Transparent Heterogeneous Backing Store for File Systems

Benjamin Marks and Tia Newhall
Computer Science Department
Swarthmore College
Swarthmore, PA, USA
(bmarks1, newhall)@cs.swarthmore.edu

Abstract*—We present Nswap2L-FS, a fast, adaptable, and heterogeneous storage system for backing file data in clusters. Nswap2L-FS particularly targets backing temporary files, such as those created by data-intensive applications for storing intermediate results. Our work addresses the problem of how to efficiently and effectively make use of heterogeneous storage devices that are increasingly common in clusters. Nswap2L-FS implements a two-layer device design. The top layer transparently manages a set of bottom layer physical storage devices, which may include SSD, HDD, and its own implementation of network RAM. Nswap2L-FS appears to node operating systems as a single, fast backing storage device for file systems, hiding the complexity of heterogeneous storage management from OS subsystems. Internally, it implements adaptable and tunable policies that specify where data should be placed and whether data should be migrated from one underlying physical device to another based on resource usage and the characteristics of different devices. We present solutions to challenges that are specific to supporting backing filesystems, including how to efficiently support a wide range of I/O request sizes and balancing fast storage goals with expectations of persistence of stored file data. Nswap2L-FS defines relaxed persistence guarantees on individual file writes to achieve faster I/O accesses; less stringent persistence semantics allow it to make use of network RAM to store file data, resulting in faster file I/O to applications. Relaxed persistence guarantees are acceptable in many situations, particularly those involving short-lived data such as temporary files. Nswap2L-FS provides a persistence snapshot mechanism that can be used by applications or checkpointing systems to ensure that file data are persistent at certain points in their execution. Nswap2L-FS is implemented as a Linux block device driver that can be added as a file partition on individual cluster nodes. Experimental results show that file-intensive applications run faster when using Nswap2L-FS as backing store. Additionally, its adaptive data placement and migration policies, which make effective use of different underlying physical storage devices, result in performance exceeding that of any single device.

I. Introduction

It is common for both sequential and parallel applications to generate temporary intermediate result files. For instance, optimizing compilers typically create temporary files containing the output of intermediate steps. Out-of-core algorithms for large data sets, such as external merge sort, use temporary files to store partial output. Similarly, MapReduce [1] computations generate temporary files of intermediate (key, value) pairs. Often, these temporary files are stored on a local file partition backed by hard disk drives (HDD).

Given the potential size of data stored in these intermediate files and the slow speeds of HDDs, the overhead of writing and reading intermediate files can be substantial, delaying the completion of an application. Nswap2L-FS is our solution for providing fast backing storage for file data in clusters and networked systems. It combines a heterogeneous set of storage options available in the cluster, including local and remote devices such as flash or disk and remote network RAM in order to provide superior performance to any single device. To the node operating system (OS), Nswap2L-FS appears as a single, fast, random access backing store device.

As node RAM capacities continue to grow, significant amounts of RAM remain idle in clusters, even during times of heavy utilization [2], [3]. Network RAM is the concept of allowing nodes in a cluster to use the idle memory of other cluster nodes as fast backing store. Because network speeds plus RAM access speeds are typically faster than flash SSD, and orders of magnitude faster than HDD devices, network RAM is increasingly attractive as an option for fast storage.

The main drawback of using this network RAM as secondary storage is that its volatility does not satisfy traditional persistence guarantees of file system data. However, we note that for some types of files, strict persistence constraints on individual file writes may not be necessary, and for many, the trade-off in weaker persistence constraints for faster file I/O is desired.

Temporary files are one example where faster file I/O may be favored over strict file persistence guarantees. Often data intensive applications create temporary files containing partial results that are processed by later phases of the computation. In many cases, generated temporary files are short lived and do not need persistence [4]. Additionally, many large parallel and data intensive applications incorporate fault tolerance to be robust to node failure and data loss. For instance, MapReduce uses backup tasks to allow computation to continue even in the presence of degraded performance and often starts multiple copies of the same task, allowing some of them to fail. Checkpointing is also commonly used by long-running applications to create a persistent snapshot. In such an environment, enforcing persistence on every file block write adds unnecessary overhead

to file I/O. Instead, Nswap2L-FS can write file blocks to fast volatile network RAM, speeding up file I/O and subsequently reducing application runtime.

While network RAM can provide higher performance than many other cluster storage devices, HDD and SSD drives are still useful. In particular, given that network RAM is not always available and that it has varying capacity based on nodes' workloads, a storage system solely using network RAM many not always be possible. Further, because RAM is volatile, file storage on network RAM alone is not sufficient if the file system it backs desires persistent storage.

In clusters there is increasingly a wide range of storage devices available, including flash SSD, magnetic HDD, and networked storage devices. A system that can incorporate network RAM alongside other cluster storage can more easily adapt to changing cluster conditions, using network RAM when available and falling back on local devices as needed. Additionally, a system incorporating many devices has the ability to move data between devices to redistribute with a goal of increasing I/O parallelism and performance.

Nswap2L-FS is our adaptable, scalable, and efficient backing storage system for providing fast access to temporary files. It builds off our earlier work exploring backing store for swap data in clusters [5]. While both transparently manage multiple heterogeneous storage technologies with the goal of providing superior performance, Nswap2L-FS extends our previous work in significant ways. In particular, the underlying architecture differs substantially in order to support high performance I/O for requests of any size, which is necessary for backing file systems. Additionally, Nswap2L-FS adds a persistent storage mechanism and optional encryption of remotely stored data.

The main goals of our system are: to provide fast backing storage for file system data by combining heterogeneous storage devices that are increasingly common in clusters; to hide the complexity of managing heterogeneous storage from node OSs, allowing this storage to be used in a wide range of contexts without modifying OS subsystems or applications; and to provide an adaptable system that easily incorporates new storage devices and provides tunable and adaptable policies to dynamically take advantage of available cluster resources.

Nswap2L-FS hides the complexity of managing the set of heterogeneous storage devices from cluster node operating systems. To node OSs, Nswap2L-FS appears as a single, fast random access device that can be formatted as a local file partition, freeing OS file system implementations from having to be optimized for different sets of backing storage media available in a cluster. When the OS issues file read, write, or discard requests to the Nswap2L-FS device, Nswap2L-FS implements data placement policies for choosing underlying devices based on dynamic performance information, cluster load, or user input. Additionally, it incorporates mechanisms for moving data from one device to another, allowing for policies that exploit the heterogeneous set of available storage resources. For instance, when network RAM is scarce, moving data away from network RAM to a slower local device can increase performance by keeping network RAM free for future write operations. This data movement is transparent to node OSs.

Nswap2L-FS is optimized for fast file I/O specifically targeting temporary file storage. Its normal operation does not guarantee persistence of individual file writes—all internal metadata are stored in memory, and written data may be stored in the memory of remote nodes. For applications that desire stricter persistence guarantees, Nswap2L-FS provides a persistence interface via `/sys`. Applications can use it to create a snapshot of the file partition backed by Nswap2L-FS, or to restore Nswap2L-FS to an existing snapshot of earlier state.

Nswap2L-FS's persistence interface represents a trade-off in strict persistence of individual file block writes for faster file I/O access speed. For temporary file systems in particular, this trade-off is desirable since long term storage is not required. Nswap2L-FS's persistent snapshot mechanism can be used by a checkpointing system as needed.

Nswap2L-FS is designed to be an adaptable and dynamic backing store device. To that end, we have built it with many extensible components that can support a rich set of flexible policies to achieve high performance. For instance, underlying devices can be added on the fly, profiling information of each storage device can be collected, as well as access frequency and recency of each block. These features are enabled and accessed through an extensive `/sys` interface.

The building blocks of our system enable a wide range of potential policies to most effectively use heterogeneous storage. For instance, device performance data could be integrated into placement policies to dynamically prioritize page placement and prefetching data between underlying physical devices. Its access pattern data could be used to identify hot pages to preferentially place on faster devices.

This paper focuses on our first implementation of Nswap2L-FS, describing initial experiments validating the benefits of its design and main goals. Our initial work opens up a rich design space of potential policies within Nswap2L-FS. We believe we have only scratched the surface and are excited to continue exploring these directions in our future work.

II. Related Work

Since the 1990s, researchers have noted the uneven distribution of workloads in computing clusters [6], [7], resulting in a substantial fraction of the computing power and memory usually being idle [2]. More recent studies [3] note that even though data-intensive computing has lead to higher RAM utilization (with a mean reported near 80% in some large data centers), there is a high degree of variability in RAM utilization and server load in these systems. Even in the era

of Big Data, there is still a large amount of idle RAM, and investigating how this can be used to improve performance is an area of ongoing research [8]–[11].

Much research has investigated the potential utility of this idle memory, resulting in systems that use remote idle RAM as backing store for swapping [12]–[14], for cooperative caching [15], [16], or as an extension of local memory [17], [18]. At the same time, researchers have investigated how to effectively incorporate RAM into systems in order to balance cost and performance. Graefe [19] concludes that with larger RAM sizes and lower costs, data accessed every 5 hours should be placed in RAM, suggesting that many intermediate, temporary files could be stored in RAM for their entire lifetime.

As parallel workloads have become more common, and the amount of data stored and processed has grown, some have examined how network RAM could be used to speed up file accesses, effectively incorporating remote RAM into the memory hierarchy as a cache for file system data between local memory and disk access [15], [20]. Some systems are more specialized: optimizing distributed caching for Hadoop [21], [22] or for commercial query accesses [16].

While caching greatly improves performance, the large disparity between memory and disk access times means that cache hit rates must be exceptionally high in order to provide performance comparable to using the cache alone [23]. In this vein, recent work has expanded the role of RAM from caches to primary store for file systems. The goal of these file systems is to provide performance as if the entire file system were in volatile memory, while still providing the reliability and persistence guarantees of traditional file systems backed by SSDs or HDDs. NOVA [24] uses non-volatile memory (NVM) for persistent storage of data while constructing non-persistent indices in DRAM in order to increase performance. Tachyon [4] proposes a checkpoint and re-execution model, where all data are stored in DRAM, with periodic flushing to persistent storage. If temporary files are created, used, and deleted before the flush to persistent storage, there is no I/O overhead associated with these files. Triple-H [25] outlines a heterogeneous storage system incorporating RAM (for fast I/O) alongside SSDs (for caching / staging) and HDDs (for long term persistence). The motivation is similar to Nswap2L [5] and Nswap2L-FS, although Triple-H provides persistence and focuses solely on speeding up Hadoop jobs, while Nswap2L-FS targets any application that generates temporary files.

The increasing size and decreasing cost of memory has led some to claim that "RAM is the new disk," with disk replacing tape storage [26]. This has motivated file systems based entirely in memory. RAMCloud [23] is a DRAM based storage system for key-value pairs in cloud systems. Primary storage for all file data is in the aggregate RAM of thousands of servers. Data durability is provided by storing copies of file data on slower persistent storage such as disk. MemEFS [27] implements a similar distributed memory file system with dynamic data redistribution based on the cluster size and load, but does not describe strategies for durability. Nswap2L-FS is similar to this work in that it implements network RAM as one of its underlying devices storing file data. However, it differs in that Nswap2L-FS is designed for smaller clusters and networked systems, adapts to changes in nodes' RAM usage, and manages a heterogeneous set of underlying devices on which file data are stored. Additionally, Nswap2L-FS implements a device interface on which a file system can be mounted, rather than a file system interface, making it a more versatile backing storage system.

III. Nswap2L-FS Design & Implementation

Nswap2L-FS is backing store for file systems in clusters. It is implemented as a substantial extension of Nswap2L [5], which provides backing store for swap in clusters. Nswap2L-FS makes significant changes to the underlying architecture of Nswap2L to add support for backing file system data. Specifically, it:

- Provides high performance I/O operations for data spanning multiple pages and sub pages.
- Defines a persistence model for file data and implements an interface that can be used by applications to trigger persistent snapshots of all live data and metadata in the Nswap2L-FS partition.
- Adds an interface for encrypting data stored remotely.
- Provides a set of tunable policies for data placement and movement between the underlying devices.

Making best used of heterogeneous storage devices is more difficult in the context of backing files than backing swap. For example, file I/O request sizes can vary from from individual blocks of 512B to multiple contiguous blocks totaling hundreds of KBs. These request size differences lead to problems in mapping granularity of data stored to underlying devices. Nswap2L-FS chooses a 4KB block size, as compromise between the amount of metadata needed store mappings, and the flexibility of placing large request over multiple devices for performance reasons. A 4KB block size is also selected because it is the default Linux page size, allowing Nswap2L-FS to back swap or file systems. Additionally, the expectation of non-volatility of file storage is in conflict with our desire to use fast network RAM storage to speed-up file I/O. This conflict resulted in our persistent snapshot solution that provides a trade-off in persistence of every file block write for faster I/O, which we argue is desired for the target uses of our system.

Main Design Goals: The implementation of Nswap2L-FS, like its predecessor Nswap2L, is driven by the goal of efficiently and transparently combining heterogeneous storage devices with a focus on extensibility and adaptability.

Pursuit of these goals leads to a primary design decision that Nswap2L-FS be implemented at the device, rather

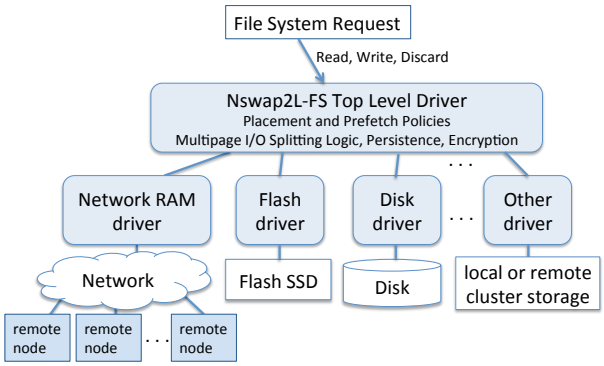

Figure 1. The two level design architecture of Nswap2L-FS (and Nswap2L). The top level device interfaces with the OS. Internally, it contains policies for choosing an underlying device for data placement and for internally prefetching data among the heterogeneous set of underlying devices it manages.

than file system, layer. This choice increases Nswap2L-FS's applicability, allowing it to be used for any type of backing store, not just for backing files.

Given that Nswap2L-FS incorporates multiple devices and variable amounts of remote network RAM, adaptability to changes in available devices or capacities is essential. Its network RAM implementation is built to adapt to changes in cluster resource availability. In addition, Nswap2L-FS defines modular policies, which can be changed on the fly, for dictating where data are placed and for prefetching data from one underlying device to another, with the goal of improving I/O performance.

We briefly present Nswap2L background as applicable to Nswap2L-FS, before discussing Nswap2L-FS in detail. For more information about Nswap2L, see [5].

Nswap2L Background: The main design goal of Nswap2L is to make best use of a heterogeneous cluster-wide storage to transparently provide fast swap space. Nswap2L frees the OS virtual memory system from having to be optimized for all possible combinations of different physical secondary storage devices. Additionally, Nswap2L is designed to be adaptable—the set of underlying storage devices can be changed on-the-fly, and its policies are tunable and designed to adjust to changes in cluster resource usage.

Nswap2L (and Nswap2L-FS) is conceptually designed as a two-level block device driver as shown in Figure 1. The top level Nswap2L driver implements the abstraction to the OS of a single, large, fast, random access storage device that can be added as a swap partition on individual cluster nodes. It manages a set of heterogeneous bottom level storage devices, including its implementation of network RAM, on which data are stored. When the OS writes to the top-level Nswap2L device, its placement policies select the bottom-level physical device on which to store the data.

The main system architecture is shown in Figure 2. While conceptually it is a two-level device driver, its implementation uses two levels of drivers for all underlying physical devices except for network RAM, which is implemented directly within the top-level Nswap2L driver.

The top-level Nswap2L driver is itself implemented in two layers. The upper layer implements the interface to the OS as a single, fast, random access storage device, and the lower layer implements mechanisms for reading and writing data to the underlying physical storage devices it manages. In addition to data placement policies, the top-level Nswap2L driver contains data prefetching policies that may result in data moving from one underlying device to another. Thus, a subsequent swap-in of a page from the Nswap2L device may result in the page data being read in from an underlying device different from the one to which it was initially written. This data movement is completely transparent to the OS.

The network RAM implementation is part of the top-level driver, and it is designed to be scalable and adaptable. Each cluster node acts as an equal peer, implementing both the client and server parts of the network RAM system. The client is active when the OS reads or writes to Nswap2L pages that have been placed on underlying network RAM. The server is active when it receives read or write requests from network RAM clients running on remote nodes. The server manages a portion of local RAM space that it makes available for network RAM storage. It grows and shrinks the amount of RAM it allocates based on the node's current workload. When the node has idle RAM, the server may allocate some for use by remote nodes; when the local workload needs more RAM, the server shrinks the amount it has allocated, releasing it back to the OS. Network RAM shrinking can result in page data being migrated to other servers or underlying storage devices. Each node uses a local estimate of available network RAM space in the cluster to find servers with available RAM space. Currently, we use LAN broadcast for servers to periodically share their available network RAM capacity with other nodes. In scaling to larger size systems, an overlay broadcast network could be used. See [12] for more details about the network RAM implementation.

As backing store for swap space, Nswap2L receives read and write requests for individual 4KB pages, each the result of a page swap-in or swap-out request by the OS. Nswap2L adds I/O requests to an internal queue to be handled by a worker pool thread. Worker threads invoke policy code and handle a read, write, or discard request to any underlying device. With multiple worker threads, multiple independent I/O requests can be handled in parallel.

Nswap2L maintains a small amount of metadata with each page of stored data, including which underlying device stores the page, in a data structure called the Slotmap. When a worker thread dequeues a read request, it looks up the device id in the Slotmap entry to find the underlying device storing the requested page. If the page is stored in

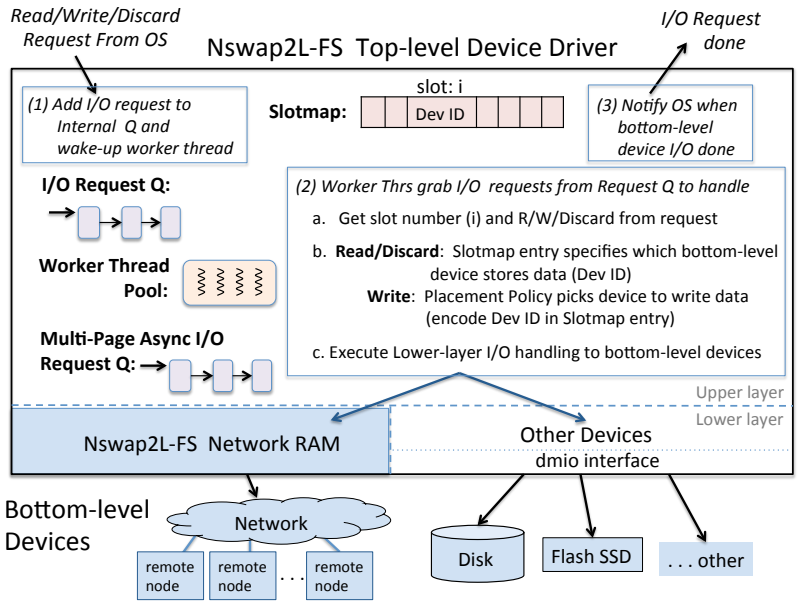

Figure 2. Nswap2L-FS (and Nswap2L) Architecture and Data Structures, including the Slotmap and Worker Thread Pool. *The upper layer implements data placement and migration policies to underlying devices. The bottom layer implements mechanisms to pass I/O requests to underlying physical devices and network RAM. Nswap2L-FS's asynchronous multi-page I/O request queue for network RAM pages is also shown.*

network RAM, the Slotmap entry encodes which network RAM server (remote cluster node) stores the page data. The worker thread directly runs the network RAM code in the lower-layer of Nswap2L to synchronously retrieve the page from the remote node. The worker then notifies the OS when the data are received and the read request is complete. If the page is stored on another underlying device, then the worker thread uses the Linux dmio [28] interface to asynchronously forward the read request to the bottom-level driver (e.g. flash SSD driver). In the callback function to the asynchronous dmio, the OS is notified that the read request is completed.

Writes are handled similarly to reads with two main differences. First, the worker thread executes policy code that chooses which bottom-level device will store the page (data placement policies are tunable via an interface exported in `/sys`). Second, when the write completes, the worker thread (for network RAM) or callback function (for dmio writes) encodes the underlying device in the Slotmap entry for the page and notifies the OS of completion.

Discards are handled by either forwarding the discard request to the underlying discardable devices (e.g. SSD), or by notifying a remote server that it can free the page stored in its network RAM. Discards update the Slotmap indicating that the page slots are unused.

While most requests handled by worker threads are issued by the OS, a portion are generated internally by Nswap2L to move data from one device to another. This internal data movement, called *prefetching*, is transparent to the OS and is a unique feature of Nswap2L. Nswap2L's ability to dynamically modify where swap data are stored has the promise of performance benefits beyond what the single fastest device can support. For instance, if there is relatively little network RAM available in the cluster, prefetching pages away from network RAM and onto a local device like flash can maintain some free network RAM space available for fast writes, while still allowing for good read performance from flash or network RAM.

Backing swap space is easier than backing file system data. For backing swap, all I/O requests are in single page-size units, thus no support for variable sized I/O requests is required. Additionally, there is no expectation of fault tolerance for swapped pages, so providing persistent backing store is not necessary.

A. Solutions for Handling File I/O Requests

To support backing filesystem data, Nswap2L-FS needs to handle requests in units of partial or multiple pages (any number of contiguous 512 byte blocks). This adds a substantial amount of complexity, especially since Nswap2L-FS, like Nswap2L, maintains mappings on page-size granularities. This decision allows us to split requests across multiple underlying heterogeneous devices and keeps the size of the Slotmap small. As a result, Nswap2L-FS must handle requests that may span multiple underlying devices and that may not be page aligned, complicating both the mapping of file data to underlying devices and the efficient implementation of handling single file I/O requests that are satisfied by multiple underlying devices.

Handling Partial Page Sized Requests: Most file system read requests to Nswap2L-FS are in units of its defined sec-

tor size (currently 4KB). However, file systems sometimes issue read requests for partial page data. These requests are generated by programs like `mount` that read portions of the first few pages to validate metadata for the file system being mounted. Because partial page size reads are unusual, our solution for handling them favors not introducing additional overheads for handing the much more common full and multi-page I/O requests.

If the underlying device storing the page data is on the dmio handling path, then the worker thread simply forwards the read request to the underlying device to handle it asynchronously—no special handling is needed in this case for partial page reads. If the underlying device storing the data is network RAM, then the worker thread handles the partial page read by issuing a full page read to the remote server storing the page, and extracts the portion requested. This implementation retains flexibility for handling small reads at a very low added cost of a full vs. a partial page read, while having no impact on the performance of the much more common case of larger I/O requests sizes.

Handling Multi-page Requests: To back a file system, Nswap2L-FS must be able to handle I/O requests for multiple contiguous pages of data, possibly spanning several underlying devices. We assign the responsibility for managing a multi-page request to the worker thread that removes the request from the internal request queue (the "shepherd thread"). This thread is responsible for handling each page of the I/O request, tracking the completion of any asynchronous page I/Os it issues, and notifying the OS of request completion once all pages have been handled.

In order to handle a multi-page I/O request, a shepherd worker thread must iterate through each page in the request and process it. For the pages in the request that are backed by network RAM, the shepherd thread directly executes the network RAM code to send or receive each page to remote server nodes, completing the necessary I/O before starting to process the next page. For pages backed by dmio devices, the shepherd thread issues asynchronous I/O calls via dmio, one for each page, to access the underlying storage device. Note that once the shepherd thread has finished iterating through the pages in the request, it must wait until all of the issued single page asynchronous dmio I/Os have completed before it can notify the OS that the full multi-page request is done. Keeping track of the status of each issued asynchronous request is crucial to ensuring correctness.

When the asynchronous dmio I/O function is called, its arguments include a callback function pointer and a context pointer. Dmio invokes the callback function when it completes the I/O operation, passing it the context pointer as an argument. For single page I/O requests, execution of the callback function indicates completion of the I/O request, and the callback function notifies the OS that its request is done. This notification requires that the callback function pass the OS a pointer to the request structure associated with its I/O request. We use the context pointer parameter to dmio to pass the address of the request structure used in the callback function (shown in the top of Figure 3).

For multi-page requests, however, execution of the callback function only indicates that a single page I/O of the larger multi-page request has completed. If the callback function were to notify the OS of completion of its request, then the OS would incorrectly be notified multiple times of its request completion, resulting in corruption of kernel-level data structures. When the callback function is executed for a page that is part of a larger request, it should only update state to indicate that one page of the request has completed.

Our solution is to assign the shepherd thread the responsibility of notifying the OS when the full multi-page request is complete. The shepherd thread communicates with threads executing asynchronous callbacks through a shared semaphore. It decrements the semaphore once for each issued asynchronous I/O, and each callback increments it. When it unblocks, the multi-page request is complete and it is safe to notify the OS.

To support this solution, the dmio callback function needs to identify if the I/O was for a single or a multi-page request. We use the context pointer parameter to encode this information. For a single page callback, the parameter's value is a pointer to the request structure from the OS. For a multi-page callback, the parameter's value is an encoding of the ID of the shepherd thread associated with the request, the ID of the underlying device storing the page (multi-page I/O requests can span multiple underlying devices), and the offset into the Nswap2L-FS device for this particular page (its Slotmap index). The encoding is shown in the bottom of Figure 3. Because the request structure is 4-byte aligned, we can use the low order bit of the context pointer to encode whether it contains information for a single page (low order bit 0) or a multi-page (low order bit 1) request. After testing this bit, the callback function extracts state from the context pointer parameter, and either notifies the OS of completion of the request (single page) or increments the shepherd thread's semaphore, signaling that one page of the request has completed (multi-page). For writes, the callback function needs to update the Slotmap entry with the underlying device's ID. For single page I/O, information about the Slotmap entry index and the underlying device are encoded in fields of the request structure; for multi-page I/O, they are encoded in the context pointer argument.

Asynchronous Multi-page I/O to Network RAM: In our initial implementation of multi-page I/O handling, individual pages backed by network RAM were synchronously read or written from remote server nodes by the shepherd thread, one page I/O after the next. Our experimental analysis found that this serial, synchronous I/O can significantly slow down applications that issue a large number of multi-page reads to Nswap2L-FS. As a result, we implemented asynchronous multi-page I/O support for the network RAM device. Our

Single Page Context:

63 bit Request Structure Pointer	0

Multi-page Context:

32 bit Nswap2L-FS Offset	16 bit Thread ID	15 bit Device ID	1

Figure 3. Asynchronous I/O callback context pointers. *For single page requests, the context pointer encodes a pointer to its associated I/O request structure. For multi-page requests, the context pointer encodes the I/O offset, the device ID, and the handling shepherd thread's ID. The least significant bit differentiates the two context types.*

solution uses an approach similar to the asynchronous interface provided by dmio to access other bottom level-devices.

Because worker threads directly execute network RAM code to read or write pages to remote server nodes, the only way to achieve asynchronous handling of multiple pages of network RAM is to have multiple worker threads simultaneously perform I/O of separate pages of the multi-page request. We add an additional internal request queue that is only for handling individual pages of multi-page I/O requests that are backed by network RAM (shown in Figure 2). The shepherd thread adds a special request structures to this second queue, one for each network RAM backed page in the multi-page request. It then wakes up other worker threads to handle the individual pages of the request in parallel. Coordination between the worker threads and the shepherd thread is handled in a similar manner as dmio callbacks: a shared semaphore is used to signal completion of each page of the multi-page request.

Experimental results comparing the synchronous and asynchronous versions of multi-page network RAM I/O show a significant improvement in performance (e.g. a speed-up of 5.5 over synchronous network RAM handling in our file I/O benchmark presented in Section IV).

B. File Data Persistence

Nswap2L was designed without support for persistent storage. While persistent storage may not be needed for backing swap data, file system data are generally assumed to persist. By design, Nswap2L-FS has relaxed persistence constraints on individual file writes, optimizing instead for fast I/O accesses by storing some file system data in the volatile RAM of remote cluster nodes, and file system metadata in the Slotmap stored in local RAM. For temporary file data that do not need to persist beyond the execution of the application that creates them, this trade-off in strict persistence of individual writes for faster file I/O is often desirable and can lead to faster application completion times. However, providing some support for file data persistence in Nswap2L-FS is useful, and even necessary for long-running applications, particularly those running on large systems where individual cluster node failure is the norm.

Nswap2L-FS provides persistence of stored data at the granularity of a persistent snapshot. The snapshot includes all live file data and metadata in the file partition backed by Nswap2L-FS. The added overhead of ensuring data persistence is only incurred if, and when, a application chooses to make a persistent snapshot. Nswap2L-FS implements an interface via `/sys` that applications can use to request, query the status of, and restore a persistent snapshot. Checkpointing systems, for example, can use this interface to save a snapshot of the file data as part of a checkpoint. The interface also can be used to save and restore Nswap2L-FS state across node reboots.

When a snapshot dump is requested, a dedicated persistence thread in Nswap2L-FS uses dmio to issue read requests to Nswap2L-FS and write requests to the snapshot destination device. An identical process, with data going from the destination device to Nswap2L-FS, occurs when a restore is requested. From the perspective of the Nswap2L-FS driver, all I/O requests issued by the persistence thread appear identical to any other read or write request from the OS.

The saved snapshot includes both a copy of every page of Nswap2L-FS storage that currently stores live filesystem data and metadata that encodes the Slotmap entry for each data page in the snapshot. Because Nswap2L-FS is loaded as a discardable device, the OS sends it discard I/O requests when it frees file blocks (pages of Nswap2L-FS backing store). This allows Nswap2L-FS to limit the amount of data included in a persistent snapshot to include only those pages that are currently in use by the filesystem.

When a restore is issued, the persistence thread reads in the metadata part of the saved snapshot and uses it to extract the correct offsets into the Nswap2L-FS device for each data page in the snapshot dump. As pages are read from the snapshot dump, the offset values are used to write the page data to its appropriate offset in the Nswap2L-FS device (the page's Slotmap index) to correctly restore the state of the filesystem saved in the snapshot.

By design Nswap2L-FS provides a course grained persistence mechanism that is optimized for fast file I/O over strict persistence guarantees of every individual write. For the types of systems and uses Nswap2L-FS targets, this is the right trade-off. In these uses, snapshots are rare and persistence may not even be desired by the application. Nswap2L-FS's snapshot feature only introduces persistence overhead when explicitly invoked by a higher-level user.

Because Nswap2L-FS is implemented at the device layer vs. at the file system layer, the granularity of its persistent snapshot must be at a full file partition level, including all live file data in the partition it backs; Nswap2L-FS does not have a higher-level interpretation of the meaning of the file blocks it stores, and thus has no way to pick blocks corresponding to specific files to back. A possible future direction is to support an persistent snapshot interface that specifies sub-ranges in our device partition. However, because our target use is for backing temporary file data, the

partition will tend to only contain live data from currently running applications. Thus, we anticipate that persistent snapshots will be taken only at checkpoint events where the full partition of Nswap2L-FS live data is the desired granularity of the persistent snapshot.

Using its snapshot and recovery interface is straightforward, and could be easily integrated into checkpointing and MapReduce implementations. The performance of the persistent snapshot and restore from snapshot depends heavily on the amount of live data in the file system, the size of the file partition, and the type of backing storage device storing the snapshot.

C. Optional Encryption

Nswap2L-FS includes an optional encryption feature that a user can enable to direct Nswap2L-FS to encrypt file page data that it places in network RAM. Encryption adds additional overhead to network RAM I/O, but is useful in cases when Nswap2L-FS is used to store sensitive file data on an insecure network. Encryption is implemented solely in the network RAM client using the Linux kernel AES library; network RAM servers and the Nswap2L-FS upper-layer are unaware of page data encryption.

IV. Nswap2L-FS Performance Results

The main goal of Nswap2L-FS is to effectively make use of the heterogeneous collection of storage available in clusters in order to provide fast backing store for filesystems, particularly those storing temporary files. Our vision is that applications that generate temporary files will run faster when their file data are backed by Nswap2L-FS, relative to being backed by any single physical storage device. We present results of experiments that evaluate the performance of Nswap2L-FS, comparing the performance benchmark programs run with filesystems backed by Nswap2L-FS to those backed by other secondary storage devices. The purpose of these experiments is to evaluate the relative performance of Nswap2L-FS compared to any single physical device. Additionally, we validate Nswap2L-FS's two level design by evaluating the performance benefits achieved through its adaptable data placement and prefetching policies to its heterogeneous set of underlying devices.

All experiments were run on a 16 node cluster running unmodified Linux 4.0.4. Each node had 16GB of RAM, a 120GB Intel 320 Series SSD, and a 500GB Western Digital HDD. Nodes were connected by 10Gb Ethernet, with a latency of approximately 0.17 ms. The Nswap2L-FS device driver was loaded on each node, formatted with the Linux `ext4` file system, and configured to manage three different bottom-level storage devices: network RAM, flash SSD, and HDD. Unless otherwise noted, the amount of network RAM available exceeded the amount necessary to hold all data stored on the filesystem.

A. Nswap2L-FS vs. Single Device

To evaluate Nswap2L-FS's two-level design, we ran several benchmark programs comparing their performance when using a file system backed by different devices. The purpose of our benchmarks is to evaluate the performance of an Nswap2L-FS backed file system in envisioned use cases. We present the results of three of these studies.

The first program simulates a common file access pattern of applications that create and use temporary files as part of a larger computational task: first a temporary file is created and written to, and later it is read in for further processing. We tested Nswap2L-FS's performance on a file I/O benchmark program that we wrote to generate this file creation, file write, and file read access pattern. The benchmark program consists of creating 200 files, each 20 megabytes in size, flushing the system caches, and subsequently computing the MD5 hash of written files, requiring a full read of each one. Flushing the system caches between the write and read phases eliminates confounding effects of OS caching across runs of each experiment.

The second program is the STXXL [29] library's implementation of external merge sort. External merge sort is commonly used for sorting data that are too large to fit into RAM, and thus involves a substantial amount of file I/O.

The third benchmark program is the varmail application from the Filebench [30] benchmarking suite. Varmail simulates another common temporary file creation and access pattern: short-lived files on a mail server where email message files are created, delivered, and then deleted. Unlike the other programs tested, the metric reported by the benchmark is the average number of I/O operations completed per second, rather than task completion time.

Each benchmark was run with three different Nswap2L-FS placement policy configurations: one placing file data to underlying network RAM, one to flash, and one to disk. We also ran the benchmark for configurations where the file system is mounted directly on flash and hard drive partitions. These runs provide measures to evaluate Nswap2L-FS two-layer overheads when performing I/O on top of flash and disk. We measure the total runtime of the file I/O and external sorting benchmarks, and the number of I/O completions per second over a three minute execution of the the varmail benchmark.

Figure 4 shows the average runtime for our file I/O benchmark. The results show statistically significant performance improvement when Nswap2L-FS places file data on its underlying network RAM vs. when it places it on underlying flash or disk partitions (343 seconds for network RAM vs. 389 for flash and 431 for disk). Also shown are measures of the overhead added by the Nswap2L-FS top-level driver on top of flash and disk. When the flash device is directly used as a backing store, the additional processing added by the Nswap2L-FS driver marginally increases the runtime by

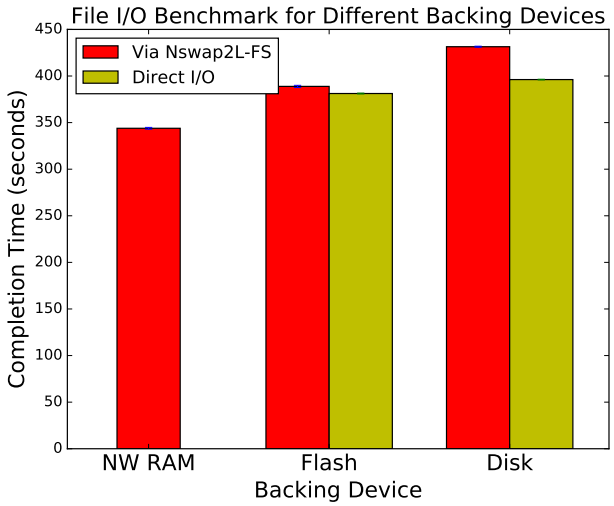

Figure 4. The runtime of our file I/O benchmark program for different backing storage devices: Nswap2L-FS to Network RAM; Nswap2L-FS to Flash; Nswap2L-FS to Disk; direct to flash; direct to disk. *The times are the average of 10 runs on each configuration. Standard deviations are also shown.*

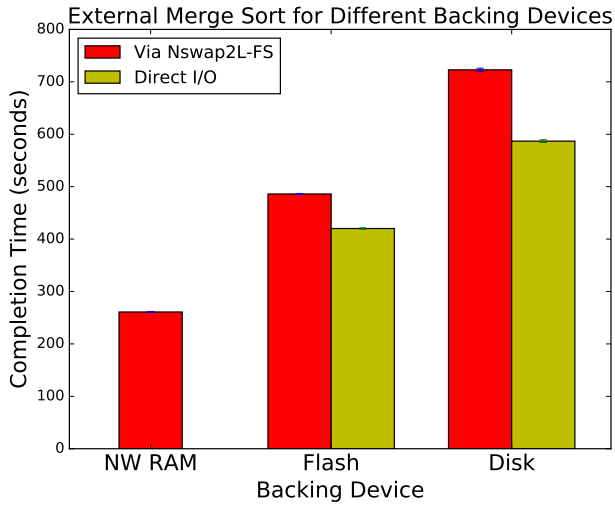

Figure 5. Total Execution Time of STXXL External Merge Sort for temporary files systems backed by different devices: Nswap2L-FS to Network RAM; Nswap2L-FS to Flash; Nswap2L-FS to Disk; direct to flash; direct to disk. *The times are the average over 10 runs, and standard deviations (very small) are shown.*

approximately two percent. On top of disk, the Nswap2L-FS overhead is higher, adding 8% to the total runtime of the file system directly mounted on the disk partition. The higher overhead on top of disk is partially due to Nswap2L-FS using the Linux `noop` scheduler vs. the disk driver using the default Linux scheduler that is optimized specifically for disk I/O. Because Nswap2L-FS is unlikely to send file data to underlying disk (doing so only when there is no network RAM or flash space available), the higher overheads on top of disk are unlikely to affect Nswap2L-FS's file I/O speeds.

Figure 5 shows the average runtime for the STXXL external merge sort program for runs on different file backing storage devices. The results show a 1.8 speed-up when using Nswap2L-FS to network RAM vs. Nswap2L-FS to flash, and speed-up of 2.8 over Nswap2L-FS to disk. Nswap2L-FS to network RAM is 1.6 times faster than direct flash, and 2.2 times faster than direct disk. Also shown are measures of the overhead added by the Nswap2L-FS top-level driver on top of flash and disk. When flash and disk devices are directly used as a backing store, the additional processing added by the Nswap2L-FS driver marginally increases the runtime. However, both direct storage devices are significantly slower then Nwap2L-FS's network RAM storage.

Figure 6 shows the average number of I/O operations per second when running the varmail benchmark for each configuration of backing device for the temporary file system. The results show an increase by a factor of 10.5 in the number of I/Os per second when using Nswap2L-FS to network RAM versus direct to local disk, and an increase by a factor of 1.7 when using Nswap2L-FS to network RAM versus direct to local flash. There is no significant overhead

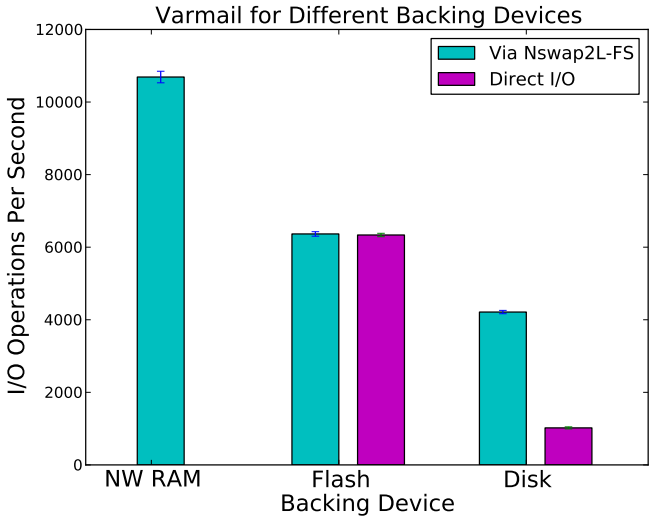

Figure 6. Total I/O operations per second for Varmail benchmark, with the destination file system backed by Nswap2L-FS or local devices. Note that since this is a measure of throughput, higher values are better. *The totals are averaged over 10 runs. Standard deviations are also shown.*

in accessing flash through Nswap2L-FS versus accessing it directly. However, accessing file data on disk through Nswap2L-FS strangely results in better performance (more I/Os per second) than when accessing file data directly on disk. We suspect that this is due to the small size of the files created and written and the different device scheduling policies used when the disk is accessed directly versus when it is accessed indirectly through Nswap2L-FS. It should be noted that Nswap2L-FS does *not* cache any stored data

internally.

Overall, these results support Nswap2L-FS as a fast backing storage device for filesystems. They show that Nswap2L-FS's network RAM can be used as a fast backing store device that outperforms local flash or disk devices on a file access pattern common to the applications that our work specifically targets. While there is some variability in the amount of overhead added by Nswap2L-FS depending on the types and sizes of requests issued, these results broadly show that its two-level device design often adds little extra overhead to directly accessing underlying devices.

B. Evaluating Nswap2L-FS Policies

A main feature of Nswap2L-FS is that it can take advantage of the strengths of different types of storage devices typically found in clusters. Its data placement policies determine which underlying devices store the file data written to Nswap2L-FS. Its prefetching policies transparently move file data from one underlying device to another in response to changes in cluster-wide storage capacity (such as changes in the amount of available idle RAM for network RAM storage), or to increase read parallelism by distributing data over a set of devices with fast read performance. Ideally, its policies result in Nswap2L-FS providing file system I/O performance that is better than that of any single backing storage device available in the cluster.

We present results from two experiments to evaluate how well the two layer architecture of Nswap2L-FS enables higher performance by more effectively adapting to changes in cluster workload and exploiting the benefits of a heterogeneous set of devices. Other adaptive policy results are omitted due to space; their results are similar to studies presented in our earlier work [5].

Our first experiment evaluating the benefits of Nswap2L-FS's adaptable policies examines its support for prefetching (moving data stored on one underlying device to another underlying device). Prefetching is transparent to the OS and is one method that Nswap2L-FS uses take advantage of the strengths of the different underlying storage devices it manages. For instance, network RAM has fast writes, while flash writes are typically slower than flash reads due to block erasure overheads. Both flash and network RAM have reasonably fast read speeds. An advantage of Nswap2L-FS's hierarchical design is its ability to transparently move, or prefetch, pages from network RAM to a local flash device, freeing up network RAM space to be available for future writes. This example of prefetching allows an application to gain the benefits of fast writes to network RAM even when the total amount of data written exceeds network RAM capacity.

We evaluated Nswap2L-FS's prefetching using the varmail benchmark program. We ran experiments for different amounts of available network RAM capacity. Nswap2L-FS's placement policy will choose to place data on network RAM first, picking flash placement only if no network RAM is available at the time of the write. In one set of experiments, Nswap2L-FS was configured to not use prefetching. As a result, data written early to Nswap2L-FS were directed to network RAM; once network RAM was full, subsequent writes were directed to flash SSD. In another set of experiments, Nswap2L-FS was configured to prefetch pages from network RAM to flash, with the goal of keeping some network RAM available for fast writes.

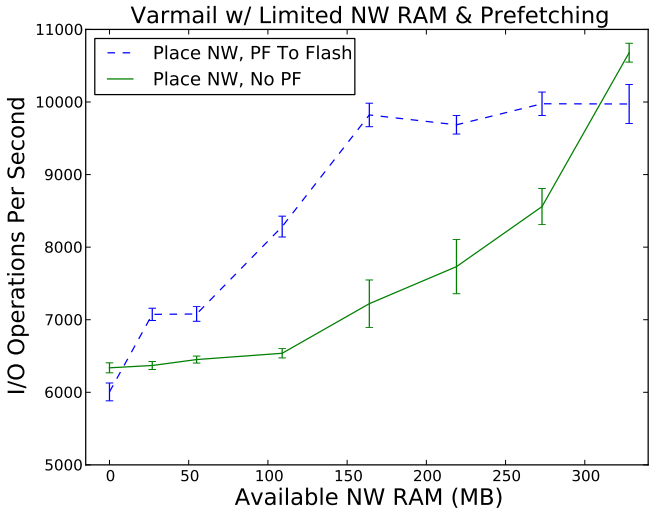

Figure 7. Nswap2L-FS with and without prefetching for different Network RAM capacity. *The number of completed I/O operations per second for the varmail benchmark when run with differing amounts of network RAM space, comparing runs without prefetching to runs with prefetching from network RAM to flash. Also note that the y-axis starts at 5,000.*

Figure 7 shows the average number of I/O operations per second for the varmail benchmark for runs with and without prefetching enabled for different network RAM capacity sizes. For all runs where there is some network RAM available, but not enough to hold the entire filesystem, prefetching from network RAM to flash results in significant improvements over the I/O throughput of Nswap2L-FS without prefetching. For example, when there is 164MB of network RAM available, prefetching leads to 9,822 I/O operations per second vs. 7,221 per second without prefetching. When network RAM is plentiful or nonexistent, prefetching adds unnecessary overheads. But outside of these two extremes, prefetching data from network RAM to local flash storage improves I/O throughput by freeing up network RAM space for future fast writes.

This result suggests that even simple prefetching policies can lead to significant performance improvements, especially in cases where network RAM capacity is not sufficient to store the entire file system backed by Nswap2L-FS. They also motivate future investigation into developing more complex policies that can enable prefetching when it is

Configuration	Runtime (s)	Std. Dev (s)
Adaptable: Flash & Network RAM	331.1	0.5
Local Flash SSD	486.1	0.9
Network RAM	260.9	0.8

Table I
TOTAL COMPLETION TIMES OF EXTERNAL MERGE SORT WITH AN ADAPTIVE DATA PLACEMENT POLICY VS. A STATIC PLACEMENT POLICY THAT ALWAYS CHOOSES FLASH AS THE UNDERLYING DEVICE. RESULTS ARE ALSO SHOWN FOR EXECUTION TIME USING ONLY NETWORK RAM AS THE UNDERLYING DEVICE (THE FASTEST WHEN AVAILABLE).

likely to be advantageous and disable it when it is unlikely to improve performance. Based on these results, a policy that uses the amount of available network RAM capacity to enable and disable prefetching would result in better Nswap2L-FS file I/O throughput.

Our second experiment evaluates the benefits of Nswap2L-FS's adaptable data placement policies. Nswap2L-FS data placement policies may choose different underlying devices in response to changing cluster conditions. Given the inherent variability in cluster workloads, the amount of network RAM available varies over time; Nswap2L-FS grows and shrinks the storage capacity of network RAM in response to changes in the amount of available idle RAM in the cluster.

We evaluated a dynamic placement policy that chooses network RAM for file writes when network RAM is available. When no network RAM is available, the policy chooses to send the page to the underlying flash device. As new network RAM becomes available, the policy will switch from sending writes to flash to sending them to network RAM. To evaluate the potential benefits of this dynamic placement policy, we ran the external merge sort benchmark with an Nswap2L-FS backed temporary file system configured to use this dynamic placement policy. In our experiments, initially no network RAM is available and the policy chooses to send pages to flash. After approximately 120 seconds of execution, we make network RAM available and Nswap2L-FS begins to place subsequent file writes on network RAM. We compared runs using this adaptive policy to runs using a static placement to underlying flash.

The results, in Table I, show that the run using the adaptive policy that chooses flash placement until network RAM becomes available performs better than the static placement policy that continues to choose flash placement (331.1 vs. 486.1 seconds). As a performance comparison, the results of a run placing only to network RAM (the fastest option) are also shown. These results validate the heterogeneous storage management design of Nswap2L-FS. They demonstrate one way in which Nswap2L-FS can adapt on-the-fly to changes in underlying device capacity to make better placement policy decisions.

The experimental results support our design of Nswap2L-FS. They show that its two-level design often adds little overhead to directly mounting file systems on the underlying devices and that these overheads are reclaimed by the performance benefits resulting from effective use of heterogeneous storage options. Flexible placement and prefetching policies that adapt to changes in cluster resource usage and take advantage of different device strengths result in faster I/O for the file system it backs and increased application performance.

V. CONCLUSION

Nswap2L-FS is our solution for taking advantage of both network RAM and storage devices in clusters to provide fast backing storage for file systems. The Nswap2L-FS device trades persistence guarantees on the granularity of individual writes for provide faster I/O accesses to the file data it stores. It is particularly useful for storing temporary file data that do not normally persist beyond the execution of the application that creates them. Its persistent snapshot mechanism can be used by applications that desire some persistence for the file system it backs. Nswap2L-FS presents itself to the OS as a single storage device, which isolates the complexity of effectively managing these storage options from the OS. Internally, Nswap2L-FS manages these devices using interchangeable policies which dictate where data should be placed, and potentially relocated, in order to achieve optimal performance. Its extensive `/sys` interface allows users to change policies at run time, request checkpoints of stored data, and monitor internal status metrics. Our experimental results support its two-level device design and its adaptable data placement and prefetching policies, which result in increased file I/O performance, even when network RAM is scarce. Future research directions include implementing and evaluating more adaptive policies, testing Nswap2L-FS with a wider range of underlying storage devices, and testing Nswap2L-FS's scalability to larger applications and to bigger cluster systems.

ACKNOWLEDGMENTS

This work is supported by the National Science Foundation grant NSF:CNS:CSR-1116224.

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] A. Acharya and S. Setia, "Availability and utility of idle memory in workstation clusters," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 27, no. 1. ACM, 1999, pp. 35–46.

[3] R. Birke, L. Y. Chen, and E. Smirni, "Usage patterns in multi-tenant data centers: A temporal perspective," in *Proceedings of the 9th International Conference on Autonomic Computing*, ser. ICAC '12. ACM, 2012.

[4] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: Reliable, memory speed storage for cluster computing frameworks," in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2014, pp. 1–15.

[5] T. Newhall, E. R. Lehman-Borer, and B. Marks, "Nswap2l: Transparently managing heterogeneous cluster storage resources for fast swapping," in *Proceedings of the Second International Symposium on Memory Systems*, ser. MEMSYS '16, 2016.

[6] T. E. Anderson, D. E. Culler, and D. A. Patterson, "A case for now (networks of workstations)," *Micro, IEEE*, vol. 15, no. 1, pp. 54–64, 1995.

[7] A. Acharya, G. Edjlali, and J. Saltz, "The utility of exploiting idle workstations for parallel computation," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 25, no. 1. ACM, 1997, pp. 225–234.

[8] D. Nevarez, V. Patwari, J. J. Rosales, and M. J. Rosas, "Distributed computing utilizing virtual memory having a shared paging space," Oct. 18 2011, US Patent 8,041,877.

[9] M. Serrano, J. Sahuquillo, S. Petit, H. Hassan, and J. Duato, "A cost-effective heuristic to schedule local and remote memory in cluster computers," *The Journal of Supercomputing*, vol. 59, no. 3, pp. 1533–1551, 2012.

[10] H. Sharifian and M. Sharifi, "Network ram based process migration for hpc clusters," *Journal of Information Systems and Telecommunication*, vol. 1, no. 1, pp. 47–53, 2013.

[11] H.-H. Choi, K. Kim, and S.-J. Bae, "A remote memory system for high performance data processing," *International Journal of Future Computer and Communication*, vol. 4, no. 1, p. 50, 2015.

[12] T. Newhall, S. Finney, K. Ganchev, and M. Spiegel, "Nswap: A network swapping module for linux clusters," in *Euro-Par 2003 Parallel Processing*. Springer, 2003, pp. 1160–1169.

[13] A. Samih, R. Wang, C. Maciocco, T.-Y. C. Tai, and Y. Solihin, "A Collaborative Memory System for High-Performance and Cost-Effective Clustered Architectures," in *Proceedings of the 1st Workshop on Architectures and Systems for Big Data*. ACM, 2011, pp. 4–12.

[14] P. Werstein, X. Jia, and Z. Huang, "A Remote Memory Swapping System for Cluster Computers," in *International Conference on Parallel and Distributed Computing, Applications and Technologies PDCAT*. IEEE, 2007, pp. 75–81.

[15] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson, "Cooperative caching: Using remote client memory to improve file system performance," in *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*. USENIX Association, 1994, p. 19.

[16] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab *et al.*, "Scaling memcache at facebook," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013, pp. 385–398.

[17] M. J. Feeley, W. E. Morgan, E. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath, *Implementing global memory management in a workstation cluster*. ACM, 1995, vol. 29, no. 5.

[18] M. Serrano, J. Sahuquillo, S. Petit, H. ne Hassan, and J. Duato, "A cost-effective heuristic to schedule local and remote memory in cluster computers," *Journal of Supercomputing*, vol. 59, no. 3, 2013.

[19] G. Graefe, "The five-minute rule twenty years later, and how flash memory changes the rules," in *Proceedings of the 3rd international workshop on Data management on new hardware*. ACM, 2007, p. 6.

[20] B. Fitzpatrick, "Distributed caching with memcached," *Linux journal*, vol. 2004, no. 124, p. 5, 2004.

[21] J. Zhang, G. Wu, X. Hu, and X. Wu, "A distributed cache for hadoop distributed file system in real-time cloud services," in *Grid Computing (GRID), 2012 ACM/IEEE 13th International Conference on*. IEEE, 2012, pp. 12–21.

[22] Y. Luo, S. Luo, J. Guan, and S. Zhou, "A ramcloud storage system based on hdfs: Architecture, implementation and evaluation," *Journal of Systems and Software*, vol. 86, no. 3, pp. 744–750, 2013.

[23] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang, "The ramcloud storage system," *ACM Trans. Comput. Syst.*, vol. 33, no. 3, Aug. 2015.

[24] J. Xu and S. Swanson, "Nova: A log-structured file system for hybrid volatile/non-volatile main memories," in *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, ser. FAST'16. Berkeley, CA, USA: USENIX Association, 2016, pp. 323–338. [Online]. Available: http://dl.acm.org/citation.cfm?id=2930583.2930608

[25] N. S. Islam, X. Lu, M. Wasi-ur Rahman, D. Shankar, and D. K. Panda, "Triple-h: A hybrid approach to accelerate hdfs on hpc clusters with heterogeneous storage architecture," in *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*. IEEE, 2015, pp. 101–110.

[26] S. Robbins, "Ram is the new disk. . . ," *InfoQ News*, June 2008.

[27] A. Uta, A. Sandu, S. Costache, and T. Kielmann, "Memefs: an elastic in-memory runtime file system for escience applications," in *e-Science (e-Science), 2015 IEEE 11th International Conference on*. IEEE, 2015, pp. 465–474.

[28] "Device mapper, red hat inc." http://sources.redhat.com/dm.

[29] R. Dementiev, L. Kettner, and P. Sanders, "Stxxl: standard template library for xxl data sets," *Softw., Pract. Exper.*, vol. 38, no. 6, pp. 589–637, 2008.

[30] R. McDougall, "Filebench," *URL: http://www. nfsv4bat. org/Documents/nasconf/2004/filebench*, p. 56, 2005.