Providing a Shared File System in the Hare POSIX Multikernel

by

Charles Gruenwald III

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2014

Author .....
Department of Electrical Engineering and Computer Science
May 21, 2014

Certified by .....
Frans Kaashoek
Professor
Thesis Supervisor

Certified by .....
Nickolai Zeldovich
Associate Professor
Thesis Supervisor

Accepted by .....
Leslie Kolodziejski
Chairman, Department Committee on Graduate Theses

Providing a Shared File System in the Hare POSIX Multikernel

by

Charles Gruenwald III

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 2014, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract

Hare is a new multikernel operating system that provides a single system image for multicore processors without cache coherence. Hare allows applications on different cores to share files, directories, file descriptors, sockets, and processes. The main challenge in designing Hare is to support shared abstractions faithfully enough to run applications that run on traditional shared-memory operating systems with few modifications, and to do so while scaling with an increasing number of cores.

To achieve this goal, Hare must support shared abstractions (e.g., file descriptors shared between processes) that appear consistent to processes running on any core, but without relying on hardware cache coherence between cores. Moreover, Hare must implement these abstractions in a way that scales (e.g., sharded directories across servers to allow concurrent operations in that directory). Hare achieves this goal through a combination of new protocols (e.g., a 3-phase commit protocol to implement directory operations correctly and scalably) and leveraging properties of non-cache coherent multiprocessors (e.g., atomic low-latency message delivery and shared DRAM).

An evaluation on a 40-core machine demonstrates that Hare can run many challenging Linux applications (including a mail server and a Linux kernel build) with minimal or no modifications. The results also show these applications achieve good scalability on Hare, and that Hare's techniques are important to achieving scalability.

Thesis Supervisor: Frans Kaashoek
Title: Professor

Thesis Supervisor: Nickolai Zeldovich
Title: Associate Professor

Acknowledgments

The author of this thesis would like to acknowledge their advisors, lab-mates, family and friends for all of their support while working on this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A current trend in modern computer architectures is increasing core counts, meaning that future architectures may contain many cores in a single machine. However, not all hardware features can scale easily due to complexity and power constraints. As a consequence, there are processors which provide limited or no support for some architectural features such as cache coherence among cores.

This thesis describes an operating system and file system designed for multicore architectures which do *not* support cache coherence between cores. Even though the hardware does not provide cache coherence, the operating system supports a broad range of POSIX style applications in a scalable manner, while providing consistency for shared files, directories, sockets, file descriptors and migrated processes.

This chapter provides an overview for the system. It describes the architectures the operating system is designed to support. Next, it describes the approach of using a multikernel design to run applications and provide operating system services on this hardware. With this, the problems which arise when trying to provide existing interfaces to applications using the multikernel approach. Finally, the research contributions that Hare uses to solve these problems.

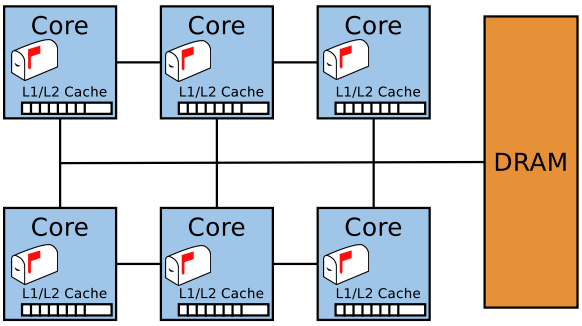

Figure 1-1: A multicore processor design with private processor caches, a shared DRAM, but without cache coherence in hardware.

1.1 Non-Cache-Coherent Multicore Processors

As the number of cores increases in a processor, there are certain features that do not scale easily. Caches provide a small area of memory that provides significantly faster access times than DRAM. When there are multiple cores in a machine, cache coherence ensures that the data contained in each of the caches is consistent with main memory across all cores. At a high level, this means if a core reads a memory location that has been written by another core, then the reader will observe the most recent changes made by the writer. Maintaining cache coherence across all cores can be expensive both in complexity as well as power as the number of cores increases, because sophisticated protocols are necessary to provide consistency at scale. As a consequence, several architectures currently available such as Intel's SCC [11], IBM Cell's SPE [10], the TI OMAP4 SoC [14], the Intel Xeon Phi [22] and GPGPUs do not support cache coherence, and future architectures as well may not support this feature.

Figure 1-1 depicts a logical representation of the hardware architecture that this thesis targets. This diagram is a representative design of the common features found in many existing commodity architectures that lack cache coherence. Cores communicate with each other using messages, while supporting reads and writes to a shared DRAM. Processors cache the results of such reads and writes in their private caches, but do not provide cache coherence between the cores.

In the absence of cache coherence, there is potential for cores to read *stale* data

from the cache which is not consistent. For example, if one core writes to a memory location and that write is cached, other cores will not see that update until it is flushed and invalidated. As a consequence, specific steps need to be taken by the software to ensure that this inconsistency does not result in bugs or unintended program behavior. Writing software to take advantage of such architectures therefore presents new challenges.

The operating system provides good context for exploring the challenges of writing software for architectures that lack cache-coherent shared-memory for several reasons. First, the operating system is an important piece of software with several challenging aspects such as scheduling processes between cores and consistently managing per-process state. Second, most applications rely on the operating system to provide shared abstractions for features such as process management, file access and networking. For example, if one process writes to a file, then reads by a different process from that file should return data from that write. Third, existing applications indicate typical workloads for operating systems which can direct the design of the system. Fourth, the operating system requires non-trivial sharing between applications and services that must be provided without relying on cache coherence.

1.2 Approach: Multikernel Operating Systems

Multikernel operating systems [2, 30] have been proposed as an alternative to traditional shared-memory operating systems for manycore processors. Multikernel operating systems run an independent kernel on each core, and these kernels communicate using messages to implement operating system services and provide synchronization. In the context of this thesis, the *kernel* refers to trusted software that provides an interface to device drivers, per-core operating system state as well as low-level features like memory management. One advantage of this design is that the operating system is viewed as a distributed system, which closely matches the underlying hardware. Additionally, since all sharing is explicit, multikernels have the potential to avoid scalability bottlenecks that have plagued shared-memory kernel designs [4]. Lastly, by

using a design that mimics a distributed system, known techniques can be adapted from distributed systems research and employed to provide interfaces in a correct, scalable manner.

Due to the fact that multikernels communicate only through message passing, they are particularly suitable for multicore processors without cache-coherent shared-memory. The independent kernels are solely responsible for managing their own core as data structures are private, or independent from kernels running on other cores. This separation means that issues such as reading stale data in a cache does not arise for the kernel.

1.3 Problem: Sharing in Multikernel Operating Systems

A challenge for multikernels is to implement services that *do* require sharing. For example, many applications rely on a shared file system, but a multikernel cannot rely on cache-coherent shared memory to implement a buffer cache, inode table, etc. This thesis contributes a scalable design for implementing a POSIX file system on a multikernel operating system. In previous work [3] we have explored a solution for sharing network sockets this context, however this thesis focuses mainly on the file system.

Previous multikernel operating systems implement file services using a distributed file system for local-area networks and data centers. For example, Barrelfish runs an NFS [28] backed file system [2]. Processes share files by making NFS RPCs to an NFS server. There are several drawbacks to this approach including performance and limited application support. First, the file server itself can be a bottleneck for meta-data updates (such as creating or renaming files). Sharding may allow concurrency for operations in different directories, however operations within a single directory are serialized. Second, there is no mechanism for clients in traditional distributed filesystems to share file descriptors, which limits processes that share file descriptors

to running on a single core. Third, the clients are not designed to take advantage of the presence of non-cache-coherent shared DRAM. This results in reduced capacity for the buffer cache because blocks are duplicated by separate clients, and as a consequence, reduced performance for applications using this approach.

Because of these limitations, the operating system cannot take advantage of much of the parallelism provided by the hardware. Applications are limited to a single core or may suffer poor performance when run across many cores.

1.4 Goal: Single System Image

From the perspective of distributed systems, a single system image is a system interface for applications that presents the logical view of a unified system. In the context of multikernel operating systems, a single system image implies access to system resources such as the file system and networking stack from any core with the same semantics that would be provided on a traditional shared-memory operating system. This also means that running processes on separate cores and sharing abstractions such as file descriptors between these processes is supported across the independent kernels.

In the absence of a single-system-image interface, applications running on a multikernel operating system are forced to view the operating system and the abstractions it provides as a distributed system. This scenario can be difficult to program as the application coordinate access to shared data structures which frequently requires distributing them across several cores in order to achieve good scalability. However, once the data has been distributed, the application is then responsible for explicitly coordinating accesses and updates to these data structures as the availability of locking to provide mutual exclusion is not available in a multikernel design. In this regard, it is advantageous to limit the distributed system interface to the operating system kernel.

In order to avoid the difficulties of programming applications as a distributed system with explicit coordination, the goal of this thesis is to provide a single system

image across the multicore processes despite independent kernels and lack of cache coherence. This allows the application programmer to build applications much as they would on a traditional shared-memory operating system. Using familiar interfaces with the same semantics allows the application to run as it would if the architecture supported cache coherence. As a result, many existing applications are able to run unmodified while new applications can be written without learning the specifics of the underlying messaging protocol.

This thesis targets supporting a single system image that implements the POSIX API [13] faithfully enough to run a broad range of applications across all of the cores. In particular, abstractions that are commonly shared between different applications such as files, directories, file descriptors, pipes and network sockets are all supported in a manner in which they may be accessed from applications running on any core. The main challenge in providing a single system image on a multikernel operating system is ensuring a sufficient level of consistency for shared data structures between the application and the operating system through the messaging layer. Certain aspects of operating system abstractions, such as path resolution, must be explicitly cached in software for performance while new protocols are needed to ensure that this cached information is consistent enough to provide a faithful POSIX interface.

1.5 Hare System Overview

This thesis contributes a multikernel operating system and file system design (called *Hare*) that supports the standard POSIX file system interface in a scalable manner across many cores. In Hare, each core runs a kernel with a Hare client library. Processes make POSIX system calls, and the local client library sends remote procedure calls (RPCs) to one or more server processes to implement the POSIX call. The central challenge in Hare's design is providing coherence for file system data structures without relying on cache-coherent shared memory. Hare addresses this challenge through a combination of new protocols and implementation techniques.

Hare's design has three novel aspects compared to networked distributed file sys-

tems. First, Hare partitions the buffer cache and stores each partition in DRAM. When a process accesses a file, its local client library can read and write the buffer cache directly; since DRAM accesses are not cache-coherent, this provides close-to-open consistency [12, 19].

Second, to allow for scalable directory operations, Hare distributes directory entries in the same directory to different servers, using hashing. Each server stores the metadata for its directory entries. Each Hare client library caches the results of directory lookup operations to avoid server communication on each path lookup. When a client library must list all of the entries in a directory, it issues RPCs to servers in parallel to reduce the latency of listing the directory.

Third, Hare supports sharing of file descriptors between processes. When a parent process spawns a child process, the child can use the file descriptors opened by the parent, even if the client runs on a different core. File descriptor state is stored in a Hare file server, and all file descriptor operations are implemented as RPCs to that server. For performance, the file descriptor is typically co-located with the server storing the corresponding file.

We have implemented Hare by logically running a Linux kernel on each core, interposing on Linux system calls, and redirecting those system calls to a local Hare client library. This implementation strategy forced us to implement a sufficiently complete POSIX file system so that Hare can run many Linux applications with few modifications, including building the Linux kernel, running a mail server, etc.

We evaluate Hare on a 40-core off-the-shelf machine. This machine provides cache-coherent shared memory. Like previous multikernel operating systems, Hare uses cache coherence purely to pass messages from one core to another. Whenever Hare uses shared DRAM to store data, it explicitly manages cache consistency in software. We run complete application benchmarks on Hare (such as compiling the Linux kernel and a mailserver benchmark), many common Unix utilities, scripting languages like bash and python as well as microbenchmarks to test specific aspects of Hare's design. Many of the parallel benchmarks as well as full applications such as the mailserver benchmark and build of the Linux kernel run in a scalable manner.

We find that some benchmarks scale well while other scale moderately. The main reason that some scale moderately is that Hare's directory operations must contact all servers. We also find that the individual techniques that Hare uses are important to achieve good scalability. Finally, we find that performance of Hare on a small number of cores is reasonable: on some benchmarks Hare is up to 3.4× slower than Linux, however in many cases it is faster than UNFS3 which represents a more realistic comparison for the architectures that Hare targets.

1.6 Thesis Contributions

The main research contributions of this thesis include the following:

- The design and implementation of Hare: A scalable multikernel operating system and filesystem which supports the following:
  - Sharing of files, file descriptors and directories.
  - Process migration through the `exec` call.
  - A POSIX style interface which supports a broad range of applications.
- Protocols to provide a single system image:
  - Close-to-open consistency with strong consistency for shared file descriptors.
  - A three-phase commit protocol to ensure consistency when removing distributed directories.
  - An invalidation protocol to allow consistent caching of directory lookup operations.
- Optimizations for performance and scalability:
  - A mechanism for distributing individual directories across multiple servers.
  - Using a shared buffer cache to improve capacity and performance.
  - Broadcast directory operations which allows listing and removal to happen concurrently on multiple servers.

  - A directory lookup cache that can reduce the number of lookup operations.

- An evaluation on commodity hardware that demonstrates the following:
  - Support for a broad range of applications.
  - Scalability up to the maximum number of cores across a variety of applications and benchmarks.
  - Competitive performance on a small number of cores.
  - The necessity of techniques and optimizations in order to achieve good performance.
  - That techniques used in Hare could improve performance for some workloads in traditional shared-memory operating systems.

Although the Hare prototype implements a wide range of POSIX calls, it has some limitations that we plan to remove in future work. The most notable is that while our evaluation shows that changing the number of servers dynamically based on workload can improve performance, Hare does not support this feature yet. We believe that implementing support for this feature should not affect the conclusions of this thesis.

1.7 Thesis Roadmap

The rest of the thesis is organized as follows. Chapter 2 relates Hare to previous file system designs. Chapter 3 describes Hare's design. Chapter 4 details Hare's implementation. Chapter 5 evaluates Hare experimentally. Chapter 6 describes our plans for future work as well as some open discussion topics. Chapter 7 summarizes our conclusions.

Chapter 2

Related Work

Hare is the first multikernel operating system that provides a single system image in a scalable manner, without relying on cache coherence. Hare's design builds on ideas found in previous multikernels and file systems, but contributes a new design point. Hare takes advantage of the properties of a non-cache-coherent machine to implement shared file system abstractions correctly, efficiently, and scalably. In particular, Hare's design contributes several new protocols for implementing shared files, shared directories, shared file descriptors, and migrating processes.

2.1 LAN Distributed File Systems

Hare's design resembles networked distributed file systems such as NFS [19] and AFS [12], and borrows some techniques from these designs (e.g., directory caching, close-to-open consistency, etc). Sprite supports transparent process migration [8] to extend the Sprite file system to multiple nodes. The primary differences are that Hare can exploit a shared DRAM to maintain a single buffer cache, that directory entries from a single directory are distributed across servers, and that file descriptors can be shared among clients. This allows Hare to run a much broader range of applications in a scalable manner than network file systems on our target architecture.

2.2 Datacenter and Cluster File Systems

Datacenter and cluster file systems are designed to support parallel file workloads. A major difference is that the Metadata Server (MDS) in these designs is typically a single entity, as in Lustre [6] and the Google File System [9], which creates a potential bottleneck for metadata operations. The Flat Datacenter Storage (FDS) [18] solution uses a Tract Locator Table to perform lookups. This design avoids the MDS on the critical path, but FDS is a blob store and not a general file system. Blizzard [17] builds a file system on top of FDS as a block store, but does not allow multiple clients to share the same file system. Ceph [29] uses a distributed approach to metadata management by using Dynamic Subtree Partitioning to divide and replicate the metadata among a cluster of Metadata Servers. As with traditional distributed file systems, they cannot exploit a shared DRAM and don't support sharing of file descriptors across clients.

Shared-disk file systems such as Redhat's GFS [21, 31] and IBM's GPFS [24] enable multiple nodes to share a single file system at the disk block level. Such designs typically store each directory on a single disk block, creating a bottleneck for concurrent directory operations. Furthermore, such designs cannot take advantage of a shared buffer cache or a shared directory cache, and cannot support shared file descriptors between processes.

2.3 File Systems for SMP Kernels

Another potential approach to designing a file system for multikernel operating systems might be to adopt the design from existing SMP kernels, such as Linux. These designs, however, rely on cache coherence and locking of shared data structures, which multikernel operating systems cannot assume. Furthermore, the SMP designs have been plagued by scalability bottlenecks due to locks on directories and reference counting of shared file system objects [5].

HFS [15] is a system designed for SMP architectures that allows the application to have fine-grained control of how files are organized and accessed to improve

performance based on a workload's access pattern. This system is focused on file layout without addressing parallelism for directory operations. Furthermore, it exposes modified interfaces as opposed to providing the POSIX API.

In other realms, such as databases, it has been suggested to treat the multicore as a cluster, and to run separate copies of the service on different cores, partitioning the data across the cores [23]. This technique makes data movement between the individual partitions difficult, whereas a design like Hare can update a directory entry while leaving file data in place, and access the data through shared-memory (with explicit cache writeback and invalidation in software to provide the required consistency).

2.4 Multikernel Designs

Hare's split design between a client library and multiple file system servers is inspired by the file system in the fos [30] multikernel. The fos design, however, is limited to read-only workloads [30]. Barrelfish uses a standard distributed file system, NFS [27]. Hare extends these designs with support for multiple servers supporting read/write workloads while also adding more support for the POSIX API, in particular sharing of file descriptors between processes running on separate cores.

K2 [16] is designed for machines with separate domains that have no cache coherence between them. Their design targets mobile platforms with several domains each of which consists of multiple cores, though the design targets a low number of domains. Hare instead targets an architecture where each coherence domain consists of a single core. Since K2 relies on distributed shared memory for sharing OS data structures across coherency domains, workloads which perform many operations in the same directory will experience a bottleneck when run in different domains. Hare does not use distributed shared memory, but relies on new protocols specifically designed for shared files, directories, etc to achieve good performance and scalability.

Cerberus [26] is designed to scale to many cores by instantiating several virtual machines and providing the view of a single system image through messaging. Hare

takes the contrasting approach of solving scalability challenges within the operating system. Due to the reliance on the underlying operating system which makes use of cache coherence, Cerberus does not provide a solution on architectures which lack this feature. Hare can run on any system which provides support for message passing, obviating the need for a virtual machine monitor. Hare also differs from Cerberus in its support for fine-grained directory distribution, whereas Cerberus delegates directories to a specific domain.

2.5 File Systems for Heterogeneous Architectures

GPUfs [25] is similar to Hare in that it uses a shared DRAM between a GPU and the host processor to implement a distributed file system. Hare differs from GPUfs in handling directories, file offsets, and other shared state exposed in the POSIX interface. Furthermore, GPUfs is focused purely on accessing file contents through a restricted interface without supporting shared file descriptors. Without sharing of file descriptors, pipes and redirection as well as make's job server will not function, limiting the number of supported applications.

Chapter 3

Design

Hare's goal is to run a wide variety of POSIX applications out-of-the-box on a machine with non-cache-coherent shared memory, while achieving good performance and scalability. This means providing a single system image as expected by applications using the POSIX API, which amounts to providing shared abstractions like a single scheduler, a shared file system, a network stack, and so on, across all cores. This goal is challenging because of the lack of cache coherence: Hare cannot rely on the hardware to keep shared data structures such as the buffer cache, the inode tables, and the directory caches, consistent across cores. The focus of this thesis is on implementing a shared file system and shared processes; Hare's network stack is described in a separate technical report [3].

3.1 Why a New Design?

One might think that the above challenge is straightforwardly met by viewing the multikernel as a distributed system, and running a distributed file system on the multikernel operating system. In fact, early multikernel designs advocated this solution and supported it [2]. However, this approach fails to support many important applications because it does not provide single-system-image semantics across the cores.

Consider a situation where a file descriptor is shared between a parent and child

```
1 dup2(open("file", ...), 1);
2
3 if(fork() == 0) //child
4   exec("echo", ["echo", "hello"]);
5
6 wait(); //parent
7 printf("world.\n");
```

code/fd_share.c

Figure 3-1: Example of file descriptor sharing not supported by NFS.

> fork(): [...] Each of the child's file descriptors shall refer to the same open file description with the corresponding file descriptor of the parent.
>
> exec(): [...] File descriptors open in the calling process image shall remain open in the new process image [...]

Figure 3-2: Excerpt from POSIX API specification [13] regarding sharing file descriptors across `fork()` and `exec()` calls.

process as provided in Figure 3-1. This idiom appears in many applications such as extracting a compressed file using `tar` or configuring a build system using `autoconf`. According the POSIX API specification [13] (as provided by Figure 3-2), the underlying file description should be shared. This means that aspects related to the file descriptor such as the file offset should remain consistent between the two processes. However, since there is no mechanism for NFS clients to share file descriptions, applications using this idiom are limited to a single core. In the example provided, the file contents should be the text "*hello*\n*world.*\n", however if the offset is not consistent between the two processes then the parent's write could shadow the child's.

Networked file systems typically lack support for accessing unlinked files through already-opened file descriptors, especially if the file is open on one machine and is unlinked on another machine. Consider the situation where one process opens a file while another process writes to and then removes that file. An example of such behavior is provided in Figure 3-3. This situation arises during a typical compilation process. According to the POSIX [13] specification (provided in Figure 3-4), the file data should remain valid for the original process that opened the file. Networked file

```c
fd = open("d/file", ...);
// script runs
ret = read(fd, buf, 1024);
```

code/orphan.c

```
#!/bin/sh
echo "hello world" >> d/file
rm -r d
```

code/orphan.sh

Figure 3-3: Example of orphan file use not supported by NFS.

> If one or more processes have the file open when the last link is removed, the link shall be removed before unlink() returns, but the removal of the file contents shall be postponed until all references to the file are closed.

Figure 3-4: Excerpt from POSIX API specification [13] regarding orphan files after `unlink()` calls.

systems typically do not support this idiom, as they cannot rely on client machines to remain online and reliably close all outstanding open files. This is due to the fact that client machines are not trusted and may crash without notifying the server.

To avoid these limitations, and to provide a single system image, Hare leverages several properties of the target architecture. First, Hare runs with a single failure domain between the file servers, applications and client libraries. This is reasonable to assume because both the clients and the servers are part of the same machine's operating system. Second, Hare leverages fast and reliable message delivery between cores. Third, Hare exploits the presence of non-cache-coherent shared memory for efficient access to bulk shared data.

3.2 Overview

Figure 3-5 illustrates Hare's overall design. Hare consists of a set of server and application processes running on a multikernel operating system. Each core runs an independent kernel. The kernels and user-level processes communicate using message passing. The server processes implement most of Hare, but each kernel has a Hare client library that performs some operations, such as accessing the buffer cache directly. To protect the buffer cache, the client library runs inside its local kernel. All

the kernels trust the other kernels to enforce this protection.

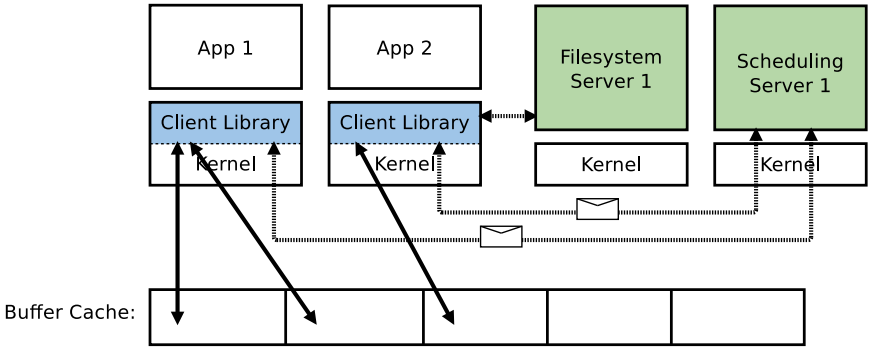

Figure 3-5: Hare design.

Hare's design does not support a persistent on-disk file system, and instead provides an in-memory file system. We made this choice because implementing a scalable in-memory file system is the most challenging aspect of designing a multikernel file system, since it requires dealing with shared state across cores. We are instead focused on the steady-state behavior of the system after blocks have been read from disk. Any operations that read or write data on disk would be orders of magnitude slower than in-memory accesses, and could be serialized by processors that are closest to the disk I/O controllers.

The scheduling server is responsible for spawning new processes on its local core, waiting for these processes to exit, and returning their exit status back to their original parents. Additionally, the scheduling server is responsible for propagating signals between the child process and the original parent. To this end, the scheduling server maintains a mapping between the channels for the original process that called `exec()` and the new child process that has been spawned locally.

Figure 3-6 shows the data structures used by Hare's file system. The file server processes maintain file system state and perform operations on file system metadata. The data structures that comprise the file system are either distributed or partitioned between the file servers. Each client library keeps track of which server to contact in order to perform operations on files, directories, or open file descriptors. For example, to open a file, the client library needs to know both the file's inode number, and the server storing that inode. The client library obtains both the inode number and the

server ID from the directory entry corresponding to the file. A designated server stores the root directory entry.

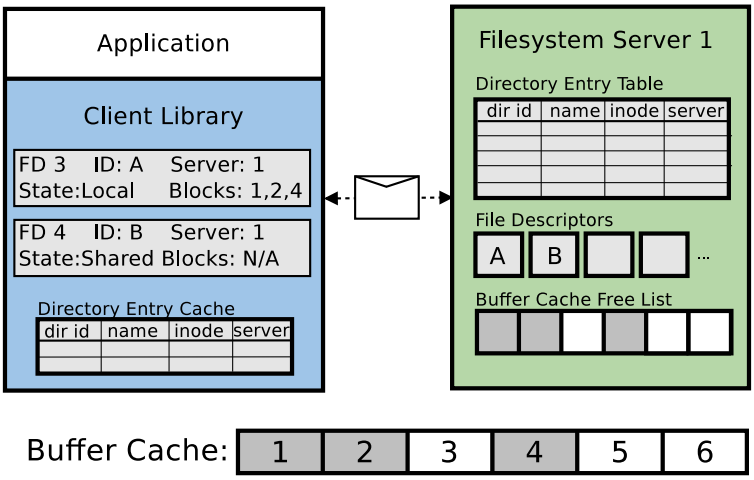

Figure 3-6: Data structures used by Hare's file system.

In the rest of this chapter, we will discuss Hare's file system and process management in more detail, focusing on how shared state is managed in Hare.

3.3 File Data

The buffer cache stores file blocks, but not file metadata. The buffer cache is divided into blocks which file servers allocate to files on demand. Each server maintains a list of free buffer cache blocks; free blocks are partitioned among all of the file servers in the system. When a file requires more blocks, the server allocates them from its local free list; if the server is out of free blocks, it can steal from other file servers (although stealing is not implemented in our prototype).

The client library uses shared-memory addresses to directly read and write blocks in the buffer cache. If an application process opens a file, it traps into its local kernel, and its client library sends a message to the file server in charge of that file. If the standard POSIX permission checks pass, the server responds to the client library with the block-list associated with that file. When an application invokes a `read()` or `write()` system call, the application's local client library reads and writes the buffer

cache directly provided the blocks are known, otherwise it requests the associated blocks before performing the operation.

The challenge in accessing the shared buffer cache from multiple cores lies in the fact that each core has a non-coherent private cache. As a result, if the application on core 1 writes to a file, and then an application on core 2 reads the same file, the file's data might be in core 1's private cache, or even if it was flushed to DRAM, core 2's cache could still have a stale copy.

3.3.1 Solution: Invalidation and Writeback Protocol

To address this problem, Hare performs explicit invalidation and writeback. To avoid having to invalidate and writeback cache data at every file read and write, Hare employs a weaker consistency model, namely, close-to-open consistency [12, 19]. When an application first opens a file, the client library invalidates the local processor's cache for the blocks of that file, since they may have been modified by another core. When an application closes the file descriptor, or calls `fsync()` on it, its client library forces a writeback for any dirty blocks of that file in the local processor cache to shared DRAM. This solution ensures that when a file is opened on any core, the application will observe the latest changes to that file since the last close.

Although close-to-open semantics do not require Hare to ensure data consistency in the face of concurrent file operations, Hare must make sure its own data structures are not corrupted when multiple cores manipulate the same file. In particular, if one core is writing to a file and another core truncates that file, reusing the file's buffer cache blocks can lead to data corruption in an unrelated file, because the client library on the first core may still write to these buffer cache blocks. To prevent this, Hare defers buffer cache block reuse until all file descriptors to the truncated file have been closed, as we describe in Section 3.5.

Note that networked distributed file systems don't have the option of having a shared buffer cache. Thus, if several processes on different machine on a network read the same file, the blocks of the file are replicated in the memory of several machines. Adopting the same design on a single shared memory system would waste DRAM, by

storing multiple copies of the same popular data. Hare ensures that each file block is stored only once in shared DRAM.

3.4 Directories

Parallel applications often create files in a shared directory. To avoid contention between operations on different files in the same directory, Hare allows the application to create a *distributed directory* by using a flag during directory creation time. Hare then distributes the entries for this directory across the file system servers. When an application creates a file in the distributed directory *dir*, the client library determines which server to contact to add the directory entry using the following hash function:

$$\texttt{hash}(dir,\ name)\ \%\ \texttt{NSERVERS}\ \rightarrow\ server\_id \tag{3.1}$$

where *name* is the name of the file or directory in the directory *dir*. To avoid re-hashing directory entries when their parent directory is renamed, Hare uses an inode number to identify each directory (and file) in the file system, which does not change when it is renamed. In the above hash computation, *dir* refers to the inode number of the parent directory.

This hashing function ensures that directory entries of distributed directories are evenly spread across all file servers. This allows applications to perform multiple operations (e.g., creating files, destroying inodes, and adding and removing directory entries) on a distributed directory in parallel, as long as the file names hash to different servers.

In the current design, the number of servers (`NSERVERS`) is a constant. As we will show in Chapter 5, it can be worthwhile to dynamically change the number of servers to achieve better performance. However, the optimal number of servers is dependent on the application workload.

One complication with distributing directories arises during an `rmdir()` operation, which must atomically remove the directory, but only if it is empty. Since the directory

entries of a distributed directory are spread across multiple file servers, performing an `rmdir()` operation on a distributed directory can race with another application creating a file in the same directory.

3.4.1 Solution: Three-phase Commit Protocol

To prevent the race between an `rmdir()` operation and file creation in that directory, Hare implements `rmdir()` using a three-phase commit protocol. The core of this protocol is the standard two-phase commit protocol. The client library performing `rmdir()` first needs to ensure that the directory is empty; to do so, it sends a message to all file servers, asking them to mark the directory for deletion, which succeeds if there are no remaining directory entries. If all servers succeed, the client library sends out a `COMMIT` message, causing the servers to delete that directory. If any server indicates that the directory is not empty, the client library sends an `ABORT` message to the servers, which removes the deletion mark on the directory. While the directory is marked for deletion, file creation and other directory operations are delayed until the server receives a `COMMIT` or `ABORT` message. The last complication with this protocol arises from concurrent `rmdir()` operations on the same directory. If the concurrent operations contact the file servers in a different order, there is a potential for deadlock. To avoid this potential deadlock scenario, Hare introduces a third phase, before the above two phases, where the client library initially contacts the directory's home server (which stores the directory's inode) to serialize all `rmdir()` operations for that directory.

Note that Hare handles directory entries differently from most networked file systems. In most distributed file systems, all entries in a directory are stored at a single server, because the round-trip time to the server is high, and it is often worthwhile to download the entire directory contents in one round-trip. Distributing directory entries across servers in a networked file system would also require an atomic commit protocol for `rmdir()` operations, which is costly and requires a highly available coordinator for the protocol.

3.5 File Descriptors

File descriptors are used to keep track of the read/write offset for an open file descriptor, which poses a challenge for Hare when several processes share a file descriptor. In a cache-coherent system, it is simply a matter of storing the file descriptor data structure with its associated lock in shared-memory to coordinate updates. Without cache coherency, though, Hare needs a mechanism to guarantee that the shared file descriptor information remains consistent. For example, suppose a process calls `open()` on a file and receives a file descriptor, then calls `fork()` to create a child process. At this point `write()` or `read()` calls must update the offset for the shared file descriptor in both processes.

A second scenario which poses a challenge for Hare is related to unlinked files. According to the POSIX API specification, a process can read and write to a file through an open file descriptor after file has been unlinked.

3.5.1 Solution: Hybrid File Descriptor Tracking

To solve this problem, Hare stores some file descriptor state at the file servers. For each open file, the server responsible for that file's inode tracks the open file descriptors and an associated reference count. The file server ensures that when a file is unlinked, the inode and corresponding file data will remain valid until the last file descriptor for that file is closed.

The file descriptor's offset is sometimes stored in the client library, and sometimes stored on the file server, for performance considerations. When the file descriptor is not shared between processes ("local" state), the client library maintains the file descriptor offset, and can perform read and write operations without contacting the file server. On the other hand, if multiple processes share a file descriptor ("shared" state), the offset is migrated to the file server, and all `read()` and `write()` operations go through the server, to ensure consistency. The file descriptor changes from *local* to *shared* state when a process forks and sends a message to the server to increment the reference count; it changes back to *local* state when the reference count at the

server decreases to one. Although this technique could present a potential bottleneck, sharing of file descriptors is typically limited to a small number of processes for most applications.

Traditional distributed file systems, including NFS and AFS, have trouble handling this scenario, especially if the client accessing the file runs on a different node than the client performing the unlink, because it's impractical to track all open files, especially in the face of client node failures. Hare is able to address this problem because all client libraries are trusted to inform the server when file descriptors are closed (thereby avoiding leaking of unlinked files) , and communication for each file open and close is relatively cheaper.

3.6 Processes

In order to take advantage of many cores provided by hardware, Hare applications must be able to spawn processes on those cores. A scheduler in a traditional shared memory operating system can simply steal processes from another core's run queue. However, in Hare, it is difficult to migrate a process from one core to another through stealing as each core has an independent kernel with its own data structures, memory allocators, etc. As a consequence, migrating a running process would require the two kernels to carefully coordinate hand-off for all data structures associated with that process.

3.6.1 Solution: Remote Execution Protocol

Hare's insight is that `exec()` provides a narrow point at which it is easy to migrate a process to another core. In particular, the entire state of the process at the time it invokes `exec()` is summarized by the arguments to `exec()` and the calling process's open file descriptors. To take advantage of this, Hare can implement the `exec()` call as an RPC to a scheduler running on another core, so that the process finishes the `exec()` call on that core before resuming execution.

Each core runs a scheduling server, which listens for RPCs to perform `exec()`

operations. When a process calls `exec()`, the client library implements a scheduling policy for deciding which core to choose for running the new process image. Our prototype supports both a *random* and a *round-robin* policy, with round-robin state propagated from parent to child, which proves to be sufficient for our applications and benchmarks. After choosing a destination core, the client library sends the arguments, file descriptor information, and process environment to the new core's scheduling server. The scheduling server then starts a new process on the destination core (by forking itself), configures the new process based on the RPC's arguments, and calls `exec()` in the local kernel to load the target process image.

Running a process on another core creates three challenges. First, when the process exits, the parent on the original core needs to be informed. Second, signals need to be propagated between the new and original core. Third, the process might have had some local file descriptors (e.g., to the console or other file descriptors specific to that core's kernel) that are not valid in the new core's kernel.

To address this challenge, Hare uses a *proxy* process. The original process that called `exec()` turns into a proxy once it sends the RPC to the scheduling server. The scheduling server will, in turn, wait for the new process to terminate; if it does, it will send a message back to the proxy, causing the proxy to exit, and thereby providing the exit status to the proxy's parent process. If the process running on the new core tries to access any file descriptors that were specific to its original core's kernel, the accesses are turned into messages back to the proxy process, which performs the operation where the file descriptor is valid. Finally, if the proxy process receives any signals, it relays them to the new process through the associated scheduling server.

3.7 Techniques and Optimizations

Hare implements several optimizations to improve performance, for which an evaluation of performance is provided in Chapter 5.

3.7.1 Directory lookup and caching

Hare caches the results of directory lookups, because lookups involve one RPC per pathname component, and lookups are frequent. Pathname lookups proceed iteratively, issuing the following RPC to each directory server in turn:

$$\texttt{lookup}(dir,\ name) \rightarrow \langle server,\ inode \rangle \tag{3.2}$$

where *dir* and *inode* are inode numbers, *name* is the file name being looked up, and *server* is the ID of the server storing *name*'s *inode*. The file server returns both the inode number and the server ID, as inodes do not identify the server; each directory entry in Hare therefore stores both the inode number and the server of the file or directory.

Hare must ensure that it does *not* use stale directory cache entries. Which may arise when a file is renamed or removed. To do this, Hare relies on file servers to send invalidations to client libraries during these operations, much like callbacks in AFS [12]. The file server tracks the client libraries that have a particular name cached; a client library is added to the file server's tracking list when it performs a lookup RPC or creates a directory entry. During a remove, rmdir or unlink operation, the invalidation message is sent to all client libraries which have cached that lookup.

The key challenge in achieving good performance with invalidations is to avoid the latency of invalidation callbacks. In a distributed system, the server has to wait for clients to acknowledge the invalidation; otherwise, the invalidation may arrive much later, and in the meantime, the client's cache will be inconsistent.

3.7.2 Solution: Atomic message delivery

To address this challenge, Hare relies on an *atomic message delivery* property from its messaging layer. In particular, when the `send()` function completes (which delivers a message), the message is guaranteed to be present in the receiver's queue. To take advantage of this property, Hare's directory lookup function first checks the incoming queue for invalidation messages, and processes all invalidations before performing a

lookup using the cache. This allows the server to proceed as soon as it has sent invalidations to all outstanding clients (i.e., `send()` returned), without waiting for an acknowledgment from the client libraries.

3.7.3 Directory broadcast

As described in Section 3.4, the hash function distributes directory entries across several servers to allow applications to perform directory operations on a shared directory in parallel. However, some operations like `readdir()` must contact all servers. To speed up the execution of such operations, Hare's client libraries contact *all* directory servers in parallel. This enables a single client to overlap the RPC latency, and to take advantage of multiple file servers that can execute the corresponding readdir operation in parallel, even for a single `readdir()` call.

3.7.4 Message coalescing

As the file system is distributed among multiple servers, a single operation may involve several messages (e.g. an `open()` call may need to create an inode, add a directory entry, as well as open a file descriptor pointing to the file). When multiple messages need to be sent to the same server for an operation, the messages are coalesced into a single message. In particular, Hare often places the file descriptor on the server that is storing the file inode, in order to coalesce file descriptor and file metadata RPCs. This technique can overlap multiple RPCs, which reduces latency and improves performance.

3.7.5 Creation affinity

Modern multicore processors have NUMA characteristics [2]. Therefore, Hare uses *Creation Affinity* heuristics when creating a file: when an application creates a file, the local client library will choose a close-by server to store that file. If Hare is creating a file, and the directory entry maps to a nearby server (on the same socket), Hare will place the file's inode on that same server. If the directory entry maps to a server

on another socket, Hare will choose a file server on the local socket to store the file's inode. Each client library has a designated local server it uses in this situation, to avoid all clients storing files on the same local server. These heuristics allow the client library to reduce the latency required to perform file operations. Creation Affinity requires client libraries to know the latencies for communicating with each of the servers, which can be measured at boot time.

Chapter 4

Implementation

Hare's implementation follows the design depicted in Figure 3-5. The kernel that Hare runs on top of is Linux, which provides support for local system calls which do not require sharing. Hare interposes on system calls using the `linux-gate.so` mechanism [20] to intercept the application's system calls and determine whether the call should be handled by Hare or passed on to the local kernel. Hare implements most of the system calls required for file system operations, as well as several for spawning child processes and managing pipes. A list of supported system calls is provided in Table 4.1. The RPC messages that the client library uses to provide these system calls is provided in Table 4.2. By using Linux for the per-core kernel, we are able to obtain a more direct comparison of relative performance as well as leverage an existing implementation for per-core functionality such as the timesharing scheduler and interrupt management. We have not placed the client library into the kernel since it complicates the development environment, although it would allow multiple processes on the same core to share the directory lookup cache.

Hare does not rely on the underlying kernel for any state sharing between cores, and instead implements all cross-core communication and synchronization through its own message passing layer [3], which uses polling, and through the buffer cache. The buffer cache is statically partitioned among the servers, and totals 2 GB in our setup which proves to be a sufficient size to hold all file data across our experiments. the applications as well as the server processes are each pinned to a core in order to limit

Category	System calls
Directory operations	access, creat, getdents, getdents64, mkdir, mkdirat, open, openat, readlink, rename, rmdir, symlink, symlinkat, unlink, unlinkat
File descriptor operations	close, dup, dup2, dup3, _llseek
Process operations	chdir, clone, execve, exit_group, fchdir, getcwd, pipe, socketcall
File data	fdatasync, ftruncate, mmap2, munmap, read, truncate, truncate64, write
File metadata	chmod, chown, faccessat, fchmod, fchmodat, fchown, fchown32, fchownat, fcntl, fcntl64, fsetxattr, fstat, fstat64, fstatat64, fsync, ioctl, lchown, lchown32, lstat, lstat64, stat, stat64, utime, utimensat

Table 4.1: List of system calls handled by Hare's client library.

Category	RPCs
Directory operations	MKDIR, RMDIR, DIRENT, ADD_MAP, RM_MAP, RESOLVE, SYMLINK, READLINK
File descriptor operations	OPEN, DUP, LSEEK, CLOSE
Process operations	CONNECT, DISCONNECT, EXEC
File data	CREATE, DESTROY, READ, WRITE, TRUNC, BLOCKS, FLUSH
File metadata	CHMOD, CHOWN, FSTAT, UTIMENS

Table 4.2: List of Hare RPCs. Note that CONNECT and DISCONNECT are implicit through the messaging library. ADD_MAP and RM_MAP add and remove directory entries.

any unintended use of shared memory. Hare uses a modification to the Linux kernel to provide PCID support. The PCID feature on the Intel architecture allows the TLB to be colored based on process identifiers, which allows the kernel to avoid flushing the TLB during a context switch. This results in lower context switch overhead which can result in faster messaging when a server is sharing a core with the application.

Using Linux as a multikernel has simplified the development of Hare: we can use debugging tools such as `gdb`, can copy files between Hare and Linux, and run applications without building a custom loader. Hare supports reading and writing files that live on the host OS by redirecting file system calls for a few directories to Linux. A client library additionally maintains a mapping of file descriptors that correspond to either the Hare or the underlying host kernel (e.g. console, sockets, virtual file systems, etc) to disambiguate system calls made on these file descriptors.

Lines of code for various portions of the system are provided in Figure 4-1; Hare is implemented in C/C++.

Component	**Approx. SLOC**
Messaging	1,536
Syscall Interception	2,542
Client Library	2,607
File System Server	5,960
Scheduling	930
Total	13,575

Figure 4-1: SLOC breakdown for Hare components.

Chapter 5

Evaluation

This chapter evaluates the performance of Hare across several workloads to answer the following questions. First, what POSIX applications can Hare support? Second, what is the performance of Hare? Third, how important are Hare's techniques to overall performance? Fourth, can Hare's design show benefits on machines with cache coherence?

5.1 Experimental Setup

All of the results in this chapter are gathered on a PowerEdge 910 server containing four Intel Xeon E7-4850 10-core processors, for a total of 40 cores. This machine provides cache-coherent shared memory in hardware, which enables us to answer the last evaluation question, although Hare does not take advantage of cache coherence (other than for implementing message passing). The machines run Ubuntu Server 13.04 i686 with Linux kernel 3.5.0. All experiments start with an empty directory cache in all client libraries.

5.2 POSIX Applications

One of Hare's main goals is to support POSIX style applications, therefore it is important to consider which applications can run on the system. Although Hare does

not support threads and `fork()` calls must run locally, Hare can still run a variety of applications with little to no modifications. Some of these are benchmarks designed to stress a portion of the system, while others employ a broader range of operations. All of the applications can run on Hare as well as Linux unmodified. Several of the applications were modified slightly to control the sharing policy of directories (i.e. to distribute shared directories) or to use `exec()` in addition to `fork()` calls to allow migration of processes across cores. Additionally a placement policy was chosen for each of the applications (random placement for *build linux* and *punzip* and round-robin for the remainder of the workloads), though the placement policy is decided within the Hare client library and therefore does not require modifying the application.

The benchmarks shown in Table 5.1 perform a wide range of file system operations; Figure 5-1 shows the breakdown of file system operations, as a percentage of the total, for each of the benchmarks. This shows the wide variety of workloads present in Hare's benchmarks. Benchmarks such as *mailbench*, *fsstress* issue many different operations, while some of the microbenchmarks such as *renames* and *directories* are limited to just a few operations and are designed to stress a specific portion of the system.

In Table 5.2, the requirements and operation count for each of the workloads used to evaluate the system is depicted. From this table, it is clear that supporting sharing of various abstractions across cores is required to support a wide range of workloads. Additionally from the high number of operations issued one can gather that these tests significantly stress the system indicating the stability of the prototype.

From Figure 5-1, it is clear that the breakdown of operations is significantly different across the various benchmarks. In addition to the breakdown provided, it is also important to note the number of operations being issued and various ways that the workloads access files and directories varies. For instance the larger benchmark *build linux* issues on the order of $1.3M$ operations with the other tests issuing tens to hundreds of thousand operations. Tests such as *extract*, *punzip* and *build linux* make use of pipes and the *make* system used to build the kernel relies on a shared pipe, implemented in Hare, in order to coordinate its job server. The broad range of appli-

Application	Description
creates	Each process creates a file, via `open()` followed by `close()`, and then removes the file, repeating the process 65535 times.
writes	A benchmark that performs 10 writes per file, at 4096 bytes per write (block-size of the buffer cache) for 65535 iterations. In this benchmark, the processes perform the writes on files in the same directory.
renames	Each process creates a file and then moves that file for $56k$ iterations in the same parent directory.
directories	Each process creates and then removes a unique directory $128k$ times.
rm (dense)	Removal of a dense directory tree that contains 2 top-level directories and 3 sub-levels with 10 directories and 2000 files per sub-level.
rm (sparse)	Removal of a sparse directory tree that contains 1 top-level directory and 14 sub-levels of directories with 2 subdirectories per level.
pfind (dense)	A parallel find executed on the dense file tree described above. Each process walks the tree in a breadth first manner, randomizing the order as it recurses to avoid any herding effects.
pfind (sparse)	A parallel find executed on the sparse file tree describe above.
extract	A sequential decompression of the Linux 3.0 kernel .
punzip	Parallel unzip which unzips 20 copies of the manpages on the machine. Each unzip process is given 1000 files to extract at once.
mailbench	A mail server from the sv6 operating system [5, 7]. This application creates a few directories and passes messages through temporary files to the queue manager. A few slight modifications were made to the application to avoid `exec()` calls and some unnecessary `fork()` calls. This increases the performance of the benchmark on both Linux and Hare, as well as further stressing the file system.
fsstress	A synthetic file service benchmark originally used to test the performance of NFS. The application repeatedly chooses a file system operation at random and executes it. The number of operations is set to 65525.
build linux	A parallel build of the Linux 3.0 kernel.

Table 5.1: Applications and microbenchmarks used to evaluate Hare's performance.

Application	Shared files	Shared dirs	Shared fds	Shared pipes	#ops
creates	√	√	×	×	9M × n
writes	×	√	×	×	1.4M × n
renames	×	√	×	×	6M × n
directories	×	√	×	×	5.7M × n
pfind (dense)	×	√	×	×	2.6M × n
pfind (sparse)	×	×	×	×	300k × n
rm (dense)	×	√	×	×	1M
rm (sparse)	×	×	×	×	40K
extract	√	×	√	×	400K
punzip	√	√	√	×	300K
mailbench	√	√	×	×	700K × n
fsstress	×	×	×	×	350K × n
build linux	√	√	√	√	1.3M

Table 5.2: Sharing properties of workloads used to evaluate Hare's performance. n represents the level of parallelism for the workload.

cations and the various ways in which they access the system demonstrate that Hare supports a full featured API which can support a variety of real world applications.

5.3 Performance

To understand Hare's performance, we evaluate Hare's scalability, compare timesharing and dedicated-core configurations, and measure Hare's sequential performance.

5.3.1 Scalability

To evaluate Hare's scalability, we measure the speedup that the benchmarks achieve when running on a different total number of cores. We use a single -core Hare configuration as the baseline and increase the parallelism of the test as well as the number of Hare servers up to the maximum number of cores in the system. Figure 5-2 shows the results.

The results demonstrate that for many applications, Hare scales well with an increasing number of cores. As seen in the figure, the benchmark that shows the

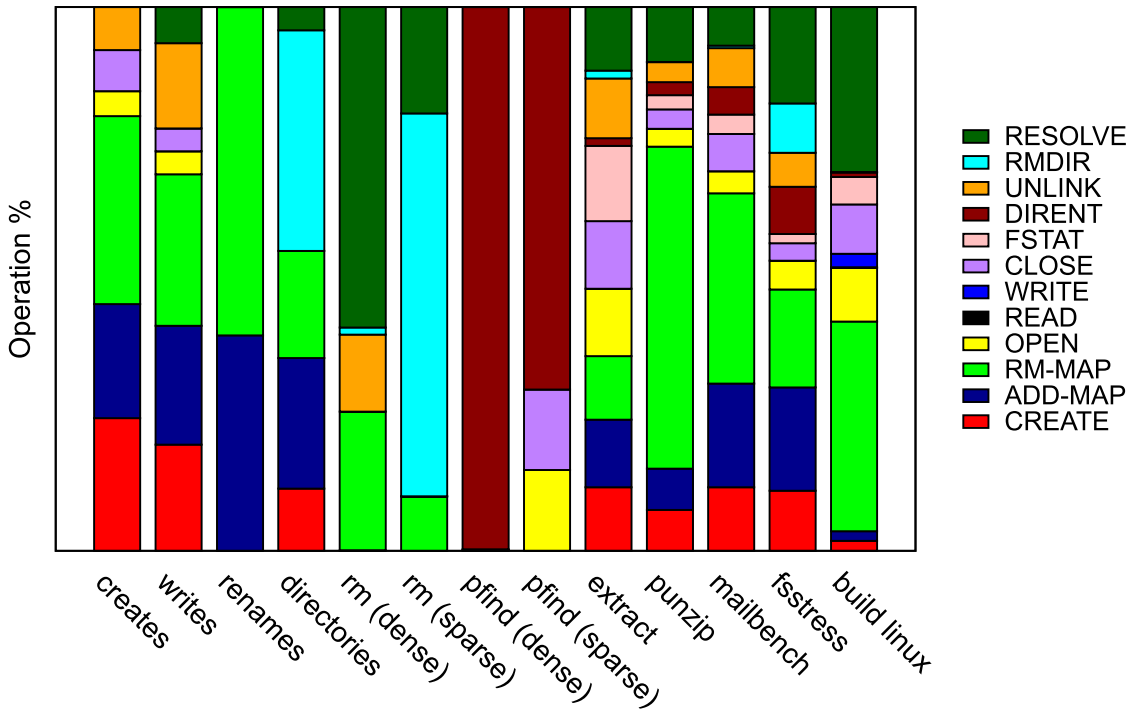

Figure 5-1: Operation breakdown per application / benchmark.

least scalability is the *pfind sparse* benchmark. In this test the client applications are recursively listing directories in a sparse tree. As the directories are not distributed in this test and there are relatively few subdirectories, each of the clients will contact the servers in the same order. This results in a bottleneck at individual servers due to a herding effect because all n clients will walk the directory tree in the same order. The remaining tests, on the other hand, show good scalability up to 40 cores and promise of further scalability to higher core counts.

5.3.2 Split Configuration

In addition to running the filesystem server on all cores, Hare may also be configured to isolate several cores that can be devoted to the filesystem server while the remaining cores run the application and scheduling server. As will be described in Section 5.3.3, there are some performance penalties associated with timesharing cores between servers and applications, though the split-configuration limits the number of

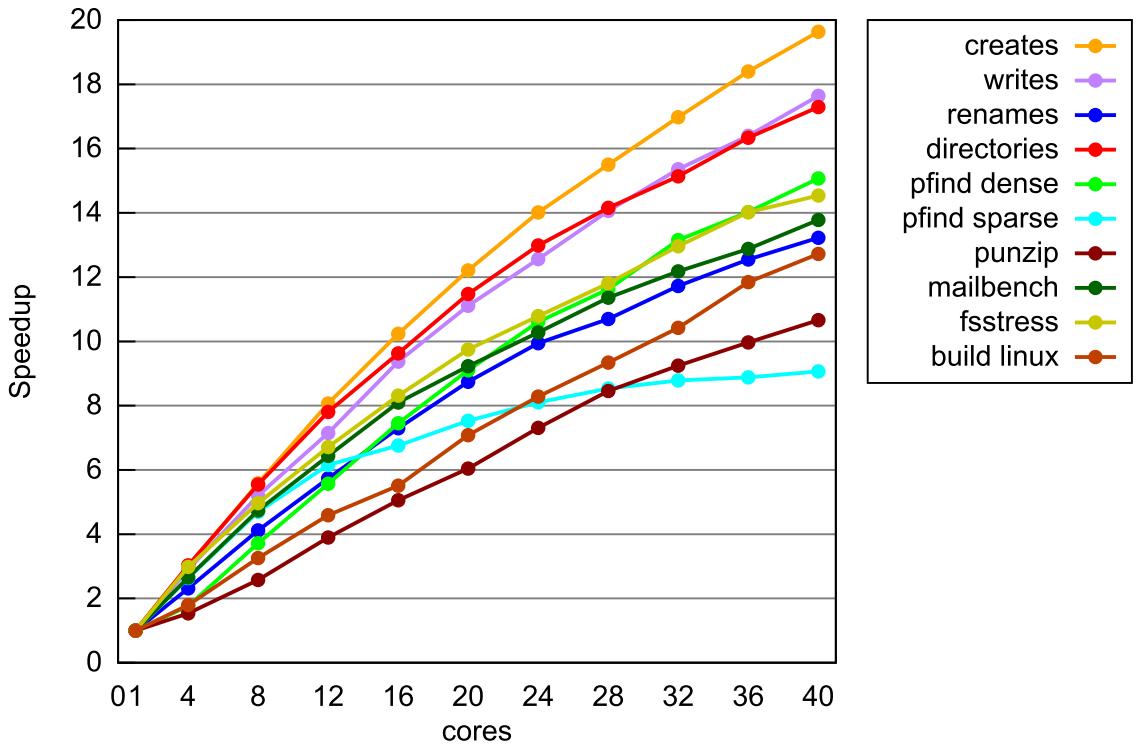

Figure 5-2: Speedup of benchmarks as more cores are added, relative to their throughput when using a single core.

cores than an application may scale to.

Figure 5-3 presents the performance of Hare in the following configurations: running the server and application on all 40 cores (timeshare), running the server on 20 cores and the application on the remaining 20 cores (20/20 split) and finally choosing the optimal split for all 40 cores between application cores and filesystem server cores (best). The optimal number of servers is presented above the bar for the best configurations, and is determined by running the experiment in all possible configurations and picking the best performing configuration. Each of the configurations is normalized against the timeshare configuration which was used for the scalability results presented in Figure 5-2.

From these results, it is clear that Hare can achieve better performance if the optimal number of servers is known a priori. The results also show, however, that the optimal number of servers is highly dependent on the application and its specific workload, making it difficult to choose ahead of time. Since choosing the optimal

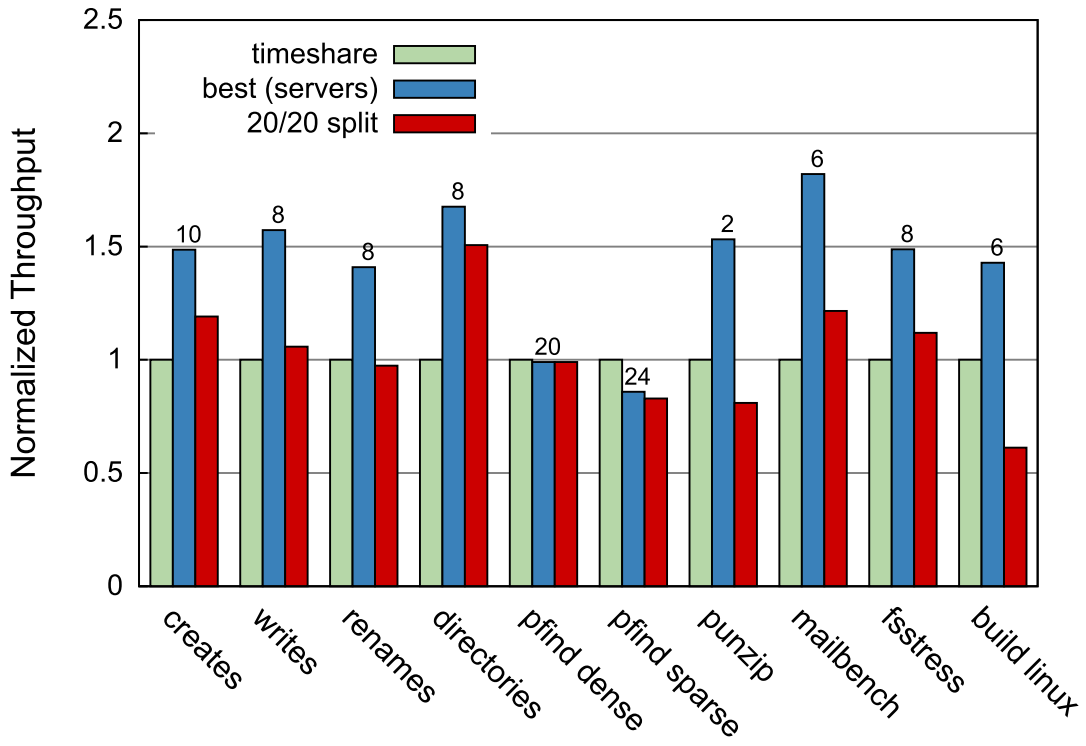

Figure 5-3: Performance of Hare in both split and combined configurations.

number is difficult in practice, we use the timesharing configuration for results presented in this chapter unless otherwise noted. This configuration achieves reasonable performance without per-application fine-tuning and provides a fair comparison for the results presented in Section 5.3.1.

5.3.3 Hare Sequential Performance

Now that it has been established that Hare scales with increasing core counts, it is important to consider the baseline that is used for the scalability results. To evaluate Hare's sequential performance (single-core baseline), we compare it to that of Linux's ramfs running on a small number of cores. In the simplest configuration, Hare may run on a single core, timesharing between the application and the server. Additionally, Hare may be configured to run in a split configuration where application process(es) run alongside the scheduling server on one core while the filesystem server runs on another core. In the split configuration, Hare's performance can be improved

significantly as the time spent performing individual operations is comparable to the cost of a context switch. When running in the split configuration, there are no other processes sharing the core with the filesystem server, and therefore the context switch overhead is eliminated. An additional performance penalty is also incurred due to cache pollution when the application and file server share a core. We note, however, that the split configuration uses twice as many resources to perform the same operations. Lastly, we have seen a noticeable performance penalty from TLB flushes on context switches, however Hare is able to avoid these by using the PCID feature of the processor to color the TLB of applications and servers independently to avoid flushing the TLB during a context switch.

We also compare Hare's performance with that of UNFS3 [1], a user-space NFS server which runs on a single core accessed via the loopback device. This comparison more closely represents a state-of-the-art solution that would be viable on a machine which does not support cache coherence and furthermore exemplifies the setup used in Barrelfish OS [2]. The UNFS3 configuration is run on a single core.

Figure 5-4 shows the performance of our benchmarks in these configurations. These results show that Hare is significantly faster than that of UNFS3 due to the high cost of messaging through the loopback interface. When compared to Linux ramfs, Hare is slower (up to 3.4×), though the ramfs solution is not a viable candidate for architectures which lack cache coherence. Our benchmarks achieve a median throughput of 0.39× when running on Hare when compared to Linux on a single core.

Much of Hare's performance on microbenchmarks comes from the cost of sending RPC messages between the client library and the file server. For example, in the *renames* benchmark, each `rename()` operation translates into two RPC calls: ADD_MAP and RM_MAP, which take 2434 and 1767 cycles respectively when measured from the client library, while only taking 1211 and 756 cycles at the server when run on separate cores. Since no other application cores have looked up this file, the server does not send any directory cache invalidations. As a consequence, the messaging overhead is roughly 1000 cycles per operation. The UNFS3 configuration will suffer a similar overhead from using the loopback device for RPC messages. In either

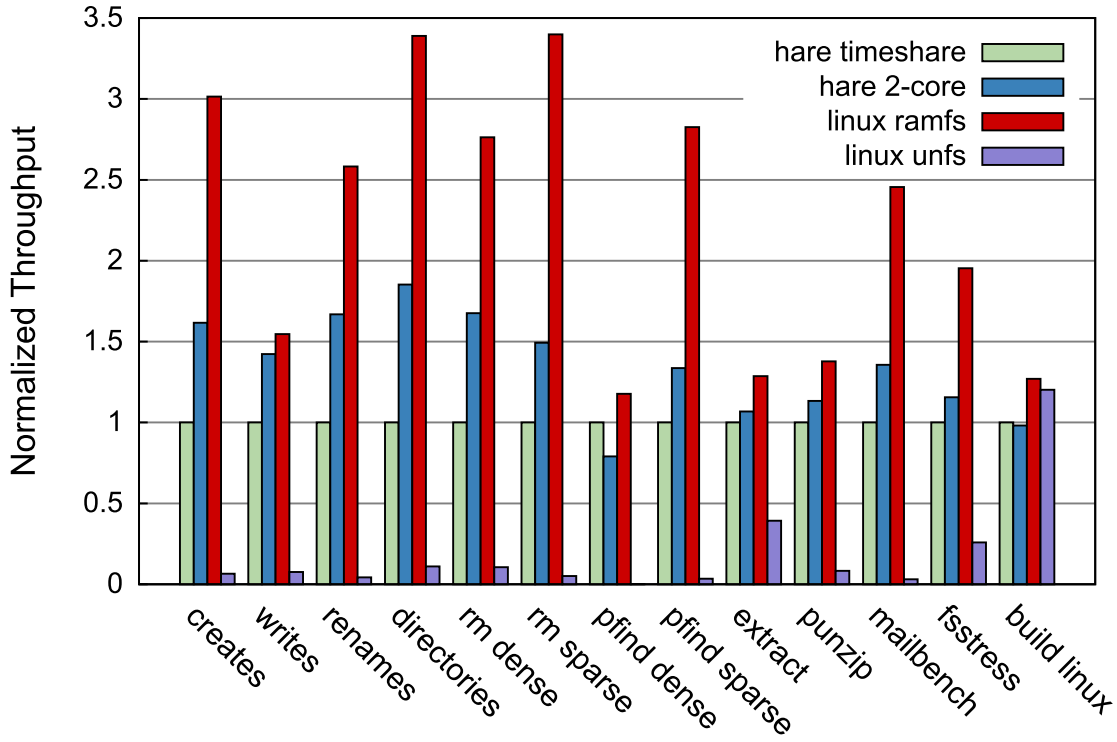

Figure 5-4: Normalized throughput for small number of cores for a variety of applications running on Linux relative to Hare.

case, the performance of a multikernel design is dependent on the performance of the messaging subsystem and hardware support could greatly increase the sequential performance.

In order to determine the component of the system most affected by running on a separate core we evaluate the performance of the `rename()` call across many iterations. When running on a single core the call takes 7.204 μs while running on separate cores the call takes 4.171 μs. Adding timestamp counters to various sections of code reveals an increase of 3.78× and 2.93× for the sending and receiving portion of the operation, respectively. Using *perf* demonstrates that a higher portion of the time is spent in context switching code as well as a higher number of L1-icache misses both contributing to the decrease in performance when running on a single core.

5.4 Technique Importance

To evaluate the importance of individual techniques in Hare's design, we selectively turn each technique off, and compare the performance of all our benchmarks on both this modified system and the full Hare design, running both configurations across all 40 cores.

5.4.1 Directory Distribution

Figure 5-5 shows the effect of directory distribution on Hare's performance. When a directory is distributed, the entries are spread across all servers, however in the common case the entries are all stored at a single server. Directory entries for different directories are always spread across servers. This decision is made by the application programmer by setting a flag at directory creation time or via an environment variable which applies the flag to all directories created by an application. Applications which perform many concurrent operations within the same directory benefit the most from directory distribution and therefore use this flag. The applications which use this flag include *creates*, *renames*, *pfind dense*, *mailbench* and *build linux*.

From the results we can see that applications which exhibit this behavior can benefit greatly from distributing the directory entries across servers as they do not bottleneck on a single server for concurrent operations. Additionally, workloads that involve `readdir()` on a directory which contains many entries (e.g. *pfind dense*) benefit from obtaining the directory listings from several servers in parallel. Conversely, obtaining a directory listing with few entries (e.g. *pfind sparse*) can suffer from distributing directory entries, therefore this benchmark leaves this feature turned off.

On the other hand, `rmdir()` requires the client library to contact all servers to ensure that there are no directory entries in the directory that is to be removed before executing the operation. As a consequence, workloads such as *rm sparse* and *fsstress* which perform many `rmdir()` operations on directories with few children perform worse with directory distribution enabled and likewise run without this feature. This demonstrates that allowing applications to choose whether to use Directory Distribu-

tion on a per-directory basis can achieve better performance.

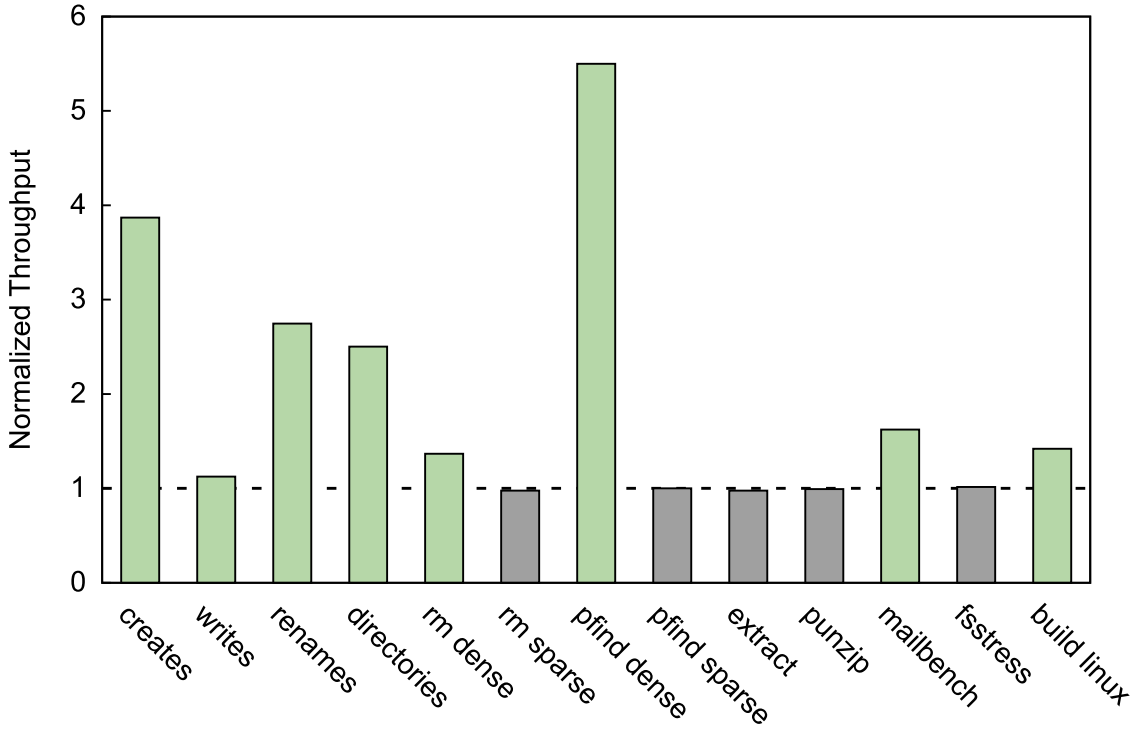

Figure 5-5: Performance of Hare with and without Directory Distribution.

5.4.2 Directory Broadcast

As mentioned above, Directory Distribution can improve performance when several operations are being executed concurrently in the same directory. One drawback to using this technique is that some operations will be required to contact all servers, such as `rmdir()` and `readdir()`. Hare uses Directory Broadcast to send out such operations to all servers in parallel. Figure 5-6 shows the effect of this optimization, compared to a version of Hare that uses sequential RPCs to each file server for directory operations. As expected, benchmarks that perform many directory listings, such as *pfind (dense)* and *pfind (sparse)*, as well as the *directories* test which removes many directories, benefit the most from this technique. On the other hand, directory broadcast can hurt performance only when repeatedly removing a directory that is not empty, as occurs in *fsstress*. However, since each of the *fsstress* processes per-

form operations in different subtrees, *fsstress* turns off Directory Distribution, and therefore directory broadcast is not used.

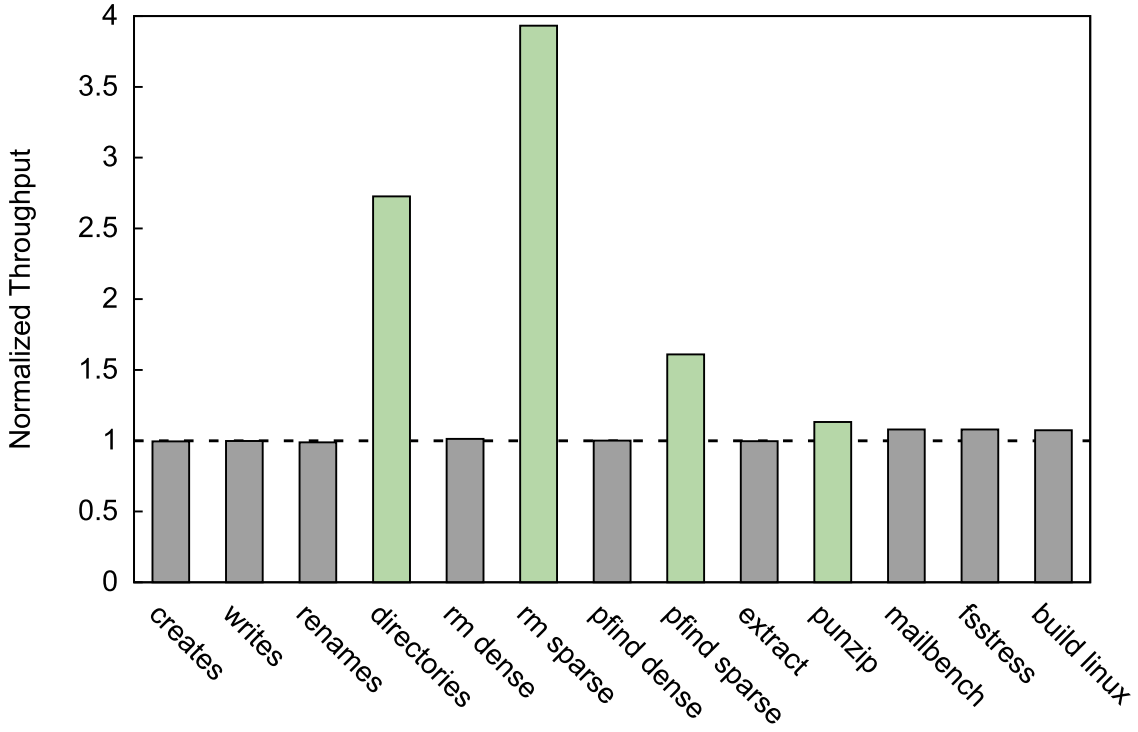

Figure 5-6: Performance of Hare with and without Directory Broadcast.

5.4.3 Direct Access to Buffer Cache

Figure 5-7 shows the performance of Hare compared to a version where the client library does not directly read and write to a shared buffer cache, and instead performs these calls through RPCs to the file server.

The performance advantage provided by directly accessing the buffer cache is most visible in tests which perform a high amount of file i/o operations such as *writes*, *extract*, *punzip* and *build linux*. Direct access to the buffer cache allows the client library to access it independently of the server and other applications, providing better scalability and throughput. Furthermore, it avoids excessive RPC calls which provides a significant performance advantage by alleviating congestion at the file server.

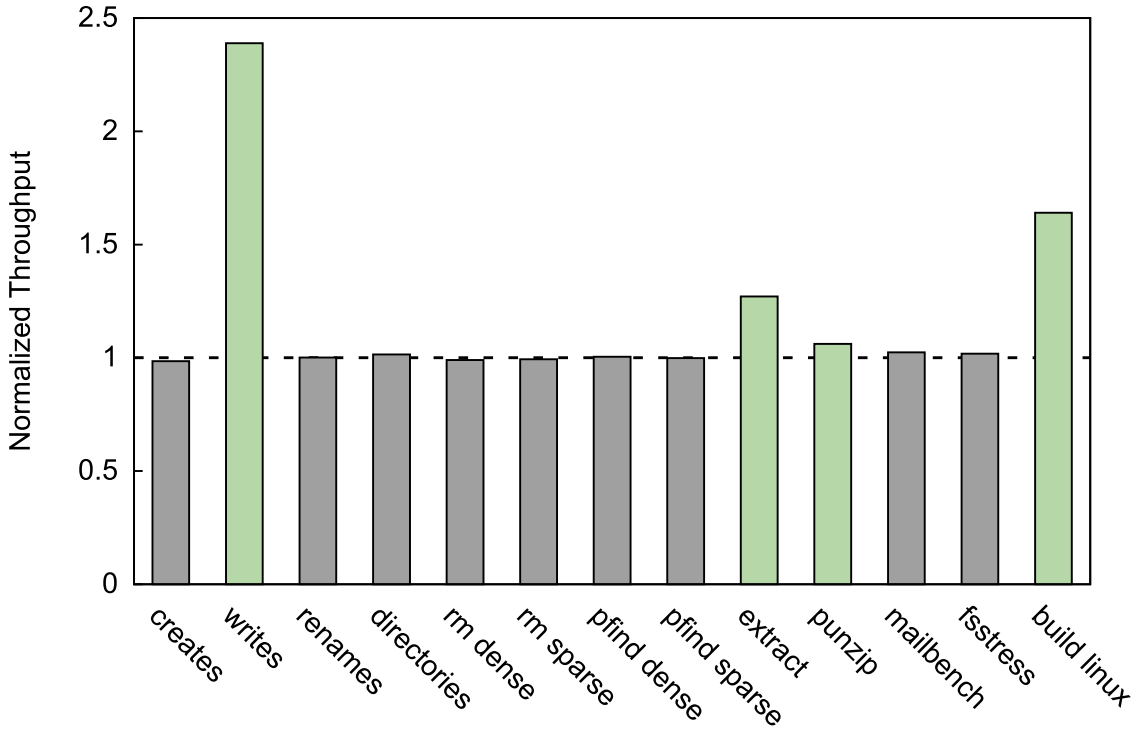

Figure 5-7: Performance of Hare with and without direct access from the client library to the buffer cache.

Another design for the buffer cache could be to use an independent cache on each core rather than a shared buffer cache across all cores. We chose to use a unified buffer cache to reduce capacity misses introduced by sharing the same blocks on multiple cores. To evaluate such effects, the *build linux* test is used as a representative workload as it has a larger working-set size. On this test we found that the number of misses is 2.2× greater when the buffer cache is not shared. As these misses would require loading blocks from disk, this increase can have a significant performance penalty. Additionally, direct access to the buffer cache is never a hindrance to performance and therefore should be used if available.

5.4.4 Directory Cache

Figure 5-8 shows the performance of Hare with and without the directory caching optimization. Many of the benchmarks demonstrate improved performance by caching directory lookups. The benchmarks which benefit the most perform multiple opera-

tions on the same file or directory such as *renames*, *punzip* or *fsstress*. In the case of *mailbench* and *fsstress*, each iteration will be executed on a file in a subdirectory which will require an extra lookup if the directory cache is disabled. The *rm dense* workload experiences a performance penalty with this optimization as it will cache lookups without using them. Overall, considering the performance advantage across all tests, it is advantageous to use this technique.

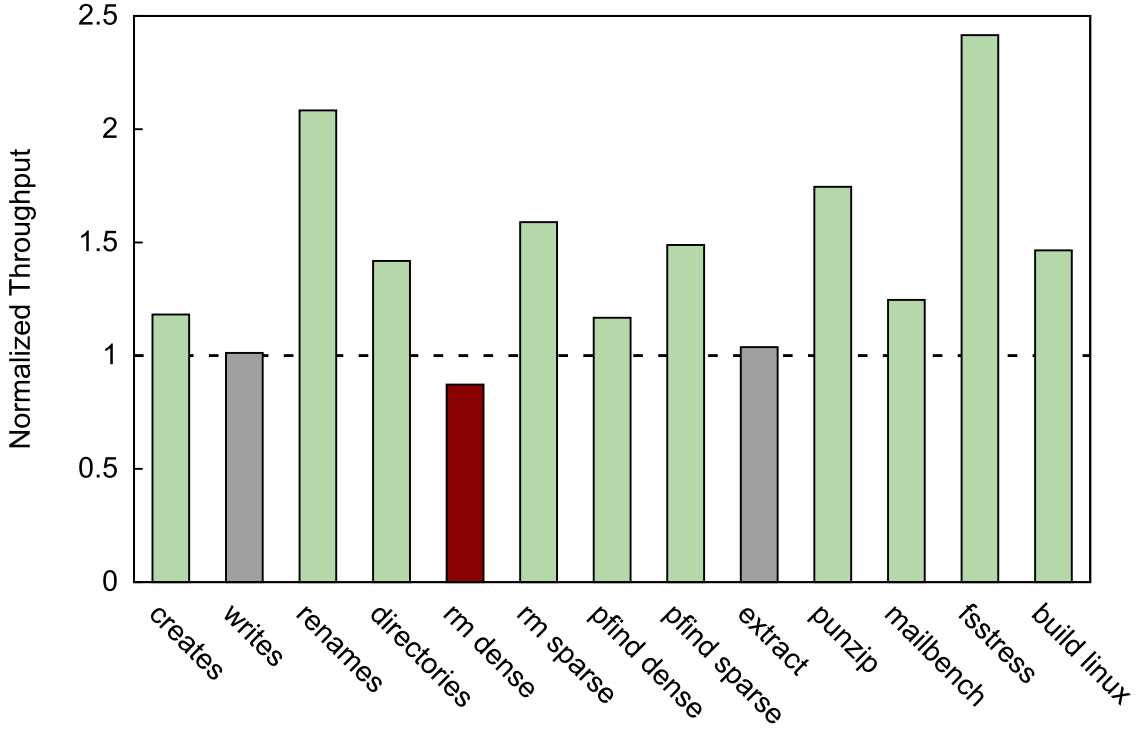

Figure 5-8: Performance of Hare with and without the Directory Cache.

5.5 Hare On Cache Coherent Machines

Figure 5-9 shows the speedup of the parallel tests on both Hare and Linux for 40 cores relative to single-core performance. Some tests scale better on Hare while others scale better on Linux. Although the Linux design is not suitable to run directly on hardware which does not support cache coherence, some of the design points that allow Hare to scale well while running on a machine that does support cache coherent shared memory could potentially be applied to the Linux kernel to improve

performance. Particularly, tests which perform many directory operations in the same parent directory show significantly better scalability on Hare. Traditional shared-memory operating systems could potentially benefit from employing the equivalent of distributed directories to increase performance for applications which perform many operations within the same directory.

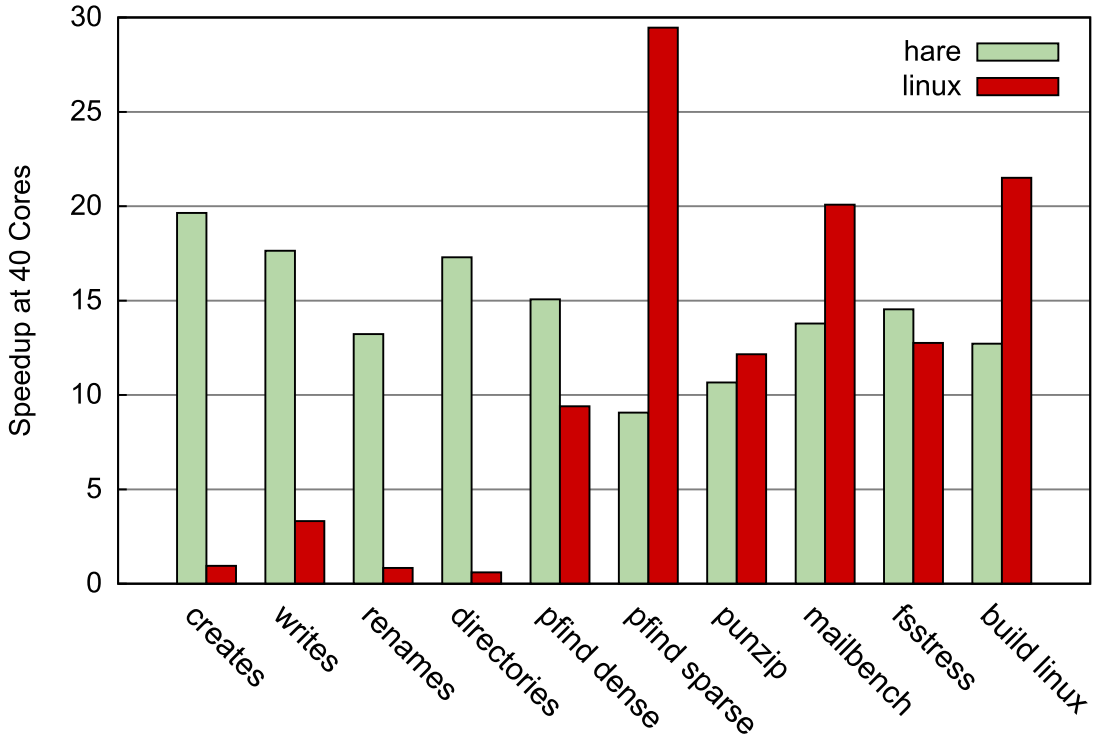

Figure 5-9: Relative speedup for parallel tests running across 40 cores for Hare and Linux respectively.

Chapter 6

Discussion and Open Problems

The previous chapter demonstrated that Hare can achieve reasonable performance and good scalability for a wide range of POSIX applications and microbenchmarks. Furthermore, Hare's specific techniques are important to achieve good performance for this wide range of benchmarks. However, there still remains several opportunities to improve the system. This chapter discusses the open problems related to the design of Hare that may be considered for future work. Additionally, several open discussion topics which arise from designing, implementing and evaluating Hare are discussed. Each of these are provided in the following sections.

6.1 Open Problems

There are a few portions of the design and implementation of Hare that can be improved in future work. The common aspect to all of these future directions is a need to dynamically change the state of the system make a decision based on the current state of the system and a prediction about future behavior. These decisions are non-trivial make as the current state of the system is globally distributed. Each of these open problems is described below.

6.1.1 Dynamic Server Resizing

The results also show that achieving the best scalability requires properly partitioning the cores between the application and the file servers. In Hare's current design, this partitioning choice remains fixed after the system starts, but this may not be practical in a real deployment. A future direction of Hare is to dynamically increase and decrease the number of file servers depending on the load. File data would not be affected, since it can remain in the shared buffer cache. However, adding or removing file servers would require migrating file descriptors and inodes between servers, and would require changing the hash function or using consistent hashing. Additionally, this solution requires a scheme for determining the load on the system and when changing the number of fileservers will improve system throughput. Though the mechanism for migrating file descriptors solved with the use of an indirection table, the other issues in incorporating such support remain non-trivial.

6.1.2 Dynamic Directory Distribution

The mechanism employed by an application to control whether a directory is distributed or stored at a single server is through a flag when creating a directory. Although it is frequently obvious to the application programmer which directories should be distributed as they are shared between processes, it would be more convenient if the system could determine this behavior live and dynamically change the distributed nature of the directory. Heuristics such as the number of entries or the number of operations in a given timeframe could be used to determine when a directory should be distributed. When the system decides to distribute a directory, the entries need to be passed to the other servers and the client caches need to be cleared for each entry from that directory.

6.1.3 Smarter Scheduling Placement Policies

For the benchmarks and applications used to evaluate Hare, one of two scheduling placement policies proves to be sufficient: random or round-robin. In a full system,

other competing factors such as the load on a given core could influence this decision. One approach to solving this is to have the servers periodically exchange their load with each other, providing an approximate view of the load in the system. Using this information the client library could choose the least loaded core. Without support for live process migration, however, this decision can have a negative effect on performance if made incorrectly for long-lived processes.

6.2 Discussion

The following sections discuss some considerations about the design of Hare providing insights gained during the building and evaluation of the system. In particular, the assumption of a shared DRAM when it is possible for it to be partitioned in hardware, the usefulness of caching file data and finally the trade-offs of building distributed systems using the message passing versus shared memory interface.

6.2.1 Shared vs Partitioned DRAM

One assumption made by Hare's design is the existence of a shared DRAM. It is possible, however, for the DRAM to be partitioned into several individual regions of memory. Hare's design is amenable to such an architectural configuration as demonstrated by the results in Section 5.4.3, where the client reads and writes file data via messages to the appropriate server instead of using direct access. As demonstrated by these results there is a performance penalty due to both the overhead of messaging as well as added contention at the server. In the situation where the DRAM is partitioned, a better solution than always sending RPCs to a server would be to use a hybrid approach where direct access is used for file accesses when the data is stored on a local DRAM and sending messages to the server when it is not. This hybrid design would be a straightforward change to the system: the client library could make the decision based on the addresses of the blocks of the file. The performance would likely fall in-between the two scenarios presented in the evaluation chapter. In such a configuration, a stronger effort to take advantage of the *Creation Affinity* heuristic

could prove to be beneficial to avoid remote reads and writes where possible.

6.2.2 Caching File Data

One important consideration in Hare's design is in regards to the usefulness of caching file data. Processors need to both flush as well as invalidate file data when closing a file descriptor to ensure proper consistency. A common access pattern for an application is to simply read or write a file in a linear fashion. As a consequence, for many workloads the application is not benefiting from caching of file data as it is simply populating the cache, then flushing it without accessing it again. There are still advantages to having a cache, however, as server data structures and application data structures will typically be used many times. Therefore, in the common case the best usage of the cache would be obtained by performing the reads and writes on file data directly to the DRAM without any caching and reserving the private cache for application and server data structures that remain private. Note, though, that the linear access pattern does not encompass all applications as there are workloads that exhibit non-sequential read or write patterns such as the linker during a compilation process, retrieving a file using the BitTorrent protocol, editing an image, etc... In these situations it is more likely for the application to take advantage caching file data. Further analysis needs to be done to determine what the typical access pattern is for applications running on the target architecture combined with the consequences of filling the cache with file data that isn't accessed again. It may also be possible for the hardware to control the caching of memory accesses so the decision could be made dynamically.

6.2.3 Shared Memory vs Message Passing

A multikernel design employs the use of message passing for communication between servers which provide operating system services and applications. Hare provides an interface that allows shared abstractions like files, directories and file descriptors to be shared between application processes. As a consequence, a question that has been

discussed many times in the past arises from this thesis in regards to the advantages and disadvantages when using message-passing or shared-memory interfaces to develop scalable distributed systems. In particular, the trade-offs related to this question can be affected by the fast messaging afforded by modern architectures in addition to the complexity of maintaining cache-coherence for a high number of cores.

At a high-level, writing software correctly using shared memory can be fairly straightforward as it is typically just a matter of placing a lock around data structures to prevent multiple processes from accessing that data structure at the same time. A contention issue arises, however, when multiple processes attempt to acquire the lock at the same time. Typically this problem is solved decreasing the granularity of the lock. This approach may introduce new issues as aspects such as lock order can result in deadlock scenarios. Furthermore, there is a limit to how fine-grained locks may be before locking overhead itself may become an issue. One approach to dealing with these issues is to use lock-free data structures. However, these can be quite difficult to program correctly and are frequently architecture dependent.

Using message passing is an alternative approach to building scalable parallel systems with different a different set of trade-offs. From an initial design, it is not simply a matter of placing a lock around a data structure. Instead a server is responsible for maintaining consistency for the data and applications send messages to that server to modify or retrieve the data. This approach can be more difficult to implement as the information that must be exchanged between the application and server must be marshalled and de-marshalled on both ends as part of the RPC. However this difficulty has a subtle added benefit. In making the sharing explicit, it becomes obvious to the programmer when and where the sharing happens and how much of the data is being shared. In particular, it is clear when a single server may become a bottleneck or when a high number of messages will be required or a high amount of data will be transferred to accomplish a given operation. As a result, considerations for performance in terms of scalability may be incorporated into the design early on.

In message-passing systems, data may be cached by separate processes to avoid additional RPCs, which results in independent copies that may be accessed concur-

rently. However, the consistency of this cache must be managed explicitly to ensure that reading stale data does not result in undesirable program behavior.

An additional drawback to using a message passing interface is that there is still a potential for deadlock depending on message ordering. This can be solved by limiting the messaging patterns. In Hare, a design decision made early on was to avoid having servers message each other. Though this can be limiting and places more responsibility in the client library, we are afforded this decision because the client library is in the same fault domain as the servers. Because the servers do not message each other and the client libraries do not message each other, the message patterns do not present a potential for deadlock. Hare does not use a strict RPC messaging pattern for all operations, however. For instance, removing a directory, will result in the server messaging other clients for invalidations. If the server were required to wait for a response from the client, there could be a potential deadlock scenario when multiple clients remove directories. However, Hare assumes reliable message delivery which allows the server to enqueue the message without waiting for a response from the client. This allows the invalidations to be incorporated without a potential for deadlock.

Though the trade-offs make it unclear as to which method may be preferred when building a distributed system, this thesis is a testament to the fact that it can be accomplished in a performant and scalable manner. The file system built on Hare was designed and implemented in roughly one year. This file system supports a broad range of applications using a POSIX interface, it is distributed and scales well with reasonable performance. Using a message passing approach influenced the design in such a way that performance considerations such as caching and scalability were incorporated as part of the design of the system. The design was influenced from the beginning based on the fact that data structures were not shared by default, forcing us to consider explicit caching of information and distributing shared structures from the initial inception, resulting in a scalable design from the start.

Chapter 7

Conclusion

Hare is a novel multikernel operating system that provides a single system image through a unique file system design. It is designed to run on architectures which do not support cache-coherent shared memory. It allows the sharing of common abstractions such as shared files, directories, pipes and file descriptors as well as supporting process migration, all through the familiar POSIX API.

Hare can run many challenging workloads including benchmarks which stress individual aspects of the system, many common Unix utilities, scripting languages such as bash and python as well as full applications such as a build of the Linux kernel and a mailserver benchmark. Furthermore, Hare can run these applications with minimal or no changes.

Hare achieves good performance and scalability through a combination of new protocols for maintaining consistency and exploiting hardware features of the target architecture, such as shared DRAM, atomic message delivery, and a single fault domain. Hare uses techniques such as distributing directories and directory lookup caching to improve performance. Hare also contributes new protocols such as a three-phase commit protocol for guaranteeing atomic directory removal as well as an invalidation protocol to ensure that directory lookup caches are consistent.

Our results demonstrate that Hare's techniques are key to achieving good performance. Hare shows good scalability across a broad range of workloads up to 40 cores, with most workloads demonstrating promise for further scalability at higher

core counts. Our results also show that Hare's techniques may also be beneficial to existing shared-memory multiprocessor operating systems.

Bibliography

[1] UNFS3. http://unfs3.sourceforge.net.

[2] Andrew Baumann, Paul Barham, Pierre Évariste Dagand, Timothy L. Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, October 2009.

[3] Nathan Beckmann, Charles Gruenwald, Christopher Johnson, Harshad Kasture, Filippo Sironi, Anant Agarwal, Frans Kaashoek, and Nickolai Zeldovich. Pika: A network service for multikernel operating systems. CSAIL Technical Report No. MIT-CSAIL-TR-2014-002, January 2014.

[4] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of Linux scalability to many cores. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, Vancouver, Canada, October 2010.

[5] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–17, Farmington, PA, November 2013.

[6] Cluster Filesystems, Inc. Lustre: A scalable, high-performance file system. http://www.cse.buffalo.edu/faculty/tkosar/cse710/papers/lustre-whitepaper.pdf, 2002.

[7] Russ Cox, M. Frans Kaashoek, and Robert Morris. xv6. http://pdos.csail.mit.edu/6.828/2012/xv6.html.

[8] Fred Douglis and John Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Softw. Pract. Exper.*, 21(8):757–785, July 1991.

[9] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, October 2003.

[10] Michael Gschwind, H. Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. Synergistic processing in Cell's multicore architecture. *IEEE Micro*, 26(2):10–24, March 2006.

[11] Jason Howard, Saurabh Dighe, Sriram R. Vangal, Gregory Ruhl, Nitin Borkar, Shailendra Jain, Vasantha Erraguntla, Michael Konow, Michael Riepen, Matthias Gries, Guido Droege, Tor Lund-Larsen, Sebastian Steibl, Shekhar Borkar, Vivek K. De, and Rob F. Van der Wijngaart. A 48-Core IA-32 Processor in 45 nm CMOS Using On-Die Message-Passing and DVFS for Performance and Power Scaling. *J. Solid-State Circuits*, 46(1), 2011.

[12] 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.

[13] The IEEE and The Open Group. POSIX API Specification. IEEE Std. 1003.1, 2013 Edition http://pubs.opengroup.org/onlinepubs/9699919799/.

[14] Texas Instruments. OMAP4 applications processor: Technical reference manual. *OMAP4470*, 2010.

[15] Orran Krieger and Michael Stumm. Hfs: A performance-oriented flexible file system based on building-block compositions. *ACM Trans. Comput. Syst.*, 15(3):286–321, August 1997.

[16] Felix Xiaozhu Lin, Zhen Wang, and Lin Zhong. K2: a mobile operating system for heterogeneous coherence domains. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 285–300, 2014.

[17] James Mickens, Ed Nightingale, Jeremy Elson, Bin Fan, Asim Kadav, Vijay Chidambaram, Osama Khan, Krishna Nareddy, and Darren Gehring. Blizzard: Fast, cloud-scale block storage for cloud-oblivious applications. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, Seattle, WA, April 2014.

[18] Edmund B. Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. Flat datacenter storage. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, October 2012.

[19] Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, and David Hitz. NFS version 3 design and implementation. In *Proceedings of the Summer 1994 USENIX Technical Conference*, Boston, MA, June 1994.

[20] Johan Petersson. What is linux-gate.so.1? http://www.trilithium.com/johan/2005/08/linux-gate/.

[21] Kenneth W. Preslan, Andrew P. Barry, Jonathan Brassow, Grant Erickson, Erling Nygaard, Christopher Sabol, Steven R. Soltis, David Teigland, and Matthew T. O'Keefe. A 64-bit, shared disk file system for linux. In *Proceedings of the IEEE Symposium on Mass Storage Systems*, San Diego, CA, March 1999.

[22] James Reinders and Jim Jeffers. *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufmann, 2013.

[23] Tudor-Ioan Salomie, Ionut Emanuel Subasu, Jana Giceva, and Gustavo Alonso. Database engines on multicores, why parallelize when you can distribute? In *Proceedings of the ACM EuroSys Conference*, Salzburg, Austria, April 2011.

[24] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the Conference on File and Storage Technologies (FAST)*, Monterey, CA, January 2002.

[25] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. GPUfs: integrating a file system with GPUs. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Houston, TX, March 2013.

[26] Xiang Song, Haibo Chen, Rong Chen, Yuanxuan Wang, and Binyu Zang. A case for scaling applications to many-core with OS clustering. In *Proceedings of the ACM EuroSys Conference*, Salzburg, Austria, April 2011.

[27] Manuel Stocker, Mark Nevill, and Simon Gerber. A messaging interface to disks. http://www.barrelfish.org/stocker-nevill-gerber-dslab-disk.pdf, 2011.

[28] Sun Microsystems, Inc. NFS: Network file system protocol specification. RFC 1094, Network Working Group, March 1989.

[29] Sage A. Weil, Scott A Brandt, Ethan L Miller, and Darrell D.E. Long. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, November 2006.

[30] David Wentzlaff, Charles Gruenwald, III, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Lamia Youseff, Jason Miller, and Anant Agarwal. An operating system for multicore and clouds: Mechanisms and implementation. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, Indianapolis, IN, June 2010.

[31] Steven Whitehouse. The GFS2 filesystem. In *Proceedings of the Linux Symposium*, Ottawa, Canada, June 2007.