Hare: a file system for non-cache-coherent multicores

Citation: Charles Gruenwald, III, Filippo Sironi, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Hare: a file system for non-cache-coherent multicores. In Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15). ACM, New York, NY, USA, Article 30, 16 pages.

As Published: http://dx.doi.org/10.1145/2741948.2741959

Publisher: Association for Computing Machinery (ACM)

Persistent URL: http://hdl.handle.net/1721.1/101091

Version: Author's final manuscript: final author's manuscript post peer review, without publisher's formatting or copy editing

Hare: a file system for non-cache-coherent multicores

Charles Gruenwald III, Filippo Sironi, M. Frans Kaashoek, and Nickolai Zeldovich
MIT CSAIL

Abstract

Hare is a new file system that provides a POSIX-like interface on multicore processors without cache coherence. Hare allows applications on different cores to share files, directories, and file descriptors. The challenge in designing Hare is to support the 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 features (such as shared file descriptors) that traditional network file systems don't support, as well as implement them in a way that scales (e.g., shard a directory across servers to allow concurrent operations in that directory). Hare achieves this goal through a combination of new protocols (including 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.

1 Introduction

As the number of cores per processor grows, it is becoming more challenging to implement cache-coherent shared-memory. Although computer architects don't agree on whether scalable shared memory is feasible [25], researchers and practitioners are exploring multicore designs that have a global physical address space but lack cache coherence. In such designs, cores have private caches and share one or more DRAMs (see Figure 1). Cores communicate with each other using messages, but can read and write from DRAM. Processors cache the results of such reads and writes in private caches, but do not provide cache coherence. Intel's SCC [19], IBM Cell's SPE [18], the TI OMAP4 SoC [21] and GPGPUs are examples of such processors, and even on commodity x86 servers, Intel's Xeon Phi [34] has non-cache-coherent shared memory.

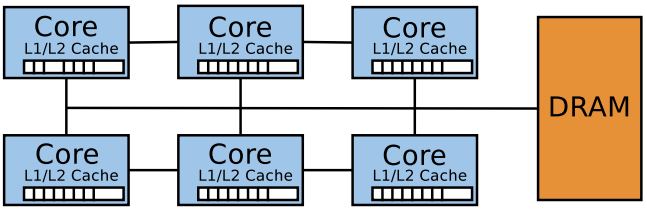

Figure 1: Hare's target multicore system: private caches, shared DRAM, but no hardware cache coherence.

Hare is a file system for such non-cache-coherent shared-memory machines. The challenge in Hare's design is that non-cache-coherent shared-memory systems require software to explicitly handle coherence. Hare strives to provide the illusion of a single, coherent system—often called *single system image*—because it makes it easy to write applications. For example, if one core creates a file, it is convenient if another core can read that file with a guarantee that it will see the last consistent version of that file. Similarly, it is convenient for programmers to be able to create a child process on a remote core that may share file descriptors with the parent process. Traditional shared-memory operating systems rely on cache-coherence and synchronization primitives (such as locks) to implement these facilities, but on non-cache-coherent shared-memory systems, there exists no good solution that scales to a large number of processors.

One possible solution is to treat the non-cache-coherent multicore as a distributed system and run a distributed file system (e.g., NFS [30] or 9P/VirtFS [22]). Processes share files by making remote calls to the NFS server. The downside of this solution is that many Unix applications cannot be run on several cores, because network file systems do not provide a single system image. For example, NFS clients lack a mechanism for sharing file descriptors, but many applications rely on shared file descriptors (e.g., building software with `make` or using redirection). As a result, such applications are restricted to a single core on an NFS-based design. In addition, this "solution" treats the underlying hardware as a message-passing distributed system, and cannot take advan-

EuroSys'15, April 21–24, 2015, Bordeaux, France.
ACM 978-1-4503-3238-5/15/04.
`http://dx.doi.org/10.1145/2741948.2741959`

tage of shared resources (such as non-cache-coherent shared memory) for performance.

This paper explores the Hare file system design, which allows applications to take advantage of multiple cores by providing a single system image on non-cache-coherent multicores in a scalable manner. To provide the single-system-image POSIX API, Hare must support the correct semantics for shared files, directories, pipes, sockets, file descriptors, etc, without relying on the hardware cache-coherence protocol. Hare implements the POSIX system call API faithfully enough that it can run many POSIX applications with little to no modifications. For example, Hare can build the Linux kernel with a small modification to `make` (to flag the pipe of the jobserver as shared) and no modifications to any of the standard Unix utilities used by the build process. Hare provides these interfaces in a way that allows applications to scale to many cores. For example, our suite of benchmarks achieves an average speedup of 14× on a 40-core machine with Hare, compared to running on a single core.

Hare achieves good performance using several techniques. First, Hare uses the shared DRAM to store the buffer cache, but each server manages its own partition of the buffer cache. This allows for high throughput for read and write operations while avoiding storing multiple copies of the same file data. Second, to allow for scalable directory operations, Hare shards directory entries in the same directory to different servers, using hashing. To ensure consistency of the directory caches, Hare employs an invalidation protocol, and to reduce the latency of invalidation, Hare relies on atomic message delivery. To handle removal of directories correctly, Hare uses a three-phase protocol. Third, Hare supports remote execution of processes. Fourth, Hare supports sharing of file descriptors between processes.

We implement Hare by running a Linux kernel, interposing on system calls that involve sharing, and redirecting those system calls to a local Hare client library. System calls that don't involve shared resources are processed by the Linux kernel locally. This implementation strategy allowed us to implement a sufficiently complete POSIX API, so that Hare can run many Linux applications unmodified, including building the Linux kernel, running a mail server, etc. In addition, this strategy enables a more direct performance comparison with Linux, because local systems calls are implemented identically.

We evaluate Hare on a 40-core off-the-shelf machine that provides cache-coherent shared memory. Like previous operating systems (e.g., Barrelfish [6]), Hare uses cache coherence purely to pass messages from one core to another. Hare uses shared DRAM to store file data, but explicitly manages file consistency in software by sending invalidate message on `open` and writing back dirty blocks on `fsync`. The advantage of using a cache-coherent shared-memory machine is that we can also run Linux as a shared-memory multiprocessor operating system on the same machine, and compare Hare and Linux in terms of performance and scalability.

We run complete application benchmarks (such as compiling the Linux kernel) as well as microbenchmarks to test specific aspects of the Hare design. We find that most benchmarks scale well when run on Hare. Hare's single-core's performance is worse than Linux's due to messaging overhead, however Hare's performance is substantially better than a user-space NFS server [2]. We also find that the individual techniques that Hare uses are important to achieve good scalability.

Although Hare's design scales well to many cores, our current prototype has several limitations. We do not support persistent storage to disk, and instead focus on the multi-core scalability of the in-memory file system. We do not support multiple threads in a single process, to keep Hare's polling IPC implementation simple. Our remote execution mechanism does not support migration between `exec` calls. Finally, we do not support the entire POSIX API, but instead focus on providing enough of POSIX to run many real applications, as we show in §5.

2 Related Work

Hare targets running standard POSIX applications on multicore processors without cache-coherent shared memory. To achieve this goal Hare builds both on work in file systems for multicore processors and distributed systems.

2.1 File systems for multicore processors

Shared-memory file systems. Widely used multicore processors support cache-coherent shared-memory and can run shared-memory multiprocessor (CC-SMP) file systems. Although CC-SMP file systems are successful, CC-SMP file systems have suffered from scalability bottlenecks due to locks on directories and reference counting of shared file system objects [11]. To achieve better scalability, CC-SMP file systems like Linux have been moving to a more distributed model, with per-core state or explicit replication for certain data structures, adopting techniques from earlier CC-NUMA systems like SGI IRIX [41], Tornado [16], and K42 [3]. However, the Linux file system takes advantage of cache-coherent shared memory for shared file descriptors, shared file caches, and the shared directory cache—something that Hare cannot do on a non-cache-coherent system.

Because Hare cannot rely on cache coherence, it introduces several new techniques that, when combined together, allow POSIX applications to run with few modifications on a non-cache-coherent multicore processor. Furthermore, because we implemented Hare on top of the Linux kernel (without using Linux's shared-memory functionality), we can compare Hare to Linux using POSIX applications, which allows us to assess the value of cache coherence. From this comparison we conclude that some techniques that Hare introduces could also be of value to CC-SMP file systems to achieve scalability.

HFS [23] allows high-performance-computing applications to adjust the file system to the needs of the application and the architecture it is running on. It gives applications fine-grained control over how files are organized and accessed to improve performance, based on a workload's access pattern and the architecture of the machine. Hare targets general-purpose SMP applications, which require many POSIX features beyond basic file operations, and often need those operations to be scalable. For example, applications like *make* stress many POSIX operations, and require directory and file operations to scale well.

Hive [10] splits a cache-coherent shared-memory multiprocessor into cells and runs an isolated kernel in each cell for fault containment. Different cells can use distributed file systems like NFS to share files across cells. Disco [9] layers a small virtual machine monitor below the cells to manage resources between the different kernels and to support both general-purpose kernels with limited scalability and special-purpose scalable kernels. Cerberus uses a combination of a virtual machine monitor and a layer on top of the cells to provide a scalable single system image on top of the cells [38]. Unlike Hare, all of these systems rely on cache-coherent shared memory.

File systems for heterogeneous multicore processors. Operating systems targeting heterogeneous multicore processors or intelligent IO devices split the system in different functional components [27] but typically do not support a single file system across the different cores. For example, the Spine [15] and Hydra [43] systems support offloading OS components to smart devices but don't provide a file system to them. Helios's satellite kernels provide a uniform kernel API across heterogeneous cores but Helios doesn't include file and directory operations [28]. Hare's techniques could be used to implement a file system using the uniform kernel API.

K2 [24] implements a "share most" operating system based on Linux for multicore mobile platforms with separate domains that have no cache coherence between them. K2 relies on distributed shared memory for sharing OS data structures across coherence domains. Hare targets architectures where coherence domains consists of a single core and Hare relies on new protocols, instead of distributed shared memory, to achieve good performance and scalability for shared files, directories, and so on.

GPUfs [37] 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 by POSIX. Furthermore, GPUfs is focused purely on accessing file contents through a restricted interface, and cannot support general-purpose POSIX applications. For example, without sharing of file descriptors, pipes, and so on, *make*'s jobserver will not function correctly.

Cosh [7] provides abstractions for managing shared memory across heterogeneous cores, including cache-coherent and non-cache-coherent cores as well as accelerators that require explicitly transferring memory to and from the shared system memory. CoshFS provides a simple file system on top of the Cosh primitives, but focuses on file data, and does not address the question of how to provide scalable directory and file descriptor operations.

File systems for multikernels. Multikernel operating systems don't require cache-coherent shared memory; each core runs its own kernel, which communicate with message passing. One approach for building a multikernel file system is to strictly partition the file system state across cores, similar to how many multicore databases work [35]. This suffers from load imbalance and makes remote data access costly. The file system in the fos multikernel [44] shares data between cores, but is limited to read-only workloads. Barrelfish uses a standard distributed file system, NFS [39], but NFS doesn't support POSIX applications that rely, for example, on shared file descriptors (as discussed below), while Hare does. Although Hare runs on top of the Linux kernel, Hare doesn't rely on shared memory and its techniques could be used in multikernel operating systems to build a scalable file system that can run a wide range of POSIX applications.

Popcorn [5] is a multikernel based on Linux specifically targeting heterogeneous platforms without cache-coherent shared memory. Popcorn is complementary to Hare: Popcorn focuses on running a separate Linux on each core while providing a shared namespace and process migration. However, Popcorn does not have a scalable design for a shared file system. We expect that Hare's file system design would fit well into Popcorn.

2.2 Distributed file systems

Hare adopts many techniques from distributed file systems, but differs in significant ways. To support the same applications as CC-SMP file systems do, Hare introduces a number of new techniques to support the necessary POSIX semantics. Hare does so by leveraging properties of the hardware that distributed file systems cannot leverage: assuming a single failure domain between the Hare file servers and client libraries, leveraging fast and reliable message delivery between cores, and using shared DRAM for efficient access to bulk shared data.

LAN Distributed File Systems. Hare's design resembles networked distributed file systems such as NFS [30], AFS [20], and Plan 9 [32], and borrows some techniques from these designs (e.g., directory caching, close-to-open consistency, etc). The primary differences are that Hare can exploit 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 standard SMP POSIX applications with almost no modifications.

Consider a situation where a file descriptor is shared between a parent and child process. 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 [1], the underlying file descriptor 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 descriptors, applications using this idiom are limited to a single core.

Distributed file systems also 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. This situation arises during a typical compilation process. According to the POSIX [1] specification, the file data should remain valid for the original process that opened the file. Networked file systems typically do not handle this situation, 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.

More broadly, distributed OSes like Sprite [14], Amoeba [40], and MOSIX [4] also aim to provide a single system image, which includes a shared file system. These systems provide process migration to place applications closer to nodes storing data, but cannot take advantage of direct access to shared DRAM, and do not provide scalable directory operations.

Datacenter and Cluster File Systems. Hare targets running POSIX applications in a scalable manner by parallelizing file system operations. 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 [12] and the Google File System [17], which creates a potential bottleneck for metadata operations. The Flat Datacenter Storage (FDS) [29] 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 [26] 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 [42] 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.

Cluster file systems rely on sophisticated protocols to achieve consistency between nodes. For example, Farsite showed how to implement rename atomically across directories [13]. Hare uses similar protocols, such as its directory removal protocol, to provide shared file system semantics on a non-cache-coherent system, but it can assume a single failure domain.

Shared-disk file systems such as Redhat's GFS [33, 45] and IBM's GPFS [36] 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.

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 goal is challenging because modern SMP applications rely on the POSIX API in complex ways. Consider an application such as *make* to compile the Linux kernel. *make* has its own internal job scheduler that creates processes to achieve good parallelism, it uses shared file descriptors, pipes, and signals to coordinate activities, it removes files that other processes opened, it creates many files in the same directory, and so on. Hare must implement all these features correctly but without relying on cache consistency as CC-SMP operating systems do. This section introduces the design and protocols that Hare uses to achieve its goal.

3.1 Overview

Figure 2 illustrates Hare's overall design. The main parts of Hare are the per-core client libraries and the file servers. Applications run on top of the Hare client library, and issue POSIX system calls such as `open`, `read`, and `write`, to this library. The library implements the Hare protocol, maintains caches, accesses data in the shared buffer cache directly via DRAM, and communicates with the file servers via message passing. Our design assumes that the client library runs in the kernel, protected from misbehaving applications. Our current prototype is implemented as a library that runs in the application process itself, but we expect it would be straightforward to migrate it into the kernel. Our prototype runs on top of the Linux kernel, so that we can make comparisons with Linux. Hare uses Linux to boot up the system and to provide system calls that do not require sharing. Hare interposes on all systems calls for shared resources (e.g., files, file descriptors, etc.) and redirects those system calls to the Hare client library.

Our prototype of Hare 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 a prerequisite for tackling persistence, and poses a set of unique challenges in maintaining shared file system state on top of non-cache-coherent shared memory;

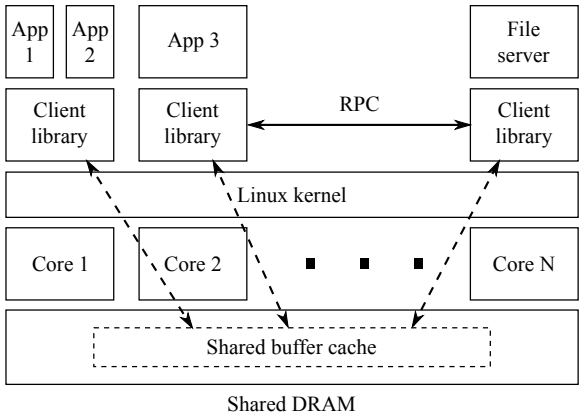

Figure 2: Hare design.

implementing persistence for a non-scalable file system is relatively straightforward. We are focused on the steady-state behavior of the system after blocks have been read from disk; when our Hare prototype reboots, it loses all data. 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. We expect that implementing `fsync` would be straightforward by serializing `fsync` at the corresponding file server and flushing the relevant file to disk (or a distributed protocol for flushing a distributed directory).

Although Hare focuses on the file system, it inevitably needs to run processes on multiple cores. To avoid relying on the Linux kernel (and its shared memory) for this, Hare introduces a scheduling server. 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 original process that called `exec()` and the new child process that has been spawned locally.

Since Hare runs on a single physical machine, it assumes a single failure domain: either all cores and DRAM are working, or the entire machine crashes. This abstraction matches the model provided by our target non-cache-coherent machines. We assume that the Hare client libraries and file servers are correct and do not crash, similar to how a monolithic kernel is assumed to never crash. If only a subset of the file servers or client libraries crash or malfunction, Hare will malfunction as a whole; it might return incorrect data to an application, corrupt the file system state, or hang.

Figure 3 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 split between all of the 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.

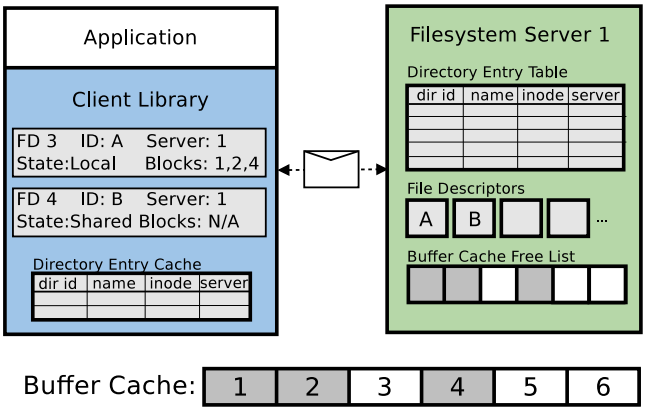

Figure 3: Data structures used by the Hare File System.

The rest of this section discusses Hare's file system in more detail, focusing on how shared state is managed.

3.2 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; each block is managed by one file server. 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 the client library, which 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.

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 [20, 30]. 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 its local processor cache to the shared DRAM. This ensures that when that file is opened on any core, the will observe the latest changes to that file since the last close. As we demonstrate in §5, many applications are compatible with close-to-open semantics.

Although close-to-open semantics do not require Hare to ensure data consistency in the face of concurrent file operations, Hare must ensure 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 is still writing 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 (see §3.4).

3.3 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 directory entries across the file system servers. When an application creates a file in a distributed directory *dir*, the client library determines which server to contact about the directory entry using a hash function: `hash`(*dir*, *name*) % `NSERVERS` $\rightarrow$ *server_id*, where *name* is the name of the new file in directory *dir*. To avoid re-hashing directory entries when the parent directory is renamed, Hare uses the inode number to identify each directory (and file) in the file system, which does not change when it is renamed. In the hash computation, *dir* refers to the parent directory's inode number.

Hashing ensures that directory entries of the directory *dir* are evenly distributed across the file servers, so that applications can 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 show in §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.

Most directory operations require contacting just one or two servers. For example, `rename` first contacts the server storing the new name, to create (or replace) a hard link with the new name, and then contacts the server storing the old name to unlink it. One exception is `readdir`, which requires contacting all servers to obtain a list of all directory entries. In case of concurrent operations, this might produce a non-linearizable result, but POSIX allows this [1].

One complication with directory distribution arises during `rmdir`, which must atomically remove the directory, but only if it is empty. Since the directory entries are distributed across file servers, performing `rmdir()` on the directory can race with another application creating a file in the same directory.

Solution: Three-phase directory removal protocol. To prevent the race between `rmdir()` and file creation in that directory, Hare implements `rmdir()` using a three-phase 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, allowing the servers to delete that directory. If any server indicates the directory is not empty, the client library sends an `ABORT` message, 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. To avoid deadlock, 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.

It might be possible to optimize this protocol further, by placing the directory's home server in charge of coordinating `rmdir()` across all of the servers. We did not implement this because Hare avoids server-to-server RPCs, which simplifies reasoning about possible deadlock scenarios between servers.

3.4 File descriptors

File descriptors are used to keep track of the read/write offset within an open file, 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 ensure that the file descriptor information remains consistent when shared. 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 both processes.

A second scenario which poses a challenge for Hare is related to unlinked files. In POSIX, a process can still read and write a file through an open file descriptor, even if the file has been unlinked.

Solution: Hybrid File Descriptor tracking. To solve this problem, Hare stores some file descriptor state at servers. For each open file, the server responsible for that file's inode tracks the open file descriptors and associated reference count. This ensures that even if the 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 synchronous RPC request to the server to increment the reference count; it changes back to *local* state when the reference count at the server drops 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.

3.5 Processes

In order to take advantage of many cores, 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, since it requires packaging up data structures related to the process on the local core, sending them over, and unpacking them on the destination core.

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` on that core before resuming execution.

Each core runs a scheduling server, which listens for RPCs to perform `exec`s. When a process calls `exec`, the client library implements a scheduling policy for deciding which core to pick; our prototype supports both a *random* and a *round-robin* policy, with round-robin state propagated from parent to child. These two policies proved to be sufficient for our applications and benchmarks. The client library then sends the arguments, file descriptor information, and process environment to the other core's scheduling server. The scheduling server in turn starts a new process on the destination core (by forking itself), configures the new process based on the RPC's arguments, and calls `exec` to load the target process image on the local core.

Running a process on another core faces 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 local file descriptors (e.g., to the console or other file descriptors specific to the original core) that are not accessible on the new core.

To address this challenge, Hare uses a *proxy* process, similar to MOSIX [4]. 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 an RPC back to the proxy, enabling the proxy to exit, and thereby providing the exit status to the parent process. If the process running on the new core tries to access any file descriptors that were specific to its original core, the accesses are turned into messages back to the proxy process, which relays them to the original core. Finally, if the proxy process receives any signals, it relays them to the new process.

If a process repeatedly `exec`s and runs a process on another core, it can accumulate a large number of proxy processes. When the remote core already has a proxy for the process calling `exec`, Hare could reuse this proxy process. We do not implement this in our prototype, since our target applications do not perform repeated `exec` calls in the same process.

3.6 Techniques and Optimizations

Hare implements several optimizations to improve performance, which we evaluate in §5.

3.6.1 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: `lookup`$(dir, name) \rightarrow \langle server, inode \rangle$, 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 must therefore store both the inode and the server of the file or directory.

Hare must ensure that it does not use stale directory cache entries. To do this, Hare relies on file servers to send invalidations to client libraries, much like callbacks in AFS [20]. 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.

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.

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, the message is guaranteed to be present in the receiver's queue. In our prototype implementation, we implement message passing on top of cache-coherent shared memory, and achieve atomic message delivery by delivering the message into the receiver's shared-memory queue before returning from `send()`.

To take advantage of this property, Hare's directory lookup function first checks the invalidation queue for incoming 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 themselves. Note that the lookup operation does not have to contact the server; it simply processes a queue of already-delivered messages.

This design ensures that if a server sent an invalidation to a client library before that client core initiated a lookup, the lookup is guaranteed to find the invalidation message in its queue. If the invalidation message was sent after the lookup started, the client might miss the invalidation, but this is acceptable because the invalidation and lookup were concurrent.

3.6.2 Directory broadcast

As described in §3.3, 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()` have to contact all of the 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 its `readdir()` in parallel, even for a single `readdir()` call. The individual RPCs to each file server are implemented as unicast messages; the benefit of this technique stems from overlapping the execution of server RPC handlers.

3.6.3 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 are sent to the same server for the same 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.

3.6.4 Creation affinity

Modern multicore processors have NUMA characteristics [6]. 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, under the assumption that the inode will be accessed more frequently than the directory entry. As mentioned in §3.6.1, Hare names inodes by a tuple consisting of the server ID and the per-server inode number to guarantee uniqueness across the system as well as scalable allocation of inode numbers. Each client library has a designated local server it uses in this situation, to avoid all clients storing files on the same local server. Creation Affinity requires client libraries to know the latencies for various servers, which can be measured at boot time.

4 Implementation

Hare's implementation follows the design in Figure 2. 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 [31] to intercept the application's system calls (by rewriting the syscall entry code in the `linux-gate.so` page) and determine whether the call should be handled by Hare or forwarded to the kernel. Hare implements most of the system calls for file system operations, as well as several for spawning child processes and managing pipes. We have not placed the client library into the kernel since it complicates the development environment, although it would allow processes on the same core to share the directory lookup cache. As a result, our prototype does not provide any security or isolation guarantees.

Hare does not rely on the underlying kernel for any state sharing between cores; all cross-core communication is done either through Hare's message passing library [8], which uses polling, or through the buffer cache, which is partitioned among the servers, and totals 2 GB in our setup. The message passing library is implemented using shared memory, but could be changed to a hardware message-passing primitive. Our prototype does not support multiple threads within a single process because of the polling-based message passing. To limit any unintended use of shared memory, the applications as well as the server processes are pinned to a core; we informally checked that Hare does not inadvertently rely on shared memory.

Because Hare relies on RPC to file servers that potentially run on the same core, Hare uses a modification to the Linux kernel to provide PCID support. Use of the PCID feature on the Intel architecture allows the TLB to be colored based on

process identifiers, which avoids flushing the TLB during a context switch. This results in faster context switch times which can result in faster messaging when a server is sharing a core with the application.

The Hare prototype strives to provide a POSIX interface, and except for close-to-open consistency, we believe that Hare adheres to POSIX. As we describe in §5, our benchmarks use a wide range of file system operations, and are able to run correctly on Hare, suggesting that Hare's POSIX support is sufficient for many applications. That said, Hare is a research prototype and other applications might expose ways in which Hare does not fully adhere to the POSIX standard. We expect that it would be relatively straightforward (if time-consuming) to find and fix cases where Hare deviates from POSIX.

Lines of code for various portions of the system are provided in Figure 4; Hare is implemented in C and 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: SLOC breakdown for Hare components.

5 Evaluation

This section evaluates Hare's performance on several workloads to answer four 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 section are from a system with four Intel Xeon E7-4850 10-core processors, for a total of 40 cores, with 128 GB of DRAM. 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. All reported results are the averages across 6 runs of each experiment.

5.2 POSIX Applications

As one of Hare's main goals is to support POSIX style applications, it is important to consider which applications 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,

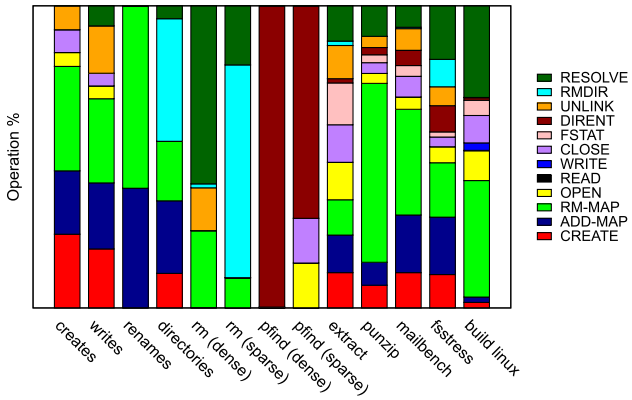

Figure 5: Operation breakdown for our benchmarks.

while others employ a broader range of operations. All of the applications can run on Hare as well as Linux without any modifications. However, to improve their performance, we made small changes to several applications; specifically, to control the sharing policy of directories or to use `exec()` in addition to `fork()` calls. Additionally, we chose a placement policy for each of the applications (random placement for *build linux* and *punzip* and round-robin for the rest), though the placement policy is within the Hare client library and does not require modifying the application.

The tests which stress individual parts of the system include *creates*, *writes*, *renames*, *directories*, *rm* and *pfind*. For each of these benchmarks an individual operation is performed many times (typically 65535 iterations) within the same directory, to reduce variance. The *dense* directory tree contains 2 top-level directories and 3 sub-levels with 10 directories and 2000 files per sub-level. The *sparse* directory tree contains 1 top-level directory and 14 sub-levels of directories with 2 subdirectories per level. The *extract* test performs a decompression of the Linux 3.0 kernel while the *build linux* text performs a parallel build of this kernel. The *punzip* test unzips 20 copies of the manpages on the machine in parallel. The *fsstress* benchmark repeatedly chooses a file system operation at random and executes it; we borrowed it from the Linux Test Project. Finally, the *mailbench* test is a mail server benchmark from the sv6 operating system [11]. The POSIX use cases described previously are present in one or more of these tests. It is also important to note that configuring and building the Linux kernel invokes a broad range of standard Unix utilities, all of which run unmodified on Hare.

From Figure 5, 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 $1.2M$ operations with the other tests issuing tens to hundreds of thousand operations. Tests such as *extract*,

punzip and *build linux* make use of pipes. *make* relies on a shared pipe implemented in Hare in order to coordinate with its jobserver. The broad range of applications and the various ways in which they access the system demonstrate that Hare can support a variety of real-world applications. For space reasons, we do not break down how each application uses Hare, but collectively they stress all aspects of Hare.

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 our benchmarks observe when running on a different total number of cores. We use a single-core Hare configuration as the baseline and increase the number of cores, as well as the number of Hare servers, up to the maximum number of cores in the system. Figure 6 shows the results.

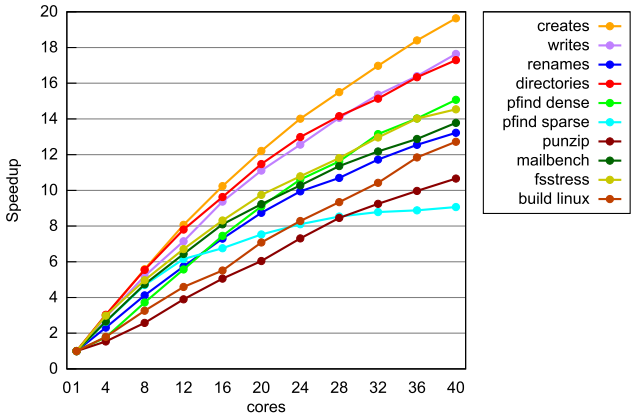

Figure 6: Speedup of benchmarks on Hare as more cores are added, relative to their throughput on a single core.

The results demonstrate that for many applications, Hare scales well with the number of cores. The benchmark that shows the least scalability is *pfind sparse*. In this test, application processes recursively list directories in a sparse tree. As the directories are not distributed in this test and there are relatively few subdirectories, each of the clients contacts the servers in the same order, resulting in a bottleneck at a single server as all n clients perform lookups in the same order. The other tests show promise of further scalability to higher core counts.

5.3.2 Split Configuration

In addition to running the filesystem server on all cores, Hare can also be configured to dedicate several cores to the filesystem server while running the application and scheduling server on the remaining cores. As will be described in §5.3.3, there are some performance penalties associated with time-multiplexing cores between servers and applications, though the split configuration limits the number of cores than an application may scale to.

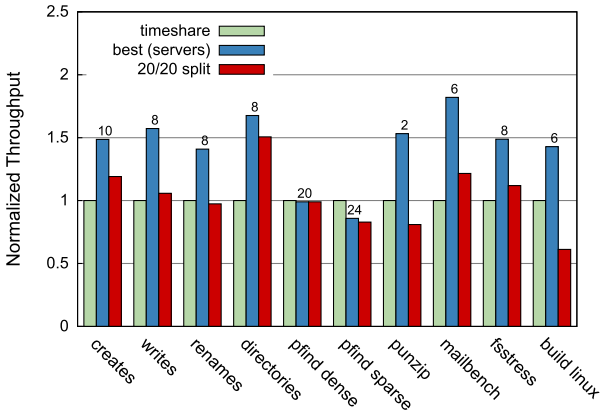

Figure 7: Performance of Hare in both split and combined configurations.

Figure 7 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 one. Each of the configurations is normalized against the timeshare configuration which was used for the scalability results presented in Figure 6.

From these results, it is clear that Hare can achieve better performance if we knew the optimal number of servers. 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. For example, always using a 20/20 split leads to significantly lower throughput for *mailbench*, *fsstress*, *directories*, and others. Conversely, always using an 8/32 split leads to significantly lower throughput for both *pfind* benchmarks. Consequently, we employ a time-sharing configuration instead, which achieves reasonable performance without per-application fine-tuning and provides a fair comparison for the results presented in §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 one core. In the simplest configuration, Hare runs on a single core, time-multiplexing between the application and the server. Hare can also be configured

to run the application process(es) on one core alongside the scheduling server while the filesystem server runs on another core. In this split configuration, Hare's performance can be improved significantly as the time spent performing individual operations is comparable to the cost of the context switch (which is significant because it involves the Linux kernel scheduling and switching to the file server, and then scheduling and switching back to the client library). When running in the split configuration, there are no other processes sharing the core with the filesystem server, and therefore the cost of the context switch is eliminated. An additional performance hit is also incurred due to cache pollution when the application and the server share a core. We note, however, that the split configuration does use twice as many resources to perform the same operations.

We also compare Hare's performance with that of UNFS3 [2], a user-space NFS server. This comparison more closely represents a state-of-the-art solution that would be viable on a machine which does not support cache coherence. In this setup, Linux uses two cores: one running the NFS server, and one running the application, accessing the NFS server via a loopback device. This experiment is intended to provide a naïve alternative to Hare, to check whether Hare's sophisticated design is necessary.

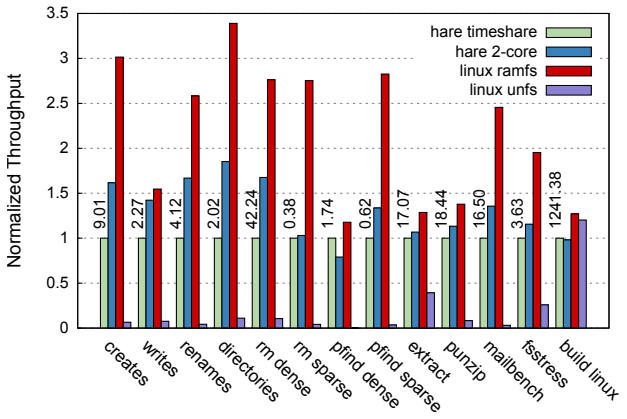

Figure 8: Normalized throughput for a variety of applications running on one core on Linux, relative to Hare ("hare timeshare"). The number above the "hare timeshare" bar indicates the total runtime of that workload in seconds.

Figure 8 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 OS and loopback interface. The only application that performs well on UNFS3 is *build linux*, since it more CPU-intensive, and Hare's scheduler is worse than Linux's. 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 with Hare achieve a median throughput of 0.39× that of Linux on one core.

Much of Hare's performance on micro-benchmarks stems 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 at the client end, 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 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 sending and receiving respectively. Using *perf* demonstrates that a higher portion of the time is being 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 each individual technique in Hare's design, we measure the relative throughput of each benchmark in Hare's normal configuration normalized to the throughput with that technique disabled. Each technique has different effects on each benchmark; Figure 9 summarizes the results, which we discuss in more detail in the rest of this subsection. For example, in Figure 10, the *creates* test is approximately 4× faster when distributing directories than without. We perform this comparison on the timesharing configuration using all 40 cores, to include the effects on scalability.

Technique	Min	Avg	Median	Max
Directory distribution	0.97×	1.93×	1.37×	5.50×
Directory broadcast	0.99×	1.43×	1.07×	3.93×
Direct cache access	0.98×	1.18×	1.01×	2.39×
Directory cache	0.87×	1.44×	1.42×	2.42×
Creation affinity	0.96×	1.02×	1.00×	1.16×

Figure 9: Relative performance improvement achieved by five techniques in Hare over the set of all benchmarks, compared to not enabling that technique. Higher numbers indicate better performance; numbers below 1× indicate benchmarks that got slower as a result of enabling a technique.

Directory Distribution. Applications have two options for how to store a directory—distributed or centralized—determined by a flag at directory creation time. When a directory is distributed, the entries are spread across all servers.

For centralized directories, the entries are all stored at a single server. 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*.

As seen in Figure 10, 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 and thus leave 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 directory distribution on a per-directory basis is a good idea.

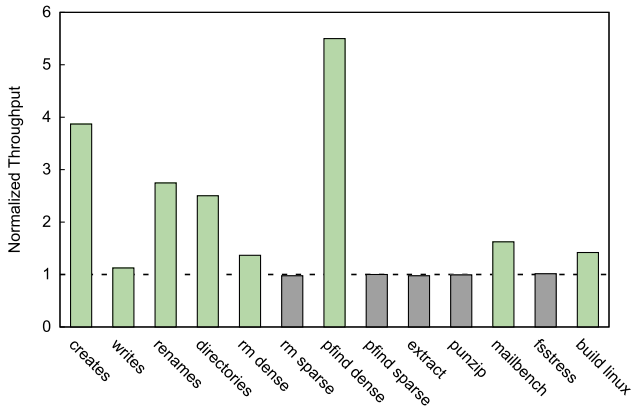

Figure 10: Throughput of Hare with Directory Distribution (normalized to throughput without Directory Distribution).

Directory Broadcast. As mentioned above, Directory Distribution can improve performance when several operations are being executed in the same directory. One drawback is that some operations must now contact all servers, such as `rmdir()` and `readdir()`. Hare uses Directory Broadcast to send out such operations to all servers in parallel. Figure 11 shows the effect of this optimization, compared to a version of Hare that uses sequential RPCs to each file server for directory operations. 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 perform operations in different subtrees, *fsstress* turns off directory distribution for this test, and therefore Directory Broadcast is not employed.

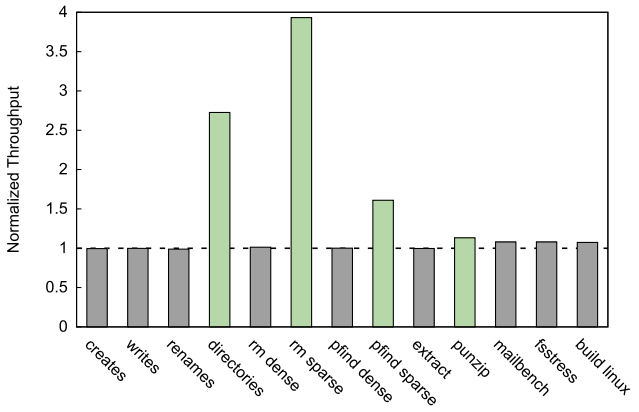

Figure 11: Throughput of Hare with Directory Broadcast (normalized to throughput without Directory Broadcast).

Direct Access to Buffer Cache. Figure 12 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 RPCs to the file server. The performance advantage of directly accessing the buffer cache is most visible in tests which perform heavy file I/O, 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 reduces RPC calls, which provides a significant performance advantage by alleviating congestion at the server.

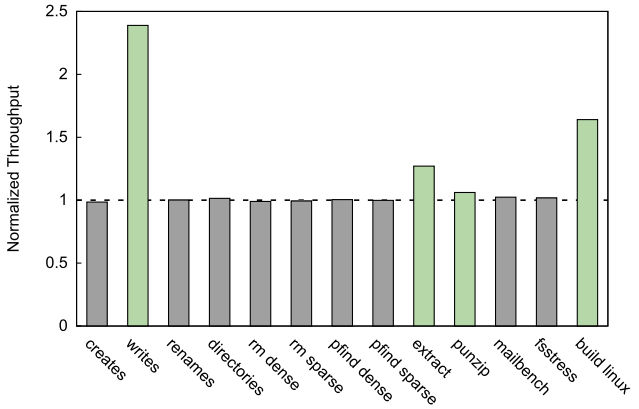

Figure 12: Throughput of Hare with direct access from the client library to the buffer cache (normalized to throughput without direct access).

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 a unified buffer cache to reduce capacity misses introduced by sharing the same blocks on multiple cores. To evaluate such effects, we use *build linux* as a representative workload, as it has a large working-set size. On this test, the number of misses is 2.2× greater when the buffer cache is not shared. In a persistent file system that stores data on disk (unlike Hare), these misses would require going to disk, and thus this increase in the number of misses would have a significant performance penalty. Additionally, direct access to the buffer cache is never a hinderance to performance and therefore should be used if available.

Directory Cache. Caching directory lookups improves the performance of many benchmarks. As seen in Figure 13, the benchmarks which benefit the most perform many operations on the same file or directory, such as *renames*, *punzip* or *fsstress*. In *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* test takes a performance hit with this optimization as it caches lookups without using them. Overall, considering the performance advantage across all tests, it makes sense to use this technique.

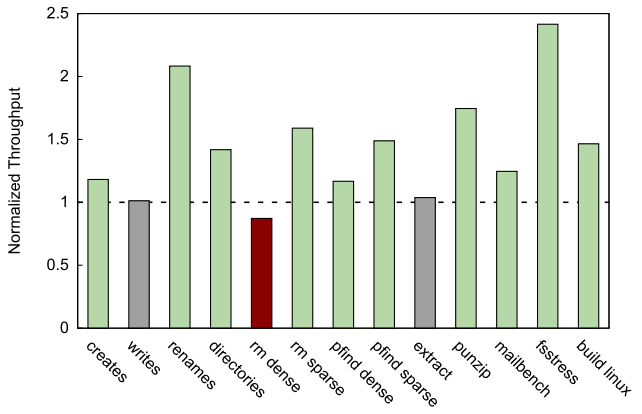

Figure 13: Throughput of Hare with the Directory Cache (normalized to throughput without the Directory Cache).

Creation Affinity. Figure 14 shows the effect of enabling the creation affinity optimization. With creation affinity disabled, Hare places newly created files on the same server that stores the corresponding directory entry. While this enables better message coalescing, it requires sending messages to server cores that are further away. The benchmarks that most benefit from Creation Affinity are *writes*, *punzip*, and *mailbench*. In these workloads, the application creating the file is likely to access the file again, and therefore benefit from nearby placement.

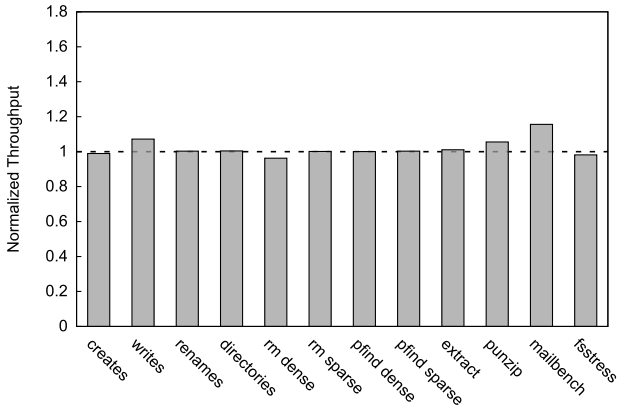

Figure 14: Throughput of Hare with Creation Affinity (normalized to throughput without Creation Affinity).

5.5 Hare on cache-coherent machines

Figure 15 shows the speedup of the parallel tests on both Hare and Linux (using tmpfs) for 40 cores. Some tests scale better on Hare while others scale better on Linux. Traditional shared-memory operating systems, such as Linux, could potentially benefit from employing distributed directories to increase performance for applications which perform many operations within the same directory.

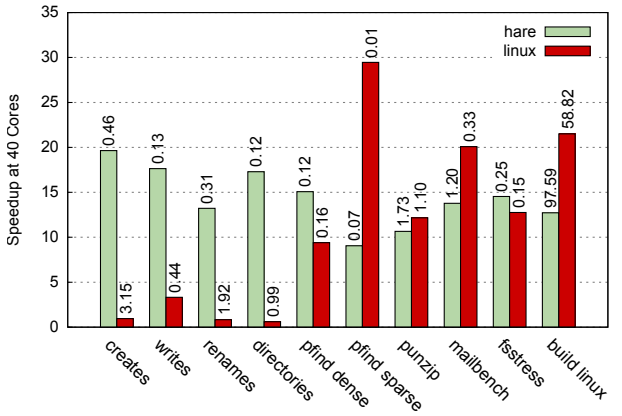

Figure 15: Relative speedup for parallel tests running across 40 cores for Hare (timesharing configuration) and Linux respectively. The absolute running time on 40 cores, in seconds, is shown above each bar.

6 Discussion

Hare contributes to the discussion of whether future processors should provide cache-coherent shared memory. As the previous section demonstrated, it is possible to provide a single system image with good scalability without relying on cache coherence. However, Hare's performance is notably lower than that of Linux, largely as a result of IPC overhead. While Linux simply cannot run on a non-cache-coherent system, this nonetheless suggests that better hardware support for

IPC (or direct hardware support for remote flushes or reads on another core) could be helpful. Hardware that provides islands of cache coherence between small subsets of cores could provide an interesting trade-off: if applications and file servers reside in the same coherence domain, Hare could switch to shared-memory data structures instead of RPC.

Every directory in Hare is either not distributed or is distributed across all cores. This makes sense with a modest number of cores, but as the number of cores grows, contacting every server may become too expensive. Such systems may want to consider intermediate choices, such as distributing a directory over a subset of cores, or automatically adjusting the directory's distribution state based on access patterns, instead of relying on programmers. Similarly, Hare could achieve better performance by dynamically tuning the number of file servers for a particular workload. This would require redistributing state between file servers and updating the client libraries' mappings from directory entries and inodes to file servers.

The previous section also showed that it's possible to support many largely unmodified applications on a non-cache-coherent system, even though they were designed with POSIX in mind. Hare's design appears to be a good fit for multikernel systems that want to provide a POSIX-like interface. Supporting these applications required us to implement complex features such as file descriptor sharing, directory distribution, directory entry caching, etc. Exploring whether alternative APIs (such as ones that expose more asynchrony) can lead to lower overheads while still providing a convenient abstraction for programmers is an interesting question for future work.

Hare relaxes the guarantees typically provided by POSIX, such as by providing close-to-open consistency in some cases. We believe this is a good fit for a large range of applications, as confirmed by the fact that Hare can run many unmodified benchmarks correctly. This tradeoff might not be a good fit for applications that concurrency access the same file from multiple cores (without inheriting a single shared file descriptor for that file), such as BerkeleyDB. We believe these applications are an exception rather than the rule, and can likely be made to work on Hare with small changes (such as explicit use of file locks or shared file descriptors).

Hare's design lacks support for on-disk persistence. Adding persistence support to Hare is challenging because the on-disk state must be consistent in case of crashes. However, since individual file servers do not communicate with one another, this is hard to achieve. For instance, flushing a single directory to disk is complicated when the directory is distributed across multiple file servers and applications are concurrently modifying parts of the directory on each of the servers. Combining scalable in-memory file systems with on-disk persistence in a crash-safe manner is an interesting area for future research, independent of whether shared memory provides cache coherence or not.

7 Conclusion

The Hare file system provides a single system image on multicore system without cache-coherent shared memory. Hare can run many challenging POSIX workloads 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. Our results demonstrate that Hare's techniques are key to achieving good performance, and may also be beneficial to existing shared-memory multiprocessor operating systems. Hare's source code is available at `https://github.com/charlesg3/faux`.

Acknowledgments

Thanks to the anonymous reviewers and our shepherd Don Porter for feedback that helped improve this paper. This paper builds upon prior work on the Pika multikernel network stack [8], and we thank Nathan Z. Beckmann, Christopher R. Johnson, and Harshad Kasture for their help in developing Pika. This research was supported by Quanta.

References

[1] POSIX API Specification. IEEE Std. 1003.1, 2013 Edition.

[2] UNFS3. `http://unfs3.sourceforge.net`.

[3] J. Appavoo, D. D. Silva, O. Krieger, M. Auslander, M. Ostrowski, B. Rosenburg, A. Waterland, R. W. Wisniewski, J. Xenidis, M. Stumm, and L. Soares. Experience distributing objects in an SMMP OS. *ACM Transactions on Computer Systems*, 25(3):6, 2007.

[4] A. Barak, S. Guday, and R. G. Wheeler. *The MOSIX Distributed Operating System: Load Balancing for UNIX*. Springer-Verlag, 1993.

[5] A. Barbalace and B. Ravindran. Popcorn: a replicated-kernel OS based on Linux. In *Proceedings of the Linux Symposium*, Ottawa, Canada, July 2014.

[6] A. Baumann, P. Barham, P. É. Dagand, T. L. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. 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, Oct. 2009.

[7] A. Baumann, C. Hawblitzel, K. Kourtis, T. Harris, and T. Roscoe. Cosh: Clear OS data sharing in an incoherent world. In *Proceedings of the 2014 Conference on Timely Results in Operating Systems (TRIOS)*, Broomfield, CO, Oct. 2014.

[8] N. Z. Beckmann, C. Gruenwald, III, C. R. Johnson, H. Kasture, F. Sironi, A. Agarwal, M. F. Kaashoek, and N. Zeldovich. Pika: A network service for multikernel operating systems. Technical Report MIT-CSAIL-TR-2014-002, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, Jan. 2014.

[9] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, Nov. 1997.

[10] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault containment for shared-memory multiprocessors. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 12–25, Copper Mountain, CO, Dec. 1995.

[11] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. 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, Nov. 2013.

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

[13] J. R. Douceur and J. Howell. Distributed directory service in the Farsite file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, Nov. 2006.

[14] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software: Practice and Experience*, 21(8):757–785, July 1991.

[15] M. E. Fiuczynski, R. P. Martin, B. N. Bershad, and D. E. Culler. SPINE: An operating system for intelligent network adapters. In *Proceedings of the 8th ACM SIGOPS European Workshop*, 1998.

[16] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, pages 87–100, New Orleans, LA, Feb. 1999.

[17] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, Oct. 2003.

[18] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic processing in Cell's multicore architecture. *IEEE Micro*, 26(2): 10–24, Mar. 2006.

[19] J. Howard, S. Dighe, S. R. Vangal, G. Ruhl, N. Borkar, S. Jain, V. Erraguntla, M. Konow, M. Riepen, M. Gries, G. Droege, T. Lund-Larsen, S. Steibl, S. Borkar, V. K. De, and R. F. V. 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.

[20] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, Feb. 1988.

[21] T. Instruments. OMAP4 applications processor: Technical reference manual. *OMAP4470*, 2010.

[22] V. Jujjuri, E. V. Hensbergen, A. Liguori, and B. Pulavarty. VirtFS - a virtualization aware file system pass-through. In *Proceedings of the Linux Symposium*, Ottawa, Canada, July 2010.

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

[24] F. X. Lin, Z. Wang, and L. Zhong. K2: a mobile operating system for heterogeneous coherence domains. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 285–300, Salt Lake City, UT, Mar. 2014.

[25] M. M. K. Martin, M. D. Hill, and D. J. Sorin. Why on-chip cache coherence is here to stay. *Communications of the ACM*, 55(7):78–89, July 2012.

[26] J. Mickens, E. Nightingale, J. Elson, B. Fan, A. Kadav, V. Chidambaram, O. Khan, K. Nareddy, and D. 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, Apr. 2014.

[27] S. Muir and J. Smith. Functional divisions in the Piglet multiprocessor operating system. In *Proceedings of the 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications*, pages 255–260, Sintra, Portugal, 1998.

[28] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: Heterogeneous multiprocessing with satellite kernels. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 221–234, Big Sky, MT, Oct. 2009.

[29] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue. Flat datacenter storage. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, Oct. 2012.

[30] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3 design and implementation. In *Proceedings of the Summer 1994 USENIX Technical Conference*, Boston, MA, June 1994.

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

[32] R. Pike, D. Presotto, K. Thompson, H. Trickey, and P. Winterbottom. The use of name spaces in Plan 9. *ACM SIGOPS Operating System Review*, 27(2):72–76, Apr. 1993.

[33] K. W. Preslan, A. P. Barry, J. Brassow, G. Erickson, E. Nygaard, C. Sabol, S. R. Soltis, D. Teigland, and M. 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, Mar. 1999.

[34] J. Reinders and J. Jeffers. *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufmann, 2013.

[35] T.-I. Salomie, I. E. Subasu, J. Giceva, and G. Alonso. Database engines on multicores, why parallelize when you can distribute? In *Proceedings of the ACM EuroSys Conference*, Salzburg, Austria, Apr. 2011.

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

[37] M. Silberstein, B. Ford, I. Keidar, and E. 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, Mar. 2013.

[38] X. Song, H. Chen, R. Chen, Y. Wang, and B. Zang. A case for scaling applications to many-core with OS clustering. In *Proceedings of the ACM EuroSys Conference*, Salzburg, Austria, Apr. 2011.

[39] M. Stocker, M. Nevill, and S. Gerber. A messaging interface to disks. `http://www.barrelfish.org/stocker-nevill-gerber-dslab-disk.pdf`, 2011.

[40] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, and S. J. Mullender. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, Dec. 1990.

[41] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. *ACM SIGOPS Operating Systems Review*, 30(5):279–289, Sept. 1996.

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

[43] Y. Weinsberg, D. Dolev, T. Anker, M. Ben-Yehuda, and P. Wyckoff. Tapping into the fountain of CPUs: On operating system support for programmable devices. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 179–188, Seattle, WA, Mar. 2008.

[44] D. Wentzlaff, C. Gruenwald, III, N. Beckmann, K. Modzelewski, A. Belay, L. Youseff, J. Miller, and A. 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.

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