Exporting Kernel Page Caching for Efficient User-Level I/O

Appears in the proceedings of the 26th IEEE Conference on Mass Storage Systems and Technologies (MSST 2010)

Richard P. Spillane, Sagar Dixit, Shrikar Archak, Saumitra Bhanage, and Erez Zadok
Computer Science Department
Stony Brook University
Stony Brook, New York 11794-4400

***Abstract*—The modern file system is still implemented in the kernel, and is statically linked with other kernel components. This architecture has brought performance and efficient integration with memory management. However kernel development is slow and modern storage systems must support an array of features, including distribution across a network, tagging, searching, deduplication, checksumming, snap-shotting, file pre-allocation, real time I/O guarantees for media, and more. To move complex components into user-level however will require an efficient mechanism for handling page faulting and zero-copy caching, write ordering, synchronous flushes, interaction with the kernel page write-back thread, and secure shared memory. We implement such a system, and experiment with a user-level object store built on top. Our object store is a complete re-design of the traditional storage stack and demonstrates the efficiency of our technique, and the flexibility it grants to user-level storage systems. Our current prototype file system incurs between a 1% and 6% overhead on the default native file system EXT3 for in-cache system workloads. Where the native kernel file system design has traditionally found its primary motivation. For update and insert intensive metadata workloads that are out-of-cache, we perform 39 times better than the native EXT3 file system, while still performing only 2 times worse on out-of-cache random lookups.**

I. INTRODUCTION

User-level programs that perform I/O are becoming ever more critical to the user. Performance of user-level database APIs and web-service infrastructure is the new storage bottleneck, and is critical to the average Internet user. However in the current I/O stack, for purposes of memory management, write ordering, and caching, databases are subordinate to in-kernel file systems.

There is a strong modularity and security argument against allowing user-level programs unchecked kernel privileges. The argument is that the file system represents a generic commonly used interface to on-disk data. Recently however, file systems have been continuously growing in number, and in complexity, with more file systems offering more features such as snapshotting, attribute indexing, checksumming, per-file striping, logical volume management, network distribution, deduplication, copy-on-write, and more. Future in-kernel storage systems, even more complex than the current generation, could benefit from the virtualizations and system abstractions afforded to user-level software.

On the other hand, user-level I/O stacks have been growing in complexity to suit the complex needs of large OLTP and Web services. Systems such as Dryad [46], Map Reduce [8], Hadoop [3], and Memcached [9] all rely on interaction with the underlying file system on each node. However the efficiency of Hadoop on a single node to perform a sort is 5 to 10% of what is possible in an efficient single-node implementation [2]. Further, power efficiency is directly related to performance efficiency [34]. As these systems seek out economies in performance and power usage, a modular user-level storage API with kernel-level speed and control will come into demand.

There has been a flurry of excitement in the research community over user-level storage stacks and APIs such as Anvil [22], Rose [33], and Stasis [32]. New indexing needs have prompted new file system architectures that break away from hierarchical name-spaces such as Perspective [31], Spyglass [20], and hFAD [36]. File system extensions to track provenance [26] also make more sense architecturally as modular user-level extensions [37], but only if the write semantics are correct and the performance is good.

Unfortunately to get proper write semantics and good performance, many implementers must rely on ad-hoc techniques to achieve proper write ordering such as accidental write ordering of the lower file system [22], non-standard `fsync` flags such as `F_FULLSYNC` [4], or complex interposition mechanisms [25]. Turning off disk caching in the physical hardware is another common practice to ensure proper write semantics for database APIs [27], but hurts performance dramatically. Finally, it is standard practice for database APIs and servers to *re-implement* page caching to ensure control over write-ordering, page pinning, cache eviction policies, and faulting [39].

We argue that the common interface exposed by the kernel should not be a host of different mounted file systems in a hierarchical name space, but instead a shared-memory page-cache service with configurable eviction, faulting, write-back, and dirtying policies. To allow kernel-level performance for in-

978-1-4244-7153-9/10/$26.00

cache workloads, we have added support to Linux for a third kind of privilege level between user and kernel space called *library* space. The MMU unit is used to protect privileged libraries in an analogous way to how it is used to protect the kernel at a different ring level. Although our system can support many different kinds of I/O interfaces besides a traditional file system, to compare with current in-kernel storage running current work loads, we have implemented a privileged library that supports standard UNIX file system calls. We have implemented our own path name resolution and caching rather than using Linux's VFS, and have implemented our own file mapping layer rather than using Linux's VM. With some minor caveats, the only kernel interface our privileged library currently utilizes is our exported page caching interface on top of a raw block device. We have evaluated both our library, and our underlying system's performance, against native kernel file systems, finding for almost all workloads we perform competitively, equivalently, or even better than the native file system.

In Section II we discuss the design and implementation of our exported page caching technique and our object store. In Section III we discuss related work. We relate our experimental findings in Section IV. Finally we conclude in Section V.

II. Design and Implementation

The object store is designed with efficiency of data transfers and metadata operations in mind. It utilizes several write-optimized indexes to perform rapid insertion, deletion, and update of metadata values in a space-efficient manner. For data transfers, it adopts a read-optimized file system format. To make calls into the object store efficiently while providing security to other users, we utilize a modified trap instruction.

A. Object Store Architecture

As seen in Figure 1, the architecture of the object store consists of a library loader and trapping mechanism to safely call into a journaling layer. This layer then calls into the namespace layer which performs path traversals to find object IDs. The namespace then calls into the object layer which exports an interface to POSIX-style sparse files with object IDs for names. Finally the object layer sits on top of a lower layer which interacts with the Linux kernel to provide efficient and recoverable write-back to disk that does not cause resource deadlock.

The namespace and object layer utilize four indexes: (1) The *dmap index*, (2) The *omap index*, (3) the *fmap index*, and (4) the *buddy index*. The dmap index stores dmaps, equivalent to `dentries`. The omap index stores onodes, which are compact versions of `inodes`. The fmap index acts like a range tree for each file, storing which ranges of file offsets in each file are backed by which physical extents of storage. The buddy index acts like a buddy allocator tree, storing which blocks on the disk are free, and splitting or merging blocks as necessary. Actual blocks are allocated from the block store.

We implement our own indexing technology for each of the four indexes using a simple 2-COLA [5] data structure. We modify the 2-COLA to support deletes, finite disk space, atomic flush to disk for ordered writes, 2-phase commit with other COLAs, and a special kind of query that avoids performing lookups on disk for faults into sparse files.

The lower layer consists of a series of `mmaps` on top of a large disk (30GiB in our evaluation). We operate only on 64-bit architectures due to address space requirements. To support proper ordering of writes, we have modified the Linux kernel to support a new flag to `mmap` called `MPIN` which pins *file* pages dirtied by writes, preventing them from being written back to disk (unlike `mlock` which only prevents pages from swapping out and not writing back to disk). To avoid memory squeezes we employ a signaling mechanism between the page allocator in the kernel and the process utilizing `mpin`. Backing our memory mappings is a single-file file system that we wrote to optimize reads and writes with `mmap`, control write-back of dirty pages, and ensure sync requests wait for a disk cache flush.

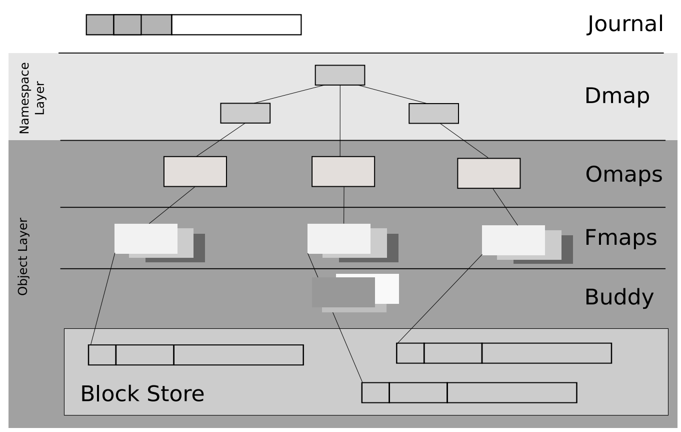

Fig. 1. The object store architecture

B. Library Loader and Trapping Mechanism

The trapping mechanism uses the same technique the kernel uses, to trap into our shared user-level library. When processes wish to call into our user library, they call a special system call `reroute` which redirects them to a system call handler in our shared library based on an integer id we store in their process block. The redirection process modifies the segment descriptor of the process so that it has access to the shared library address range. Without trapping into the kernel it would not be able to access this address range. Individual stacks are maintained for each client that enters into the shared library as in a typical kernel. To return from a privileged library call, the library calls `reroute` again to reset the segment descriptor. In our current implementation, the `reroute` system call makes all necessary traps but does not modify the segment descriptor.

If the user traps into the kernel, but has loaded arbitrary code at the reroute point (where the library should have been) he could execute code at an unauthorized privilege level. To stop this, the kernel only `reroutes` to *privileged* address ranges. An address range is considered privileged if its in-kernel mapping struct (`vma` in Linux) is marked as privileged. If a user process wants to mark an address range as privileged it must use the `seal` system call. After a user process loads a library into an address space using `mmap`, it `seals` the entire library's address range before using `reroute`. During `seal`, the kernel disables read, write, and execute permissions on the range, and then it checksums the range. To ensure that the contents of the loaded library have not been tampered with, the kernel marks the mapping corresponding to the library's address range as privileged only if the checksum is found in a list of valid checksums that the kernel maintains. The library's mapping does not permit reads, writes, or executions except via `reroute`. The list of valid checksums is hard-coded within a separate kernel module that is loaded before any privileged libraries are loaded.

This allows for arbitrary users to directly read from or write into a user-level storage system cache with minimal message passing overhead and by utilizing a context-switching mechanism which is already highly optimized (`sysenter/exit` on Intel). We show in our evaluation that the overhead of our `reroute` routine is negligible.

C. Alternative Storage Stack

The typical user of an exported page cache would be a user-level database or storage server. We experiment with the flexibility and performance characteristics of our approach by re-implementing the VFS in C++ where objects are cached and written back to disk using the four 2-COLA [5] indexes. We find that our alternative storage stack has interesting performance properties regarding metadata update and search performance for extremely large numbers of file objects.

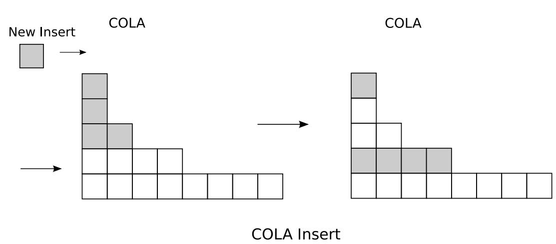

Fig. 2. The COLA performs a single insertion.

COLA: We utilize the cache oblivious look-ahead array data structure to provide our indexing needs. We have made some alterations to the data structure to allow journaling and locking.

The cola maintains for n key-value pairs (typically stored as `structs`) $log(n)$ sorted arrays of packed key-value pairs. As seen in Figure 2, when a new element is inserted into the COLA, if the top level is free, it is inserted there directly; otherwise we find the first free level by searching from the top down: this is called the *target level*. All levels above the target level are merged into the target level, thus freeing them all for insertions.

By ensuring the levels increase in size by a factor of two, we are guaranteed to always have enough space to merge all levels above a target level into that target level The amortized cost of insertion into this structure is asymptotically superior to the B-Tree, and has been shown to insert up to 700 times faster for random insertions [5]. Lookups in the COLA are slower, up to ten times slower. Fractional cascading can optimize lookups [5], but we do not implement this optimization in this work.

The COLA has several other attractive properties, including the tendency to place recently inserted and updated items at the top where the backing page is more likely to be in RAM and even CPU cache. In addition the COLA adopts a naturally log-structured layout on disk which is ideal for rapid insertion to other high-latency high-bandwidth mediums such as a network.

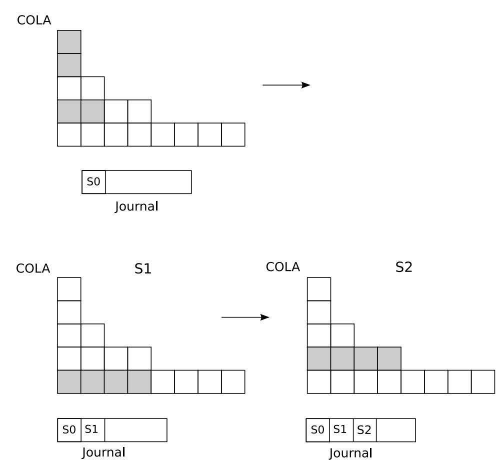

Fig. 3. The COLA performs a journal commit.

COLA journaled: The COLA does not typically support deallocating disk space after deleting elements. We modified the data structure to percolate into a higher level after a merge. Since all levels above the target level become free after a merge, if the target level can fit into a higher level, we copy it upward, reclaiming space after a series of deletes. This process is called *percolation* and in Figure 3 we see the target level in $S1$ percolate upward in $S2$ after finding a higher level that fits.

The output of a merge can be smaller than the input if there are *cancellation* keys which are inserted during deletes to cancel out existing key-value pairs during a merge. To track the amount of elements in each level, we maintain a *cursor* which all readers share, and which writers must take a lock on to update after performing a merge into a target level. These cursors are appended to the journal as part of the flush procedure.

The journaling layer of our system instantiates a thread which periodically executes the flush procedure. To provide recovery of the object store in case of a crash, we modified the COLA to flush to disk in an atomic manner. We serialized the cursors for each level into a journal as part of a 2-phase commit protocol as seen in Figure 3. The flush procedure is:

1) Take a write lock on *all* COLAs, waiting for existing readers to leave.
2) For each COLA, flush all levels to the level below the lowest level; sync that target level to disk.
3) Write the state of all COLAs' cursors to the journal followed by a sector containing a checksum of the newly added cursors; sync the journal.
4) For each COLA, percolate the target level to any level that fits; sync that level to disk.
5) Write the state of all COLAs' cursors to the journal followed by a sector containing a checksum of the newly added cursors; sync the journal.

An example is seen in Figure 3. The COLA begins with a cursor that points to the data stored at state $S0$ in the COLA. A flush is requested, so the COLA transitions to state $S1$ where all contents are merged into the level underneath the lowest level. A new cursor is appended to the journal. If anything happens, the data $S0$ points to is still on disk and was not overwritten by the merge in $S1$. Finally we percolate upward in $S2$, with the data from $S1$ again being unaffected leaving the $S1$ cursor usable in case of a crash.

This series of actions allows us to write elements to a COLA index and know they will hit the disk in the order we wrote them, assuming callers take the appropriate locks, and assuming dirty pages are not written back until flush (guaranteed by our page pinning implementation). The additional sync step is needed as we may write over an existing level pointed to by the last-known good cursor in the journal while percolating up. Our multi-state approach to avoid this scenario is similar to protocols used in log truncation of a transaction manager [14].

Our file system upon which our object store is backed supports a truly synchronous `msync` that explicitly flushes the disk caches by issuing a `SATA FLUSH_CACHE` command and waiting for its successful return and full flush of any pending writes or commands before issuing any more writes to the disk cache. This allows us to *guarantee* full atomicity of writes in our system without having to turn off the disk caches [27]. We would not have to hold a write-lock on the COLAs during flush if we utilized copy-on-write, this is a subject of future investigation. Additionally, our flush protocol is currently linear with respect to the size of the index, this can be improved and is a subject of future work. Currently we at least allow readers to continue performing reads while a merge into the target level is ongoing. We wait for them to leave so we can update the COLA's shared cursor state.

D. Namespace and object layer

The namespace stores each `dentry` as an element in the dmap index. Elements in this index consist of the *parent-id*, the *component*, and the *child-id*. The parent-id is equal to the child-id of the parent `dentry`. The primary ordering of elements is parent-id, followed by the component, and the value is the child-id. The child-id is equal to the object ID of the object in the object layer that the `dentry` points to.

The object layer supports POSIX like semantics for manipulating file objects. A file object is similar to a POSIX file except the file name (object ID) is chosen by the system, and is returned to the namespace layer to link a `dentry` against, analogous to an `inode` number. Otherwise the file can be read from, written to, and sought through. The object store supports sparse files, and rewards sequential appends with increasingly larger contiguous allocations. File objects can be pre-allocated in advance, and can specify a preferred block size during writes that would require allocating new blocks to back the file. We have designed our object layer in a peculiar way to optimize performance on the COLA.

a) Omaps and fmaps: All files in the system are associated with a single omap object, whose primary key in the omap index is its object ID. Rather than using a radix tree of block pointers to support sparse files, we adopt an approach similar to XFS. We introduce the idea of an fmap which contains information about a contiguous range of virtual offsets in a file. Newly allocated files start off with a single fmap that indicates the entire range is empty and is not backed by any physical blocks from the block store. An fmap which indicates a range is not backed and is empty is called a *gap*. As virtual offsets are faulted by write requests, the fmap containing the virtual offset is broken into two or three new fmaps, where the ranges not containing the faulted offset are associated with fmaps set as gaps. The range containing the faulted offset is associated with a *backed* fmap. Backed fmaps point to extents in the block store, allocated by the buddy index. Fmaps are stored in the fmap index, and utilize our *bucket query*.

Bucket queries can take any virtual offset, and find the fmap in the fmap index that contains it. This is done without performing a merge of all levels in the COLA, making faulting performance fast for fmaps still in cache.

To reward serial writes with increasingly larger extents allocated from the block store, each gap is given a link boost. When the virtual offset at the beginning of the gap is faulted, the link boost is used to determine the size of the allocated extent. This is called a *serial allocation*. When a serial allocation occurs, the subsequent gap is given an even larger link boost. Gaps created by random faults are always given a link boost of 1. Over-allocation of extents can be dealt with by having the buddy allocator reclaim unused portions of extents past the end of the file when free space is low.

b) Buddy allocator: The buddy allocator is implemented within an index and does not use a binary tree. During deallocation we simulate traversing the binary tree, pushing would-be buddies onto the call stack, and attempting to merge with them. Blocks allocated by the buddy allocator have offsets relative to the start of the allocation region. Reading from such a block is equivalent to doing a memory copy from the read block to the reader's buffer, a write is the same.

c) Optimizing for COLA journaling: Our object store is careful to write out updates to the lower COLAs such that dependencies are always written before dependent objects. In case of crash we may utilize an asynchronous back-ground cleaner thread to perform a sequential range query through the indexes to garbage collect objects not pointed to. This would not be possible if our COLAs did not guarantee proper ordering of insertions (e.g., inserting A into the fmap COLA and then inserting B into the omap COLA should result in A hitting disk before B). We therefore rely on our journaling protocol to ensure this property is held.

d) Optimizing for COLA write-optimization: Performing any kind of query in the COLA is slow, and the slowest queries are those that perform range queries, or non-exact match queries (e.g., lower bound). This is because the next key in a cursor increment could be in the coldest cache level. On the other hand, queries which look for exact matches can stop searching downward the moment they find a match. Also insertions, updates, and deletes are inserted from the top, and so a key which is updated frequently, or was deleted frequently results in a fast exact-match search. We therefore have designed the object layer to perform all operations in terms of insertions, updates, deletes, and exact match queries.

For example, in our name space layer dmaps refer to their child directory entries via the parent-id in each child's key, so we can perform pathname lookup without performing a range query. File creation consists of performing a path lookup, and if no entry is found, creating an object id in the omap, followed by creating a gap fmap. A special extension offered by our object store is a *blind create* which performs no path lookup, and if errors arise, they are dealt with asynchronously later on during a COLA merge. This allows for near to disk-throughput file creations that are actually faster than the native EXT3 file system's creation throughput.

E. Lower Layer: Exported Page Caching

To minimize the performance penalty of a user-level page cache, we use the kernel's page cache for our object store. Rather than maintain a page table which is checked on every access, we `mmap` the entire store, and use the hardware to fault cache misses. This allows the object store to avoid paying for a software hash table or software lookup calculation for cache hits. However, Linux does not permit applications to prevent write-back of dirty file mappings to disk. Linux relies on this to prevent memory squeezes, and has been a sticking point for user-level file systems. We resolve this issue by adding a new flag to `mmap` called `MPIN`, along with a new system call `set_vma`, and a custom file system 1ffs. Among other things, 1ffs has been especially modified to provide a full disk cache flush on `msync`.

e) 1ffs and swapping: The file system we use underneath our object store library is a simple file system that when mounted, appears to have only one file in its root directory. The file has a hard-coded inode and when the file system is mounted, the file is always considered to be owned by the `root` user. 1ffs provides a direct interface to the disk for applications that use `mmap`. When 1ffs is asked to perform page-writeback, it first determines if page write-back is due to memory pressure or a periodic flush for durability. If the write-back is due to memory pressure, 1ffs writebacks the dirty pages to a special swap partition it maintains. Integrating this swapping mechanism with the system swap would require careful understanding of the interaction between these two kinds of swapped pages, and how it would effect thrashing on the system, and is a subject of future work. If the page-writeback is a periodic flush for durability, 1ffs does not write back the pinned pages.

In this manner 1ffs makes a best effort to not write-back pinned pages, while still offering a last recourse to the kernel during a memory squeeze. Further, if the user-level storage component ensures it never pins more pages than what could be swapped out, 1ffs can guarantee the process will not be killed due to pinned pages.

f) `MPIN`: The `MPIN` flag that we add to `mmap` marks pages belonging to that mapping as pinned. When a pinned page is written to, it is marked dirty, and is written back by the VFS as part of a periodic flush for durability, or in response to memory pressure. 1ffs' page write-back checks if pages are pinned before writing them back, and does not write back pinned pages. If the sync bit is set in the write-back control parameter, then 1ffs syncs the pages. This bit is only set when the pages are synced as part of an explicit user-invoked `msync` request on the page mapping.

g) 1ffs and `set_vma`: The Linux out-of-memory killer is responsible for killing processes to reclaim pages in case of a memory squeeze that will cause resource deadlock. Although 1ffs is able to prevent processes from being killed by the out-of-memory killer, thrashing from writing back swapped pages can harm performance. User processes can not determine if they will fault a page on a read or a write, or they will be forced to maintain their own page table and perform lookups on every access. The purpose of exporting the kernel page cache was to avoid this. Therefore 1ffs offers processes a new system call `set_vma` which allows a process to set a high water mark. When the total number of dirty pinned pages belonging to a particular process exceeds the threshold set in `set_vma`, the kernel sends a `SIGUSR1` signal to the process, to which it can respond by immediately flushing its dirty pages to disk, or face being swapped out.

To write into the exported page cache, we simply use `memcpy` into the mapped address corresponding to the physical block on disk allocated by the buddy allocator. To prevent a read fault on the first copied bytes (as the kernel tries to fill the page to prevent data inconsistencies visible to the user), for future work we could modify the kernel to detect a read or write fault, and to *not* read in the page on a write fault.

We implement `set_vma` by modifying the `mm_struct` handle, which aggregates all the virtual memory mappings belonging to a process. During a page fault, the kernel descends a red-black tree of virtual memory areas (`vmas`) until it finds the one containing the fault. This `vma` is passed into the fault handler which determines if the page is anonymous (e.g., from `malloc`) or backed by a file. At this point we use a back pointer in the `vma` to access the `mm_struct` which contains the global count of dirty pages and increment it. If the count is over the set threshold, we send the SIGALRM signal.

By utilizing the features provided by 1ffs and our trapping mechanism, a user-level process can efficiently use the kernel's page cache to fault in pages on reads and writes to disk while working with the kernel to keep memory pressure on the system low. The kernel is not deprived of total control of the system, and can swap pinned pages if necessary, or just kill the process and release the pinned pages (without writing them back) if swap space is exceeded. User processes can trap into privileged shared memory libraries that use these facilities to efficiently modify the caches of these shared-memory database and storage servers without the use of costly message passing mechanisms. We have used these features to implement a simple but scalable object store which competes favorably to native file system performance.

III. Related Work

Previous work related to our object store built on top of our exported page caching technique can be categorized as works dealing with current mmap semantics and design decisions in commodity kernels, external paging, user-level file systems (namely FUSE), other object-store and indexing file systems, and alternative indexing technologies.

h) Existing In-Kernel Mechanisms: The virtual memory management component of the kernel has grown considerably since Linux 2.4. Memory management of anonymous memory has been merged with management of dirty file pages. The swapper has been implemented to act like a special file system to write-back dirty MAP_ANONYMOUS pages to the swap. Newer features such as MAP_POPULATE and MAP_NONBLOCK were introduced and not reserving swap space for read-only pages has been introduced since 2.5. The mlock system call and its cousin flag in the mmap system call are typically referred to as page *pinning* system calls; they are not that however. The mlock system call simply ensures that a page will never be written to swap, and will never be released/evicted. It does *not* guarantee that it will not be written back. This is in fact difficult to design around due to the fact that Linux still treats dirty file pages and dirty anonymous pages quite differently in its virtual memory management code.

Other kernels such as Solaris and MacOS X also provide the standard POSIX mmap and mlock system calls, but they have the same semantics on these systems [42]. For instance, Solaris also handles file pages and anonymous pages differently, since Solaris 8 dirty file pages are placed on a separate list to reduce pressure on anonymous memory [6]. Solaris and Mac OS X also provide the same POSIX standard mlock semantics. Solaris and Linux offer Direct IO to give user-level database processes the opportunity to perform their own caching. However these cache implementations remain separate, distinct, and are difficult to make as efficient as the in-kernel page cache, requiring careful implementations [32, 44] and tuning cache size to avoid thrashing [29]. In addition they do not benefit from the review and testing that a kernel component receives.

i) Micro-Kernel Approaches: The body of work on micro-kernels includes a large and extensive list of operating systems. The major contributions include L4 [21], L4-Verified recently [19], Spring [24], Exokernel [15], Pebble OS [11], VINO [35], Synthesis [23], Accent [30], and Mach [1]. Each of these projects are all new operating environments, some include modular APIs to re-use kernel components at the user-level, such as VINO. L4 and Mach are the canonical micro-kernels, offering a practical implementation of the concept. Exokernel utilizes an even smaller micro-kernel that only handles permission and resource availability. Our page cache exporting technique is designed with monolithic kernels in mind. It is an explicit endorsement of the memory mapping to backing store model. Page cache exporting is part of a mature and fully developed modern monolithic kernel and is not an alternative operating system or hyper-visor like substrate.

Page cache exporting utilizes some features from other micro-kernels. The idea of trapping into a privileged library with less privilege than the main kernel by utilizing the hardware segment descriptors of the CPU is an alteration of one of the existing ideas in Pebble OS: portals. Pebble OS portals allow applications to transfer control to other applications using special automatically generated trap routines. Like Synthesis, Pebble OS generates these trap routines automatically based on a specification language that has semantic restrictions that protect the system (e.g., from infinite loops). Page cache exporting utilizes a trap instruction only as a practical way to context switch efficiently to a shared user-level storage stack to access that server's page cache for a lookup or write. The privileged library must be authorized by the main kernel with the seal system call discussed earlier. Granting privilege to new libraries can only be done by re-compiling the seal module to include the new library. No other security mechanisms or context switching primitives are needed or employed by page cache exporting.

j) External paging: External paging is an ongoing field of research trying to find a better abstraction between processes and memory management. In a similar vein to micro-kernel approaches, the majority of this work focuses on introducing new operating systems with alternative memory manager designs.

The issue of giving applications efficient page-caching is long-standing. Stonebraker in 1981 discusses the inappropriateness of LRU (default in Linux) page replacement for database workloads [40]. Several architectures to repair this have been proposed, including external paging in Mach [13] [10], an extension to a communication-oriented kernel by Young [45] and a further extension to Mach external paging by McNamee [7]. Other works include a scheme to partition physical memory, and grant user applications direct control in a novel kernel (V++) by Harty [15]. Haeberlen and Elphinstone discuss a combination of MAP and GRANT that provides a super-set of the functionality offered by Linux's splice system call [21].

Mach external paging is an alternative to `mmap` that allows micro-kernel servers the ability to map pages into other processes [10]. Clients make requests to servers via RPC and receive an authorization token in reply. This token can be exchanged with the memory manager server to have a page mapped into the client's address space. Eviction policies are not configurable in this environment. McNamee proposes to solve this by using RPC to signal page faults and allow processes the ability to specify eviction policies. Unlike McNamee's approach, we do not require RPC and instead use a shared memory approach which they deemed too complex to implement. Haeberlen and Elphinstone propose an extension similar to `splice` with the exception that processes can gift pages to other processes, not just the kernel, and that processes will receive a message from the kernel on a page fault. Evaluation of this work was scant. Unlike Harty, we do not physically partition the memory, but let the kernel retain full control over all aspects of memory, and instead use a soft signal-handler to signal page-writeback to maintain good disk throughput, and use a swapper to ensure liveness and performance guarantees to applications when necessary (by swapping out pages that need to be evicted but are dirty). Unlike McNamee, we do not allow alternative eviction policies to be selected, and this is a subject of future work.

Our approach is fundamentally different from preexisting works in this area in that we have modified and extended the existing virtual memory implementation of UNIX (Linux) to achieve kernel-like I/O performance, rather than replace it or start again from scratch with generalized approaches. Our focus is on using this technology for user-level storage and file systems in existing commodity operating systems. We are able to abide by a simple architecture that is a better fit for existing UNIX-like operating systems. Unlike much work in this area, we focus on file system and I/O benchmarks, not memory transfer performance or faulting overheads [7, 13, 15, 21].

k) User-level file system support: The file system abstraction is simply one kind of storage stack; however, it is an important one. It is one of the most widely used abstractions to interact with on-disk data. Although there are many user-level file system frameworks, including NFS interceptors, shared-libraries, and even one of our own [38] based on `ptrace`, the framework which behaves most like a native kernel file system is FUSE. The FUSE file system is broken into two parts: (1) an in-kernel file system that behaves like an NFS client, and (2) a user-level library that simplifies the process of creating FUSE daemons that act like NFS servers, responding to this client. They communicate across a shared memory interface. FUSE and NFS interceptors and custom NFS file systems only export the POSIX service requests of processes. FUSE does not have a mechanism for allowing client file systems to participate in write-back of dirty pages, and has no mechanisms to allow file systems to interface with the page cache like kernel-level file systems can [25]. Further FUSE incurs context switching and message passing overheads for most in-cache workloads, this is confirmed by our evaluation. FUSE supports a caching mode that mostly eliminates these overheads, but then the FUSE daemon will not receive every read request, making custom or alternative cache implementations like our object store impossible.

l) System metadata indexing mechanisms and object stores: Our object store system is an example of a storage stack which is considerably different than what a typical VFS provides. We utilize several cache-oblivious indexes to achieve caching of metadata in RAM and do not have `inode` or `dentry` caches.

Existing indexing systems on Linux (e.g., inotify) and other OSs provide user applications with an event queue to signal when a directory or file has changed. User-space indexing systems use these mechanisms but pay heavy message-passing costs.

In hFAD, the authors propose a B-tree index to store offsets to extents and argue that the file system should be reduced to an object store [36]. Their prototype uses FUSE, and it is unclear in their short paper how they will achieve proper recoverability in crash.

Perspective [31], a distributed object-store–based file system with metadata indexing and query capabilities uses FUSE [25] and MySQL; MySQL's InoDB [16] back-end employs write off-loading to handle write bursts and uses a B-Tree to index data. Perspective argues for more flexible namespaces and metadata capabilities. Its performance is limited by its user-space implementation and the authors focus instead on a user-study.

Spyglass [20] optimizes metadata queries on large file systems by improving performance when queries are limited to a file system namespace subtree. Spyglass partitions the file system namespace into subtrees that maintain separate indexes. Each partition index is kept small enough to fit into RAM. Spyglass maintains an index of partition indexes which is structured as a traditional file system tree, using a block-allocation strategy similar to a B-Tree. Spyglass's insertion, delete, and update speed depends on its partition index, which utilizes a B-tree like structure that will scale poorly for inserts and updates compared to a COLA, especially when RAM is full.

m) Alternative fast indexing technologies: The COLA [5] is one example of a write-optimized indexing technology; other write-optimized indexing technologies also exist. The log-structured merge tree (LSM) [28] maintains an in-RAM cache, a large B-Tree called c_1 on disk which is R times larger than the in-RAM cache, and an even larger B-Tree also on disk called c_2 which is R times larger than c_1, ideally. When RAM is full of insertions, they are merged in sorted order into c_1. When c_1 is full, it is merged in sorted order into c_2. Amortized insertion time here is $\mathbf{O}\left(\left(\sqrt(N)\log(N)\right)/B\right)$ [33]. As there are only two trees, query time is optimal $\mathbf{O}\left(\log_{B+1}(N)\right)$ but in practice is slower than a B-Tree as two trees are searched. LSM is a classic write-optimized data structure, but the COLA maintains an asymptotically better amortized insertion time. Rose [33] is an LSM-based database which exploits compression to improve throughput.

Partitioned exponential file trees [17] are similar to LSM trees and like Spyglass, include an optimization for bursts of insertions with a limited range of keys by relying on a partitioning scheme. Such an optimization can be easily added to the COLA.

IV. Evaluation

We tested the performance of our object store based on an exported kernel page cache by running standard intensive file system workloads with the FileBench utility [41]. We analyzed several cache intensive workloads, as well as larger system benchmarks including a video server, a file server, and a web server. We also analyzed the cost of our trapping mechanism compared to standard system calls and FUSE, and evaluated our metadata indexing against standard Linux file systems.

n) Experimental setup: All benchmarks were run on six identically configured machines each with a 2.8GHz Xeon CPU and 1GB of RAM for benchmarking. Each machine was equipped with six Maxtor DiamondMax 10 7,200 RPM 250GB SATA disks and ran CentOS 5.3x86-64 with the latest updates as of December 28, 2009. To ensure a cold cache and an equivalent block layout on disk, we ran each iteration of the relevant benchmark on a newly formatted file system with as few services running as possible. We ran all tests at least three times or until the standard deviation was within 5% of the mean, except where explicitly noted. To compensate for disk geometry and partitioning irregularities, and to take into account the ZCAV effect, all benchmarks were run on newly formatted identically sized 30GiB partitions [43].

We ran several configurations during our benchmarks, including:

- `ext3` is a default EXT3 file system.
- `fuse` has caching disabled and is a pass-through file system mounted on top of REISER 3.
- `xfs` is a default XFS file system.
- `reiserfs` is a default REISER 3 file system.
- `btrfs` is a default B-TREE FS file system.
- `hook` is a pass-through file system using only `reroute` to intercept file system calls, and call down into REISER 3, and then `reroute` back on completion.
- `exp-pc` is our whole storage stack, utilizing `reroute` to trap into our object store, which runs on top of 1FFS, which runs directly on top of the disk device.

The object store was compiled with all optimizations turned on, and so were all benchmarks for all configurations. Only the `hook` and `exp-pc` benchmarks used our modified kernel; all other standard file systems used the version of Linux we forked from during development: 2.6.31.6. The in-kernel watermark for the `set_vma` system call was set to 500MiB. Similarly, the dirty page ratio for the Linux kernel on all in-cache configurations was set to 50% to equal the cache size used by our system, and the default 10% for out-of-cache workloads to keep the disk plugged with I/O. Journal flush and dirty page write-back of all file systems was set to 30s. We observed during experimentation that with regularity, we were asked to flush due to memory pressure, and responded promptly by performing a journal commit of our indexes and a full data flush.

o) In-cache, out-cache: All benchmarks that we ran fall into one of two categories: (1) *in-cache*, or (2) *out-of-cache*. Due to the extreme variation in benchmark results across hardware and file systems [34], we focused on in-cache benchmarks where almost all operations can be serviced without accessing the disk, and out-of-cache benchmarks, where almost no operations can be serviced without accessing the disk. Both workloads are important (e.g., Facebook's Haystack [18] relies heavily on memcached [9], a distributed cache). We confirmed the presence or absence of block I/O when appropriate using `vmstat`. We also ensured the working set size was large enough that random accesses almost always had to access the disk.

p) Interposition: The cost of only intercepting user-level file system calls and passing them down to the lower file system can be very high. It is important to minimize this cost as it establishes an upper-bound on in-cache throughput where context switching is a critical path. Our `reroute` system call re-directs every relevant user-level system call to the privileged shared library. To precisely measure the cost of an equivalent context switch to the user-level FUSE daemon, we enabled its `-odirect_io` feature which simply forwards every user-level system call to the user-level file system, rather than utilizing its in-kernel caches. This is identical to what `reroute` does. Without receiving every file system call, a user-level storage server can not implement its own cache (as we have to), or modify the cache's semantics, as it would implicitly be using the kernel's cache.

The rest of Section IV discusses experimental results concerning in-cache performance (Section IV-A), and out-of-cache (Section IV-B) performance.

A. Shared Memory In-Cache Performance

The Hotset workload consists of randomly reading 128B from a randomly selected file from a set of 65,536 4KiB files. An operation is opening a file, reading 128B, and closing it. The total workload of 256MiB easily fit in cache for all file systems. We confirmed this by monitoring zero block I/O (except for periodic write-back) and decreasing the workload size until throughput did not increase. Figure 4 shows that `hook` (11,723 ops/sec) was equivalent in performance to `reiserfs` (11,721 ops/sec), which implies that `reroute` has a negligible overhead in this workload. On the other hand, `fuse` (7,960 ops/sec) was only running at 65% of the throughput of `reiserfs` which it is based on. Every single operation including `open`, `read`, and `close` must call down into the FUSE kernel file system, queue a message, wait for the daemon to be scheduled, call down into the lower file system (REISER 3), and then reverse the process to return to the caller. We saw that other kernel level file systems performed comparably to `exp-pc` for both single and multi-threaded performance.

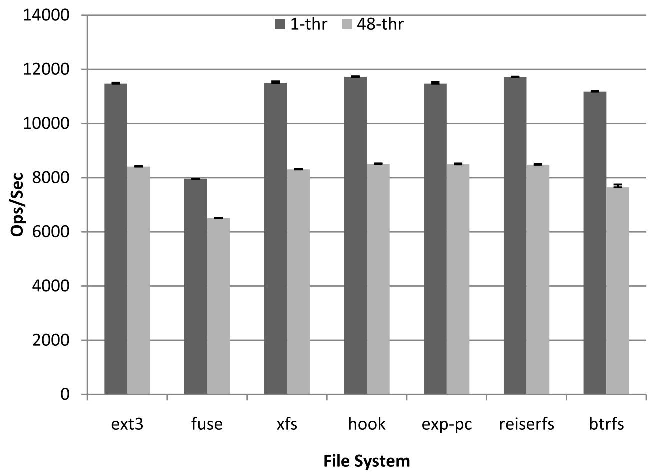

Fig. 4. Hotset benchmark.

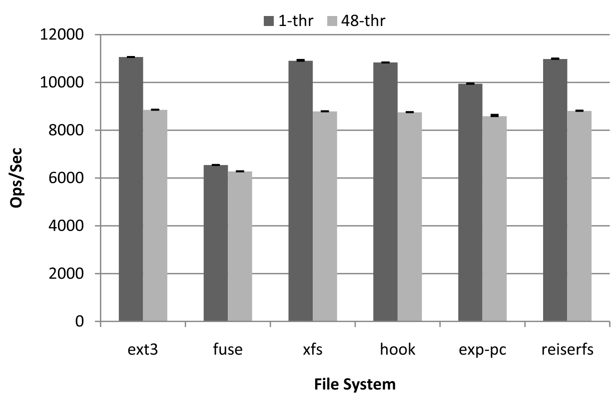

Fig. 5. In-cache webserver workload.

Our multi-threaded run consisted of the same workload as above, with 48 threads in parallel. On our hardware setup, multi-threaded performance was worse for all file systems (a 27% drop in performance) as our machine has a single core, and the overhead of locking and scheduling was eating away at useful good-put through our single core. Due to its shared-memory exported page cache, `exp-pc` was able to maintain equivalent performance to the native file system. We believe the primary bottleneck in the Hotset workload was the VFS, since all operations accessed files and `inodes` which should have been cached. This explains the almost uniform performance across all native kernel file systems and `exp-pc`. The largest discrepancy in performance among the kernel file systems was about 10% between `reiserfs` and `btrfs`.

q) Read-heavy system workload: The Webserver workload represents a read-heavy workload where again the working set was confirmed to fit into cache. In the webserver workload, threads open, do a whole read, and close ten files, and then append a 16KiB block to a log file. Each open, whole read, close and append are an operation. The in-cache workload had 1,000 files of 4KiB. The multi-threaded workload had 48 threads. Figure 5 shows that `exp-pc` was running at 9,940 ops/sec, where `reiserfs` and `ext3` respectively maintained 10,980 and 11,061 ops/sec, `exp-pc` incurred a 10% overhead. We measured `exp-pc` with no memory transfers in the page cache, and throughput increased to 10,788 ops/sec. Currently, when flushing, we stop all readers and writers, and appends are dirtying enough pages to cause the synchronous write to disk on flush to damage our throughput. Readers are stopped for a shorter period than writers during flush due to our flushing protocol, so in the multi-threaded workload this affects our performance less.

`Fuse` at 6,541 ops/sec for single-threaded, and 6,276 ops/sec for multi-threaded was again bottlenecked on context switches, which were slowing down every op in this in-cache workload. As in Hotset, the other file systems were primarily doing little, with the Linux VFS handling most of this workload.

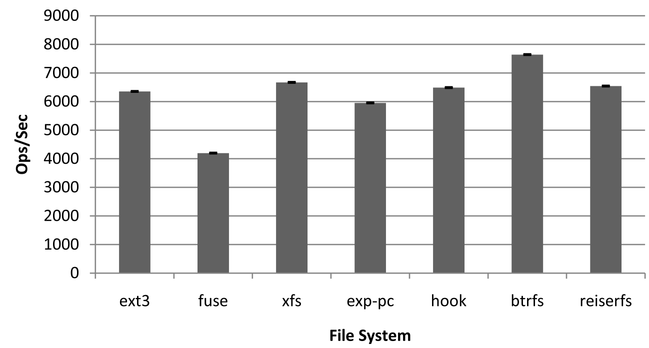

Fig. 6. In-cache fileserver workload.

r) Mixed read and write system workload: Unlike the webserver workload, the in-cache fileserver workload stressed an equal number of reads and writes, as well as creating and deleting files within subdirectories. The fileset consisted of 1000 files, each 64KiB large. Figure 6 shows that `fuse` incurs a 34% overhead on `ext3` since it suffers from the same context-switching bottleneck as in other in-cache workloads. `exp-pc` suffered a 6% overhead on `ext3` due to writers (including unlink and create) stopping due to flushing. The 1% overhead of `hook` on `reiserfs` demonstrates that `reroute` has negligible overhead in both read-heavy, and mixed read-write in-cache workloads that include unlinks and creates.

B. *Out-of-Cache Performance*

For out-of-cache performance, we stressed the on-disk format of the file system, as well as the efficiency of the Linux `mmap` implementation's read-ahead, faulting, and writing compared to direct in-kernel block device requests.

The videoserver workload consisted of a single uploading thread, pushing a queue of up to 194 new media files to the server, while one or more clients downloaded a different media file at the same time from a set of 32 pre-existing media files. Each media file was 50MiB. `1-thr` is a configuration with one client, `4-thr` is with four clients, and `48-thr` is with forty-eight clients. Figure 7 shows that all systems have equivalent

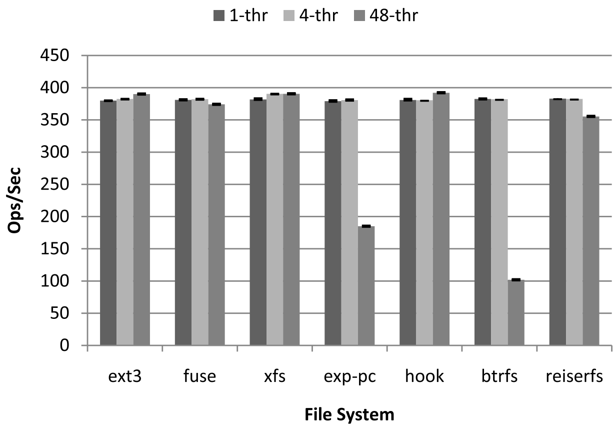

Fig. 7. Out-of-Cache videoserver system workload.

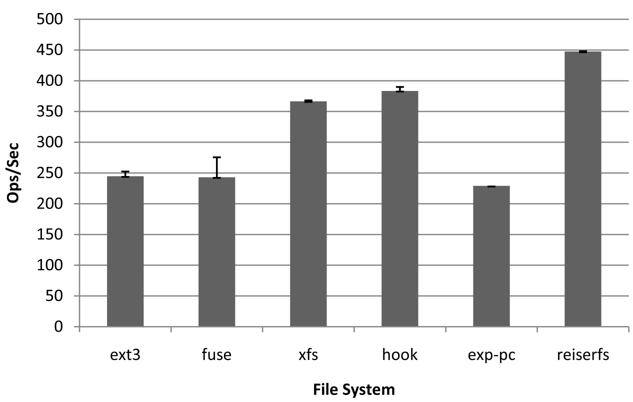

Fig. 8. Out-of-cache webserver workload.

performance (380 ops/second). `fuse` was not slower here because context-switching was no longer the bottle neck: disk I/O performance was. `exp-pc` uses extents for serially written files, so blocks of video files are mostly contiguous, a common design decision used by the other file systems. For small numbers of reader threads, the performance of `exp-pc` was good since `mmap` read-ahead was still able to pre-fetch blocks; but as the number increased to 48, `mmap` read-ahead stopped working effectively and our throughput decreased by 51%.

s) Read-heavy out-of-cache workload: The out-of-cache webserver workload was identical to the in-cache workload, but stressed I/O-bound random reads. Due to high variance in disk performance, `fuse` had a standard deviation of 13% of the mean. We used 100,000 files of 32KiB each, and appends of 16KiB. We found that during our runs both `ext3` (244 ops/second) and `exp-pc` (228 ops/second) spent 9.8ms in each read operation as seen in Figure 8. This latency was very close to the disk-arm latency to perform a seek on our hardware. We monitored block I/O and found a steady stream of block reads, not at disk throughput. This implies that both systems were performing a block read on each read system call. `Reiserfs` (447 ops/second) is designed for random reads to a large number of small files due to its global S+-tree it can quickly perform lookups on objects, and can keep all the parent nodes in this tree cached in RAM. `Xfs` (366 ops/second) has a similar advantage. Both of these file systems were able to fit a larger amount of the workload in cache. Our `hook` instrumentation induced a 14% overhead as the additional call to `reroute` took long enough that there was time for the kernel to evict more pages to make room for new reads, decreasing the chance that it can avoid a disk I/O on a read even further. `Fuse` at a 46% overhead over `reiserfs` magnified this problem with even longer context switching times, further harming throughput.

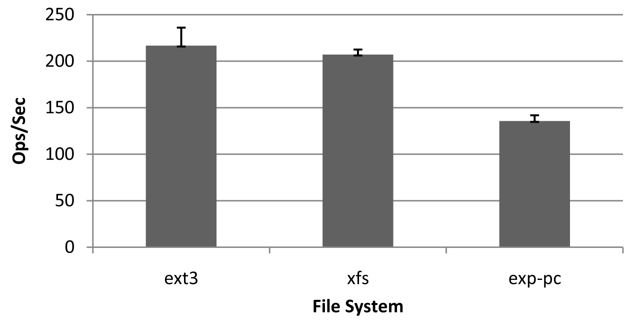

Fig. 9. Out-of-Cache fileserver workload.

t) Mixed read and write out-of-cache workload: In this benchmark `ext3` had a standard deviation within 8% of its mean. We found that read performance of `exp-pc` was competitive, but write performance was lacking due to the kernel's current inability to distinguish between full page overwrites, and partial page writes (which require preceding read faults). Therefore in all write workloads, `exp-pc` performed reads of pages before dirtying them with writes. This was confirmed by temporarily modifying the kernel to not fault in pages on writes. Before this change, we would see disk-throughput block reads (50,000 to 60,000 blocks per second) preceding large flushes of our page cache. Afterward, we saw no block reads (except on read cache misses). This caused an increased lag for flushes that were performing writes. In Figure 9 we see that reading pages before writing them, and longer flush times induced a 37% overhead over `ext3`, and a 34% overhead over `xfs`.

u) Metadata insertion performance: The performance argument for user-level storage is workload specialization. While `exp-pc` performs extremely competitively in other workloads compared to native kernel file systems, it is specialized for rapid metadata updates. We discuss the *blind create* optimization. A blind create is a create that does not return an error if the file already exists, or if a part of the path does not exist. This allows us to avoid performing an initial lookup during create and defer error handling until merge, at which point the error condition can be appended to a log. The application level semantics are changed as the error is not reported during file creation but rather asynchronously later,

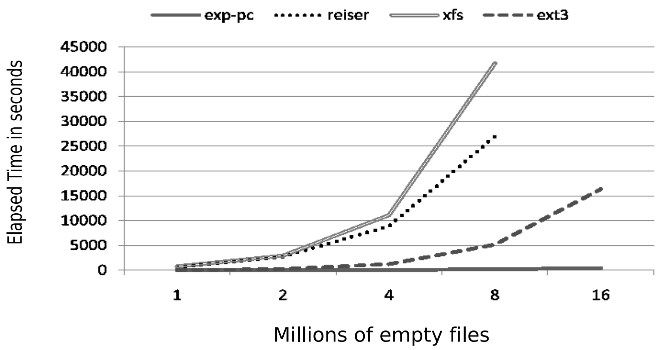

Fig. 10. Out-of-Cache object creation and lookup.

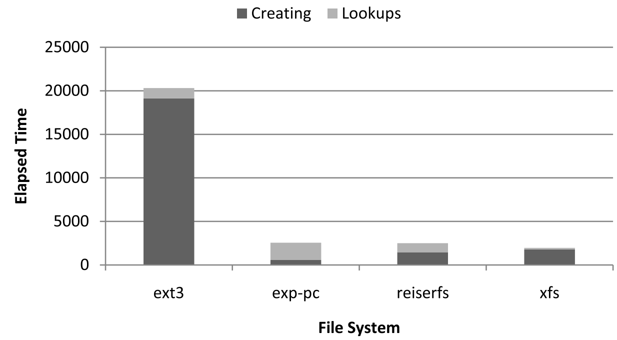

Fig. 11. Partially out-of-cache object creation, delete, and lookup.

during merge. For some applications (e.g., those that rely on `O_EXCL`) this is not a viable option.

In this workload we created 1,000,000 to 16,000,000 files with randomly generated names in the same directory, and then performed 100,000 random lookups within this directory. Due to the excessive amount of time to perform runs, we only ran each benchmark once. Since we intend to measure only the performance of indexing, we disabled journaling in all file systems. In `ext3` and `xfs` this means using a RAM device for the journal, and `reiserfs` and `exp-pc` were configured to have their journals disabled. Further `xfs` was configured to aggressively batch super-block updates. Figure 10 shows the performance of `xfs` and `reiserfs` started to decrease rapidly as the file set size grew beyond 1 million files. This workload forced each file system to index its `dentry` to `inode` mappings or suffer intractable lookups. For these file systems which are using read-optimized indexing, this incurred random writes. Our write-optimized indexing is sorting and merging, and is better exploiting disk throughput. Figure 10 shows that at 1 million files we were 20 times the speed of `ext3`, 58 times the speed of `reiserfs`, and 69 times the speed of `xfs`. These results are similar to those found in the COLA paper when comparing random B-tree insertions to 2-COLA insertions [5], and to our own results in previous experiments with write-optimized structures. At 4 million, we were 62 times faster than `ext3`, 150 times faster than `reiserfs`, and 188 times faster than `xfs`. For 8 million, `reiserfs` was 163 times slower than `exp-pc`, and `xfs` was 262 times slower than `exp-pc`. Runs of 16 million took more than 20 hours to complete. The `exp-pc` configuration inserted 16 million random keys in 422 seconds. It performed 100,000 lookups afterward in 676 seconds. The massive performance delta is the difference between serial reads and writes and more efficient use of cache for inserts and updates, and random reads and writes with very inefficient cache use. XFS utilizes a B+-tree for its `inodes`, as well as its free extents and `dentries`. REISER 3 uses one B+-tree for everything, with the primary ordering being the directory id, and subsequent orderings being the object id, and the offset in the object. This is to induce grouping by directory on the disk. This allows REISER 3 to perform its inode allocation, dentry insertion, and block allocation in the same leaf node. XFS must update multiple leaf nodes for each of its trees, inducing additional random writes per create. EXT3 allocates `inodes` serially, but due to the `dir_name` option uses a B-tree to store mappings of hashes of path components to `inode` numbers, and this B+-tree will induce similar random writes induced by the stress of the other file systems.

To compare lookup performance, we ran another workload where 20,000,000 path names were inserted in stripes, which is much closer to sorted order. In Figure 11 We see an immediate increase in performance in all the B+-tree file systems compared to `ext3`. Due to the pseudo-sorted order of insertions the B+-tree of the indexing file systems is making much better use of its cache, staging updates in the same leaf nodes before writing them out. `Reiserfs` and `xfs` are now inserting within 1,456 and 1,777 seconds each, compared to `exp-pc` performing its insertions in 571 seconds. Now `exp-pc` is only 3.1 times faster than `xfs` and 2.5 times faster than `reiserfs`. We are still seeing a 33.5 difference for `ext3` though. Its inserting hashes of path components, so the insertion workload still appears random to `ext3` and it has comparable performance to the previous workload. Lookups are different though. We saw `xfs` as the clear winner with 175 second lookup time for 100,000 random lookups, and `reiserfs` and `ext3` with 1043 and 1187 second lookup times each. The `exp-pc` pays for its fast inserts with slower lookups, weighing in with an elapsed time of `1187` seconds. Lookup performance is bounded by random block read performance, and the B+-tree based file systems have large fan-out and can keep all or almost all of the parent nodes in their trees in RAM, performing only a single block-read per lookup. Our 2-COLA based implementation uses $log_2 N$ binary trees, and due to the smaller fan-out and multiple trees, can not contain as many parent nodes in RAM. This causes `exp-pc` to exit the cache sooner, and to perform more block reads when out of the cache.

V. CONCLUSIONS

The argument for user-level storage is a non-POSIX interface, ease of development, separability from the kernel, and optimizing performance for important workloads. We have shown that one can implement an efficient file system that is as fast as or comparable to in-kernel file systems for standard workloads both in and out of cache, and yet is optimized for high-throughput metadata insertions and updates. We have shown how to do this without compromising the security of the kernel, or of the data cached in shared memory. The implication of this research is that future file systems designers should seriously consider development at the user-level, that user-level storage services can have kernel-like efficiency, and that future operating system design should consider focusing on more general access to the page cache and block device, rather than a host of different POSIX file systems.

v) Future Work: We plan to further develop our write-optimized indexing approach for metadata storage. Indexing in file systems is not new, however designing around a cache-oblivious architecture poses new challenges, but could reduce the complexity of scalable storage systems immensely. We plan to further explore the modularity and flexibility of our system by implementing a re-configurable user-level VFS composed of interchangeable modules where caches with different performance characteristics for different workloads can be mixed and matched. We plan to develop a simple distributed file system based on our cache-oblivious technology and exploit exported page caching to maximize our efficiency and compare it with existing distributed file systems like Google FS [12]. We expect our performance will be equivalent for a RAM and disk machine where the index fits in RAM, but we are interested in comparing performance with machines with additional types of media or with workloads where the index is larger than what fits in RAM. We also plan to benchmark existing user level file systems using exported kernel page caching and measure their performance improvements.

We would like to acknowledge the useful assistance and help of Zhichao Li.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proc. of the Summer USENIX Technical Conf.*, pp. 93–112, Atlanta, GA, Jun. 1986.

[2] Eric Anderson and Joseph Tucek. Efficiency matters! *SIGOPS Oper. Syst. Rev.*, 44:40–45, Mar. 2010.

[3] The Apache Foundation. Hadoop, Jan. 2010. http://hadoop.apache.org.

[4] Apple, Inc. *Mac OS X Reference Library*, 2009.

[5] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J.Nelson. Cache-oblivious streaming b-trees. In *SPAA '07: Proc. of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pp. 81–92, New York, NY, USA, 2007

[6] J. L. Bertoni. Understanding Solaris Filesystems and Paging. Technical Report TR-98-55, Sun Microsystems Research, Nov. 1998. http://research.sun.com/research/techrep/1998/abstract-55.html.

[7] K. Armstrong D. McNamee. Extending the mach external pager interface to accommodate user-level page replacement policies. In *Proc. of the USENIX MACH Symposium*, 1990.

[8] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pp. 137–150, 2004.

[9] B. Fitzpatrick. Memcached. http://memcached.org, Jan. 2010.

[10] Free Software Foundation. External pager mechanism. http://www.gnu.org/software/hurd/microkernel/mach/external_pager_mechanism.html, May 2009.

[11] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz. The Pebble Component-based Operating System. In *Proc. of the 1999 USENIX Annual Technical Conf.*. USENIX Association, 1999.

[12] S. Ghemawat, H. Gobioff, and S. T. Leung. The Google file system. In *Proc. of the 19th ACM Symposium on Operating Systems Principles*, pp. 29–43, Bolton Landing, NY, Oct. 2003.

[13] D. B. Golub and R. P. Draves. Moving the Default Memory Manager out of the Mach Kernel. In *Proceeding of the Second USENIX Mach Symposium Conf.*, pp. 177–188, Monterey, CA, Nov. 1991.

[14] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA, 1993.

[15] Kieran Harty and David R. Cheriton. Application-controlled physical memory using external page-cache management. In *ASPLOS-V: Proc. of the fifth international conference on Architectural support for programming languages and operating systems*, pp. 187–197, New York, NY, USA, 1992

[16] InnoDB. Innobase oy. www.innodb.com, 2007.

[17] C. Jermaine, E. Omiecinski, and W. G. Yee. The partitioned exponential file for database storage management. *The VLDB Journal*, 16(4):417–437, 2007.

[18] N. Kennedy. Facebook's photo storage rewrite. http://www.niallkennedy.com/blog/operations, Apr. 2009.

[19] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, Jun. Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proc. of the 22nd ACM Symposium on Operating Systems Principles*, pp. 207–220, Big Sky, MT, USA, Oct 2009

[20] A. W. Leung, M. Shawo, T. Bisson, S. Pasupathy, and E. L. Miller. Spyglass: Fast, scalable metadata search for large-scale storage systems. In *FAST '09: Proc. of the 7th USENIX conference on File and Storage Technologies*, Berkeley, CA, USA, 2009.

[21] Jochen Liedtke, Uwe Dannowski, Kevin Elphinstone, Gerd Lieflander, Espen Skoglund, Volkmar Uhlig, Christian Ceelen, Andreas Haeberlen, and Marcus Volp. The l4ka vision, Apr 2001.

[22] M. Mammarella, S. Hovsepian, and E. Kohler. Modular data storage with anvil. In *SOSP '09: Proc. of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 147–160, New York, NY, USA, 2009

[23] H. Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Computer Science Department, Columbia University, 1992.

[24] J. G. Mitchel, J. J. Giobbons, G. Hamilton, P. B. Kessler, Y. A. Khalidi, P. Kougiouris, P. W. Madany, M. N. Nelson, M. L. Powell, and S. R. Radia. An overview of the Spring system. In *CompCon Conf. Proc.*, San Francisco, CA, Feb. 1994. CompCon.

[25] D. Morozhnikov. FUSE ISO File System, Jan. 2006. http://fuse.sf.net/wiki/index.php/FuseIso.

[26] K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-aware storage systems. In *Proc. of the Annual USENIX Technical Conf.*, pp. 43–56, Boston, MA, Mar./Apr. 2006.

[27] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the sync. In *Proc. of the 7th Symposium on Operating Systems Design and Implementation*, pp. 1–14, Seattle, WA, Nov. 2006.

[28] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Inf.*, 33(4):351–385, 1996.

[29] Oracle. Database administrator's reference. http://download.oracle.com/docs/cd/B19306_01/server.102/b15658/tuning.htm, Mar. 2009.

[30] Richard F. Rashid and George G. Robertson. Accent: A communication oriented network operating system kernel. In *In Proc. 8th Symposium on Operating Systems Principles*, pp. 64–75, 1981.

[31] B. Salmon, S. W. Schlosser, L. F. Cranor, and G. R. Ganger. Perspective: Semantic data management for the home. In *FAST '09: Proc. of the 7th USENIX conference on File and Storage Technologies*, Berkeley, CA, USA, 2009.

[32] R. Sears and E. Brewer. Stasis: Flexible Transactional Storage. In *Proc. of the 7th Symposium on Operating Systems Design and Implementation*, Seattle, WA, Nov. 2006.

[33] R. Sears, M. Callaghan, and E. Brewer. Rose: Compressed, log-structured replication. In *Proc. of the VLDB Endowment*, volume 1, Auckland, New Zealand, 2008.

[34] P. Sehgal, V. Tarasov, and E. Zadok. Evaluating Performance and Energy in File System Server Workloads extensions. In *Proc. of the Eighth USENIX Conf. on File and Storage Technologies*, pp. 253–266, San Jose, CA, Feb. 2010.

[35] M. Seltzer, Y. Endo, C. Small, and K. Smith. An introduction to the architecture of the VINO kernel. Technical Report TR-34-94, EECS Department, Harvard University, 1994.

[36] M. Seltzer and N. Murphy. Hierarchical file systems are dead. In *Proc. of the 12th Workshop on Hot Topics in Operating Systems*, 2009.

[37] R. Spillane, R. Sears, C. Yalamanchili, S. Gaikwad, M. Chinni, and E. Zadok. Story Book: An Efficient Extensible Provenance Framework. In *Proc. of the first USENIX workshop on the Theory and Practice of Provenance*, San Francisco, CA, Feb. 2009.

[38] R. Spillane, C. P. Wright, G. Sivathanu, and E. Zadok. Rapid File System Development Using ptrace. Technical Report FSL-06-02, Computer Science Department, Stony Brook University, Jan. 2006.

[39] R. P. Spillane, S. Gaikwad, E. Zadok, C. P. Wright, and M. Chinni. Enabling transactional file access via lightweight kernel extensions. In *Proc. of the Seventh USENIX Conf. on File and Storage Technologies*, pp. 29–42, San Francisco, CA, Feb. 2009.

[40] Michael Stonebraker. Operating system support for database management. *Commun. ACM*, 24(7):412–418, 1981.

[41] Sun Microsystems. Filebench. www.solarisinternals.com/si/tools/filebench.

[42] Inc. Sun Microsystems. *man pp. section 2: System Calls*. Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, CA 95054, USA, 2009.

[43] R. Van Meter. Observing the effects of multi-zone disks. In *Proc. of the Annual USENIX Technical Conf.*, pp. 19–30, Anaheim, CA, Jan. 1997.

[44] C. P. Wright, R. Spillane, G. Sivathanu, and E. Zadok. Extending ACID Semantics to the File System. *ACM Transactions on Storage (TOS)*, 3(2):1–42, Jun. 2007.

[45] M. Young. Exporting a user interface to memory management from a communication-oriented operating system. In *PhD Thesis*, Pittsburgh, PA, USA, 1989. Carnegie Mellon University.

[46] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, lfar Erlingsson, Pradeep Kumar, and Gunda Jon Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language.