, global name tree cache was implemented in the VFS component of K42.

3.4 VFS – Virtual File System

The virtual file system is a layer of indirection to the underlying physical file system and enables multiple file systems to be easily used by the operating system. See **Figure 3.6**. All file systems on K42 must adhere to a single simple interface, described in Section 3.1.4. The VFS provides directory name lookup caching and co-ordination of concurrent conflicting accesses to files or directories.

The VFS should have adequate flexibility to enable simple uniprocessor, single-threaded file systems to operate under a multiprocessor, multi-threaded environment. Since data structures designed for uniprocessors may require protection against concurrent access and modification (which can lead to corrupt data), the VFS should handle all necessary locking and synchronization issues outside the uniprocessor file systems, thus hiding the concurrency. However, for sophisticated, multiprocessor, multi-threaded file systems, the VFS should do very little and, in fact, expose the concurrency of the underlying file system. The VFS layer should have adequate flexibility to handle all scenarios, allowing various file systems to be easily ported to K42 without significant modifications. These issues are beyond the scope of the thesis and were not investigated.

3.5 K42 Interprocess Communication

With K42 running on a multiprocessor system, HFS code can be executed on any processor. Due to the characteristics of the K42 *interprocess communication* (IPC)[5] facility, all code executed on behalf of an IPC

[4]Known more commonly as the directory name lookup cache.

[5]The terms *interprocess communication* (IPC) and *remote procedure call* (RPC) are equivalent under the context of K42 and may be used interchangeably.

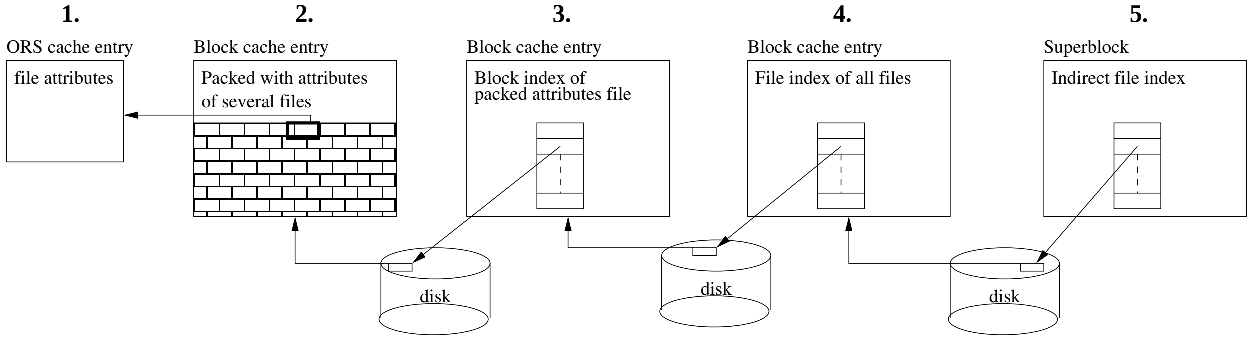

Figure 3.7: Cache trace of getAttribute().

remains local to the same processor from which the call was made. Hence, a caller will reach the callee on the same processor. In terms of file system servicing, a user application utilizing the file system will initiate an IPC to the VFS, which in turn will initiate an IPC to HFS, all of which execute on the same processor. There is no need to determine which processor is best able to handle a request from a particular processor.

3.6 Trace of Fundamental File System Operations

To gain a better understanding of HFS, we will trace the code path of two basic file system operations: obtaining file attributes and path name lookup. These operations cover most of the important code paths and will demonstrate the role of the meta-data caches and the general operation of HFS.

3.6.1 getAttribute()

This file system call, which is equivalent to the Unix/Linux/GNU stat() system call, returns the attributes of a target file or directory identified by a file token. The file system must traverse various indirect chains to reach the target meta-data. The goal is to attempt to access the meta-data as quickly as possible. If the meta-data exists in the ORS cache, it is simply accessed. If the meta-data is not in the ORS cache, the block cache is searched for indirect meta-data that will lead to the desired data. This process repeats further down the meta-data chain if necessary. The caches are accessed as follows, as shown in **Figure 3.7**.

1. The ORS cache is first searched using the file token as the hash key.

2. If the attributes are not found in the ORS cache, the block cache is searched next for a packed version of the data. Attributes of all files are stored in one large fixed-sized-record file on disk, with attributes of several files packed into each block. Since these blocks may be present in the block cache, we search for it there.

3. If the block cache entry is not found, the block cache is searched a second time but for the block index of the packed attributes file. The block index translates logical file block numbers to physical disk

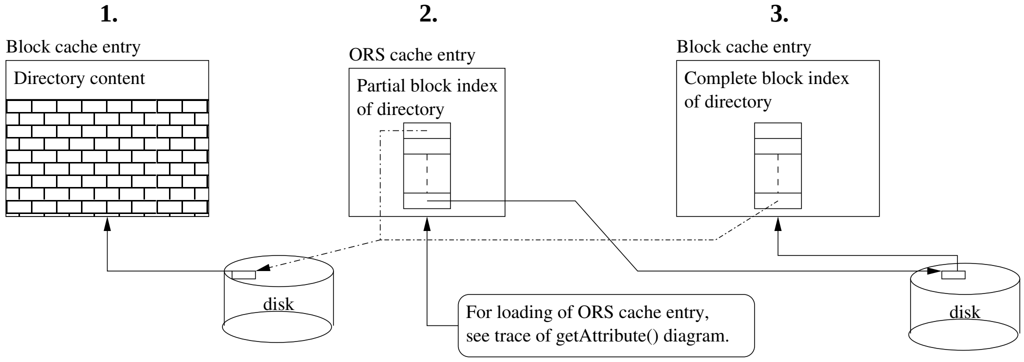

Figure 3.8: Cache trace of path name lookup.

block numbers. It identifies the disk block that contains the packed data and allows it to be loaded into the block cache.

4. If the block index is not found, the block cache is searched a third time but for the file index. The file index identifies the disk block that contains the block index of the packed data file and allows it to be loaded from disk into the block cache.

5. If the file index cannot be found, the indirect file index in the superblock is consulted, which is always resident in memory. At this point in the call chain, the execution is reversed and the missing block cache and ORS cache entries are filled from disk. The indirect file index leads to the direct file index, which leads to the block index of the packed data file. The packed data block cache is filled and the ORS cache extracts the packed data from the block cache to fill the ORS cache entry. The ORS cache now contains the attributes of the target file.

The general idea is that in the common case, meta-data is in the caches and that is why the caches are always checked first. If the meta-data is not found in the cache, it is placed into the cache by following the meta-data chain and using that information to access the disk. This is a recursive process that repeats as necessary. The read and write file system operations interact with the cache system in a similar fashion using steps 3, 4, and 5.

3.6.2 Path Name Lookup

The path name lookup operation returns a file token identifier for a given path name, such as `/usr/bin/diff`. The file token, which is similar to an inode number under Berkeley FFS, allows for easy identification of the target file by the file system and is used by clients to identify specific files. The token is a concatenation of the file number, file type, and disk number. The root directory is a well-known file token and this is where the operation begins. The task involves tracing the path name recursively by using the file token of the current path component, such as `/`, to extract the subsequent file token of the next path component, such as `/usr/`. The caches are accessed as follows (**Figure 3.8**):

1. The directory contents of the current path component is acquired from the block cache. The directory contents may occupy several block cache entries and each one is searched until the path name component is found or until the end of the directory contents is reached.

2. If a directory block is not found in the block cache, the ORS cache is searched for the partial directory block index. If the ORS cache entry cannot be found, the procedure for ORS cache miss handling is similar to the previous description of the getAttribute() file system call. The partial directory block index indicates the disk block of the desired directory block and allows it to be loaded from disk into the block cache.

3. If the complete directory block index is required, the block cache is searched. If the index cannot be found, is it loaded from disk and placed into the block cache. The target disk block, which contains the complete directory block index, is indicated in the very last entry of the partial directory block index. Once loaded into the block cache, the complete index can be used to load the appropriate directory content block from disk into the block cache.

The above steps are repeated recursively for each component of the directory path until the path name is resolved and the final target file token is found. The getDents() file system call, which returns the contents of a directory rather than the file token of a particular file of directory, interacts with the cache system in a similar fashion.

The getAttribute() and path name lookup operations cover most of the important code paths and their traces have demonstrated the role of the meta-data caches and the general operation of HFS.

Chapter 4

Experimental Setup

This chapter describes the environment in which we ran our experiments. Topics we discuss include our measurement approach, workload selection, simulated hardware setup, simulated disk setup, system environment configuration, graph interpretation, and simulated hardware characteristics.

In general, the experiments consisted mainly of a thread per processor accessing a disk exclusively. As the number of threads is increased, the number of processors and disks are increased proportionately. The time for each thread to perform its task is measured.

4.1 Measurement Approach

The goal of our experiments was to obtain an understanding of file system performance scalability from the bottom up. In this way, we could gain an understanding of scalability at various levels of complexity, from simple operations to complicated real-world workloads. We began with simple and easy to understand microbenchmarks in order to obtain basic performance numbers and an idea of scalability trends. Scalability bottlenecks were identified and incremental optimizations were implemented. We then iteratively added more complexity in the workload and repeated the process several times. Each time, we verified expectations and ensured that the results had logical explanations. The gradual progression helped in dealing with the many variables of the file system and workload. We were primarily interested in the scalability of meta-data management since in our experience, this is the source of most performance problems. File cache management and latency hiding using file block prefetching are responsibilities of the K42 file cache manager and were not examined in this thesis.

Speed-up, in terms of throughput, is the typical unit of measurement used in scalability results, however it is not as intuitively meaningful under our particular workloads. We were concerned with maintaining constant execution time per thread as the number of requesting threads, processors, and disks were increased. For example, let us assume that it takes X cycles to execute 1 request on a system with 1 processor and

1 disk. Ideally, on system with 12 threads of requests, 12 processors, and 12 disks, each request should also take X cycles to execute. However, this perfect speedup is not usually achieved because of problems such as false-sharing and lock contention, which increase execution time. Since we were concerned with maintaining the current level of performance for all threads of execution, we focused on the slow-down factor of the file system as the number of concurrent requests were increased. On a given number of processors, we calculated the average number of cycles required by each thread to execute its task. An ideal graph would have a perfectly horizontal, flat curve from 1 processor to N processors. These results allow for a more direct comparison across all processor configurations since, if there are scalability problems, we can directly see the increased average execution time of a thread.

4.2 Workload Selection

Selecting an appropriate workload to stress the file system is non-trivial. Different types of workloads have characteristics that stress the file system in different ways. General workload types include scientific, commercial/transaction based, and event-driven. Unfortunately, K42 is in the development stages and lacks infrastructure required to run many applications and benchmarks. Instead, custom file system benchmarks were developed that could run on the existing K42 infrastructure.

Although non-standard benchmarks were used, these were very quick to implement on K42 and offered a fully controllable, understandable, simple environment to gather performance information. Microbenchmark workloads include parallel file (1) reading, (2) writing, (3) creating, (4) statting[1], and (5) name lookup. These microbenchmarks stress the fundamental operations of the file system and form a common base that is applicable to all types of workloads.

The macrobenchmark we used consisted of a Web server simulator that replayed a portion of the World Cup 1998 aggregated Web log. It served to briefly verify that optimizations applied during the microbenchmarks were applicable to higher-level and more realistic workloads.

The custom benchmarks were designed to be I/O-bound, as suggested by Chen and Patterson [15]. According to them, when evaluating file systems, I/O-bound workloads should be used since they stress file systems more than other types of workloads. I/O-bound workloads spend the majority of the time performing file system operations rather than user-level computation.

The experiments were run using an optimistic configuration of 1 workload thread per processor and 1 disk per processor. Each workload thread is independent, executes in parallel, and only accesses its local disk. This configuration presents an easily scalable workload and enabled us to examine a parallel file system requirement mentioned by Kotz et al. [51, p. 13], in that file systems need to provide high performance access

[1] "Statting" refers to the common Unix file system call stat() that returns the attributes of a file. Some of these attributes include access permissions, owner, group, size, access time, and modification time.

Parameter	Value
CPU model	MIPS R4000
Number of CPUs	1–12
Clock frequency	150 MHz
Number of clocks	1
Memory bus bandwidth	177 MB/s
Memory size	128 or 512 MB
Number of memory banks	1
Disk model	Fixed
Fixed latency disk delay	0 or 15 ms
Number of cells	1
Number of consoles	1–12
Number of ethernet interfaces	1
Perfect memory latency	0 ns
Standard memory latency	2100 ns
NAK retry time	1 cycle

Parameter	Value
L1 data cache size	16 kB
L1 data cache line size	32 bytes
L1 data cache associativity	1 way
L1 instruction cache size	16 kB
L1 instruction cache line size	32 bytes
L1 instruction cache associativity	1 way
L2 cache size	1024 kB
L2 cache line size	128 bytes
L2 cache associativity	1 way
L2 cache access time	100 ns
L2 cache write buffer size	0 entries
Interprocessor cache transfer	1600 ns
Cache line upgrade time	1800 ns
Interprocessor interrupt latency	1000 ns
Short interprocessor send latency	1500 ns

Table 4.1: SimOS parameters.

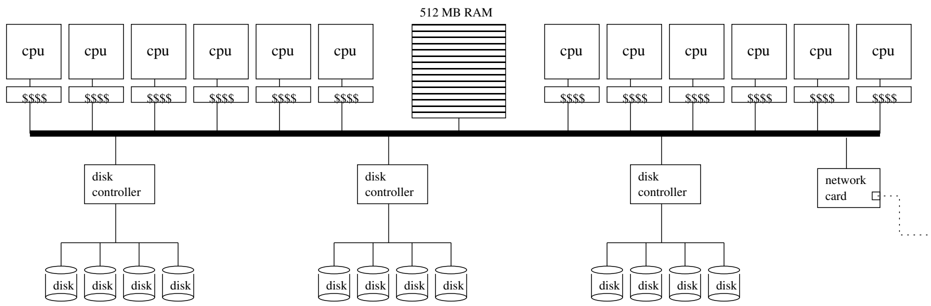

Figure 4.1: Maximally configured hardware.

to many concurrent and possibly unrelated jobs.

4.3 SimOS Software Simulator

The experiments were executed on the SimOS complete machine simulator [62]. SimOS is a software simulator of computer systems developed at Stanford University. It performs complete machine simulation of computers based on MIPS R4000, DEC Alpha, and IBM 64-bit PowerPC processors, running unmodified operating systems such as SGI Irix, Tornado, and K42. When simulating the MIPS R4000 processor, it is capable of presenting a cycle-accurate, validated [4, p. 73] [22, p. 24] simulation of the computer system.

Many simulator parameters can be changed, such as the number of processors, speed of processors, types of processors, size of caches, types of caches, speed of caches, size of memory, speed of memory, number of memory banks, speed of bus, type of bus, number of disks, speed of disks, size of disks, etc. The SimOS parameters used in the experiments are shown in **Table 4.1**. A diagram of the maximally configured hardware used for the experiments is shown in **Figure 4.1**. Due to temporary limitations in the K42 disk device driver software, a maximum of 12 disks were supported. For most of our experiments, disks were modeled as having zero latency, while a few experiments used 15 ms fixed latency disks. Using a 15 ms disk latency, while more realistic, greatly increased the time needed to run experiments on SimOS. Due to time constraints, we mainly used zero latency disks. We will later show that disk latency does not affect the scalability of our benchmarks. The *uniform memory access* (UMA) bus on the SimOS SMP multiprocessor uses a snoopy write-back invalidation protocol and its characteristics are best described by the following passage from the SimOS user guide [26].

> "This memory system models UMA memory system over a split-transaction out-of-order completion bus. Cache coherence is provided by snoopy caches with an invalidate protocol. Cache to cache transfers only occur when data is dirty in another processor's cache, in which case a sharing writeback is also issued to memory. Accesses to an address that's already outstanding will be merged or they will be delayed until the previous transaction completes. BusUma also models writeback buffers, and it has a fixed number of overflow buffers if all the writeback buffers are full."

This versatile simulator allowed us to easily scale the size of the computer system and experiment with parameters such as cache associativity and disk latency.

4.4 Disk Configuration

In the server-level microbenchmarks, accesses to the disk device drivers for file data were short-circuited to create infinitely fast devices so that stress could be placed on the file system rather than the disk device driver and disks, eliminating them as a potential scalability bottleneck source. To short-circuit the disk, source code to initiate calls from the disk interface component of HFS to the K42 disk device driver were deleted. However, meta-data accesses to disk were enabled to allow meta-data to populate the two meta-data cache systems.

Once meta-data is accessed from disk, it is placed in the meta-data cache for reuse. By running experiments twice, the first run allowed all meta-data to be loaded from disk into the meta-data cache. Since meta-data cache flushing was disabled, subsequent runs of the experiment used meta-data solely from the meta-data cache. However, disk access for file data blocks were still triggered but they were short-circuited so that disk accesses were not executed when running the server-level microbenchmarks. Using this methodology, we stressed the core functionality of HFS to the extreme. By removing the bottleneck created by the disks and disk device drivers, we could investigate the bottlenecks of the file system core. For other experiments, we re-enable disk access to determine its impact on performance scalability.

4.5 System Environment Configuration

To create a simple, controlled test environment, the experiments were mainly executed in the file system address space (server-level) and no other major servers were running at the time. For instance, the NFS server, RAM file system, toy file system, device file system, pipe server, and pty server were all disabled. For each microbenchmark, the computer system was booted, HFS was initialized, and the appropriate benchmark was executed. Running at the server-level allowed the file system to be stressed without invoking the VFS and FCM components of the operating system.

For the few user-level experiments, the services of the pipe server and pty server were required and enabled. When running the experiments at user-level, the K42 file cache manager cached all file data blocks and all requests for file data blocks were satisfied from the file cache from the second run of the experiment onward[2]. Also, the VFS component cached portions of file meta-data such as file attributes. Therefore, the user-level experiments stressed the FCM and VFS components of the operating system.

The version of K42 used did not have FCM page eviction (i.e. page-out) enabled because of problems in the implementation at the time the experiments were run. The workloads used in the experiments are not affected by this missing feature.

4.6 Measurements Taken and Graph Interpretation

We measured scalability for 1, 2, 4, 8, and 12 processor configurations. For the microbenchmarks, each experiment was repeated 10 times under each processor configuration. Each repetition consisted of an initial warm up run, followed by a subsequent identical measured run. For each thread, we measured the number of cycles taken to complete its task. For a measured run of the experiment on a uniprocessor, we obtain one number from the single thread of execution. On a dual processor, we obtain two numbers, one from each of the two threads of execution. On a four processor computer, we obtain four numbers from each of the four threads of execution.

After 10 measured runs, we have 10, 20, 40, 80, and 120 values for processor configurations of 1, 2, 4, 8, and 12, respectively. We plot the average of these corresponding numbers to obtain a curve. We also note the minimum and maximum cycle counts for each configuration.

The y-axis indicates the average number of cycles required by each thread while the x-axis indicates the number of processors. The ideal scalability graph would be a flat, perfectly horizontal curve, indicating that thread execution time is independent of system size.

Due to time constraints and speed limitations of the SimOS simulator, the macrobenchmark experiments were repeated only twice.

[2]Main memory is sized adequately to allow all file data to be cached.

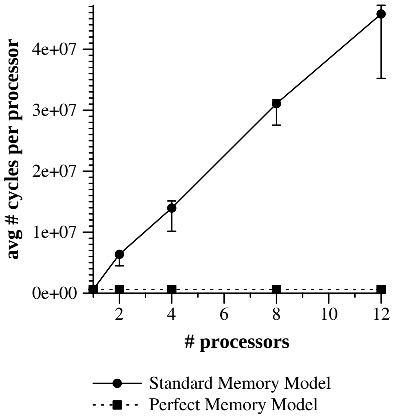

Figure 4.2: Ping-pong effect on memory models.

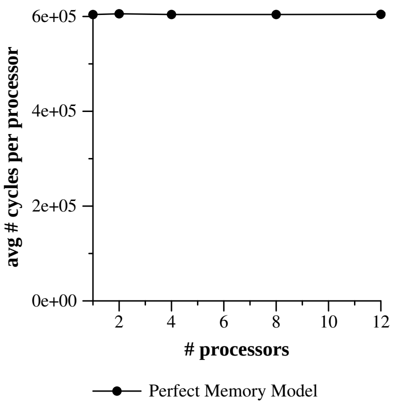

Figure 4.3: Ping-pong effect on perfect memory model, magnified.

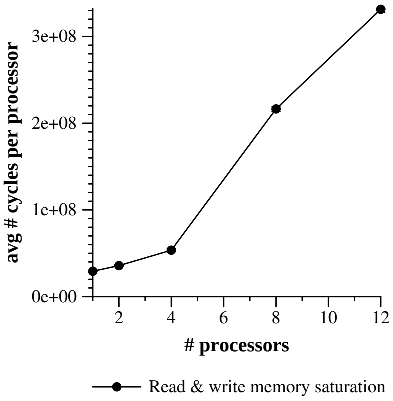

Figure 4.4: Memory system saturation.

4.7 Hardware Characteristics

Having a good understanding of the underlying hardware characteristics is the first step in understanding results from the benchmarks. This section experimentally demonstrates the basic scalability limitations of the chosen simulated hardware. These characteristics should be kept in mind when examining subsequent scalability results. In this section, we describe the SimOS "standard" and "perfect" memory models and identify the saturation point of the memory bus and memory bank.

4.7.1 Perfect Memory Model Characteristics

SimOS contains two memory models of simulation, called standard and perfect. The standard memory model simulates the timing of a realistic memory system while the perfect memory model simulates an ideal memory system where memory accesses require zero time. While the standard memory model was used in most experiments, the perfect memory model was used as an exploratory tool, since it eliminates memory contention effects and allows other problems to be revealed, such as lock contention. Under the perfect memory model, memory bus and memory bank access times are zero, and cache consistency operations incur zero overhead. As a result, contention does not exist.

A quick experiment was done to demonstrate the nature of the SimOS perfect memory model. The experiment contained a global array of 8-bit counters, where each processor incremented a designated counter 100,000 times in a tight loop. This access pattern generates a lot of cache coherence traffic since the counters are tightly packed together and reside on 1 primary data cache line. The resulting false sharing should cause a "ping-pong" effect on the cache line.

As shown in **Figure 4.2**, the standard memory model had linear performance degradation, as expected. However, the perfect memory model had no performance degradation at all. A magnification of the perfect memory results is shown in **Figure 4.3**. These results demonstrate that the perfect memory model eliminated memory bank access time, bus access time, and cache coherency overhead.

4.7.2 Memory System Saturation

The SimOS memory system parameters are given in **Table 4.1**. For instance: 177 MB/s bus bandwidth, 1 memory bank, and 2100 ns memory access time are critical memory parameters. However, these parameters alone do not describe memory system behavior under increasing contention. We ran a simple experiment to depict the characteristics and limitations of the memory system. File system scalability is limited to the maximal throughput of these hardware components. In the experiment, each processor executes a thread that sequentially traverses a separate, independent 2 MB array, reading and modifying each array element. The nature of the modification is a simple increment of the current value in the array element. Each element is 1 byte in size. The array is allocated local to each processor, using K42's memory allocation routines. The allocation performs padding at the array boundaries to prevent false sharing of cache lines. The 2 MB array size ensured the data set would not fit in the 1 MB secondary cache so that read and write accesses to main memory were necessary. The goal of the experiment was to saturate the bus or the sole memory bank with memory traffic from the array accesses to determine the maximum load that can be placed on the system.

Figure 4.4 shows that performance was satisfactory for up to 4 processors but degraded drastically beyond this point. Under continuous, intense pressure, the bus or memory bank could adequately service a maximum of 4 processors. In practice, this worst-case scenario should rarely occur since the processor caches should reduce the amount of memory bus traffic.

Chapter 5

Microbenchmarks – Read Operation

5.1 Purpose

One of the basic tasks of a file system is to act as a translation layer between the operating system and the disk device driver. The file system receives a logical file block number from the operating system, translates it into a physical disk block number, and asks the disk device driver to operate on the physical disk block. The first microbenchmark measures the scalability of this fundamental operation.

5.2 Experimental Setup

The first benchmark attempts to determine the maximum concurrency and throughput of the file system under the following configuration. In this experiment, a worker thread is allocated to each processor. In parallel, each thread sequentially reads a separate 12,845,056 byte extent-based file located on separate disks that it accesses exclusively. **Figure 3.2** on page 17 shows an example of the block index of an extent-based file. The particular file size was chosen so that the block index occupied exactly 1 meta-data cache entry. A 12,845,056 byte file contains 3136 blocks. Each thread performs the following general operations 3136 times. (1) Obtain the file status on the target file. (2) Perform a block read operation on the target block. (3) Update the file status information. Steps 1 and 3 are required to reflect the time-stamping activity of the file system during a read operation.

An ideal file system would scale perfectly under this workload since each thread is operating on its own processor and disk. A conceptual view of the experimental setup is shown in **Figure 5.1**. 128 MB of RAM was provided. Optimizations and variations are explored in Section 5.4 and Section 5.5.

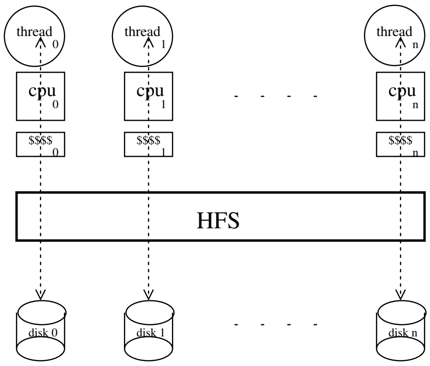

Figure 5.1: Experimental setup.

CPU #	Hash List #s
0	12, 13
1	20, 21
2	28, 29
3	36, 37
4	44, 45
5	52, 53
6	60, 61
7	4, 5
8	12, 13
9	20, 21
10	28, 29
11	36, 37

Table 5.1: ORS hash lists accessed.

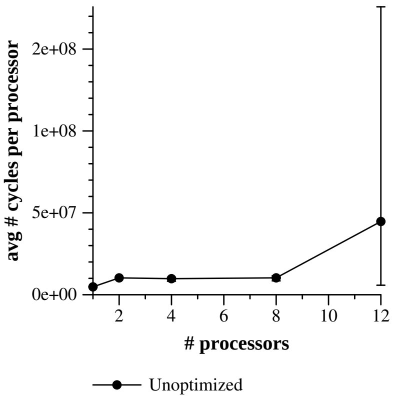

Figure 5.2: Extent read – unoptimized.

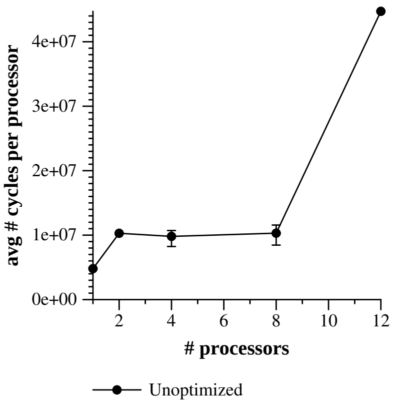

Figure 5.3: Extent read – unoptimized.

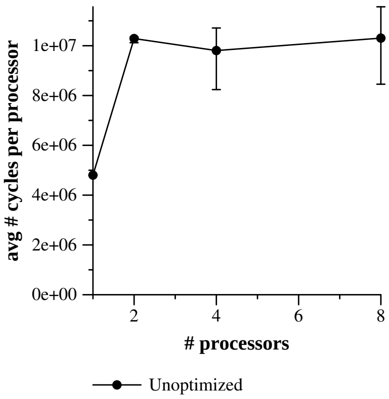

Figure 5.4: Extent read – unoptimized, magnified.

5.3 Analysis of Results

The results of the experiment are shown in **Figure 5.2** and **Figure 5.3** with and without error bars on the 12 processor configuration, respectively. Performance degraded drastically between 8 processors and 12 processors. On average, it took each thread more than five times as long to complete the task when 12 threads were running compared to when 8 threads were running.

Simulation traces identified the ORS cache to be responsible for the degradation. More specifically, the hash function used by the ORS cache was the source of the problem, causing severe hash list lock contention. While the ORS cache hash table contained 64 hash lists, under the microbenchmark workload used, the hash function did not adequately distribute the cache entries among the lists, leading to collisions and hash list lock contention. As shown in **Table 5.1**, a number of hash lists were shared between processors resulting in contention on hash list locks. Cache tracing showed that each thread repeatedly accessed 2 ORS cache entries. These 2 entries resided in separate hash lists. With 12 threads on a 12 processor system, 24 hash lists should have been occupied in the ideal case. However, the hash function used by HFS caused only

16 hash lists to be used, resulting in 8 hash lists that were shared. For example, processors 0 to 3 shared hash lists with processors 8 to 11. The wide variability on 12 processors, as shown by the large error bars in **Figure 5.2**, was the result of some threads sharing hash lists while other threads had exclusive access to hash lists. Nevertheless, the performance degradation on a contended hash list was higher than expected, considering that only 2 processors were sharing any contended hash list. Performance did not simply slow down by a factor of 2, as one may intuitively expect. The choice of lock types was examined in a later experiment to determine if it was the culprit behind this severity. Regardless, it was necessary to modify the ORS hash function to reduce hash list collisions under a variety of workloads. It was surprising that hash collisions occurred under the simple microbenchmark workload.

Figure 5.4 magnifies the left portion of **Figure 5.2**. It shows that there is also a serious performance degradation between 1 processor to 2 processors. The costs of transitioning from a uniprocessor to multiprocessor architecture can be clearly seen. Overheads may include memory bus contention, memory bank contention, multiprocessor locks, cache-coherence misses, and cache-coherence traffic. These problems do not exist on uniprocessors. As a result, two concurrent threads required twice as long to complete their independent tasks compared to one thread. Performance between 2 processors and 8 processors showed ideal scalability since hash collisions did not occur in this range of processors.

5.4 Optimizations

We implemented a number of optimizations to improve scalability. These optimizations include (1) a modified ORS hash function, (2) larger ORS hash table, and (3) padded ORS hash list headers and cache entries. The fully optimized results are shown in **Figure 5.5**[1]. This section describes the investigation process that led to these optimizations. We used the perfect memory model in Section 5.4.1 to examine memory bottlenecks. A base line experiment was run in Section 5.4.2 to verify scalability of the underlying infrastructure. The impact of lock types and cache associativity were examined in Sections 5.4.3 and 5.4.4, respectively. As described in Section 5.4.5, modifying the ORS hash function alone was not adequate and introduced other performance problems. Section 5.4.6 indicates that padding the ORS hash list headers alleviated these problems. Increasing the number of hash lists (increasing the size of the hash table) can be done without detrimental effects, as shown in Section 5.4.7. Further padding of the ORS cache entries significantly improved absolute performance in Section 5.4.8. In the investigation process, we discovered a potential problem with the K42 padding and alignment facilities. Finally, Section 5.4.9 describes the fully optimized configuration in detail.

The experimental configurations used here are based on those used in Section 5.2 with minor modifications as described in each subsection. We continue to use extent-based files in the experiments to explore ORS

[1] Error bars were removed from the unoptimized 12 processor configuration.

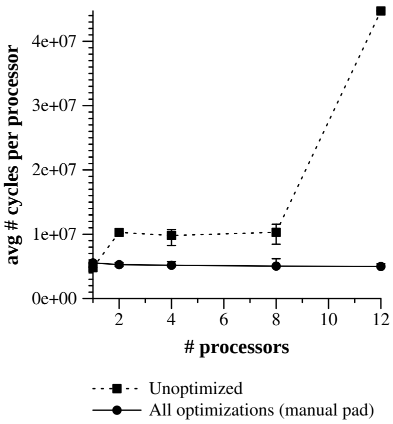

Figure 5.5: Extent read – fully optimized.

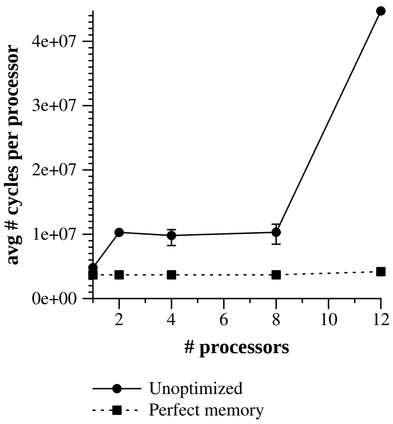

Figure 5.6: Extent read – perfect memory.

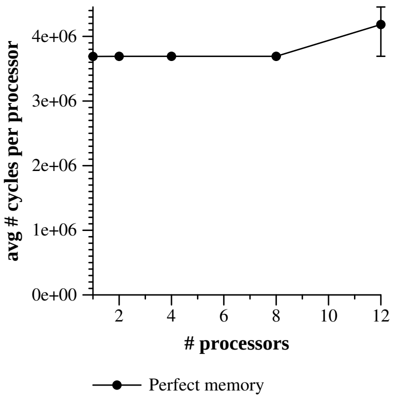

Figure 5.7: Extent read – perfect memory, magnified.

cache performance. Regular, random-access, block-based files are used in the subsequent section (Section 5.5) to explore block cache performance.

5.4.1 Perfect Memory Model

This experiment was based on the previous experiment (Section 5.2), but uses the perfect memory model rather than the standard memory model. The differences between the perfect and standard memory model results provide an indication of the time spent waiting for memory operations and any possible memory contention. The results, shown in **Figure 5.6**[2], indicate that the number of cycles required by each thread remained fairly constant under all processor configurations, especially when compared against the standard memory model results. On 2 to 8 processor configurations, memory system access time in the standard memory model contributed approximately 6,300,000 additional cycles to each thread, more than doubling thread execution time when compared against the perfect memory model.

Hash list lock contention, shown in the 12 processor configuration, caused only a minor performance degradation, increasing average thread execution time by approximately 500,000 cycles. A magnification of the results is shown in **Figure 5.7**. The increase is due solely to hash list lock contention, that is, the sharing of hash list locks as described in Section 5.3. Operations performed under the protection of a hash list lock occur very quickly due to the perfect memory model. Since the hash list lock is held for only a short period of time, contention is low, leading to a relatively small time increase in the 12 processor configuration.

These results suggest that the severe degradation seen in the standard memory model on 12 processors may be largely attributed to hash list lock contention, exacerbated by memory contention problems.

[2]Error bars were removed from the unoptimized 12 processor configuration.

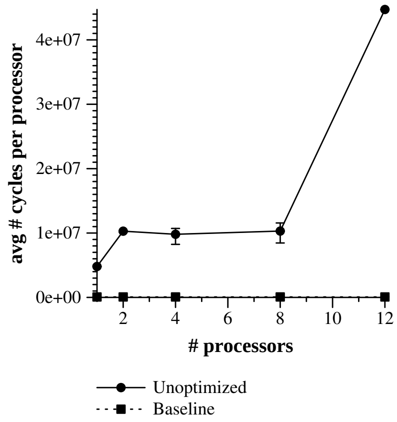

Figure 5.8: Extent read – baseline.

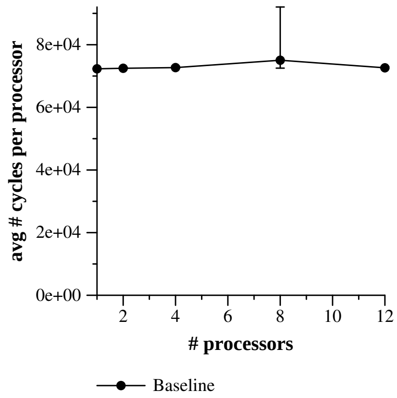

Figure 5.9: Extent read – baseline, magnified.

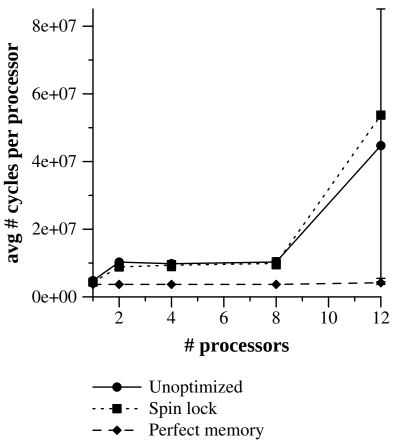

Figure 5.10: Extent read – spin-only locks.

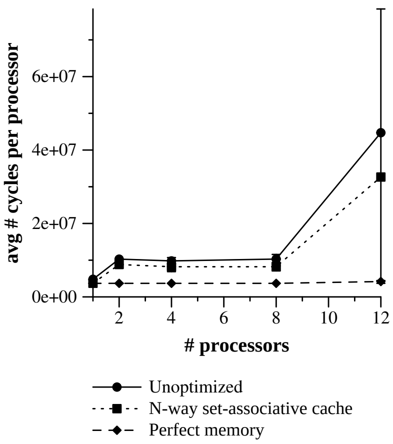

Figure 5.11: Extent read – 4-way primary and 2-way secondary cache.

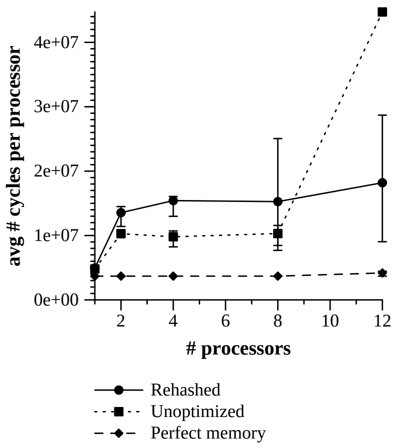

Figure 5.12: Extent read – rehashed.

5.4.2 Baseline

This experiment assesses whether K42 and the microbenchmark facilities themselves were scalable without considering HFS. Due to the rapidly changing development environment and the experimental nature of K42, the performance and scalability of K42 needs to be frequently verified. The configuration for this experiment was the same as Section 5.2 where the standard memory model was used. The code path was identical except that calls to the core of HFS were short-circuited. Calls reached as far as the HFS interface layer but no further. More specifically, calls to the HFS core to acquire and update meta-data and perform the read operation were excluded. The results are shown in **Figure 5.8**[3]. A magnification of the results is shown in **Figure 5.9**. As expected, the results showed ideal scalability, indicating that K42 and the microbenchmark mechanisms were indeed scalable, and were a negligible portion of the overhead.

[3]Error bars were removed from the unoptimized 12 processor configuration.

CPU #	New Hash List#s	Old Hash List#s
0	12, 13	12, 13
1	15, 16	20, 21
2	18, 19	28, 29
3	21, 22	36, 37
4	24, 25	44, 45
5	27, 28	52, 53
6	30, 31	60, 61
7	33, 34	4, 5
8	36, 37	12, 13
9	39, 40	20, 21
10	42, 43	28, 29
11	45, 46	36, 37

Table 5.2: New ORS hash lists accessed.

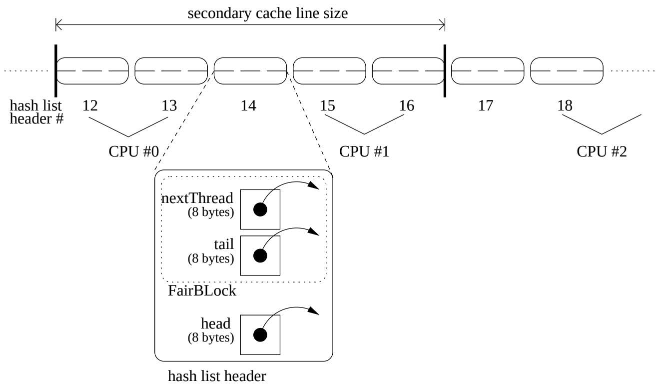

Figure 5.13: An example of false sharing.

5.4.3 Spin Lock

The spin-then-block locks that were used in the ORS hash lists of the original configuration were a potential source of the performance problems. In this experiment we replaced the spin-then-block locks with spin-only locks. Perhaps the overhead of blocking and waking up threads that were contending on the hash list locks could be reduced by using spin-only locks. Except for the lock modifications, the configuration was the same as in Section 5.2. The results indicate that performance did not improve, as shown in **Figure 5.10**[4]. The spin-then-block locks were not a source of the performance problems.

5.4.4 N-Way Set-Associative Cache

This experiment used the same configuration as Section 5.2, except that n-way set-associative caches were used rather than 1-way set-associative caches. 4-way set-associative primary instruction and data caches, and a 2-way set-associative secondary unified cache were used[5]. This configuration was used to determine the sensitivity of the benchmark and HFS to processor caching. The results are shown in **Figure 5.11**[6]. Execution time improved slightly and by a constant amount when compared to the original results. There were no fundamental improvements in scalability. Increasing cache set-associativity did not solve scalability problems.

5.4.5 Improved Hash Function

In an attempt to improve scalability, the hash function was modified so that each thread hashed to exclusive hash lists. **Table 5.2** shows the hash lists accessed under the modified hash function. The results are shown

[4]Error bars were removed from the unoptimized 12 processor configuration.
[5]We had difficulty in getting SimOS to execute using a 4-way set-associative secondary cache.
[6]Error bars were removed from the unoptimized 12 processor configuration.

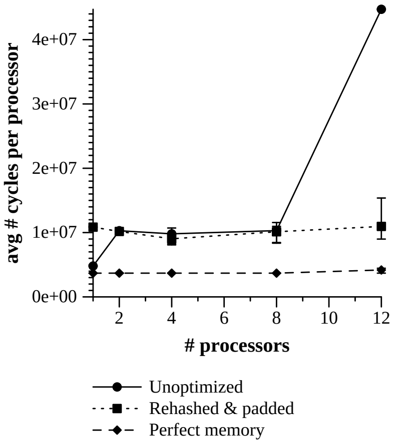

Figure 5.14: Extent read – rehashed & padded ORS hash table.

CPU	New Hash List#s	Old Hash List#s
0	12, 13	12, 13
1	20, 21	20, 21
2	28, 29	28, 29
3	36, 37	36, 37
4	44, 45	44, 45
5	52, 53	52, 53
6	60, 61	60, 61
7	68, 69	4, 5
8	76, 77	12, 13
9	84, 85	20, 21
10	92, 93	28, 29
11	100, 101	36, 37

Table 5.3: ORS hash lists accessed.

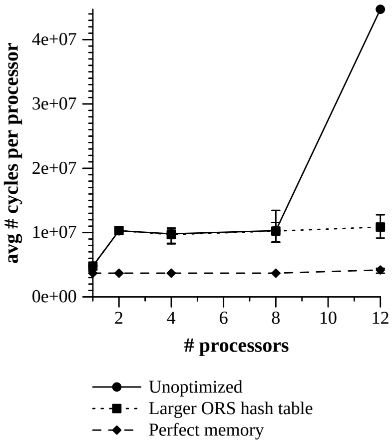

Figure 5.15: Extent read – larger ORS hash table.

in **Figure 5.12**[7]. The performance was worse than the original unoptimized configuration between 1 to 8 processors but significantly better on 12 processors. The degradation between 1 to 8 processors was due to the denser usage of the hash table. Since the hash list headers and corresponding embedded locks reside contiguously in memory due to the K42 memory allocation algorithm, a denser usage of the locks led to increased false sharing of secondary cache lines. **Figure 5.13** illustrates an example where false sharing can occur. In this example, processors 0 and 1 cause false sharing because they concurrently access hash list headers 12, 13, and 15, 16, respectively. Since these hash list headers all occupy the same cache line, this cache line will "ping-pong" between the two processors and result in reduced performance.

The increased false sharing caused increased memory and cache coherence traffic. Performance on 12 processors was similar to performance on 8 processors. A major scalability bottleneck was not introduced when moving from 8 to 12 processors, unlike in the original unoptimized configuration, where hash list lock contention occurred. In the rehashed experiment, the only difference between 8 and 12 processors was a marginal increase in memory bus traffic and memory bank contention. The increasing range of the error bars in **Figure 5.12** between 4 and 12 processors was due to the increasing memory bus contention from additional false sharing between additional pairs of processors.

5.4.6 Improved Hash Function & Padded ORS Hash Table

False sharing can often be eliminated by properly padding the affected data structures. This experiment is based on the configuration in Section 5.4.5 with the modified ORS hash function, where each hash list header was padded to the end of the secondary cache line to eliminate false sharing problems. The results

[7]Error bars were removed from the unoptimized 12 processor configuration.

are shown in **Figure 5.14**[8]. The results between 2 to 8 processors closely mirror the results of the original configuration in Section 5.3, since under both configurations, false sharing did not occur. Results on up to 12 processors show near-ideal scalability. Uniprocessor performance was worse than the original unoptimized version, taking twice as long to complete the task. We suspect that the K42 padding facilities that were used may have added additional overhead to the uniprocessor configuration since this pattern of behavior appeared in other subsequent experiments, but the exact reason has not been determined.

5.4.7 Larger ORS Hash Table

An important scalability principle is outlined by Peacock et al. [56, p. 86], "namely that queue sizes should remain bounded by a constant as the system size is increased". In terms of HFS, this meant that the ORS hash lists should each maintain a constant average length as the size of the multiprocessor increases. We attempted a one-time increase in the ORS hash table size that should be adequate in meeting this requirement for up to 12 processors. To examine the effects of a larger ORS hash table, the original configuration (Section 5.2) was used in this experiment and the hash table was doubled in size from 64 hash lists to 128 hash lists. Increasing the size of a hash table generally reduces the probability of hash collisions under a variety of workloads and the subsequent sharing of hash lists and locks. The hash function code was not modified since it automatically accounted for hash table size. This change maintained the original density of hash table entry usage. **Table 5.3** shows the hash lists accessed under this configuration. Based on this table and **Figure 5.13**, there was no possibility of false sharing because the hash list headers accessed by each processor were adequately spaced apart in memory to prevent the sharing of cache lines. The results are shown in **Figure 5.15**[9] and are similar to the results of Section 5.4.6 since false sharing did not occur, unlike in Section 5.4.5. Results on up to 12 processors show near-ideal scalability. The combination of a larger, rehashed, and padded ORS table should produce optimum scalability and will be explored in Section 5.4.9.

The uniprocessor anomaly that was present in Section 5.4.6 did not appear in the results of this section. Since the K42 padding facilities were not used in this section, these results further suggest that the source of the anomaly in Section 5.4.6 was the K42 padding facilities.

5.4.8 Padded ORS Cache Entries & Hash List Headers

This experiment explored the impact of eliminating more potential sources of secondary cache line false sharing. Both the ORS hash list headers and the ORS cache entries themselves were padded. Other than these changes, the original configuration from Section 5.2 was used. The results are shown in **Figure 5.16**[10].

[8]Error bars were removed from the unoptimized 12 processor configuration.
[9]Error bars were removed from the unoptimized 12 processor configuration.
[10]Error bars were removed from the unoptimized 12 processor configuration.

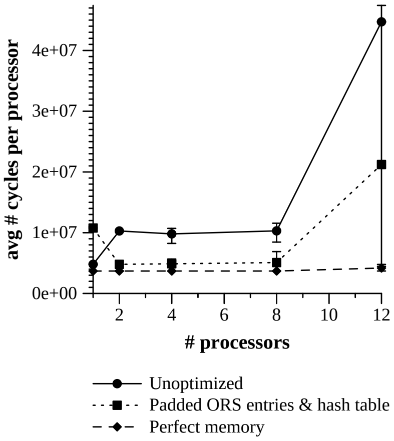

Figure 5.16: Extent read – padded ORS cache entries & hash list headers.

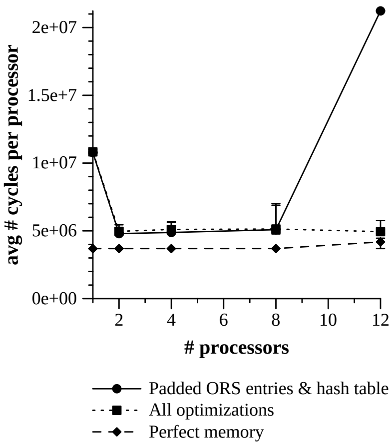

Figure 5.17: Extent read – all optimizations.

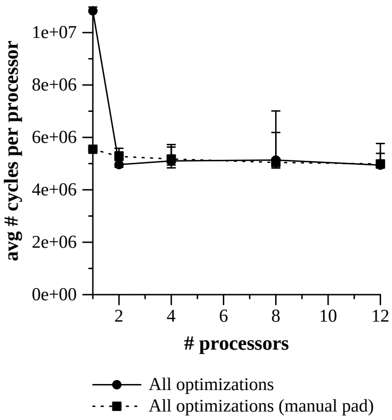

Figure 5.18: Extent read – fully optimized.

Uniprocessor performance was worse than the original unoptimized version, taking twice as long to complete the task. This unexpected behavior was also seen in Section 5.4.6. We briefly ran a similar experiment where the hash list headers were manually padded instead of using the K42 padding facilities and found that the uniprocessor anomaly was no longer present. Again, this is further evidence that the K42 padding facilities may have added unnecessary overhead to the uniprocessor configuration.

As shown in **Figure 5.16**[11], results between 2 and 8 processors approach the performance of the perfect memory configuration. Secondary cache line conflicts from the ORS cache entries caused execution time to double. The original hash function was used so the expected poor performance for 12 processors appeared again. A change in the hash function could lead to near-ideal performance and scalability and is attempted in Section 5.4.9.

5.4.9 All Optimizations Combined

This experiment combined the individual optimizations to determine whether ideal performance and scalability could be achieved. The original configuration from Section 5.2 was used along with the following modifications.

1. Modified ORS hash function (Section 5.4.5).
2. Increased ORS hash table size (Section 5.4.7).
3. Padded ORS hash list headers (Section 5.4.6).
4. Padded ORS cache entries (Section 5.4.8).

For simplicity, the K42 padding facilities were used rather than manually padding and aligning the data structures. The results are shown in **Figure 5.17**[12]. Performance mirrors that from Section 5.4.8 between

[11]Error bars were removed from the unoptimized 12 processor configuration.
[12]Error bars were removed from the padded 12 processor configuration.

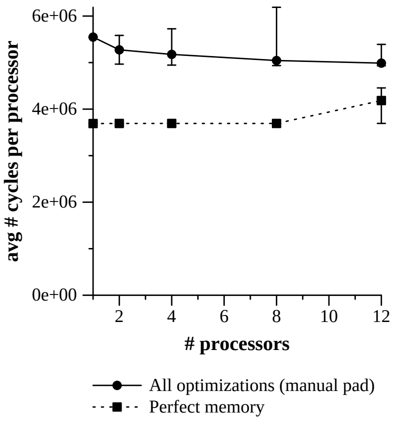

Figure 5.19: Extent read - fully optimized, magnified.

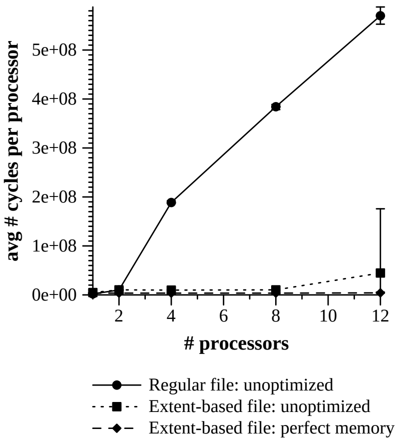

Figure 5.20: Regular read.

CPU	**New Hash List#s**	**Old Hash List#s**
0	0	1
1	1	1
2	2	1
3	3	1
4	0	1
5	1	1
6	2	1
7	3	1
8	0	1
9	1	1
10	2	1
11	3	1

Table 5.4: Hash lists accessed in the block cache.

1 to 8 processors. Except for the uniprocessor anomaly due to the K42 padding facilities, ideal scalability was achieved on 2 to 12 processors.

Manual padding of the ORS hash list headers was attempted and the results are shown in **Figure 5.18**. As expected, the uniprocessor anomaly was largely eliminated. Performance closely mirrors the perfect memory model results in Section 5.4.1, as shown in **Figure 5.19**, indicating that the experiment executed almost completely from secondary cache and with very few cache line conflicts. Finally, **Figure 5.5**[13] compares the fully optimized results against the original unoptimized results.

5.4.10 Summary

Ideal scalability was achieved for the extent-based read experiments using the combination of a modified ORS hash function, a larger ORS hash table, and padded ORS hash list headers and cache entries. The K42 memory padding and alignment facilities may have uniprocessor performance problems, as demonstrated in these experiments, and further investigation is needed.

5.5 Variations: Regular, Random-Access Files

This section describes results from using regular, random-access, block-based files rather than extent-based files. Block-based files use a simple block index where each entry maps a logical file block number to a physical disk block number. **Figure 3.2** on page 17 shows an example of such a block index. These regular files are intended for more common use in HFS than the more specialized extent-based file type. Using regular files enabled us to stress the block cache system and examine its performance.

[13]Error bars were removed from the unoptimized 12 processor configuration.

5.5.1 Unoptimized HFS

This experiment was based on the configuration of Section 5.2, except that regular 4 MB files were used rather than extent-based 12 MB files. There were two reasons for this selection. First, the file size was chosen so the block index occupied exactly 1 meta-data cache entry, making it simple to trace cache entries. Second, initial trials using 12 MB regular files were very time consuming. Using 4 MB files was adequate to show the performance characteristics.

The results, shown in **Figure 5.20**, indicate extremely poor performance for more than 2 processors, especially when compared with results from the extent-based file experiments in Section 5.2, despite reading only 1/3 as much data (4 MB vs 12 MB). The obvious bottlenecks identified by code inspection were (1) the hash function of the block cache system, (2) the limited number of hash lists (only 4) in the block cache system, and (3) the global lock used in the block cache system. The original hash function resulted in all processors sharing a single hash list. The reason for having only a limited number of hash lists in the original HFS design has not been determined. The global lock is used to protect a variety of data structures from conflicting concurrent accesses.

Outline of Optimizations

We subsequently applied a number of optimizations. These include (1) padded hash list headers, cache entries, and other critical data structures, (2) modified hash functions, (3) larger hash tables, (4) modified free list, and (5) use of fine grain locks. The fully optimized results are shown in **Figure 5.21**. With the optimizations, good scalability was achieved. Sections 5.5.2 to 5.5.9 describe the investigation process that led to these results. A modified block cache hash function or an increased number of block cache hash lists were ineffective, as shown in Sections 5.5.2 and 5.5.3, respectively. Surprisingly, implementing fine grain locks in Section 5.5.5 along with the padding of critical data structures in Section 5.5.6 did not result in scalability either. As demonstrated in Section 5.5.7, the first critical bottleneck was the block cache free list, followed by the global block cache lock, as shown in Section 5.5.8. Finally, Section 5.5.9 describes the fully optimized configuration in detail.

Many of the experiments are described in part to demonstrate how non-trivial it is to find bottlenecks in large, complex, parallel system software.

5.5.2 Improved Hash Function

This experiment examined the impact of modifying the hash function of the block cache system. The configuration is similar to Section 5.5.1 except that the hash function was modified to better distribute the use of hash lists, as shown in **Table 5.4**. The hash table size used by the block cache remained unmodified at

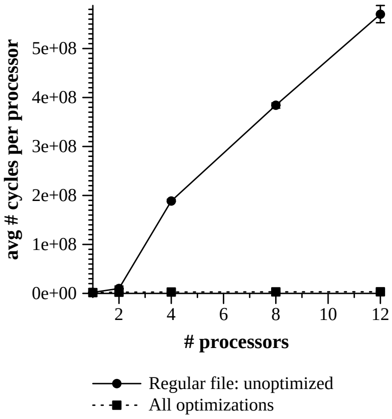

Figure 5.21: Regular read – fully optimized.

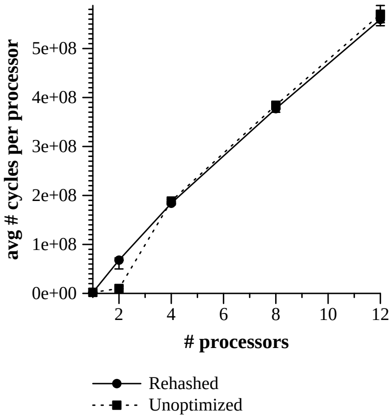

Figure 5.22: Regular read – rehashed block cache.

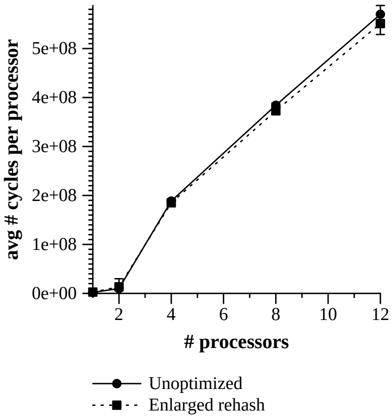

Figure 5.23: Regular read – larger rehashed block cache.

4 hash lists. The results, shown in **Figure 5.22**, were very similar to the unoptimized version in Section 5.5.1. We expected performance to improve slightly for up to 4 processors since each processor used a separate hash list, but performance remained poor. These results suggest that the global lock in the block cache system was the critical bottleneck.

5.5.3 Improved Hash Function & Larger Hash Table

This experiment provided more evidence to determine whether the global lock in the block cache system was the critical bottleneck. Based on the modified hash function configuration in Section 5.5.2, the hash table size was also increased from 4 hash lists to 64 hash lists, causing each processor to hash to a different list. This increase should help maintain the constant queue length principle outlined by Peacock et al. [56, p. 86].

If scalability did not improve, we would have further evidence that the global lock in the block cache was inhibiting concurrent access to unrelated hash lists. However, if scalability improved, we could conclude that the global lock was not a critical bottleneck point under our particular workload and configuration.

The results are shown **Figure 5.23**. As expected, performance did not improve and was very similar to the unoptimized results in Section 5.5.1. These results further suggest that the global lock in the block cache system was the critical bottleneck.

5.5.4 Fine Grain Locks

The design decision to use a single global lock in the block cache may have been chosen to reduce the number of locks that must be acquired when accessing the cache. This design decision, described by Peacock in [55, p. 32], can improve performance when the amount of contention on the global lock is low. However, when lock contention is high, this design can lead to poor performance.

The block cache contains 1 global lock to protect all of its queues, as shown in **Figure 3.5** on page 21.

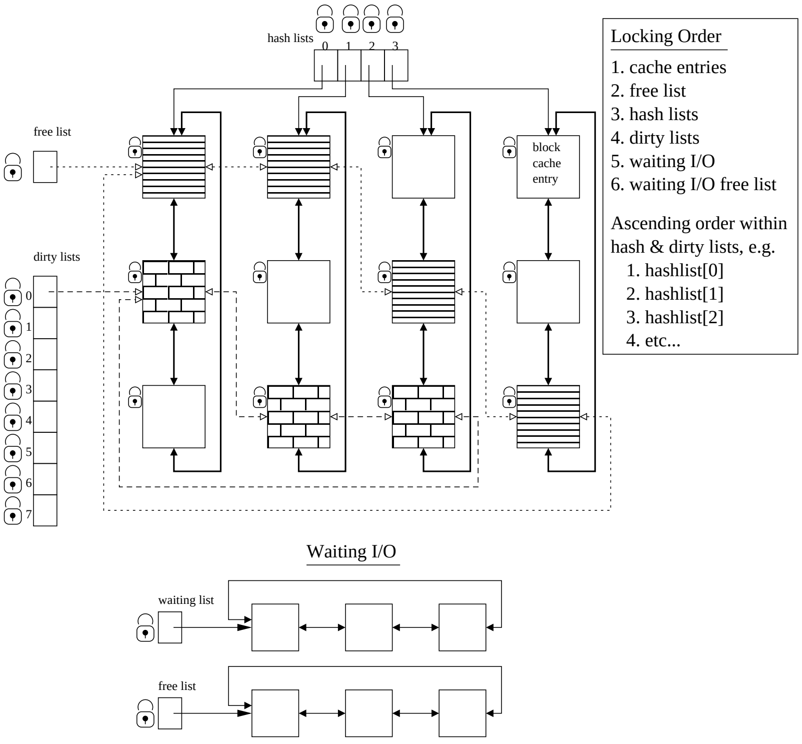

Figure 5.24: Optimized block cache.

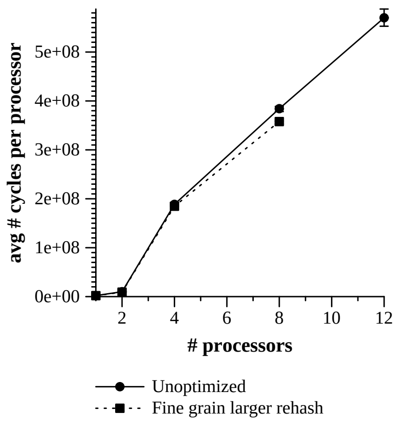

Figure 5.25: Regular read – fine grain locks, larger rehashed block cache.

These queues include the hash lists, free list, dirty lists, waiting I/O list, and waiting I/O free list. Traversals and modifications of these lists require the global lock to be acquired first. The elements of the block cache entries themselves are protected from simultaneous modification by a busy bit flag. An asserted busy bit indicates that the cache entry is currently being modified by another thread. To ensure correctly functionality, the busy bit can only asserted when the global lock is acquired. Using a single global lock makes it easy to ensure multiprocessor and multi-threaded correctness, but potentially at the cost of parallelism.

To optimize the block cache, the global lock was eliminated and replaced by several finer-grain locks, as shown in **Figure 5.24**. A separate lock was introduced for each hash list, dirty list, free list, waiting I/O list, and waiting I/O free list. In addition, a busy lock was introduced to replace the cache entry busy bit flag. In order to avoid deadlock, a locking order was defined. Various techniques such as lock releasing, re-acquiring, and retry logic were added to ensure proper operation. The implementation of the fine grain locking is similar to Peacock's design [55, p. 22].

We recognize that our implementation may not be the most optimal[14] but we were more concerned with developing a quick solution and examining the impact on scalability. Further optimizing the fine grain lock implementation was left as a future task. Since hash list locks are sometimes dropped and re-acquired, rescanning hash lists is sometimes necessary. As suggested by LoVerso et al. [40], time-stamps could be used to further improve performance. This additional optimization was not implemented because we wanted to

[14] For instance, a detailed analysis to find the optimal locking order was not conducted.

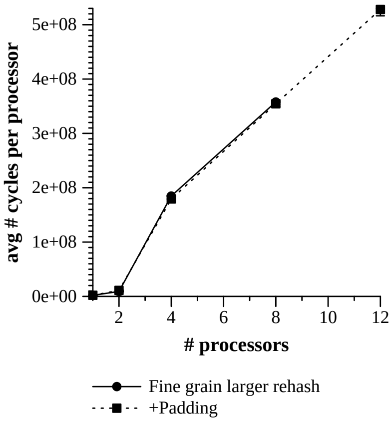

Figure 5.26: Regular read – previous optimizations +padding.

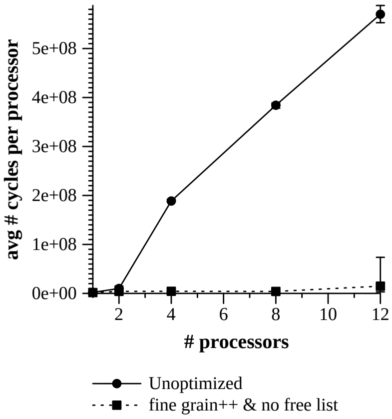

Figure 5.27: Regular read – fine grain locks++, no free list.

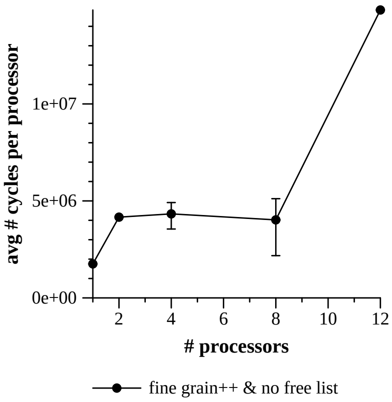

Figure 5.28: Regular read – fine grain locks++, no free list, magnified.

obtain performance results on the initial optimizations before adding more code complexity. This task was left for future work.

5.5.5 Fine Grain Locks, Improved Hash Function, & Larger Hash Table

This experiment examined the impact of using finer grain locks in the block cache, a modified hash function, and larger hash table. The results are shown in **Figure 5.25**[15]. Unfortunately, decomposition of the global lock into finer grain locks did not improve performance. Results were identical to the unoptimized case in Section 5.5.1. The global lock was not the critical bottleneck. Perhaps the global lock provided the advantages of code simplicity without sacrificing performance after all.

5.5.6 + Padding

This experiment explored whether additional optimizations could improve scalability. The configuration is based on the configuration of Section 5.5.5, using fine grain locks on the block cache, the modified hash function, and larger hash table. Additional optimizations include:

1. All critical data structures were padded to the size of the secondary cache line. These data structures included the ORS hash list headers, ORS cache entries, block cache entries, locks, and waiting I/O data structures (shown in **Figure 3.4** on page 20 and **Figure 3.5** on page 21).

2. Included the modified ORS hash function described in Section 5.4.5 on page 39.

For simplicity, the K42 padding facilities were used rather than manually padding and aligning the data structures. The results are shown **Figure 5.26**. Again, scalability remained poor and the optimizations had no effect on the results.

[15]12 processor results could not be obtained due to OS deadlock problems.

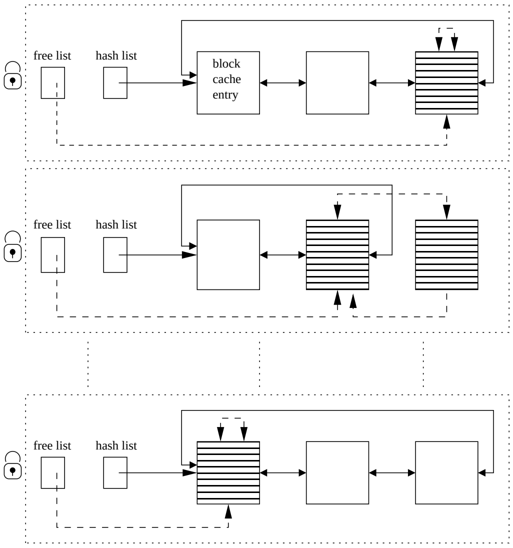

Figure 5.29: Software set-associative cache.

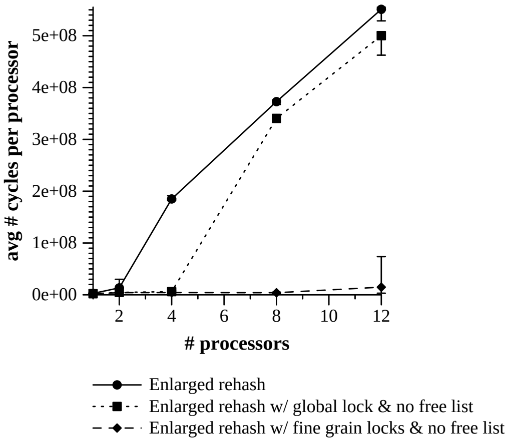

Figure 5.30: Regular read – global lock++, no free list.

5.5.7 Fine Grain Locks, Larger Rehashed Table, No Free List

This experiment examined the performance impact of the free list. The block cache used a single free list to enqueue freed cache entries and this could be a bottleneck. The configuration for this experiment is similar to Section 5.5.5, except that the block cache free list was not used and HFS simply left the freed cache entries in their current hash lists. This modification did not affect the correctness of the file system under this particular workload. Since the cache entries were not reused, additional free cache entries were automatically allocated when needed.

The results, shown in **Figure 5.27**, showed major improvements in scalability when compared against the unoptimized version in Section 5.5.1. We conclude that the free list was a critical bottleneck under this workload. However, a magnification of the results, shown in **Figure 5.28**[16], indicate that scalability problems were not completely eliminated since it took approximately 8.5 times as long for each thread to run on 12 processors than on 1 processor.

A consistent anomaly was seen on the 8 processor configuration. Across all 10 runs of the experiment, processor #7 consistently had a lower than average cycle count of approximately 2,300,000 cycles, causing the lower error bar seen in the graph. This was despite all processors having access to exclusive hash lists and memory, implying that all threads should have completed at approximately the same time. Due to the small scalability impact of this anomaly, further investigation was not a priority and was not pursued in this thesis.

An obvious solution to the free list bottleneck is to use one free list per hash list. However, balancing the number of entries on the free lists may need to be considered. A potential solution is the software set-associative cache structure described by Peacock [56, p. 86] and shown in **Figure 5.29**. Each hash list

[16]Error bars were removed from the 12 processor configuration.

is implemented as an array, containing a fixed number of elements. If all elements of an array are occupied when a new element is needed from the array, a victim element must be picked for recycling and reuse. A free list is associated with each hash list to track free elements in the hash list. Rather than having a lock for each free list and hash list, each pair of hash and free list is protected by one lock. Array indices rather than memory pointers are used because elements do not migrate between hash lists or free lists. This lack of migration combined with the fixed and equal number of elements on each hash list partially solves the balancing issue, since the maximum potential length of a free list is equal to the maximum size of an array. Free lists cannot become too unbalanced, only slightly unbalanced. An important parameter value to determine is the size of the hash list arrays, since the number of elements is fixed. The static nature of the SSAC organization makes this solution fairly rigid and limits the ability to handle load imbalances. For example, heavily used hash lists cannot exploit free elements located in hash lists that are mostly idle.

Rather than implementing a particular solution and radically modifying the block cache system once more, we decided to focus our efforts on exploring the remaining scalability problem shown in **Figure 5.28**. Implementing a free list solution is a subject of future work.

5.5.8 Global Lock, Larger Rehashed Table, No Free List

This experiment examines the relevance of the global lock once the free list bottleneck is eliminated. The configuration was the same as in Section 5.5.7 except that the single global lock was used rather than the fine grain locks in the block cache system. The results are shown in **Figure 5.30**. Performance was similar to the results in Section 5.5.3 (where a modified hash function and larger hash table were used) except that performance was better on 4 processors. Since the free list bottleneck was eliminated in the block cache system, the next performance encountered was the single global lock. The global lock was effective for up to 4 processors but became the critical bottle bottleneck beyond this point despite the distributed usage of 64 hash lists. Fine grain locks were indeed required to eliminate the bottleneck.

There is a potential concern with the number of locks that must be acquired in the fine grain lock implementation. The overhead of acquiring locks, even if they are completely uncontended, may accumulate to a significant amount of time. The results from this experiment demonstrate that the benefits of fine grain locks in the file system far out-weigh the costs of acquiring approximately 3 additional locks.

5.5.9 All Optimizations Combined

All optimizations were applied in this experiment to determine whether ideal performance and scalability could be achieved. These optimizations included the following.

1. Padded ORS hash list headers.

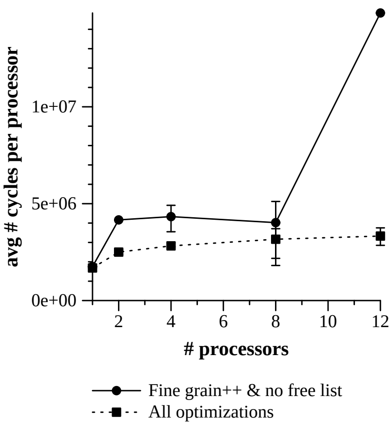

Figure 5.31: Regular read – all optimizations.

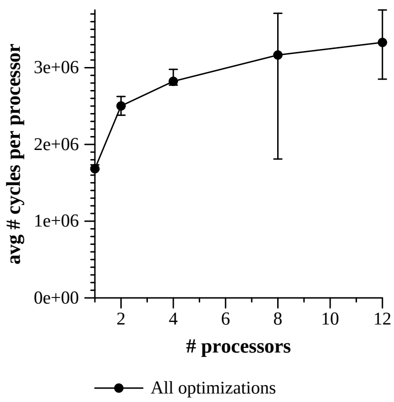

Figure 5.32: Regular read – all optimizations, magnified.

2. Padded ORS and block cache entries.
3. Padded all other critical data structures in the ORS and block cache systems, such as locks and waiting I/O data structures (shown in **Figure 3.4** and **Figure 3.5**).
4. Modified hash functions. (ORS and block cache)
5. Increased size of hash tables. (ORS and block cache)
6. Elimination of the block cache free list.
7. Use of fine grain locks in the block cache.

For simplicity, the K42 padding facilities were used rather than manually padding and aligning the data structures. The results are shown in **Figure 5.31**[17]. Scalability showed further improvement when compared to Section 5.5.7. A magnification of the results, shown in **Figure 5.32**, indicate that scalability is good although not quite ideal. The 8 processor results exhibited the same anomaly as described in Section 5.5.7, resulting in the lower error bar in the graph. **Figure 5.21** compares the fully optimized results against the original unoptimized results. At this point, rather than attempting to achieve ideal scalability, we decided to examine other workloads that stress other basic file system operations.

5.5.10 Summary

For regular, random-access files, good scalability was achieved in the read experiments using a combination of modified hash functions, large hash tables, padded hash list headers, padded cache entries, other padded data structures, fine grain locks, and a modified block cache free list.

5.6 Summary

Determining and eliminating bottlenecks in parallel system software is difficult, requiring a long process of experimentation. In general, we have found it important to apply the principles of maximizing locality and

[17]Error bars were removed from the 12 processor configuration.

concurrency that have been developed in about 10 years of multiprocessor software research by our research group [22]. In addition, the constant average queue length principle outlined by Peacock et al. [56, p. 86] is a fundamental pre-requisite to scalability. More specifically, these principles led to the use of techniques such as padding critical data structures to avoid false sharing of secondary cache lines, using appropriate hash functions to reduce hash collisions, increasing the size of hash tables to reduce hash list contention, using fine grain locks in the cache systems to reduce lock contention, and avoiding the use of global queues, such as the global free list. These optimizations have dramatically improved scalability under the read test microbenchmarks.

Chapter 6

Microbenchmarks – Other Fundamental Operations

6.1 Purpose

The purpose of the microbenchmarks in this chapter is to measure the scalability of fundamental file system operations that are the basis of higher level file system behavior. Performance of these core operations should provide us with a good idea of file system performance in higher level benchmarks such as application benchmarks. These experiments mainly used regular, random-access files rather than extent-based files since regular types are expected to be more common in the file system. Except for file reading, which was examined in the previous chapter, the fundamental file system operations that will be examined in this chapter include file creation, name lookup, statting, and writing.

6.2 File Creation

This experiment examines the scalability of the file creation operation. The first experiment is based on the unoptimized configuration in Section 5.2. 1000 regular, random-access files of zero size were sequentially created by each thread. All threads were executed in parallel and each was assigned to its own processor and disk. The disks were initially empty and files were created in the root directory of the disk. The results are shown in **Figure 6.1**. There were major scalability problems. There was a slow-down of 2 orders of magnitude between 1 and 12 processors. A heavy dependence on the unoptimized block cache was the main contributor to the results.

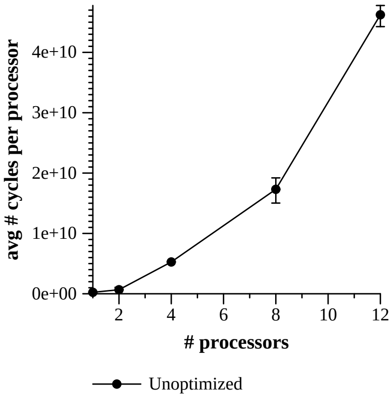

Figure 6.1: Create.

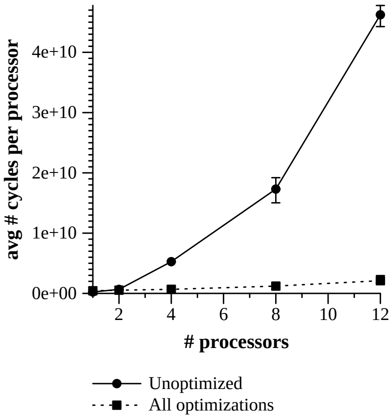

Figure 6.2: Create – all optimizations.

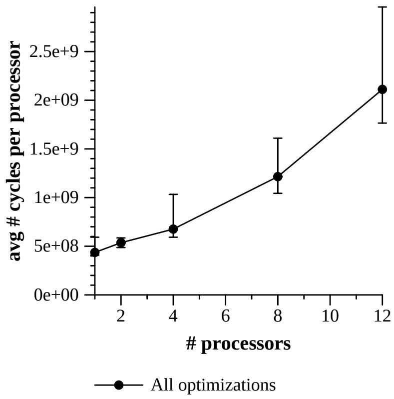

Figure 6.3: Create – all optimizations, magnified.

6.2.1 All Optimizations

The subsequent experiment examined the effectiveness of the optimizations developed for the read test workload. All optimizations were enabled in this experiment by using the same configuration as in Section 5.5.9. These optimizations included the following.

1. Padded ORS hash list headers.
2. Padded ORS and block cache entries.
3. Padded all other critical data structures in the ORS and block cache systems, such as locks and waiting I/O data structures (shown in **Figure 3.4** and **Figure 3.5**).
4. Modified hash functions. (ORS and block cache)
5. Increased size of hash tables. (ORS and block cache)
6. Elimination of the block cache free list.
7. Use of fine grain locks in the block cache.

For simplicity, the K42 padding facilities were used rather than manually padding and aligning the data structures. The results, shown in **Figure 6.2**, indicate drastically improved scalability by approximately an order of magnitude. The optimizations that applied to the read test workload were also effective on the file creation workload. However, a magnification of the graph, shown in **Figure 6.3**, indicates that scalability was not ideal since it took more than 3 times as long for a thread to execute on a 12 processor system compared to a 1 processor system. Additional optimizations are required for this type of workload and are explored in the following subsections.

In the end, attempts to further improve scalability were unsuccessful. Using 4-way set-associative primary caches and a 2-way set-associative secondary cache failed to yield any performance improvement, so these results are not shown. Modifications to the dirty lists (Section 6.2.2) and waiting I/O lists (Section 6.2.3) did not affect scalability. Increasing the initial pool of block cache entries (Section 6.2.4) was not effective either.

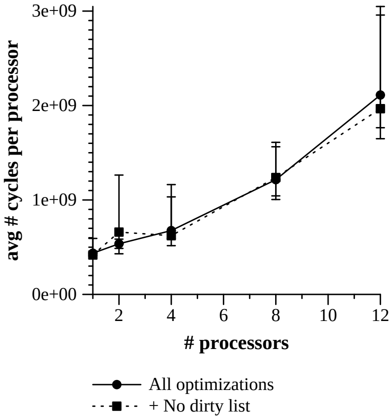

Figure 6.4: Create – all optimizations, no dirty list.

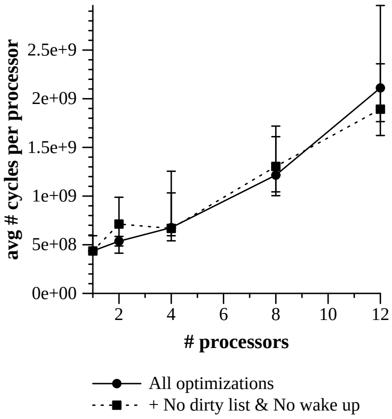

Figure 6.5: Create – all optimizations, no dirty list, no wake up.

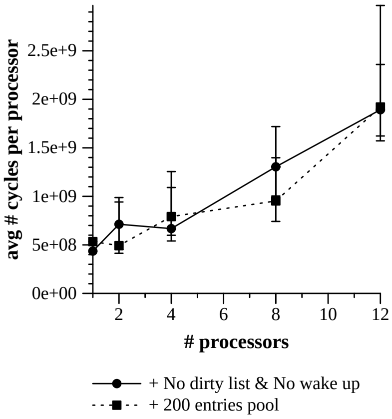

Figure 6.6: Create – +200 entries pool.

6.2.2 + No Dirty List

Despite the creation of a zero byte new file, several block cache and ORS cache entries are either created or modified on each create file operation. These actions occur because meta-data of the new file is recorded into the file system. These cache entries are marked as dirty by the cache systems and are placed onto the dirty lists. Inspired by the results of eliminating the free list bottleneck in Section 5.5.7, to improve scalability, the dirty list was short-circuited for both the ORS and block cache systems. This experiment is similar in configuration to Section 6.2.1 except that the dirty lists of the block cache and ORS cache were not used. Since the creation of 1000 files per thread caused many entries to be marked dirty and placed onto the dirty lists in parallel, this data structure may have been a critical bottleneck. However, the results shown in **Figure 6.4** indicate that the dirty list was not a critical bottleneck.

6.2.3 + No Wake Up

In another attempt to improve scalability, both the dirty list and the waiting I/O list were short-circuited for both the ORS and block cache systems. By short-circuiting the waiting I/O list, HFS no longer examined the waiting I/O list to wake up threads waiting on busy cache entries. This waiting I/O list was a potential bottleneck since all threads shared the single list. This experiment is similar in configuration to Section 6.2.2.

As well, additional free list avoidance was added to prevent the free list from becoming a bottleneck in both the ORS cache and block cache. More specifically, in the file creation workload, the free list is searched on every file creation in an attempt to allocate a free cache entry to the request. Since only one free list exists in each cache system, it may be wise to avoid this potential bottleneck. Instead of recycling an already allocated free cache entry, memory is allocated for a new cache entry.

Results shown in **Figure 6.5** indicate that the waiting I/O list was not the critical bottleneck. Performance was similar to the results in Section 6.2.2.

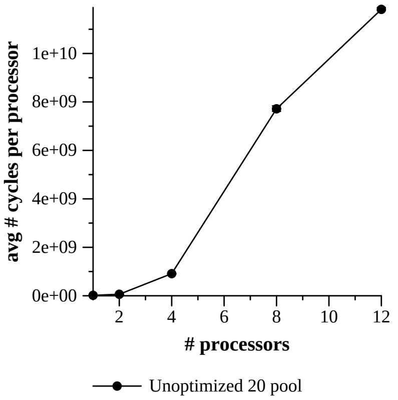

Figure 6.7: Lookup – unoptimized.

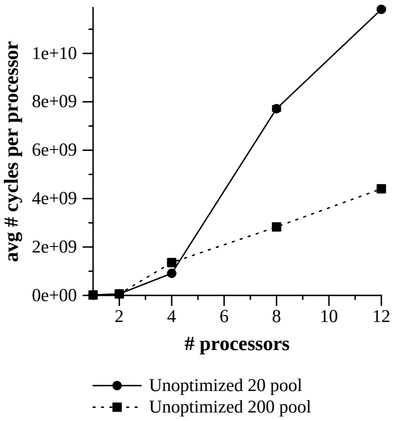

Figure 6.8: Lookup – 200 initial pool.

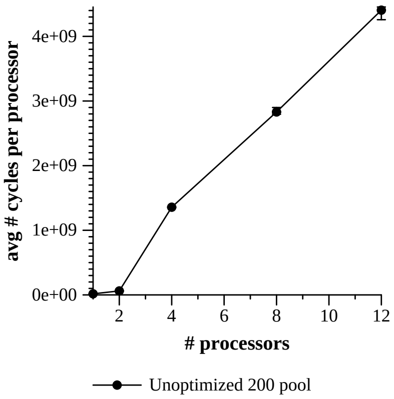

Figure 6.9: Lookup – 200 initial pool, magnified.

6.2.4 + 200 Entries Pool

Influenced by the results that will be shown in Section 6.3.1 for the file name lookup workload, the initial pool of block cache entries was increased from 20 to 200 to resolve a potential anomaly in the block cache entry allocation algorithm. This experiment is otherwise identical in configuration to Section 6.2.3. The results, as shown in **Figure 6.6**, indicate that the small initial pool of block cache entries was not the cause of the scalability problem.

6.2.5 Summary

The optimizations implemented from file read experiments were applicable to the file creation operation. File creation scalability was improved drastically. On 12 processors, approximately an order of magnitude of improvement was achieved. However, ideal scalability was not achieved since threads performed 3 times slower on a 12 processor configuration than on a uniprocessor configuration. Even worse, performance trends indicate that thread execution time will continue to grow linearly past 12 processors. We were unable to improve scalability any further for this particular workload

6.3 File Name Lookup

In this set of experiments, we examine the scalability of the file name lookup operation. As described in Section 3.6.2 on page 25, the name lookup operation returns a file token identifier for a given path name. The file token allows for easy identification of the target file by the file system and is used by clients to identify specific files.

In the first experiment, the unoptimized HFS configuration of Section 5.2 was used. Each thread performed a file look up of the same, one file: `DiskXXX:/directory1/directory2/directory3/directory4/`

`lookatme.txt` 1000 times, where XXX refers to the processor that the thread is executing on. The files were regular, random-access files. All threads were executed in parallel and each was assigned its own processor and disk. The file was located several levels down the directory tree to provide a more realistic file location that would cause a more realistic coverage of cache entries examined. However, the parent directories were empty except for the very leaf directory that contained the target file. This configuration may have provided a slightly unfair workload since, in practice, each directory should have been populated, causing the lookup scheme to linearly traverse each directory to find the target entry that represented the next directory down the path. Since we were not concerned with absolute performance results but with relative scalability results, we considered the configuration to be appropriate.

The results, shown in **Figure 6.7**, indicate that scalability was extremely poor beyond 2 processors. Since the experiment was run from within the file system address space (server-level), the normal file name caching system provided by the VFS was not utilized. Under normal usage, the VFS would handle the majority of repeated lookups of the same path names, making the performance of the file lookup operation less critical.

6.3.1 Increased Initial Pool of Block Cache Entries

This experiment examined the effect of increasing the initial pool of block cache entries from 20 to 200. Under the chosen workload, each thread required approximately 10 block cache entries in its working set. Due to the method of managing block cache entries, the characteristics of the chosen workload, and a temporary patch to avoid reader/writer problems in the original HFS implementation, the number of block cache entries never increased and remained at 20. The optimal behavior of the block cache system should have resulted in an increased number of block cache entries, from 20 entries to 200 entries. When running the experiment on more than 2 processors, this flaw caused the 20 block cache entries to thrash between all processors.

The results from increasing the initial pool of block cache entries from 20 to 200 is shown in **Figure 6.8**. Performance improved but the scalability problems were not resolved, as shown in the magnified version of the results in **Figure 6.9**.

6.3.2 Applying the Read Test Optimizations

In this experiment, we additionally applied the optimizations developed from the read test workload. This configuration included all the optimizations from Section 5.5.9 and the increased initial block entry pool size from Section 6.3.1. These optimizations included the following.

1. Padded ORS hash list headers.
2. Padded ORS and block cache entries.

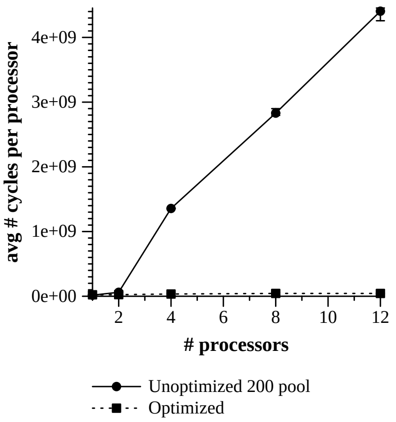

Figure 6.10: Lookup – all optimizations.

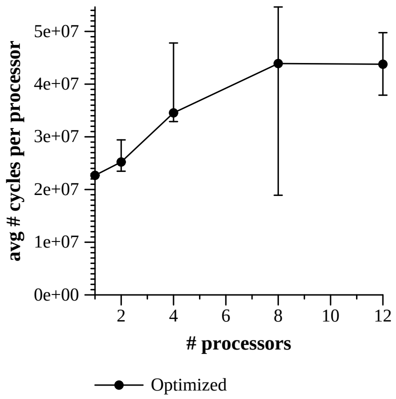

Figure 6.11: Lookup – all optimizations, magnified.

3. Padded all other critical data structures in the ORS and block cache systems, such as locks and waiting I/O data structures (shown in **Figure 3.4** and **Figure 3.5**).
4. Modified hash functions. (ORS and block cache)
5. Increased size of hash tables. (ORS and block cache)
6. Increased initial pool of block cache entries from 20 to 200.
7. Elimination of the block cache free list.
8. Use of fine grain locks in the block cache.

For simplicity, the K42 padding facilities were used rather than manually padding and aligning the data structures. The results, shown in **Figure 6.10**, indicate good scalability. A magnification of the graph, shown in **Figure 6.11**, indicates that scalability was good but not ideal. The 8 processor results exhibited the same anomaly as described in Section 5.5.7, resulting in the lower error bar in the graph.

The optimizations that applied to the read test workload were effective on the file creation and file lookup workloads. The optimizations of padding, modifying the hash functions, increasing the size of hash tables, eliminating the global free list, and using fine grain locks improved performance because the interactions with the cache systems are based on the same basic operations. These operations include hashing, hash list traversal, cache entry access and modification, dirty list enqueuing, and cache entry freeing.

6.3.3 Summary

Problems with the block cache entry allocation algorithm were evident with this simple microbenchmark workload. Increasing the size of the initial pool of block cache entries helped to improve scalability. For instance, on 12 processors, threads executed in approximately half the time as the unoptimized version. The optimizations implemented from the read file experiments were applied to this workload and dramatically improved scalability. Performance trends show that thread execution times appear to stabilize from 8 processors onwards.

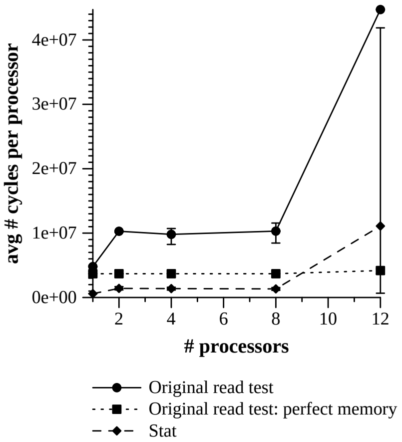

Figure 6.12: Stat.

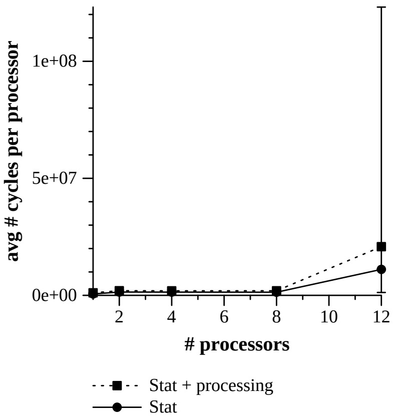

Figure 6.13: Stat – +processing.

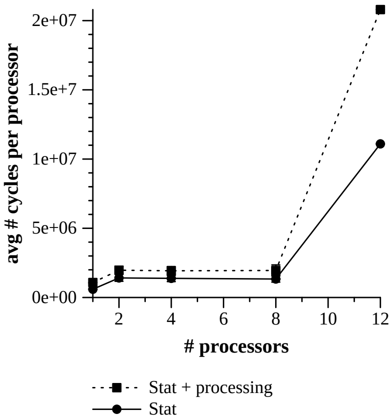

Figure 6.14: Stat – +processing.

6.4 File Stat

We examined the scalability of the getAttribute() operation in the next set of experiments. As described in Section 3.6.1 on page 24, getAttribute() returns the attributes of a target file or directory identified by a file token. This experiment was basically a subset of the unoptimized extent-based read test in Section 5.2 since, in the file stat experiment, the file status operation is executed but file block reading and file status updating is not performed. Each thread executed on its own processor, disk, and file. For each thread, the getAttribute() file system operation was performed 3136 times on its own extent-based file, matching the number of times the operation was performed on the superset of the experiment (Section 5.2), and allowing for easy comparison of results. The results of using unoptimized HFS, shown in **Figure 6.12**[1], are similar to the extent-based read test (Section 5.3), since the experiment was simply a subset of the read test.

6.4.1 + Processing

This experimental configuration was a slightly more realistic file stat test. The configuration is similar to Section 6.4, except that it processes the file status data into the standard Unix format before returning from the call. More specifically, it converts the HFS format of file status to the more widely used Unix/Linux/GNU file status format. As shown in **Figure 6.13**[2] and **Figure 6.14**[3], the results were fairly similar to the previous file stat experiment in Section 6.4 as might be expected. It took slightly longer for the extra processing, however the maximum error bar range on 12 processors was much greater.

[1]Error bars were removed from the unoptimized 12 processor configuration.
[2]Error bars were removed from the unprocessed 12 processor configuration.
[3]Error bars were removed from the 12 processor configurations.

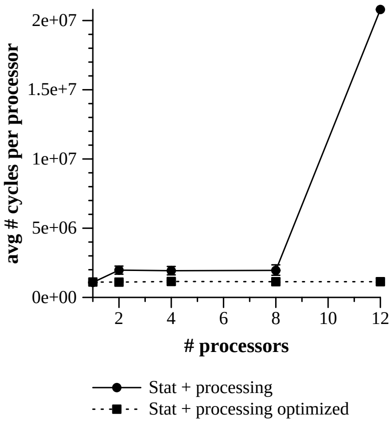

Figure 6.15: Stat – +processing, optimized.

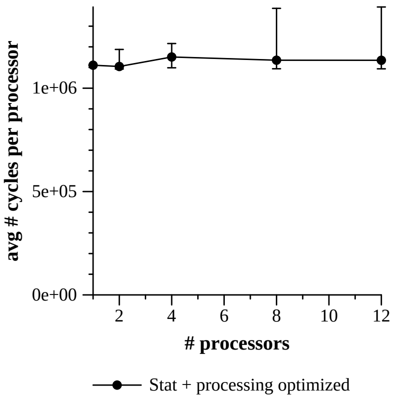

Figure 6.16: Stat – +processing, optimized, magnified.

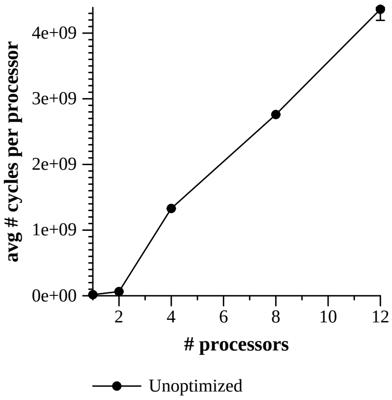

Figure 6.17: Stat – +name lookup.

6.4.2 + Processing, Optimized

This experiment examined the impact of implementing all optimizations. These optimizations included all the optimizations of the read experiments:

1. Padded ORS hash list headers.
2. Padded ORS and block cache entries.
3. Padded all other critical data structures in the ORS and block cache systems, such as locks and waiting I/O data structures (shown in **Figure 3.4** and **Figure 3.5**).
4. Modified hash functions. (ORS and block cache)
5. Increased size of hash tables. (ORS and block cache)
6. Increased initial pool of block cache entries from 20 to 200.
7. Elimination of the block cache free list.
8. Use of fine grain locks in the block cache.

For simplicity, the K42 padding facilities were used rather than manually padding and aligning the data structures. The results, shown in **Figure 6.15** and **Figure 6.16**, indicate good scalability.

6.4.3 Stat with Name Lookup

The purpose of the next experiment was to more accurately emulate a standard Unix stat() file system call[4]. Each operation of the test begins with a name lookup to obtain the target file token and is followed by an operation to obtain the file status. This experiment is similar to the configuration in the file lookup test (Section 6.3.1) using unoptimized HFS. Each thread performed a file look up of `DiskXXX:/directory1/directory2/directory3/directory4/lookatme.txt` to obtain the file token, followed by a getAttribute() file system call and attribute processing, all repeated 1000 times, allowing for easy comparison with the

[4]int stat(const char *file_name, struct stat *buf)

results in Section 6.3.1. The files were regular, random-access files[5] and the initial pool of block cache entries was set to 200 instead of the default of 20. All threads were executed in parallel and each was assigned its own processor and disk.

The results, shown in **Figure 6.17**, are similar to the results for the file name lookup experiment in Section 6.3.1. Scalability was poor beyond 2 processors.

6.4.4 Stat with Name Lookup, All Optimizations

This experiment examined the impact of implementing all optimizations in the getAttribute() with name lookup workload. This configuration included all the optimizations we used previously. The results, shown in **Figure 6.18** and **Figure 6.19**, indicate that scalability was good but not ideal since it took twice as long for each thread to execute on 12 processors than on 1 processor. The results were very similar to the results in Section 6.3.2 that examine the optimized name lookup operation, indicating that the name lookup portion of the task was the rate limiting component where as the getAttribute() portion scaled well, as shown in Section 6.4.2. The 8 processor results exhibited the same anomaly as described in Section 5.5.7, resulting in the lower error bar in the graph. Fortunately, performance trends show that thread execution times appear to stabilize from 8 processors onwards.

6.4.5 Summary

The optimizations implemented from the file read experiments were applicable to the file stat operation of the file system. File stat scalability was improved drastically. Ideal scalability was achieved in the getAttribute() portion of the task, whereas the name lookup portion performed relatively well but did not achieve ideal scalability.

6.5 Regular File Write Test

This section examines the scalability of the write operation on regular, random-access-based files. Each thread sequentially writes a 4,096,000 byte regular, random-access-based file to its own disk while running on its own processor. The operation causes meta-data to accumulate in the caches. The flushing of the caches to disk was disabled to measure ideal file writing conditions. No optimizations were implemented for this initial base test. The results, shown in **Figure 6.20**, indicate that scalability was poor beyond 2 processors, closely mirroring the results of the unoptimized regular file read experiment in Section 5.5.1.

[5]In terms of the actions taken by the file system during the getAttribute() operation, there is no difference between extent-based or regular, random-access files. Consequently, either file types can be used for the getAttribute experiments.

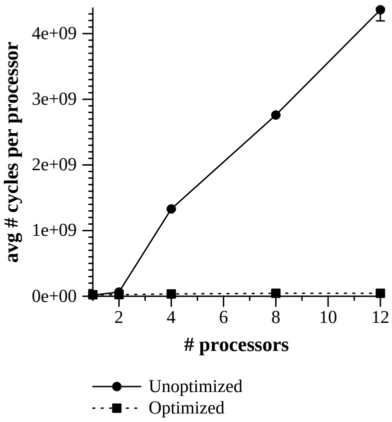

Figure 6.18: Stat – +name lookup, optimized.

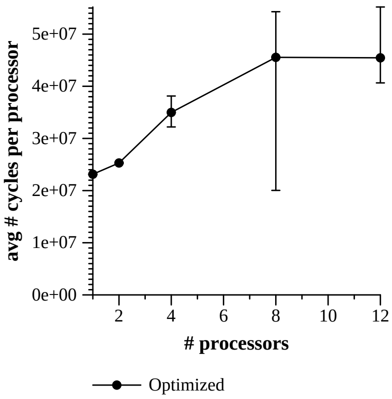

Figure 6.19: Stat – +name lookup, optimized, magnified.

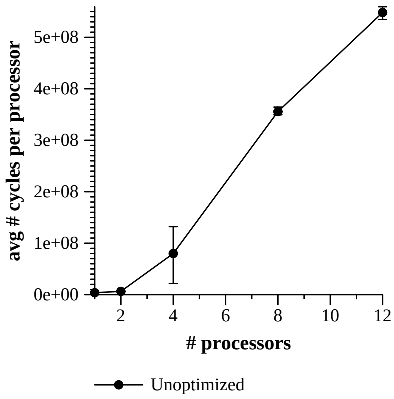

Figure 6.20: Regular write.

6.5.1 ORS Optimizations

In the regular, random-access-based file write workload, the ORS cache system is used when modifying file attributes. Since the ORS cache system may have been the critical bottleneck, this component was examined in isolation by implementing only ORS optimizations and observing the performance impact. All ORS optimizations were included while no block cache optimizations were included. The optimizations include the following.

1. Padded ORS hash list headers and ORS cache entries.
2. Modified ORS hash function.
3. Increased ORS hash table size.

For simplicity, the K42 padding facilities were used rather than manually padding and aligning the data structures. The results, shown in **Figure 6.21**, indicate that the ORS cache system was not the critical bottleneck. Only a slight performance improvement was observed, in that scalability was satisfactory for up to 4 processors. Such a result suggests that the bottleneck was due to the inadequate number of hash lists (4) in the block cache system.

6.5.2 All Optimizations

This experiment examined whether the aggregate of all previously applied optimizations were effective on the regular, random-access-based write test workload. The results, shown in **Figure 6.22** and **Figure 6.23**, indicate that good scalability was achieved, although a thread on average took almost twice as long to execute on a 12 processor configuration than on a 1 processor configuration.

The large error bar range on 8 processors was a result of a consistent anomaly on processor #6 and processor #7. This anomaly may be related to the one found described in Section 5.5.7 due to the fact

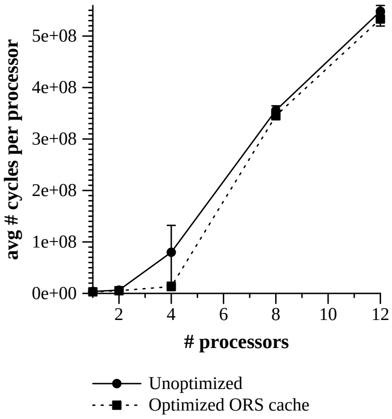

Figure 6.21: Regular write – optimized ORS.

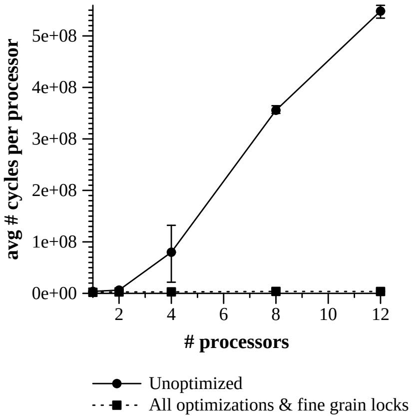

Figure 6.22: Regular write – all optimizations.

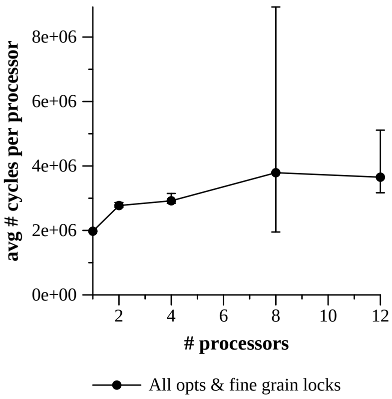

Figure 6.23: Regular write – all optimizations, magnified.

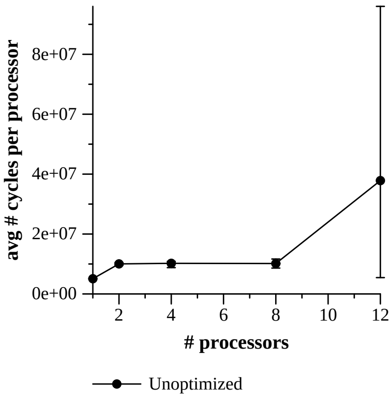

Figure 6.24: Extent write.

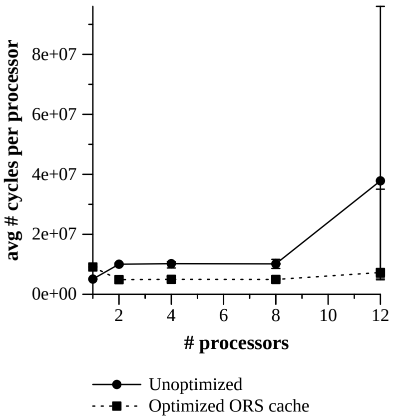

Figure 6.25: Extent write – optimized ORS.

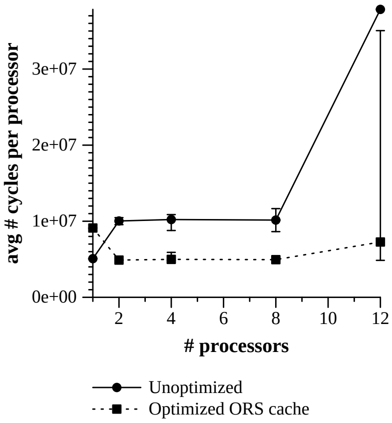

Figure 6.26: Extent write – optimized ORS.

that all noted anomalies occur on an 8 processor configuration. Over 10 runs of the experiments, the thread execution time on processor #7 was typically around 1,900,000 cycles, creating the lower error bar region, while the thread execution time on processor #6 was typically around 8,900,000 cycles, creating the upper error bar region. It appears the benefits gained by processor #7 may be at the expensive of processor #6. Finding the source of the anomaly was not attempted in this thesis but may be investigated in future work.

6.5.3 Summary

The optimizations implemented from the file read experiments were applicable to the regular, random-access-based file write operation. File write scalability was improved drastically. Scalability was close to ideal and was similar to the scalability achieved in the regular, random-access file read results.

6.6 Extent-based Write Test

This experiment examined the scalability of writing extent-based files. No optimizations were included in the initial configuration, which was based on the unoptimized extent-based read test setup in Section 5.2. Each thread, running on its own processor, writes a 12,845,056 byte extent-based file to its own disk. The results, shown in **Figure 6.24**, were almost identical to the extent-based read test in Section 5.2, as expected. Scalability was poor beyond 8 processors.

6.6.1 ORS Optimizations

This experiment examined the impact of the ORS optimizations on the extent-based write test workload. The optimized test case included only optimizations to the ORS cache, similar to those of Section 5.4.9.

1. Modified ORS hash function (Section 5.4.5).
2. Increased ORS hash table size (128 hash lists, Section 5.4.7).
3. Padded ORS hash list headers (Section 5.4.6).
4. Padded ORS cache entries (Section 5.4.8).

For simplicity, the K42 padding facilities were used rather than manually padding and aligning the data structures. The results, shown in **Figure 6.25** and **Figure 6.26**[6], indicate good scalability. Except for the uniprocessor anomaly, as reported previously that is possibly due to the K42 padding facilities, ideal scalability was achieved between 2 to 12 processors. Under this particular workload of extent-based files, only the ORS cache was stressed while the block cache was not, so fine grained block cache locking was not required in this scenario.

The large maximum error bar was a result of a consistent anomaly on processor #4, appearing in 9 out of 10 runs of the experiment. The typical thread execution time on processor #4 was approximately 35,000,000 cycles while on all other processors, the execution time clustered much closer to the average time. Finding the source of the anomaly was not attempted in this thesis but may be investigated in future work.

6.6.2 Summary

The optimizations implemented from the extent-based file read experiments were applicable to the extent-based file write operation of the file system. Extent-based file write scalability was greatly improved. Scalability was close to ideal and was similar to the scalability achieved in the extent-based file read results.

[6] Error bars were removed from the unoptimized 12 processor configuration.

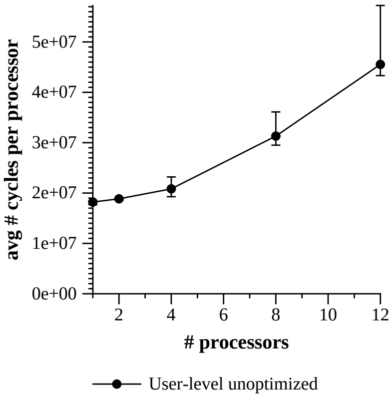

Figure 6.27: User-level read.

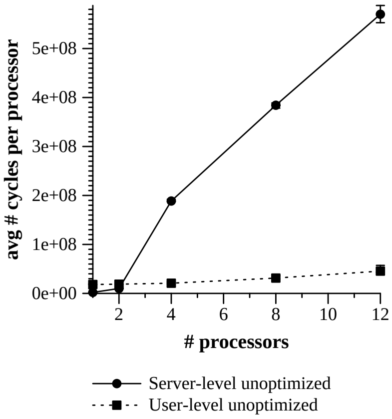

Figure 6.28: User-level vs server-level read.

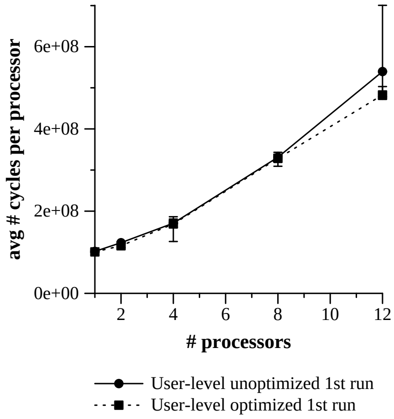

Figure 6.29: User-level read – optimized.

6.7 User-Level Read Test

6.7.1 Unoptimized

In this section, we briefly examine the impact of executing workloads from within a user-level application rather than from within the file system address space (server-level). Such an environment is more realistic than executing from the server-level. The read experiment from Section 5.5.1, which used regular, random-access files, was executed from within a separate user-level application.

As a user-level application, the workload made use of K42 kernel services and data structures in order to access the file system. For example, the application used the K42 GNU/Linux environment since standard GNU/Linux functions were called, such as fopen(), fclose(), and fread(). K42 IPC was used to communicate between the application and the K42 VFS layer, which in turn used K42 IPC to contact the file system. The VFS provided path name caching and the FCM provided data file block caching, all of which were not available in the server-level experiments. This configuration enabled the scalability of the IPC, VFS, and FCM components of K42 to exercised by the experiment. FCM and IPC scalability were demonstrated by Gamsa et al. [23] on the Tornado Operating System, which is closely related to K42.

Similar to previous experiments, each thread executed on its own processor and disk. The workload was executed twice to ensure the FCM and VFS facilities had cached as much data as possible. The first run was used to warm up the caches and the second run was measured.

As shown in **Figure 6.27**, there were scalability problems running the workload this way. At 12 processors, threads took more than twice as long to complete their tasks compared to a 4 processor system. These results suggest that critical bottlenecks may exist in the FCM and VFS components.

By executing the experiment at user-level, the FCM and VFS components provided caching of meta-data and data originating from the file system. These components helped reduce the amount of stress placed on the file system and prevented file system scalability inadequacies from appearing. This reduction of file system stress can be see in **Figure 6.28**, as the scalability of the unoptimized user-level configuration is significantly better than the unoptimized server-level configuration.

6.7.2 Optimized

All optimizations, such as those described in Section 6.4.2, were enabled to determine their effectiveness in the user-level experiment. The results are shown in **Figure 6.29**. We examined the results from the first run of the experiment (the warm up run) rather than the second run because we wished to view the effects of reduced caching by the FCM component. Unfortunately, the optimized and unoptimized user-level experiments showed no significant performance difference. Perhaps the user-level microbenchmark workload did not place adequate stress on the file system for scalability problems to appear. Perhaps larger and more complicated workloads are required. The next chapter investigates such a workload under user-level execution.

6.7.3 Summary

Running the experiment at user level caused the FCM and VFS components to be utilized, thereby reducing stress on the file system. Consequently, scalability problems of the file system did not appear in the results. For larger and more complicated workloads, such as the one used in the following chapter, file system scalability will become an issue.

6.8 Summary

Scalability improvements were achieved for all fundamental operations of the file system. In some cases, near-ideal scalability was obtained, while in others, ideal scalability was not achieved but dramatic scalability improvements were achieved relative to the unoptimized version of the file system. These improvements to the core operations of the file system should easily transfer to the macrobenchmark of the next chapter.

Chapter 7

Macrobenchmark

7.1 Purpose

The purpose of the macrobenchmark experiment was to determine the impact of a realistic workload on the file system and to verify that the optimizations implemented during the microbenchmark experiments were beneficial to more realistic applications. A Web server workload was chosen but an actual Web server application, such as Apache Web Server or Microsoft Internet Information Server, was not used because the K42 operating system lacked infrastructure to support more complex off-the-shelf applications. Instead, a simple simulated Web server was executed to generate the file system workload, using traces from a real workload.

7.2 Experimental Setup

7.2.1 Workload

The World Cup 1998 Web trace [5] was chosen as the workload. This Web trace has been studied by Arlitt and Jin [6], resulting in detailed quantitative analysis of the trace. The Web trace is a time-sorted aggregation of requests to all World Cup Web servers during the 1998 tournament. It includes both static and dynamic content requests. The complete Web trace consisted of 90 days of log files, however only a small portion of the Web trace was used for our experiments, namely from June 8, 1998, 10:00:01pm GMT to 10:21:02pm GMT. This date corresponds to day 45 of the Web trace and was chosen because it was the middle point of the tournament. The more intense final games during the final days of the tournament were not used since they targeted a smaller set of Web pages and files. They would not have offered as much breadth as traces from the middle.

The chosen Web log segment was reverse engineered to recreate the contents on disk. Files of correct size were created but the actual file contents were filled with artificially generated data since they were not used for any real purpose. The resulting disk contents occupied approximately 125 MB of storage space, not including the associated meta-data which occupied approximately 30 MB (24% of the file data size). This log segment corresponded to approximately 1 GB of data transfer from the file system by the Web server. 1 GB seemed to be an adequate sample size. However, due to the execution speed of SimOS, replaying the chosen Web log segment would have involved 1 GB of data transfer (205,454 requests) and would have taken a very long time to simulate (several months). Instead, only the initial 4000 requests were traced, which is equivalent to approximately 14 MB of data transfer and 37 seconds of Web log time. Although this sample was very small, it was large enough to demonstrate scalability problems in the file system.

To examine scalability, a thread and a disk containing the full data set were allocated to each processor, similar to the microbenchmark configuration. Each thread independently simulated the file system workload according to the 4000 entry Web log in its entirety. The same 4000 requests were made by each thread, rather than using unique segments of the Web log. This decision allowed us to maintain a uniform set of requests for calculating average thread execution time. Rather than allowing all threads to start at the beginning of the Web log, the following approach was taken to avoid simultaneously stressing the file system in exactly the same manner repetitively. Each thread traversed the Web log from a unique initial offset, wrapping around the end of the Web log to the beginning, and stopping when it reached the initial offset. This technique maintains a uniform set of requests for all threads, giving the average thread cycle time a valid meaning. For each Web log entry, the contents of the corresponding file was requested from the file system.

7.2.2 Web Server Simulation

We had previously studied Web server workloads on a multiprocessor in [76] for static Web pages. For such a workload, we found a simple repetitive sequence of file system calls consisting of (1) obtaining the file status of the requested file, (2) opening the file for reading[1], (3) reading the file, and (4) closing the file. This sequence was repeated for all requests[2]. These observations were based on examining the system call log of an SGI Challenge running a portion of the Webstone 2.5 benchmark on Apache Web Server 1.3.9. The Web server simulator was designed to imitate these actions by interpreting each Web log entry and following the above steps. For additional realism, bad requests were included to exercise the error paths since the associated file system operations would have been performed on a real Web server.

Using a Web server simulator has certain advantages and disadvantages. On the one hand, the simulator does not stress all aspects of the file system, such as application-code paging and application-specific buffering of files. On the other hand, the Web server simulator is completely scalable in contrast to an actual Web

[1]Reading is done using memory-mapped I/O via the mmap() system call.
[2]In our experiment, if a directory is requested, directory contents are gathered and a stat of each entry is conducted.

server application that has scalability limits and thus cannot offer as intense a workload on the file system as the Web server simulator.

Static Web content requests are straight-forward to model. A requested static file is simply transferred from the file system to the Web server application. Modelling dynamic content, such as dynamically generated pages from CGI-Perl scripts, is not as straight forward. Since almost anything can be done from within the script to generate a dynamic Web page, the impact of such activities on the file system can vary greatly. For example, a dynamically generated Web page may simply obtain the system date and generate a Web page to display it, which has little impact on the file system. Or, it may gather data from a large number of files and display user-requested statistical information, placing a heavy demand on the file system.

Since the possibilities are endless, we arbitrarily chose a simple solution. We model file system access when generating dynamic content by noting the size of the returned result (N bytes) as indicated in the Web log and transferring N bytes of the corresponding file from the file system to the Web server application. This method was only a very rough model. For example, for a Web log entry that indicated that N bytes were transferred as a result of a request for `cgi-bin/time.pl?TZ=500`, the Web server simulator would read N bytes from a file named `cgi-bin/time.pl`. In practice, however, the loaded program may access other files, but we do not model this characteristic, since it was not captured in any log. When the `time.pl` file was created during the disk creation phase, the size was chosen to be the maximum size noted in the Web log from all requests for `cgi-bin/time.pl*`, where "*" matches any string of characters, including the null string.

The Web server simulator was executed from within the file system address space (server-level) in the majority of the experiments and from a separate address space as a user-level application in a few subsequent experiments. The user-level version included the overheads associated with running in user-level mode, and stressed components of the operating system including the file system, FCM, VFS, and IPC mechanism.

7.2.3 Implementation Details

The SimOS configuration was similar to the configuration used in the microbenchmarks except that (1) 512 MB of RAM was used instead of 128 MB, and (2) disk access for both meta-data and file data was enabled due to technical reasons that required Web server simulator code to be loaded from the disks. 512 MB of RAM was used to ensure an adequate amount of memory was available to run the user-level experiments since (1) the pinned kernel image was 71 MB and (2) on a 12 processor configuration, the FCM could potentially cache 168 MB[3] of file data. The benefit of allowing full disk access was that it exercised and stressed the complete code path from the application to the disk, from the user-level application down to the device driver. In contrast, the microbenchmark configuration short-circuited the reading of file data from disk and enabled

[3] 12 CPUs X 14 MB of data transfer = 168 MB

meta-data access only.

Zero latency disks were used for most experiments while 15 ms latency disks were briefly used in a few experiments to verify that scalability problems remained and were solvable by the implemented optimizations. Using a 15 ms disk latency, while more realistic, greatly increased the time needed to run experiments on SimOS. Due to time constraints, we mainly used zero latency disks. Disk latency should have a small impact on scalability under this particular workload, since any time that disk access is required, the cache systems release all locks except for the targetted cache entry lock.

The simulation of logging Web requests to a log file was disabled for most experiments due to concerns about lengthy SimOS execution times. We briefly enabled Web logging in a few subsequent experiments and found that file system scalability was not affected.

Due to time constraints and speed limitations of the SimOS simulator, the Web server simulator experiments were not repeated 10 times, but only 2 times. Similar to the microbenchmarks, a warmup run was executed to warm up various caches, such as the file system cache, VFS, and FCM[4]. The results from the second and third runs were recorded and analyzed. The averages from the two runs were plotted, along with error bars indicating the minimum and maximum values observed.

In the user-level implementation, for requests that target a directory rather than a file, the GNU/Linux readdir() function call was used instead of scandir() to prevent directory entry sorting. At the server-level, sorting was not an issue since this task was delegated to higher-level layers such as the VFS or the user-level I/O libraries. This decision allowed the Web server simulator to artificially reduce its computational demand and thereby increase the request rate to the file system.

7.2.4 Caveats

There are several limitations to the Web server workload we used. The World Cup Web server was an information providing site and not a transaction site. This designation should not have a significant impact on the applicability of the workload since our criteria was simply workload realism, not sophistication. Read access represents the common case, so we did not model updates to Web files; therefore the model ignores concurrent read and write requests to the same file. Web servers have a narrow access pattern within a file, accessing file blocks sequentially and commencing at the first block. Random access within a file does not occur. Whole file accesses are typical, although partial file accesses sometimes occur. Our Web server simulator performs both whole file and partial file accesses as dictated by the transfer size field of the Web log entries.

The technique of traversing the Web log as quickly as possible may not be the most realistic way of modelling the dynamic behavior of a Web workload. On real Web servers, there are request timing depen-

[4]In the server-level experiments, the VFS and FCM were not access and hence their caches were not warmed up.

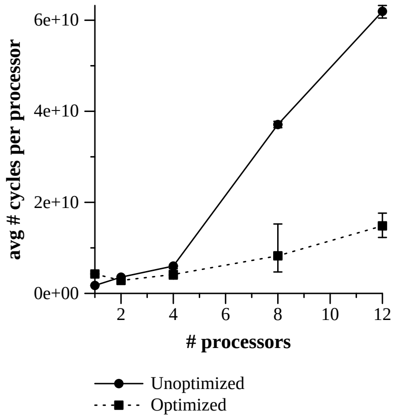

Figure 7.1: Web server.

dencies. A faster Web server may see a different ordering of the requests as well as possibly different requests as the surfing habits of Web users may change due to the responsiveness of the Web server. Clients may not increase their rate of requests simply because the Web server is faster. Client bandwidth may have an upper-bound since some clients are connected by traditional analog modems. As well, human reading speeds may limit the maximum rate of user requests. A better model would have clients issue requests at a fixed rate and simply increase the number of clients until throughput fails to increase any further or until response time is greater than some threshold. SPEC Web99 [73] uses such a technique, however, we believe that our simulations are a good stress test of the file system, especially as it relates to file system meta-data.

7.3 Server-Level Experiments

7.3.1 Results

The experiments described in this section were run at the server level, that is, within the file system address space. **Figure 7.1** shows the performance of the unoptimized and optimized versions of HFS. The optimized version contained all optimizations implemented from the previous microbenchmarks, including:

1. Padded ORS hash list headers.
2. Padded ORS and block cache entries.
3. Padded all other critical data structures in the ORS and block cache systems, such as locks and waiting I/O data structures (shown in **Figure 3.4** and **Figure 3.5**).
4. Modified hash functions (ORS and block cache).
5. Increased size of hash tables (ORS and block cache).
6. Increased initial pool of block cache entries from 20 to 200 entries.
7. Elimination of the block cache free list.
8. Use of fine grain locks in the block cache.

For simplicity, the K42 padding facilities were used rather than manually padding and aligning the data structures. Performance was similar between 1 to 4 processors for the unoptimized and optimized versions of HFS. Beyond 4 processors, the unoptimized version exhibited extremely poor scalability. Although scalability was much better in the optimized version, it was not ideal.

In the unoptimized version, there appeared to be a correlation between the number of hash lists in the block cache system (4) and the resulting knee of the curve (at approximately 4 processors). A brief experiment was conducted using the unoptimized version augmented with 32 rather than 4 hash lists in the block cache system. Results for 8 processors (not shown) did not have any performance improvement, signifying that scalability could not be achieved by simply increasing the size of the hash tables in the unoptimized version.

Due to the nature of the Web server workload, many files are accessed by each thread (one thread per processor), causing their associated meta-data to be scattered randomly and uniformly across all hash lists in the ORS cache. In contrast, the microbenchmark read test consisted of each thread accessing only one file, and hence utilizing approximately two hash lists per thread in the ORS cache. The scattered hash list usage caused by the Web server workload led to a greater number of hash collisions between threads. In addition, as the number of processors, and hence threads, were increased, the number of hash collisions increased further. The macrobenchmark enabled us to examine the impact of the hash function on a more realistic workload. It highlighted the inadequacies of the hash function in the ORS cache.

7.3.2 Optimizations

A modified hash function in the ORS cache was implemented along with a larger hash table. The new hash function assigns an exclusive group of hash lists to each disk and an offset within the group depending on the physical file number that is embedded in the file token of the request. The number of hash lists per group was set to 64. Under a general workload, this scheme eliminates any chance of hash collisions from requests targetted at different disks. However, if two requests are targetted to the same disk, there is the possibility of a collision, but the probability is reduced since there are 64 available hash lists. The distribution within a hash list group should be fairly uniform because a simple modulo function is applied to the physical file number of a request to determine the target hash list, which is equivalent to taking the 6 lower-order bits of the file number. The logical organization and physical implementation of the new hashing scheme is shown in **Figure 7.2**. Although logically organized into 2 levels, it is physically implemented in one level. For simplicity, we statically sized the ORS hash table to the maximum configuration of 12 processors and 12 disks. Ideally, the size of the hash table should scale accordingly to the size of the multiprocessor, as implied by Peacock et al. [56, p. 86]. Calculating the target hash list involves simple bit shifting, masking, and assembly.

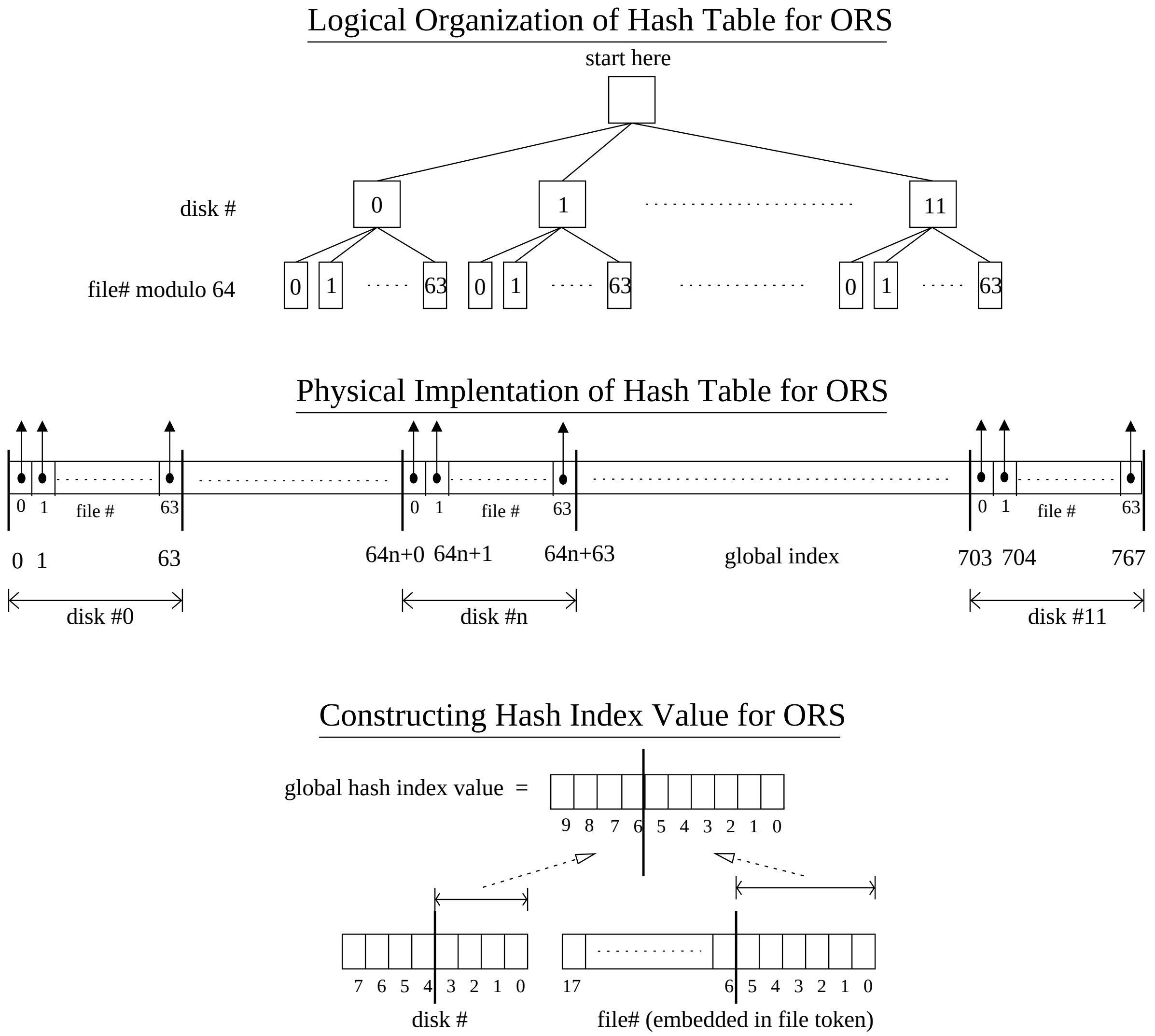

Figure 7.2: Modified ORS hash function.

In the microbenchmarks, the modified hash function in the block cache system simply assigned one hash list to each disk. However, on the Web server workload, this hash function had a detrimental effect on performance because the number of entries in each hash list was fairly large since many files were accessed. For each request, after hashing to a particular hash list, a linear search through an extremely long hash list was necessary, leading to poor absolute performance. To solve this problem, hash function modifications and a larger hash table were applied to the block cache system as well. For the block cache system, the new hash function assigns an exclusive group of hash lists to each disk and an offset within the group depending on the physical file number and logical file block number, as with the ORS cache hash table. The number of hash lists per group was set to 256. Within each group of hash lists, 16 subgroups were designated for physical file number indexing. Finally, each subgroup contained 16 hash lists that were indexed by file block number. The logical organization and physical implementation of the new hashing scheme is shown in **Figure 7.3**. Once again for simplicity, we statically sized the block cache hash table to the maximum configuration of

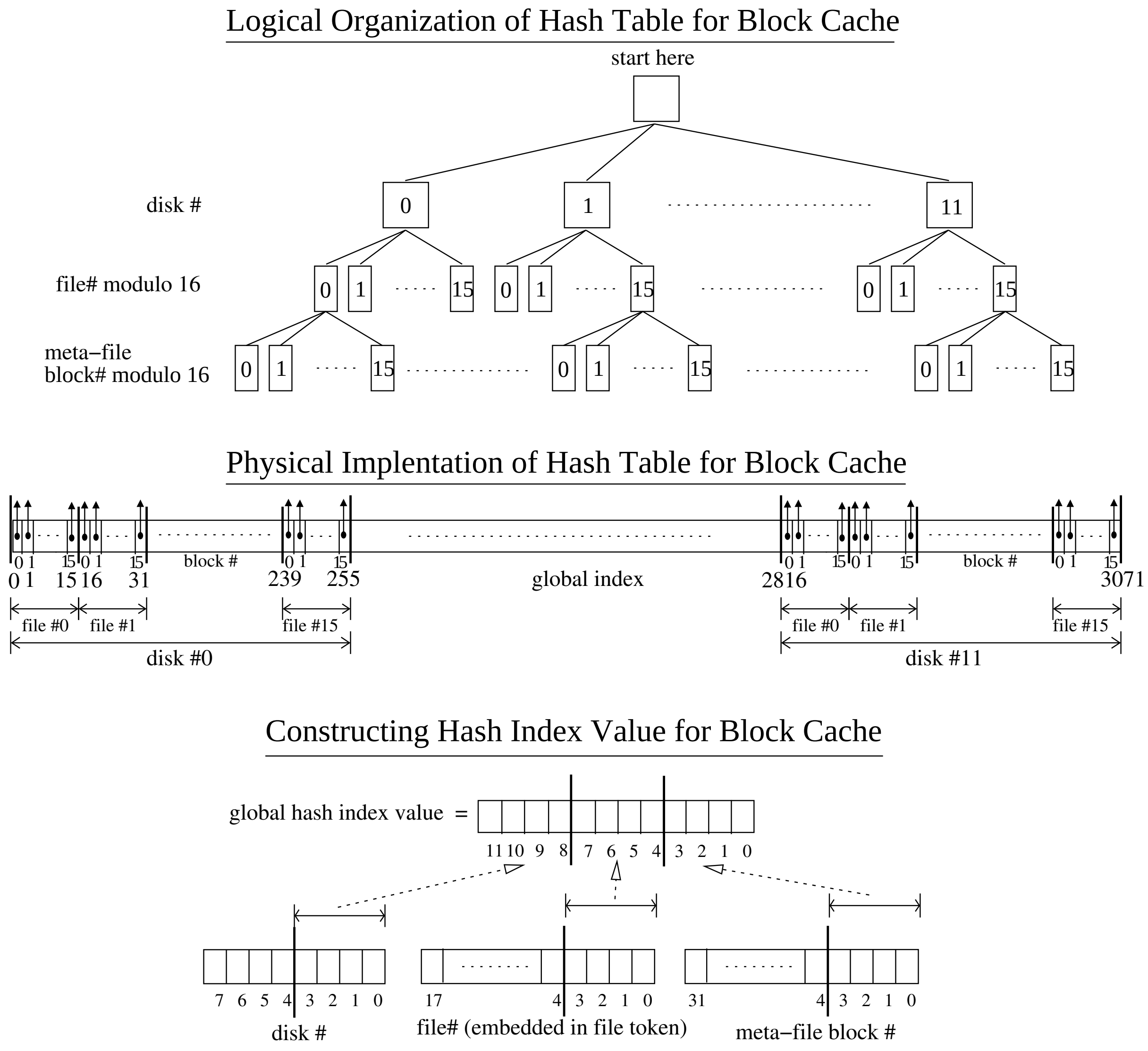

Figure 7.3: Modified block cache hash function.

12 processors and 12 disks. Ideally, the size of the hash table should scale accordingly to the size of the multiprocessor, as implied by Peacock et al. [56, p. 86].

There are effectively 3 chances for hash collision avoidance, based on disk number, file token number, and file block number. Consider 2 threads that are executing in parallel on separate processors. If the threads access different disks, then hash collisions will not occur. However, if they access the same disk[5], the chance of collisions is mitigated by the availability of 16 hash list groups. If the threads access different hash list groups, then hash collisions will not occur. The threads may access the same hash list group if either (1) they are accessing the same file, or (2) they are accessing different files on the same disk but the corresponding file numbers cause them to hash to the same hash list group. Even if the threads hash into the same hash list group, there are 16 hash lists within each group to reduce the probability of hash collisions. If the two

[5] This cannot occur in our experiments.

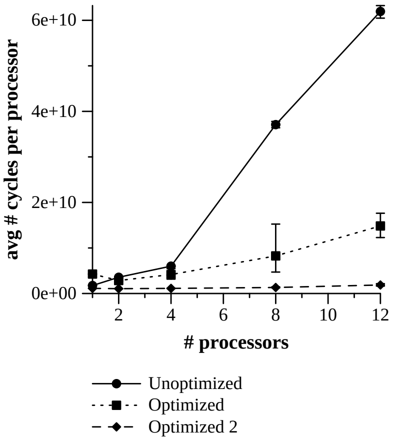

Figure 7.4: Web server – further optimized.

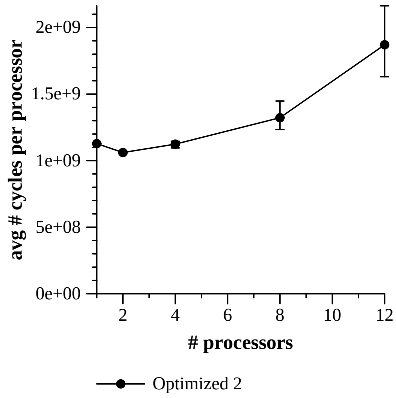

Figure 7.5: Web server – further optimized, magnified.

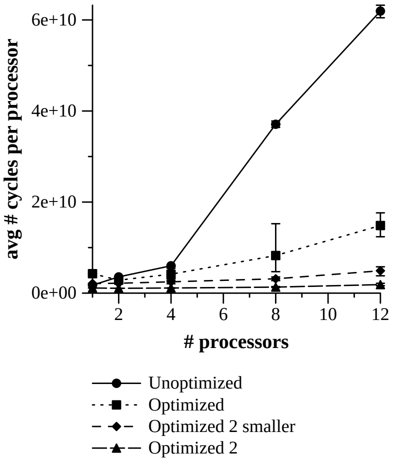

Figure 7.6: Web server – further optimized, smaller.

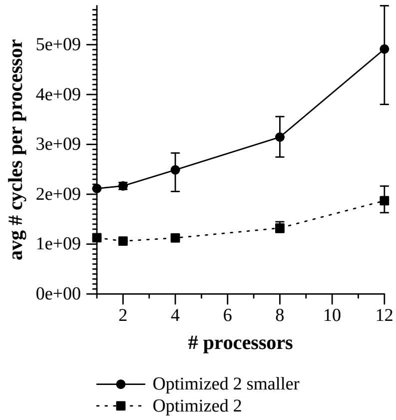

Figure 7.7: Web server – further optimized, smaller, magnified.

threads are accessing the same file, hash collisions may still not occur if they access sufficiently different locations of the file, since access to the group of 16 hash lists is roughly based on file block number.

The results from these additional optimizations are shown in **Figure 7.4** with the curve labeled "Optimized 2". Scalability was greatly improved, however, ideal scalability was not achieved. The resulting curve, magnified in **Figure 7.5**, indicates a minor scalability degradation. Average thread execution time on 12 processors was 67% greater than on a uniprocessor. To briefly examine sensitivity to the number of hash lists in a grouping, the ORS cache hash list group size was reduced from 64 to 16, and the block cache hash list group size was reduced from 256 to 64. The results are shown **Figure 7.6** and **Figure 7.7**[6] with the curve labeled "Optimized 2 smaller". As expected, the curve was shifted higher, but minor scalability problems unexpectedly reappeared since the slope of the curve has increased slightly. The modified hash function guaranteed exclusive hash list access under our particular workload so the scalability problems were not caused by hash list sharing. We did not investigate these scalability problems further since the smaller

[6] 64 MB of RAM was used on the dual processor configuration instead of 128 MB due to OS stability problems.

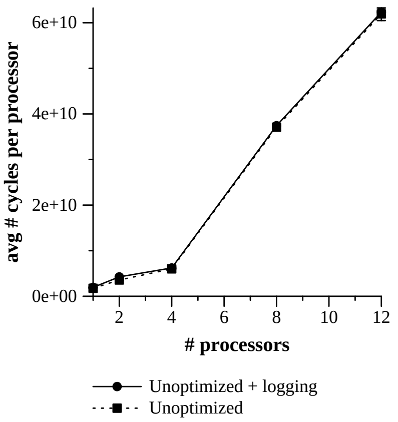

Figure 7.8: Web server – unoptimized +logging.

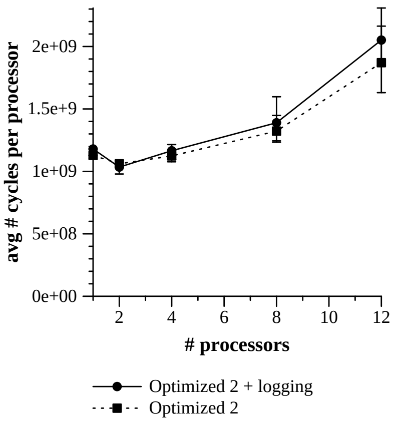

Figure 7.9: Web server – optimized 2 +logging.

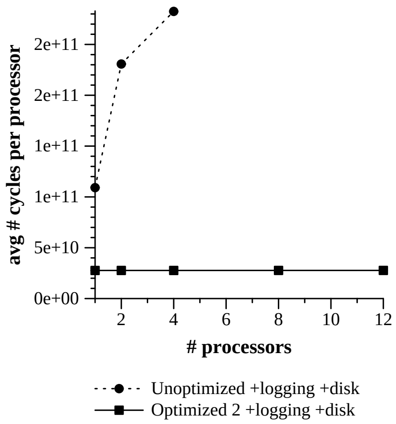

Figure 7.10: Web server – +logging +disk latency.

configuration was not intended to be the final parameters. Cache capacity misses may have been the culprit. Since the number of hash lists was reduced by a factor of 4, the remaining hash lists may have become 4 times longer. This change would have led to lengthier hash list traversals that accessed more cache entries more frequently, leading to a greater chance that the hash list elements were not resident in the secondary physical caches. The resulting additional contention on the memory bus from each processor could have led to the scalability problems seen in **Figure 7.7**.

7.3.3 Web Logging

Web logging was implemented to create a slightly more realistic scenario and determine the impact of contending read and write requests to disks. To simulate efficient Web logging, a new block was written to a Web log file every 50 requests, emulating the actions of writing a file block to disk once it has been filled with Web logging data. We used 80 bytes as the typical size of a Web log entry. Each thread appended to a Web log file located on its designated exclusive disk. We believe this distributed logging technique would naturally be used in the real world under such a scenario, since using a single global log file located on a single shared disk would be quite naive and bring an obvious imbalance to the system. Results for the unoptimized and optimized versions are shown in **Figure 7.8** and **Figure 7.9**. According to these graphs, Web logging did not affect file system scalability. The microbenchmark results for reading and writing were accurately reflected under this workload.

7.3.4 Web Logging and 15 ms Disk Latency

In addition to Web logging, a 15 ms disk access latency was introduced to determine the approximate impact of disk performance on file system scalability. Since disk accesses were made at page granularity,

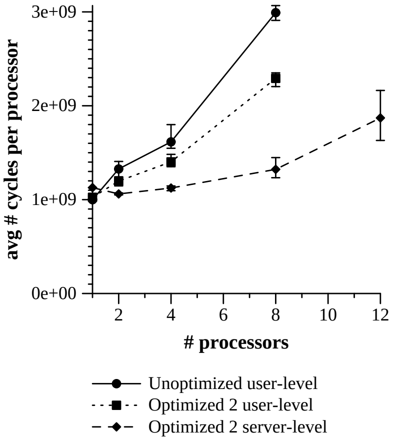

Figure 7.11: Web server – user-level.

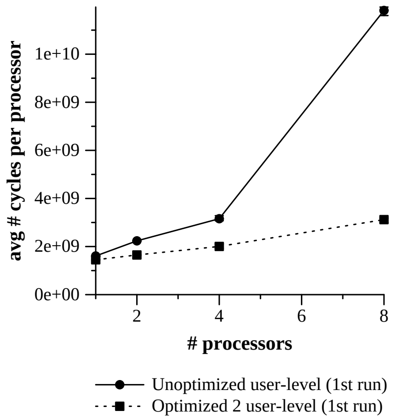

Figure 7.12: Web server – user-level, 1st run.

every 4096 bytes of disk transfer suffered a 15 ms access latency. The results, shown in **Figure 7.10**[7], indicate that there were scalability problems with the unoptimized version and that the optimizations were effective in improving scalability. More importantly, these results demonstrate that the optimizations were applicable to non-zero latency disks, despite the threat that all scalability benefits would be masked by disk latency.

7.4 User-Level Experiments

To provide a more realistic execution environment, the Web server simulator was executed as a user-level application in a separate address space instead of within the file system. Due to time constraints, zero latency disks were used and Web logging was disabled. This configuration introduced a very different operating environment compared to the Web server simulator running from within the file system. Direct communication with the file system was no longer available. As a user-level application, the Web server simulator accessed the file system using user-level I/O libraries, which communicated with the VFS and the FCM using IPC mechanisms. These components then communicated with the file system using IPC. Execution at the server-level did not require these steps since the file system functions could be called directly. The advantage of execution at user-level was that the FCM cached file data while the VFS cached name tree and file attribute meta-data. The user-level configuration allowed a larger set of the operating system components to be stressed.

[7]Due to time constraints, results for the unoptimized 8 and 12 processor configuration have not been obtained.

7.4.1 Results

With the user-level configuration, the amount of stress placed on the file system was significantly reduced, migrating it to the VFS and FCM components of the operating system instead. As shown in **Figure 7.11**[8], user-level scalability was largely unaffected by the optimizations. The user-level unoptimized and optimized versions performed similarly. Further investigation revealed that, in the unoptimized user-level version on 8 processors, only about 9% of a thread's execution time was spent in the file system. At best, we could reduce file system overhead down to 0% and improve user-level performance by at most 9%[9]. In fact, in the fully optimized user-level version (optimized 2), only 2% of a thread's execution time was spent in the file system. Based on cycle counts, these improvements reduced file system usage time by 90% (on an 8 processor system). From this perspective, the optimizations were effective, however, in the larger context of overall execution time, the optimizations had little impact.

When the Web application was run at user-level, scalability problems were not caused by the the file system, but rather were caused by other components of the operating system, such as the FCM and the VFS. Optimizing these components were beyond the scope of this thesis.

Since only 9% of a thread's execution time was spent in the file system, a larger Web log trace could have enabled the Web server simulator to add the necessary stress to the file system, causing this portion of time to increase significantly. A larger workload would cause more data to be cached and accumulated in the FCM until physical memory was depleted to the point where cache pages must be evicted. The subsequent reloading of previously evicted file data in the FCM would invoke the file system and increase its load. It could transform the FCM/VFS-bound workload into a HFS-bound workload. However, this configuration was not possible because of time constraints, speed limitations of the SimOS simulator, and the lack of page eviction support in the version of K42 used. Regardless, it is important to ensure that the file system component is scalable when highly stressed, as the server-level experiments have demonstrated.

Since the measurements were taken during the second run of the experiment, we were curious as to the results from the first run. There should be more stress on the file system since the FCM and VFS caches are initially empty and must be filled by acquiring information from the file system. The results, shown in **Figure 7.12**, indicate that the unoptimized version experienced scalability problems while the optimized version scaled much better. Upon further examination, we found at least 74% of a thread's execution time was spent in the file system on an unoptimized 8 processor configuration. Such a large proportion meant that the file system was adequately stressed. Using the fully optimized configuration reduced this portion down to less than 25%. These results indicate that, under appropriate conditions, the optimizations were effective on the user-level Web server simulator.

[8] User-level results on 12 processors could not be obtained due to OS stability problems.

[9] In fact, over-all user-level performance improved by 23%. The additional improvement was most likely due to over-all reduced memory-bus contention.

7.5 Other Configurations

This section discusses and predicts the scalability impact of modifications to our chosen Web server configuration. Possible configuration changes include (1) scaling the computer system to a larger size, (2) having multiple threads per processor, (3) reducing the ratio of disks to processors (4) using a single shared disk rather than one disk per processor, (5) Web logging to a single shared disk, (6) using a different allocation of data to disk, such as allocating exclusive portions of the Web site to particular disks, and (7) eliminating the Web server application binding of an exclusive disk to a specific processor.

Due to the complex interactions between the computer system simulator (SimOS) and the K42 operating system, we were limited to a maximum of 12 processors and 12 disks. To sustain scalability, further scaling of the Web server workload would simply require sizing the hash tables accordingly to maintain a constant ratio of demand versus resources, and to adhere to the scalability principle of Peacock et al. [56, p. 86]. As suggested by the results in this chapter, the fully optimized configuration should scale reasonably well beyond 12 processors. Contention on the multiprocessor shared system bus or memory modules would be the factor that limits scalability.

With multiple threads per processor, the multi-level hash table design would maintain a low probability of hash collisions. Although multiple threads may share a particular group of hash lists in the block cache system, there are opportunities for distributed hash list access because there are subgroups indexed by physical file number and sub-subgroups roughly indexed by file block number. Similar performance results would also be obtained if the number of disks were reduced, such as designating one disk for every two processors, which is equivalent to having two threads per processor.

With a single shared disk, hash list usage would not be optimally distributed across all hash lists. Hash list collisions would increase significantly since access would be concentrated in a small portion of the hash table. The hash function would need to be modified to accommodate this configuration. Using a single shared disk is contrary to the efforts of maintaining a well-balanced and evenly-distributed system design. The same comment can be applied to Web logging using a global shared disk. A better design would distribute the logging task to all disks and perform off-line Web log aggregation.

Altering the data to disk allocation, so that each disk contains an exclusive portion of the Web site, would have varying results. If the working set of Web files is fairly static and can be distributed evenly across all available disks, then the system can maintain good balance and should scale well. The hash lists would be evenly utilized, leading to good scalability. However, determining such a working set, and handling a dynamic and frequently changing working set is a complex problem to solve. Simply replicating all content on all disks, as was done in our configuration, may be the most practical solution.

In our experiments, we arbitrarily restricted each thread and corresponding processor to an designated, exclusive disk. Eliminating this restriction could improve scalability and absolute performance. This binding is made at the Web server application level and is not an issue of the file system core. Various technique can be used due to the nature of Web server files and the fact that all disks are identical replicas. If a thread cannot find the target meta-data in a particular hash list or if the target hash list is highly contended, it could search for the meta-data in other sections of the hash table by modifying the target disk number. This technique is only possible because all disks are identical and all threads perform similar work, suggesting that the target meta-data may exist under other sections of the hash table.

7.6 Applicability of Clustered Objects

The K42 *clustered object* mechanism [4] may be useful in maintaining scalability in HFS. The *clustered object* mechanism is used for designing, implementing, and managing execution concurrency and data locality in a structured, formalized fashion. A clustered object presents a globally shared, uniform, external interface but is internally distributed in nature. The nature of the distribution and data update policies are completely customizable by the programmer. Clustered objects can be applied to two major structures in HFS: (1) ORS and block cache hash tables, (2) the block cache free list.

Dynamically sizable, distributed hash tables can be designed and applied to the ORS and block cache hash tables. Dynamic hash table sizing enables HFS to be robust, allowing it to handle load imbalances, such as hot spots in a section of the hash table, and variable resource availability, such as available free memory, number of disks, and number of processors. Once resized, existing elements in the hash table could migrate to their new locations if necessary.

Clustered objects can offer a solution to the free list problem in the block cache without requiring major modifications to the existing code base. The clustered object free list can maintain the appearance of a single, globally shared list, but internally be organized as a distributed list. Internally, each processor could maintain a local free list. Local free lists could trigger balancing between processors as needed. In reality, only the free list header is local to a particular processor. The entries of the free list are actually block cache entries in the block cache hash table[10]. Therefore, they are globally shared entries that may exist on any processor. The free list contains several properties that make it a good candidate for clustered object implementation. First, the free list requires fast FIFO[11] capabilities, which can be satisfied solely from the local free list. Second, the free list is never traversed, meaning that there is never a need to perform global traversals[12]. There is only one scenario that requires partial global co-ordination from the clustered object. An entry in a free list may request to be removed because it no longer considers itself to be free. With such a request, the clustered object mechanism must determine which processor's local free list the entry resides (since it is really a block cache entry, which is globally shared) and acquire the corresponding free list lock

[10] As described in Section 3.3.2, the free list is threaded throughout the block cache hash lists.
[11] First in, first out – Enqueuing onto the tail of the list and dequeuing from the head of the list.
[12] A global traversal of the free list would be implemented by traversing all local free lists.

to dequeue safely.

7.7 Summary

For the Web server experiments, we have chosen only one of many possible configurations. We believe we have chosen a well-balanced hardware configuration that is appropriate for the intended workload. From this hardware configuration, we investigated whether the software can scale in unison with the hardware.

The initial optimizations developed during the microbenchmarks improved Web server performance but were not adequate in fully resolving scalability problems. Additional optimizations enabled the Web server workload to achieve good, although not ideal, scalability. These additional optimizations were the modified hash functions and the larger hash tables in both the ORS and block cache systems. The hash table played a crucial role in scalability. Further scaling of the Web server workload would simply require sizing the hash tables accordingly to maintain a constant ratio of demand versus resources, and to adhere to the scalability principle of Peacock et al. [56, p. 86].. As suggested by the results in this chapter, the fully optimized configuration should scale adequately beyond 12 processors. Contention on the multiprocessor shared system bus or memory modules would be the factor that limits scalability.

In the user-level configuration, due to the caching of file data by the FCM and the caching of name tree and file attribute meta-data by the VFS, the second run of the experiments did not adequately stress the file system to cause scalability problems. On the other hand, the optimizations were effective on the first run of the experiment because adequate stress was placed on the file system. Finally, the user-level configuration revealed scalability problems in the FCM and VFS components of the operating system.

Chapter 8

Conclusions

We have studied the scalability characteristics of HFS and have applied numerous optimizations to the meta-data cache system in the form of (1) finer grain locks, (2) larger hash tables, (3) modified hash functions, (4) padded hash list headers and cache entries, and (5) a modified block cache free list. As a result, the performance scalability of HFS has been improved significantly. As the number of processors and disks increase, the file system can handle proportionately more concurrent requests while maintaining the same throughput per request.

The methodology used to optimize HFS was effective. We started with simple experiments to enable us to understand the fundamental behavior of HFS and gradually moved to more complicated experiments. We determined the cause of performance problems at each stage, developing a thorough understanding of the dynamic behavior of the file system. This approach enabled us to identify potential problem areas during the more complicated macrobenchmark experiments.

Most of the optimizations were quick to implement and required only minor modifications. The complexity stemmed from identifying the scalability bottlenecks in a large and complicated software system. The only complicated optimization was the implementation of fine grain locks. The complication arose from implementing locks correctly so that deadlocks would not be possible. Developing an effective distributed free list, which was not done in this thesis, may be a complicated process and is a subject of future work.

The problem of achieving scalability mainly translated into the problem of ensuring that the meta-data hash lists were evenly used and distributed. Except for the global lock and global free list in the block cache system, the ORS cache system and block cache system were designed adequately in general to enable good scalability without requiring a major redesign of HFS.

8.1 General Optimization Techniques Applied

In our optimization process, we routinely used general principles of maximizing locality and concurrency [22], including:

1. Reduce sharing whenever possible.
2. Minimize the use of global data structures and locks.
3. Use of distributed data structures and locks whenever possible.
4. Minimize hash collisions.
5. Reduce lock hold times.
6. Eliminate false sharing of secondary cache lines.

The techniques listed inherently overlap since they begin quite general and become more specific. Peacock et al. [56, p. 86] had insight into multiprocessor scalability, stating that, "This problem illustrates a general principle which needs to be followed for reasonable scalability of a multiprocessor kernel, namely that queue sizes should remain bounded by a constant as the system size is increased." This principle was applied to the hash lists of the ORS and block cache system. Another important characteristic to maintain is that, in the case where true sharing cannot be fundamentally avoided, performance should degrade gracefully (linearly with the degree of sharing).

8.2 Lessons Learned

A number of lessons were learned during the production of this thesis.

1. Achieving scalability is a non-trivial task. While the optimizations themselves are simple to implement, the complexity lies in identifying the scalability bottleneck points. A significant investment in time and effort is required to understand the dynamic behavior of a large system containing complex interactions between the operating system, file system, hardware, workload, and run-time environment.

2. The user-level operating environment is quite different from the server-level environment, so different workloads may be needed for each environment to properly stress the system.

   The Web server workload provided adequate file system stress in the server-level configuration, whereas it was not sufficient for the user-level configuration. We resorted to measuring the warm up run of the Web server workload to provide adequate stress on the file system.

3. Amdahl's law [1] of parallel computing and "divide & conquer" is helpful in determining the critical scalability bottleneck points. In very simple terms, Amdahl's law states that given a program consisting of segments that can and cannot be optimized, the performance improvement is bounded by the segments that cannot be optimized.

In the user-level Web server experiments, Amdahl's law enabled us to realize that, on an 8 processor configuration, HFS scalability was not a factor in the scalability problems that arose in the second and subsequent runs of the workload since less than 10% of the execution time was spent in the file system. With this bit of wisdom, we investigated a more appropriate configuration, which led us to the first run of the workload, where at least 74% of the execution time was spent in the file system.

4. Microbenchmarks alone are inadequate in predicting real-world performance. Microbenchmarks are mainly useful for stressing specific parts of a system but not the system in its entirety.

   Optimizations developed solely during the microbenchmark experiments were not completely adequate in achieving scalability in the Web server experiments. Since additional optimizations were necessary, we experienced first-hand the well-known fact that microbenchmark workloads are not adequate in reflecting macrobenchmark and real-world workloads. The characteristics of the Web server workload were quite different from the synthetic microbenchmark workloads.

8.3 Contributions

Through the production of this thesis, the following contributions were made.

1. The Hurricane File System was ported to the K42 operating system. HFS was ported from one unconventional, research-based OS to another. This task was unlike a simple application port from one flavor of Unix to another flavor of Unix where interfaces are mostly identical. In terms of complexity, it may be similar to porting a file system between Unix and Windows OS, where interfaces are very different, and many lines of "glue" code must be introduced. Through the porting process, a clear boundary has been drawn between the operating system dependent and independent portions, leading to a fairly portable file system. In fact, the core of HFS has been embedded in several of the HFS file system tools that run on Irix and AIX operating systems. For example, on Irix, HFS runs within the core of the HFS copying tool for copying files from Irix to an HFS disk image. This flexible and scalable file system can now be ported to other operating systems more easily.

2. We began the process of examining and enabling file system scalability on a research-based, scalable, flexible operating system.

3. With the availability of a flexible and scalable file system, further file system research is now possible on K42. For instance, the building block composition [35] and object-oriented features of HFS have yet to be investigated under the context of the K42 operating system. These features are the foundations for the flexibility and customizability aspects of HFS. HFS supports a variety of file structures and policies. As described by Krieger in [34], file structures are available for sequential, random access,

read-only, write-only or read/write access, sparse or dense data, large or small file sizes, and various degrees of application concurrency and dynamic data distribution. A variety of policies are available for locking, prefetching, compression/decompression, and file meta-data cache management. These capabilities, combined with a scalable file system, offer a rich area of file system research.

4. Having a flexible and scalable file system available as a component of a scalable operating system is important. It provides a tool for further operating system research by eliminating the file system as the scalability bottleneck of the system. Larger, more realistic, operating system workloads that require a scalable file system can now be executed on K42. Components of the K42 operating system can now be stressed in a manner not possible in the past, potentially identifying performance problem areas and helping to verify complete system scalability.

5. The exercising and testing of K42 code was indirectly performed through the process of running various experiments. The experiments that were executed stressed the K42 operating system in a manner different from the past, such as relatively large workloads that stressed the I/O subsystem and relatively large memory requirements that stressed the memory management system. For instance, the experiments stressed K42 in new ways that revealed scalability problems in the FCM and VFS components.

8.4 Future Work

There are many remaining tasks. The important ones related to this dissertation include:

1. Implementation of a distributed free list in the block cache system.
2. Further optimization of the fine-grain locks implementation in the block cache.
3. Performance analysis and optimization of the VFS and FCM components in the context of Section 7.4.1.
4. Examination of other configurations of disks, processors, and threads under the context of the Web server workloads.
5. Examination of mechanisms required in HFS for graceful degradation in scalability when sharing is inherently necessary.
6. Execution of the benchmarks on real hardware and validation of simulation results.
7. Execution of benchmarks that contain file sharing.
8. Execution of industry benchmarks, such as PostMark [30].

A number of potential bottlenecks related to single shared-disk scenarios were realized during the examination of the HFS source code. These bottlenecks have not yet been examined. For example, consider when 12 processors simultaneously request access to disk number 0. Various shared data structures will be accessed by these processors. One such data structure is the per disk super block. A disk's super block is protected by a lock, which must be acquired to allocate and free disk blocks. Parallel file creation or file writing to a single shared disk may experience scalability problems. Tasks to further increase parallelism include the following.

1. Completing the redesign of HFS meta-data organization.
2. Implementing per disk queues in the disk interface objects (**Figure 3.1**), as found in the original version of HFS, to enable more capabilities and control of disks, leading to disk-based optimizations such as prefetching and scheduling.
3. Porting the HFS user-level library (**Figure 3.1**) to enable the flexibility of HFS to be fully exploited.

Now that the core of the file system has been optimized, we need to focus on the interaction between the file system and the operating system. It is currently an open research issue of finding the optimal way for the K42 operating system to interact with various file systems. How do we design a single file system interface that contains enough flexibility to fully exploit the capabilities of a variety of file systems and fully exploit the capabilities of K42? HFS was designed for micro-kernel operating systems and therefore, its operating behavior is very compatible with the K42 operating system. Other file systems may not integrate as smoothly.

This thesis represents an initial step in file system research on K42. Hopefully, it will provide guidance on further improving file system scalability.

Bibliography

[1] Gene M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485, Atlantic City, New Jersey, April 1967. AFIPS.

[2] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. In *15th ACM Symposium on Operating Systems Principles*. ACM Press, December 1995.

[3] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The interaction of architecture and operating system design. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 108–120, Santa Clara, California, April 1991. ACM Press.

[4] Jonathan Appavoo. Clustered objects: Initial design, implementation, and evaluation. Master's thesis, University of Toronto, Department of Computer Science, Toronto, Ontario, Canada, 1998.

[5] Martin Arlitt and Tai Jin. 1998 World Cup Web site access logs. Available at `http://www.acm.org/sigcomm/ITA/`., August 1998.

[6] Martin Arlitt and Tai Jin. Workload characterization of the 1998 World Cup Web site. Technical Report HPL-1999-35R1 991006 External, Hewlett-Packard Laboratories, Palo Alto, California, September 1999.

[7] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 198–212. ACM Press, 1991.

[8] Steve Best. JFS log: How the journaled file system performs logging. In *4th Annual Linux Showcase & Conference*, pages 163–168, Atlanta, Georgia, October 2000.

[9] Steve Best and Dave Kleikamp. *JFS Layout: How the Journaled File System Handles the On-Disk Layout*. IBM Linux Technology Center, May 2000.

[10] Rajesh Bordawekar, Alok Choudhary, and Juan Miguel Del Rosario. An experimental performance evaluation of Touchstone Delta Concurrent File System. In *Proceedings of the 7th ACM International Conference on Supercomputing*, pages 367–376. ACM Press, 1993.

[11] Rajesh Bordawekar, Steven Landherr, Don Capps, and Mark Davis. Experimental evaluation of the Hewlett-Packard Exemplar File-System. *ACM SIGMETRICS Performance Evaluation Review*, 25(3):21–28, December 1997.

[12] Ray Bryant, Ruth Forester, and John Hawkes. Filesystem performance and scalability in Linux 2.4.17. In *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*, Monterey, California, June 2002. USENIX Association.

[13] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, Georgia, October 2000.

[14] Miguel Castro and Barbara Liskov. Proactive recovery in a byzantine-fault-tolerant system. In *Fourth Symposium on Operating Systems Design and Implementation*. USENIX Association, October 2000.

[15] Peter M. Chen and David A. Patterson. A new approach to I/O performance evaluation – self-scaling I/O benchmarks, predicted I/O performance. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. ACM Press, May 1993.

[16] Peter M. Chen and David A Patterson. Storage performance – metrics and benchmarks. In *Proceedings of the IEEE*, volume 81:8. IEEE, August 1993.

[17] Peter F. Corbett and Dror G. Feitelson. The Vesta Parallel File System. *ACM Transactions on Computer Systems*, 14(3):225–264, August 1996.

[18] Jeffrey M. Denham, Paula Long, and James A. Woodward. DEC OSF/1 symmetric multiprocessing. *Digital Technical Journal*, 6(3), Summer 1994.

[19] Peter C. Dibble, Michael L. Scott, and Carla Schlatter Ellis. Bridge: A high-performance file system for parallel processors. In *Proceedings of the Eighth International Conference on Distributed Computer Systems*, pages 154–161, San Jose, California, June 1998. IEEE Computer Society, IEEE.

[20] John Douceur and William Bolosky. A large scale study of file-system contents. In *Proceedings of the SIGMETRICS'99 International Conference on Measurement and Modeling of Computer Systems*, pages 59–70, Atlanta, Georgia, May 1999. ACM Press.

[21] Steven L. Gaede. Perspectives on the SPEC SDET Benchmark. Technical report, Lone Eagle Systems, Inc., January 1999.

[22] Ben Gamsa. *Tornado: Maximizing Locality and Concurrency in a Shared-Memory Multiprocessor Operating System*. PhD thesis, University of Toronto, Department of Computer Science, 1999.

[23] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings to the 3rd Symposium on Operating Systems Design and Implementation*. USENIX Association, February 1999.

[24] Robin Grindley, Tarek Abdelrahman, Stephen Brown, Steve Caranci, Derek DeVries, Ben Gamsa, Alex Grbic, Mitch Gusat, Robert Ho, Orran Krieger, Guy Lemieux, Kevin Loveless, Naraig Manjikian, Paul McHardy, Sinisa Srbljic, Michael Stumm, Zvonko Vranesic, and Zeljko Zilic. The NUMAchine Multiprocessor. In *Proceedings of the 2000 International Conference on Parallel Processing*, pages 487–496, Toronto, Ontario, August 2000. IEEE.

[25] Roger L. Haskin. Tiger Shark - a scalable file system for multimedia. *IBM Journal of Research and Development*, 42(2):185–197, March 1998.

[26] Steve Herrod, Mendel Rosenblum, Edouard Bugnion, Scott Devine, Robert Bosch, John Chapin, Kinshuk Govil, Dan Teodosiu, Emmett Witchel, and Ben Verghese. *The SimOS Simulation Environment*. Stanford University, Stanford, California, 1998. `http://simos.stanford.edu/userguide/`.

[27] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51 – 81, February 1988.

[28] Sitaram Iyer and Peter Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *18th ACM Symposium on Operating Systems Principles*, Operating Systems Review, pages 117 – 130. ACM Press, December 2001.

[29] Michael K. Johnson. Red Hat's new journaling file system: ext3. White paper, Red Hat, Raleigh, North Carolina, 2001.

[30] Jeffrey Katcher. PostMark: a new file system benchmark. Technical report TR3022, Network Appliance, October 1997.

[31] David Kotz and Carla Schlatter Ellis. Caching and writeback policies in parallel file systems. *Journal of Parallel and Distributed Computing*, 17(1–2):140–145, January & February 1993.

[32] David Kotz and Nils Nieuwejaar. Dynamic file-access characteristics of a production parallel scientific workload. In *Proceedings of Supercomputing 1994*, pages 640–649, November 1994.

[33] David Kotz, Song Bac Toh, and Sriram Radhakrishnan. A detailed simulation model of the HP 97560 disk drive. Technical report PCS-TR94-220, Department of Computer Science, Dartmouth College, July 1994.

[34] Orran Krieger. *HFS: A flexible file system for shared-memory multiprocessors*. PhD thesis, Department of Electrical and Computer Engineering, University of Toronto, Toronto, Ontario, Canada, October 1994.

[35] Orran Krieger and Michael Stumm. HFS: A performance-oriented flexible file system based on building-block compositions. *ACM Transactions on Computer Systems*, 15(3):286–321, August 1997.

[36] Orran Krieger, Michael Stumm, and Ronald Unrau. The Alloc Stream Facility: A redesign of application-level stream I/O. *IEEE Computer*, 27(3):75–83, March 1994.

[37] Nancy P. Kronenberg, Henry M. Levy, and William D. Strecker. VAXclusters: A closely-coupled distributed system. *ACM Transactions on Computer Systems*, 4(2):130–146, May 1986.

[38] Thomas T. Kwan and Daniel A. Reed. Performance of the CM-5 Scalable File System. In *Proceedings of the 8th ACM International Conference on Supercomputing*, pages 156–165, Manchester, England, 1994. ACM Press.

[39] Paul Leach and Dan Perry. Standardizing internet file systems with CIFS. *Microsoft Internet Developer*, November 1996.

[40] S. LoVerso, N. Paciorek, A. Langerman, and G. Feinberg. The OSF/1 Unix filesystem (UFS). In *Proceedings of the Winter 1991 USENIX*, pages 207–218. USENIX Association, January 1991.

[41] Evangelos P. Markatos. Speeding up TCP/IP: Faster processors are not enough. In *21st IEEE International Performance, Computing, and Communication Conference*, pages 341–345, Phoenix, Arizona, April 2002. IEEE.

[42] Chris Mason. Journaling with ReiserFS. *Linux Journal*, 2001(82), February 2001.

[43] Jim Mauro and Richar McDougall. *Solaris Internals: Core Kernel Architecture*. Prentice Hall PTR, Upper Saddle River, New Jersey, 2001.

[44] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for Unix. *ACM Transactions on Computer Systems*, 2(3):181–193, August 1984.

[45] L. W. McVoy and S. R. Kleiman. Extent-like performance from a UNIX file system. In *USENIX*, Dallas, Texas, Winter 1991. USENIX Association.

[46] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, April 1965.

[47] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[48] Nils Nieuwejaar. *Galley: A New Parallel File System*. PhD thesis, Department of Computer Science, Dartmouth College, Hanover, New Hampshire, 1996.

[49] Nils Nieuwejaar and David Kotz. The Galley Parallel File System. In *ACM International Conference on Supercomputing*, pages 374–381. ACM Press, May 1996.

[50] Nils Nieuwejaar and David Kotz. Performance of the Galley Parallel File System. In *Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 83–94. ACM Press, May 1996.

[51] Nils Nieuwejaar, David Kotz, Apratim Purakayastha, Carla Schlatter Ellis, and Michael L. Best. File-access characteristics of parallel scientific workloads. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1075–1089, October 1996.

[52] John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of the Summer USENIX Conference*, pages 247–256. USENIX Association, June 1990.

[53] John K. Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the UNIX 4.2 BSD File System. In *Proceedings 10th ACM Symposium on Operating Systems Principles*. ACM Press, 1985.

[54] Arvin Park, Jeffrey C. Becker, and Richard J. Lipton. IOStone: a synthetic file system benchmark. *Computer Architecture News*, 18(2):45–52, June 1990.

[55] J. Kent Peacock. File system multithreading in System V Release 4 MP. In *Proceedings of the Summer 1992 USENIX*, pages 19–29. USENIX Association, June 1992.

[56] J. Kent Peacock, Sunil Saxena, Dean Thomas, Fred Yang, and Wilfred Yu. Experiences from multithreading System V Release 4. In *SEDMS III. Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 77–91. USENIX Association, March 1992.

[57] Apratim Purakayastha, Carla Schlatter Ellis, David Kotz, Nils Nieuwejaar, and Michael Best. Characterizing parallel file-access patterns on a large-scale multiprocessor. In *Proceedings of the Ninth International Parallel Processing Symposium*, pages 165–172, April 1995.

[58] Stephen Tweedie Remy Card, Theodore Ts'o. Design and implementation of the Second Extended Filesystem. In Frank B. Brokken et al, editor, *Proceedings of the First Dutch International Symposium on Linux*, Amsterdam, December 1994.

[59] Erik Riedel, Catharin van Ingen, and Jim Gray. A performance study of sequential I/O on Windows NT 4. In *Proceedings of the 2nd USENIX Windows NT Symposium*, Seattle, Washington, August 1998. USENIX Association.

[60] Rodrigo Rodrigues, Miguel Castro, and Barbara Liskov. BASE: Using abstraction to improve fault tolerance. In *18th ACM Symposium on Operating Systems Principles*, Operating Systems Review, pages 15 – 28. ACM Press, December 2001.

[61] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A comparison of file system workloads. In *Proceedings of 2000 USENIX Annual Technical Conference*. USENIX Association, June 2000.

[62] Mendel Rosenblum, Edouard Bugnion, Scott Divine, and Alan Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Transactions on Modeling and Computer Simulation*, 7(1):78–103, January 1997.

[63] Mark Russinovich. Inside Win2K NTFS, Part 1. *Windows 2000 Magazine*, October 2000.

[64] Mark Russinovich. Inside Win2K NTFS, Part 2. *Windows 2000 Magazine*, November 2000.

[65] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun Network Filesystem. In *Proceedings of the Summer USENIX Technical Conference*, pages 119–130. USENIX Association, Summer 1985.

[66] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, April 1990.

[67] Frank Schmuck and Roger Haskin. GPFS: A shared-disk fle system for large computing clusters. In *Proceedings of the Conference on File and Storage Technologies*, pages 231–244, Berkeley, California, January 2002. USENIX Association.

[68] Margo Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl Staelin. An implementation of a log-structured file system for UNIX. In *Winter USENIX*. USENIX Association, January 1993.

[69] Margo Seltzer, David Krinsky, Keith Smith, and Xialan Zhang. The case for application-specific benchmarking. In *The 7th Workshop of Hot Topics in Operating Systems*, Rio Rico, Arizona, March 1999. IEEE.

[70] Anish Sheth and K. Gopinath. Data structure distribution & multi-threading of Linux file system for multiprocessors. In *Proceedings of HiPC'98 5th International Conference on High Performance Computing*, pages 97–104, Chennai (Madras), India, December 1998.

[71] Keith A. Smith and Margo I. Seltzer. File system aging – increasing the relevance of file system benchmarks. In *Proceedings of the 1997 ACM SIGMETRICS Conference*. ACM Press, June 1997.

[72] Keith Arnold Smith. *Workload-Specific File System Benchmarks*. PhD thesis, The Division of Engineering and Applied Sciences, Harvard University, Cambridge, Massachusetts, January 2001.

[73] SPEC. SPECweb99 release 1.02. Technical report, Standard Performance Evaluation Corporation, July 21, 2000.

[74] SPEC. SFS 3.0. Whitepaper 1.0, Standard Performance Evaluation Corporation, Warrenton, Virginia, 2001.

[75] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In *USENIX*, San Diego, California, January 1996. USENIX Association.

[76] David Tam. Performance analysis of Apache in a multiprocessor environment. Course project, University of Toronto, Department of Electrical and Computer Engineering, Toronto, Ontario, Canada, December 1999.

[77] Diane L. Tang. Benchmarking filesystems. TR-19-95, Harvard University, April 1995.

[78] K42 Team. K42 overview. Technical report, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, October 2001.

[79] Rajeev Thakur, William Gropp, and Ewing Lusk. An experimental evaluation of the parallel I/O systems of the IBM SP and the Intel Paragon using a production application. In *Proceedings of the 3rd International Conference of the Austrian Center for Parallel Computation with Special Emphasis on Parallel Databases and Parallel I/O*, September 1996.

[80] Stephen Tweedie. Ext3, journaling filesystem. Presentation at Ottawa Linux Symposium, July 2000.

[81] Ron Unrau, Michael Stumm, Orran Krieger, and Ben Gamsa. Hierarchical clustering: A structure for scalable multiprocessor operating system design. *Journal of Supercomputing*, 1995.

[82] Werner Vogels. File system usage in Windows NT 4.0. In *Proceedings of the 17th International ACM Symposium on Operating Systems Principles*, volume 34 of *Operating Systems Review*, pages 93–109. ACM Press, 1999.

[83] Zvonko G. Vranesic, Michael Stumm, David Lewis, and Ron White. Hector – a hierarchically structured shared memory multiprocessor. *IEEE Computer*, 24(1):72–80, January 1991.