Nswap2L: Transparently Managing Heterogeneous Cluster Storage Resources for Fast Swapping

Tia Newhall

E. Ryerson
Lehman-Borer

Benjamin Marks

{newhall,elehman2,bmarks1}@cs.swarthmore.edu
Computer Science Department
Swarthmore College
Swarthmore, PA, USA

ABSTRACT

To support data intensive cluster computing, it is increasingly important that node virtual memory (VM) systems make effective use of available fast storage devices for swap or temporary file space. Nswap2L is a novel system that transparently manages a heterogeneous set of storage options commonly found in clusters, including node RAM, disk, flash SSD, PCM, or network storage devices. Nswap2L implements a two-level device driver interface. At the top level, it appears to node operating systems (OSs) as a single, fast, random access device that can be added as a swap partition on cluster nodes. It transparently manages the underlying heterogeneous storage devices, including its own implementation of Network RAM, to which swapped out data are stored. It implements data placement, migration, and prefetching policies that choose which underlying physical devices store swapped-out page data. Its policies incorporate information about device capacity, system load, and the strengths of different physical storage media. By moving device-specific knowledge into Nswap2L, VM policies in the OS can be based solely on typical application access patterns and not on characteristics of underlying physical storage media. Nswap2L's policy decisions are abstracted from the OS, freeing the OS from having to implement specialized policies for different combinations of cluster storage—Nswap2L requires no changes to the OS's VM system. Results of our benchmark tests show that data-intensive applications perform up to 6 times faster on Nswap2L-enabled clusters, and show that our two-level device driver design adds minimal I/O latency to the underlying devices that Nswap2L manages. In addition, we found that even though Nswap2L's Network RAM is faster than any other backing store, its prefetching policy that distributes data over multiple devices results in increased I/O parallelism and can lead to better performance than swapping only to a single underlying device.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

MEMSYS '16, October 03-06, 2016, Alexandria, VA, USA

© 2016 ACM. ISBN 978-1-4503-4305-3/16/10...\$15.00

DOI: <http://dx.doi.org/10.1145/2989081.2989107>

CCS Concepts

•Computer systems organization → Distributed architectures; •Software and its engineering → Virtual memory; Secondary storage; Distributed memory;

Keywords

Cluster Systems; Backing Storage; Network RAM; Flash

1. INTRODUCTION

Data intensive workloads running on general-purpose clusters may create or process data that are too large to fit in the memory of individual nodes, resulting in swapping. Alternatively, these data may be divided into smaller sets, with partial computational results stored in temporary files [7, 11], resulting in file I/O on individual nodes. As the disparity continues to grow in the speed of magnetic disk versus other devices such as RAM, network, flash and phase change memory (PCM), the cost of accessing disk – either as a backing store for swapped pages or files – is increasingly the bottleneck to achieving good performance in these systems.

To support efficient data intensive computing, it is important that cluster node OSs make the most effective use of all types of available fast backing store in a cluster. To this end, newer versions of the Linux kernel have added features that better support flash SSD, including TRIM/discard notifications for freed pages, which helps to maintain extents of erased blocks on the flash device, resulting in improved write performance. Devices can also specify the Linux `noop` scheduler that performs better for flash than the default device scheduler that is optimized for disk I/O.

Even with these additions, much of the Linux virtual memory system is still designed assuming that backing store is magnetic disk. This assumption is increasingly incorrect: with flash, PCM, and network storage, cluster backing store is, and will continue to be, more heterogeneous. Backing store may also be remotely located in a cluster, and thus not under the control of a single node's OS. These growing trends in physical storage will result in either cluster node VM page swapping policies that are progressively less well designed to take advantage of this mix, or will require introducing complicated device-specific policies into OS swapping and file subsystems that are specialized for each type of physical storage in the cluster.

Nswap2L is our novel solution for managing heterogeneous cluster backing store. It implements an efficient, scalable, extensible, and adaptable backing storage system that can efficiently utilize existing and emerging storage technologies to provide fast backing store in general purpose cluster systems. Our goal is to take advantage of all types of storage in a cluster while providing a simple, single device interface to node OSs of fast storage that is independent of the underlying physical storage on which data may reside. Nswap2L is transparent to the OS and requires no modifications to the OS running on each node. Additionally, our storage system is able to adapt to changes in resource utilization and capacity while easily incorporating new storage devices, and it takes advantage of the strengths of different types of storage media based on tunable policies for managing data placement. Ultimately, we aim to provide a fast backing storage system that can be used for swapping, for temporary filesystems, and for other types of abstractions on top of stored data. Currently, our work focuses on using our system as backing store for VM swap space.

The goals of our system inform the design of Nswap2L as a virtualization and management layer on top of cluster-wide storage. It is designed as a two-level storage device to transparently provide access to a heterogeneous set of cluster backing storage to cluster node OSs.

To cluster node OSs, Nswap2L appears as a single, fast, random access device that can be added as backing store for swap. It transparently manages an underlying set of heterogeneous storage devices to which swapped out data are stored. It implements data placement, migration, and prefetching on the set of physical storage devices it manages, choosing location based on device capacity, system load, and the strengths of different storage media. As an example, fast writes to Network RAM and fast reads from flash may result in a placement policy that initially writes swapped out pages to Network RAM. A prefetching policy may later move some pages from Network RAM to flash to take advantage of fast reads from flash. Data placement and prefetching policies such as these are likely to improve performance by maintaining free Network RAM space to support fast writes, and to improve read performance, increasing read parallelism by distributing data over more underlying devices with similar fast read performance.

In general purpose clusters or LANs, Network RAM is viable for use as backing store for swapped out pages [2]. Because general purpose clusters are not optimized for running a single, fixed workload, there is a large amount of variability in cluster resource utilization due to dynamic changes in the workload. Additionally, because resource scheduling is difficult in this environment, there will often be imbalances in resource usage across cluster nodes. For example, some nodes may have idle RAM while others have over-committed RAM, resulting in swapping. The idle RAM space on some nodes can be used by a Network RAM system to improve the performance of applications running on the cluster. Network RAM storage can be used by individual cluster nodes with over-committed RAM to swap their pages out over the network to store in the idle RAM of other nodes. Nswap2L's Network RAM provides free and fast storage in cluster systems by locating and making use of cluster-wide RAM that is currently idle.

Although Network RAM provides backing store that is faster than local disk and is often faster than local flash SSD,

there are some difficulties in designing an efficient Network RAM system to ensure that it improves performance of the cluster workload. A Network RAM system must adapt to changes in a node's RAM needs, and not impede the performance of processes running on individual nodes. If a node currently needs more RAM for its local processes, then a Network RAM system should relinquish some or all of the RAM pages it is using on that node back to the local paging system. If a node currently has some idle RAM, a Network RAM system should also detect this, and allocate a portion of it for storing remotely swapped page data from other nodes.

The Nswap2L implementation of Network RAM is designed to adapt to these types of changes in cluster nodes' RAM use; it grows and shrinks the amount of RAM a node makes available for storing remotely swapped page data. It also supports page data migration from the RAM of one node to another, or from Network RAM to another underlying storage device such as flash, in response to changes in a node's RAM usage.

Currently, Nswap2L can be loaded as a device driver on cluster nodes running unmodified 4.0 versions of Linux, and added as a swap device on individual nodes. It will transparently store swapped out page data on any number of underlying physical storage devices, including its Network RAM implementation. Its implementation includes an extensive `/sys` interface that allows users to change the set of underlying devices, and to perform data placement and prefetching policy tuning and configuration changes at runtime.

Our experimental results¹ show data-intensive benchmark programs performing up to 6 times faster on Nswap2L-enabled clusters, and that Nswap2L's two-level device driver design adds minimal overhead to underlying device access times.

The rest of this paper is organized as follows: in Section 2 we discuss related work; in Section 3 we present the design and implementation of the Nswap2L system; in Section 4 we present our results; and we conclude and discuss future directions of this project in Section 5.

2. RELATED WORK

Nswap2L is related to other work that investigates using Network RAM as backing store in cluster and networked systems. It is also complementary to work that examines how fast storage technologies, such as flash SSD, can be incorporated into VM or file systems.

One early cluster computing project, Berkeley's NOW [3], proposed the idea of using Network RAM as backing store for swap. Subsequent studies of cluster resource usage have supported the viability of Network RAM, showing that there is an abundance of free RAM in clusters even during times when the RAM on many of the nodes is overload [2, 4, 9]. As the size of RAM continues to increase and its cost decreases over time, some recent work has referred to "RAM [as] the new disk," and disk as the new tape storage [23]. This trend in RAM capacity and cost has motivated filesystems based entirely in memory, including MemEFS [28] and RAMCloud [14]. Another study [10] concludes that data accessed every 5 hours should be placed in RAM, suggesting

¹Experiments were run on a 16 node, Linux 4.0.4 cluster, with 10-Gbit Ethernet, and flash SSD, disk, and Network RAM as underlying devices.

that many intermediate and temporary files created by large data-intensive computations could be stored in memory for their entire lifetime, avoiding secondary storage completely.

There have been many previous Network RAM projects that have investigated the use of idle RAM in clusters and networked systems for storing swapped out page data [12, 15, 16, 17, 21, 25, 30], or for cooperative file caching [6, 20]. Some of the prior work in Network RAM includes centralized [17] and decentralized [16, 30] solutions, implementations that run entirely at kernel-level [16, 17, 21], and others that run at user-level [25]. Some work also focuses on adding fault tolerance to remotely swapped page data [16].

The Network RAM system that is part of Nswap2L [18] is similar to some of these projects, employing a completely decentralized design and running entirely at kernel-level. However, among this set of work, our Network RAM has a unique focus on adapting to changes in nodes’ RAM usage; when a node needs more RAM space for its local processes, it releases RAM allocated for storing remotely swapped page data back to the VM system, and when idle RAM space becomes available on a node, some is allocated for Network RAM space. This adaptability allows Nswap2L to be loaded persistently on a cluster without interfering with the RAM needs of applications running on individual nodes.

There has also been much recent work examining how to incorporate new and emerging fast storage technology into systems [5, 22, 24, 27, 29]. Different uses include incorporating flash into the memory hierarchy, or using it as a fast out of core buffer cache in database systems. In [22], the authors measure scientific workload execution times comparing different backing storage devices: two different flash devices and disk. They find that for sequential workloads, the flash drives offer little improvement over disk, but for parallel workloads flash significantly outperforms disk, primarily due to increased opportunity for parallel I/O. In [5], the authors model several new storage technologies, including phase-change memory and spin-torque transfer memory. They find that these technologies can lead to significant performance improvements, especially if the OS is modified to remove the assumption that I/O is very slow compared to computation.

FlashVM [26], in particular, focuses on changes to the VM system to better support using flash devices as secondary storage for the VM system. The VM policy and design changes they propose give insight into how we can effectively incorporate flash devices into the Nswap2L swapping device. Unlike FlashVM, however, our work does not require changes to the OS, but rather Nswap2L exists as an intermediate layer between the OSs VM system and underlying storage, which may include flash.

Finally, Triple-H [13] describes a heterogeneous storage system specifically for Hadoop applications. It chooses data placement on underlying storage including RAM, flash SSD, and HDDs. Nswap2L similarly manages placement of data on a set of heterogeneous cluster-wide storage. However, it is not limited to Hadoop applications, and it transparently implements data placement, prefetching, and migration on top of the heterogeneous storage.

3. NSWAP2L SYSTEM

We present the high-level design of our system, describe details of its implementation, and discuss its policies, extensibility, and adaptability.

Figure 1: Nswap2L’s conceptual two-level device design. The single upper-level device is the interface to the OS. It contains internal policies and mechanisms for managing a heterogeneous set of underlying devices that store the data.

The Nswap2L system design is based on an earlier prototype system [19] that evaluated the feasibility of our two-level device approach. Conceptually, Nswap2L is designed as a two-layer block device driver as shown in Figure 1. The top-level device implements the abstraction to the OS of a single, large, fast, random access device that can be added as a swap partition on individual cluster nodes. The bottom-level devices are the set of heterogeneous physical storage that the top-level manages. When the OS writes to the Nswap2L device, Nswap2L invokes placement policies to select the underlying physical device to which to store the data. Later on, its prefetching and data migration policies may result in data moving from one underlying device to another. Thus, a subsequent swap-in of a page from our top-level device may result in the page data being read in from an underlying device other than the one to which it was initially written. This data movement is completely transparent to the OS.

Although our system is designed as a two-level device driver, its implementation uses a two-level driver design for most underlying devices, but not for its own Network RAM. Instead, Network RAM is integrated into the top-level Nswap2L driver.

3.1 Implementation

Nswap2L is implemented as a multi-threaded Linux block device driver. It is a loadable kernel module that can be loaded into unmodified Linux 4.0 kernels, and added as a swap partition on individual cluster nodes. Nswap2L’s multi-threaded design allows it to simultaneously handle multiple read, write or discard requests to the set of underlying physical storage devices it manages. It exports an extensive `/sys` interface that allows for underlying devices to be added, and for policies to be tuned and changed at runtime. The `/sys` interface also is used to export statistics about Nswap2L and its underlying devices.

Figure 2 shows some details of the Nswap2L implementation. The Nswap2L top-level device is divided into two layers. The upper layer exports the block device interface to the operating system, so when Nswap2L is loaded, it appears to the OS as a single fast underlying swap partition.

Figure 2: Nswap2L Architecture. Its upper layer includes policy code and data structures including the Devices Array, Slotmap, and Worker Thread Pool. Functions in its lower layer pass I/O requests to underlying storage devices and to Network RAM. An I/O request from the OS to the Nswap2L device is added to an internal queue (1), and handled by a worker thread that uses policy code and Slotmap information to direct the request to a specific underlying storage device (2). The OS is notified when an I/O request is complete (3).

It receives read, write, or discard requests from the OS to swap-in, swap-out, or free stored page data. It also contains policies for managing a heterogeneous set of storage devices.

The lower layer implements the mechanism for reading, writing, or discarding pages stored on underlying devices. The upper layer may issue I/O requests to the lower layer in response to I/O request from the OS, or it may generate internal I/O requests to move (prefetch) data between different underlying storage devices; these prefetching I/Os are initiated by its upper layer policies designed to optimize read and write performance of our Nswap2L device. Its Network RAM implementation contains additional policies that may result in pages moving (migrating) from one remote node's RAM to another's in response to changes in available idle RAM on individual cluster nodes. Migration and prefetching I/O operations are transparent to the OS.

Functions in the Linux device mapper system (dmio) [1] are used to pass I/O requests to most of the underlying physical devices that Nswap2L manages. Device mapper has support for stacking device drivers, and is used to implement software RAID and LVM. The lower right of Figure 2 shows the lower level dmio layer. We use asynchronous dmio to pass I/O requests to underlying devices. Callback functions in Nswap2L driver notify the OS when the I/O request is complete.

The Nswap2L I/O path for storing data in Network RAM does not use dmio. Instead, its Network RAM support is implemented as part of the Nswap2L driver (shown in the lower left of Figure 2). There are three reasons why we im-

plemented Network RAM support in the Nswap2L driver. The first is for efficiency; by integrating the Network RAM implementation into the top-level Nswap2L driver, we avoid the extra overhead of having to pass I/O requests through the dmio layer. In addition, both the top-level driver and its Network RAM implementation can share data structures, avoiding duplication of common state. The second reason for this integration is because Network RAM is the one underlying storage device that is always provided by Nswap2L. The final reason for this integration, is that the access path to reading and writing to Network RAM is synchronous (as opposed to the asynchronous dmio I/O). By integrating Nswap2L Network RAM into the top-level driver, Nswap2L worker threads can directly execute Network RAM code to send and receive page data to remote nodes.

3.1.1 Worker Threads, Devices Array, and the Slotmap

Figure 2 shows the main data structures used by the Nswap2L mechanism and policy code. When Nswap2L is loaded as a kernel module, a pool of worker threads is spawned. When it receives an I/O request from the OS, it enqueues it on an internal queue and wakes up a worker thread to handle it (step 1 in Figure 2). Each worker thread is capable of handling enqueued read, write, and discard requests to any underlying device. Using multiple threads improves performance by allowing Nswap2L to process multiple independent requests in parallel.

The Devices Array is used to maintain information about each underlying storage device. Devices Array entries in-

clude the `/dev` path to the device, its capacity, and whether it is a discardable device. Additionally, it maintains usage and performance statistics about the device that are used by Nswap2L policies. Nswap2L requires at least two underlying devices: a local backing storage device (usually a hard drive partition) and Network RAM. Additional underlying devices can be added at runtime through its `/sys` interface.

For each page of stored data, Nswap2L must maintain enough information to retrieve the page from the underlying device currently storing it. This information is stored in a data structure called the Slotmap, which shadows the OS's swap map for our device. The Slotmap records which underlying device stores the page, and also includes a busy bit and a timestamp used to control concurrent access to a slot by multiple worker threads; internal prefetching I/O and external I/O from the OS could overlap, and care must be taken to not lose page data when this uncommon case occurs. The Network RAM implementation shares the Slotmap data structure, using it to encode which remote cluster node (which Network RAM server) stores the swapped-out page.

3.1.2 Handling I/O Requests

Read, write and discard requests to Nswap2L typically originate from the OS swapping subsystem. I/O requests can also occur as part of an internal prefetch between devices. A prefetch consists of a read from one underlying device followed by a write to a different underlying device (and depending on the underlying devices involved, may require a discard to the old device after the write).

When a worker thread handles a read request, it maps the offset into our top-level device to an entry in the Slotmap to find which underlying device stores the page. It sets the busy-bit on the slot to prevent a prefetch operation from moving the page during the read, and either executes Network RAM code to read in the page from a remote server, or issues an asynchronous dmio read to an underlying device. When the read is complete (either by a read callback for reads to dmio or by receiving the page data from the remote Network RAM server), the thread clears the busy-bit on the Slotmap entry and completes the read.

Writes to Nswap2L trigger executing policy code to determine which underlying device should store the data. Writes fall into one of two categories: new writes or overwrites. For new writes, Nswap2L stores no previous version of page data associated with a slot. For overwrites, an underlying device currently stores an older version of the page data. New writes trigger placement policy code and overwrites trigger overwrite policy code.

Currently, Nswap2L implements fixed and alternating placement and overwrite policies. Fixed policies implement an ordering of underlying devices. For example, one fixed ordering policy could order Network RAM first, flash second, and disk third. Another could order flash first, Network RAM second, and disk third. In fixed policies, only when there is no storage capacity in the first device is the second device chosen for placement. For alternating policies, round-robin selection of underlying device is used among the specified set of underlying devices.

An overwrite policy may be different than a placement policy. For example, if a page is currently stored in Network RAM, then it is more efficient to send the new version of the page data to the server currently storing the old version; by sending the write to the server that stores the old copy

of the page, the server can simply overwrite the RAM page storing the old data with the new data. If a new version of the page is sent to another device or server, the server storing the old version must be notified to free its copy of the old page.

Nswap2L's placement and overwrite policies can be changed on-the-fly through an interface it exports in `/sys`. Additionally, Nswap2L collects device profiling data that could be used to implement more dynamic placement and overwrite policies.

Nswap2L is a discardable device, and thus receives notifications from the OS indicating that the OS has freed data stored in certain slots. We use discard operations in two ways. First, we pass discards to underlying devices, like flash, that need discards to maintain extents of clean block for fast writes. Second, we use it for freeing garbage pages from Network RAM. When a discard operation includes pages stored to underlying Network RAM, we notify the Network RAM servers storing the pages that they can free the RAM space holding the pages, making it available to store future swapped-out data.

3.2 Network RAM

Nswap2L's Network RAM implementation builds on our earlier work developing a stand-alone Network RAM system. An in-depth description of its underlying protocols and implementation can be found in [18].

Nswap2L abstractly views Network RAM as one of many possible underlying devices – conceptually, its Network RAM is just like any other underlying device that it manages. In reality, the division between the top-layer Nswap2L device and the bottom-layer Network RAM device is not implemented as an interface between two drivers. Instead, Network RAM is implemented in the lower layer of the Nswap2L driver. As discussed above, this decision was made for efficiency reasons, to allow the Network RAM implementation to share data structures and state with Nswap2L, and because Network RAM is always the one underlying device that Nswap2L will support.

Nswap2L's Network RAM implementation, shown in the lower left of Figure 3, is designed to be scalable, adaptable, and fast. To that end, it is implemented as a completely decentralized system where all nodes are equal peers. Nodes act as both clients (requesting that other nodes store their swapped-out data) and servers (storing swapped out pages from other nodes in its RAM). At any point in time a node may be acting either as a client or as a server depending on the node's current RAM usage. The amount of RAM set aside to store data from other nodes changes dynamically based on local workload. Nodes periodically broadcast information about the amount of free memory set aside to hold pages from other nodes; these messages allow for discovery of other nodes on the network and may inform policy decisions about which nodes should be sent requests. When data are written to Network RAM, the client part uses only local information that it has about other nodes' capacities to choose a server to send the page. There is no central cluster-wide allocator of Network RAM; clients use the information received from other nodes' broadcast messages to learn about remote servers and available Network RAM space.

The Network RAM layer is multi-threaded. Nswap2L worker threads run Network RAM client code to issue reads, writes or discards for different pages to remote Network

Figure 3: Nswap2L Network RAM architecture and data structures, including references to the upper layer data structures and the IPTable. Incoming requests use cached sockets to communicate with remote servers. Servers handle incoming requests, manage cache sizes, and broadcast state information.

RAM servers. The server side spawns threads to handle requests from remote client nodes to read, write, or discard data stored in its RAM. The server threads are separate from the worker threads that are shared by the top-level Nswap2L driver.

The Network RAM client handles read, write, and discard requests from the upper layer of Nswap2L, potentially updating the Slotmap to reflect the result. The client maintains an array of metadata for each remote server node (the IPTable) that includes its IP address, an estimate of the current amount of free RAM it has for Network RAM storage, and a cache of open sockets (to avoid some TCP connection overheads).

To handle a read request, the Network RAM client uses information stored in the Slotmap shared with the Nswap2L upper layer to identify the remote server holding the page. The client acquires a socket from the cache of open sockets to that server and initiates a messaging protocol to retrieve the page from the remote server storing it. Any errors in retrieving the page are returned to the higher layer of Nswap2L for handling.

To handle a write request, the Network RAM client must first choose a server to send the page data based on policies that use IPTable data about nodes' available Network RAM space. The client then sends a request to the server, and if the server accepts the page data, the client updates the Slotmap entry to encode the server to which it sent the page. If a server cannot take the page, the client tries another server or passes the write request back to the upper layer of Nswap2L that will send the write to a different underlying physical device.

Placement policies in the upper-layer of Nswap2L avoid choosing Network RAM when it is full or not available. As Network RAM space becomes available, the Network RAM layer notifies the top-level of Nswap2L so that its policies can resume choosing to write pages to Network RAM.

Discards requests are handled by sending the Network RAM server currently storing the page a message that it

can free the page from its RAM. The Slotmap entry is then cleared for this page; a subsequent write to this slot will trigger the Nswap2L placement policy.

The server side of Network RAM is responsible for managing a cache of local RAM that it makes available for storing swapped-out page data from clients on remote nodes. The server has three primary tasks: to respond to client requests to read, write, and discard data local in the RAM it currently has allocated for Network RAM; to broadcast information about the amount of RAM it has available for storing page data; and to grow and shrink the total capacity of RAM it makes available in response to the current RAM needs of its local workload.

A novel feature of our Network RAM system is that it adapts to changes in RAM usage on individual nodes. The server side monitors local RAM usage and grows and shrinks the amount it makes available for Network RAM based on local usage. As RAM becomes idle, it allocates more for Network RAM, and as the local workload needs more RAM space, it releases RAM it has allocated for Network RAM back to the local paging system. Thus, our Network RAM system will not result in increased swapping on a node due to its allocating some RAM for Network RAM storage. When a server shrinks its Network RAM size, it may migrate some of the remotely swapped pages data it currently stores to other server nodes. It may also send the data back to the originating node (the client who originally sent it the page), and Nswap2L running on that node will write the page onto a different underlying storage device.

3.3 Prefetching

As a two layer device driver, Nswap2L has the ability to move data between underlying devices transparently to the OS. We call this process prefetching. Prefetching data is a unique feature of Nswap2L; through the use of placement and prefetching, Nswap2L has the potential to take advantage of the best performance characteristics of different underlying devices. For example, Network RAM has very fast

writes, but its availability fluctuates and may be very limited at times depending on the cluster workload. Flash capacity is fixed, and has relatively fast reads, but writes are slow because of the compulsory erase cycle. If Nswap2L chose to originally write pages to Network RAM, then prefetch to flash, we could experience the benefits of each underlying device: fast writes to Network RAM, fast reads from flash, and relatively low usage of Network RAM in the long run.

The 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 may maintain some free Network RAM space available for faster writes, while still allowing for good read performance from flash or Network RAM. Alternatively, pages initially written to a slower underlying device like disk could be prefetched to flash or Network RAM later on in order to benefit from faster subsequent reads. Finally, we might want to spread pages between multiple underlying devices in order to benefit from read parallelism.

Prefetching I/O is initiated internally by the Nswap2L policy rather than as a request from the OS. At any given time a page that was originally placed on one underlying device could be prefetched to a different underlying device (and its associated Slotmap entry updated to reflect its new location).

Nswap2L defines prefetching policies in terms of four distinct questions. The first prefetching question is when (or how often) to prefetch. If Nswap2L is busy handling I/O requests from the OS, prefetching may impede swapping performance. Whenever Nswap2L is not busy handling I/O requests from the OS, it has more opportunity to re-optimize the distribution of pages stored on underlying devices.

The second prefetch policy question is which pages to prefetch. Our current policies include LRU (least recently swapped to slot) MRU, Round-Robin and Random. Ideally, frequently used pages are available as quickly as possible once the OS requests them, so moving pages from slower underlying devices to faster ones could help achieve this goal. LRU might perform well in workloads involving repeated sequential scans, while MRU typically performs well on workloads exhibiting high degrees of temporal locality. Round Robin and Random are simple policies that may perform well in most cases.

The third question is how many pages to prefetch. Policies for this question may consider how busy the worker threads are, how frequently prefetching is occurring, and how many pages have been recently swapped out to our device.

The fourth and final policy question is which source and destination devices to prefetch between. This answer depends on the current placement policy, the distribution of pages between underlying devices, the current load on Network RAM, and the relative speeds of the underlying devices.

A dedicated prefetching thread in our top-level Nswap2L driver runs policy code and creates a set of prefetch I/O operations that are submitted to Nswap2L's internal request queue and handled by its worker threads. We use the same mechanism to execute prefetch I/O operations as we do to handle I/O requests that originate from the kernel. The difference is in how we end requests: the OS should not be notified on completion of a prefetch I/O, but must be on

completion of a regular I/O.

3.4 /sys Interface

One of the core features of Nswap2L is that it is extensible and tunable; changes to policies and changes to the set of underlying devices should be supported as Nswap2L is running on a node. We implement a tunable interface to our device using the Linux /sys interface. Users can read or write to /sys files associated with our device to trigger Nswap2L to change policies, to add a new underlying device, or export state and profiling data.

For example, in order to change its placement policy, a user simply needs to write the new policy number into the /sys placement_policy file. Similarly, to add a new underlying device to Nswap2L, a user just writes the device path to the /sys add_device file and Nswap2L will immediately add it as a new underlying device. Parameters of specific placement, overwrite and prefetching policies are also tunable through /sys.

Additionally, Nswap2L exports information and statistics through /sys. These include information about the upper layer (number of active worker threads, number of pending I/O operations to be handled, and placement policy information), the Network RAM module (connected servers, and estimated free space on each server), and performance data about underlying devices. For each underlying device, Nswap2L gathers statistics on the number of read, write, and discard operations handled, along with information about the average parallelism of the underlying device. Nswap2L optionally gathers timing information about the completion time for each type of I/O operation. These statistics about performance and load can be used by policy code in Nswap2L.

4. EVALUATION

We present results of performance studies of our Nswap2L system. We assessed overheads of its two-level design, measured performance of benchmark programs that use Nswap2L as a swap device, and evaluated the performance benefits of its policies. Our experiments were run on a 16 node cluster with 1 Gbit or 10 Gbit Ethernet. Nswap2L was loaded as a swap device on individual cluster nodes that were running an unmodified Linux 4.0.4 kernel. Nswap2L was configured to manage a set of underlying devices, including its own implementation of Network RAM, disk HDD, and flash SSD partitions.

4.1 Nswap2L Baseline Performance

We measured the performance and overhead of Nswap2L to establish correctness, to evaluate baseline running behavior, and to motivate policy development and tuning. For these experiments, we ran two simple benchmark programs that execute sequential or random access patterns over a large heap memory allocation, resulting in swapping.

In our first study, we compared the relative performance of swapping to different underlying storage devices through Nswap2L. We used the sequential benchmark to collect raw underlying device I/O times for swapping to Network RAM, flash and disk through Nswap2L. Table 4.1 shows the results for runs on a 10-Gbit network. As anticipated, individual page I/O is fastest to Network RAM and slowest to disk.

We next evaluated varying the number of worker threads, varying the underlying network speed, and varying the un-

Backing Device	Time Per Op (μ s)
Network RAM	3.58
Flash	5.28
Disk	9.60

Table 1: The average completion time per page (4KB) I/O comparing placing on Network RAM, flash or disk. These data were for 128 worker threads on 10 Gbit Ethernet.

Figure 4: Completion time for randomized workload benchmark, for varying numbers of threads on different backing stores. Network RAM on 10Gbit is faster than flash, and both are much faster than disk. Note that the effect of additional threads differs between Network RAM and other backing stores.

derlying devices to which Nswap2L swaps page data. Adding more worker threads will increase the upper-bound on the degree of parallelism in handling read, write and discard requests, but more threads also adds additional overhead, requiring nominal extra per-thread state and incurring additional context switching.

Figure 4 shows average runtime results of the random benchmark program, comparing different numbers of worker threads (the horizontal axis) swapping to different underlying devices (disk, flash, Network RAM on 1Gbit Ethernet, and Network RAM on 10Gbit Ethernet). The results show that swapping to Network RAM on 10-Gbit Ethernet outperforms all other devices, and that swapping to disk is slowest. Flash performs better than swapping to Network RAM on 1-Gbit Ethernet.

The overall results in Figure 4 match our expectation of the relative performance of each underlying device and provides validation that Network RAM systems can provide good performance in a cluster environment. The results also show that increasing the number of threads to between 64 and 128 works well for Network RAM and flash.

Although it is difficult to tell from this graph, the 256 thread results for flash were slightly slower than the 128 thread flash results, and adding threads beyond 128 does not result in improvement when swapping to Network RAM. These results indicate that while there is a benefit to increased I/O parallelism, there is also a limit to this benefit as the number of threads grows beyond some size. These results also hint at an interesting performance relationship between adding additional threads and the synchrony of the underlying device. Specifically, for devices using the asyn-

chronous `dmio` interface, adding additional threads beyond some threshold hinders performance, increasing the overall completion time. However for synchronous Network RAM, additional threads reliably decrease the completion time. This result makes sense, as threads interfacing with synchronous underlying devices (such as Network RAM) can only complete one I/O at a time; additional threads increase the degree of parallelism of the underlying device. However, asynchronous interfaces allow multiple I/O requests to be pending at once, so the underlying device parallelism is not limited solely by the number of threads.

Our baseline performance studies support our Nswap2L design. The Nswap2L Network RAM on 10Gbit Ethernet provides the fastest storage. The results also hint at the benefit of the adaptability and tune-ability of Nswap2L to make best use of a heterogeneous set of storage. Our study varying the number of threads suggests that mechanisms to dynamically adjust the number of worker threads depending on the underlying devices could improve overall performance.

The observation that Network RAM performance is dependent on network speeds suggests two important points for future policy development. First, the relative performance of network RAM to other backing store devices, like flash, differs depending on network bandwidth; placement and overwrite policies at the Nswap2L upper layer choosing whether to direct requests to remote nodes may incorporate information about low-level device performance. Second, within the Network RAM module, policies may benefit from incorporating information about relative proximity to other servers, perhaps preferentially choosing to request storage from servers in the same rack or switch in order to achieve the fastest completion time for network requests.

4.2 Data Intensive Benchmarks

The main motivation for Nswap2L is to efficiently make use of a heterogeneous collection of cluster-wide backing store to improve the performance of data-intensive workloads running on clusters, currently by providing fast backing store for VM system swap space.

To evaluate our main goal, we ran experiments measuring the performance of two large parallel applications: the Linpack HPL benchmark [8], and an MPI implementation of the Odd-Even sorting algorithm. Both benchmarks were tuned to sizes that would result in swapping on some cluster nodes. We ran experiments measuring the performance of these benchmark programs when swapping to our Nswap2L device, and compared Nswap2L placement policies to underlying Network RAM, flash and disk.

Figure 5 displays average run-times for the HPL benchmark. The results show a significant performance improvement when swapping to Nswap2L’s Network RAM. The HPL benchmarks shows a speed-up of 1.3 times over swapping to flash and a speed-up of 1.7 over swapping to disk.

The HPL benchmark also includes results that measure overheads added by our two-level design; we ran this benchmark swapping directly to flash and to disk, as well as swapping to flash and disk through Nswap2L. Swapping to disk through Nswap2L adds 5.0% to the total runtime over swapping directly to disk, and swapping to flash through Nswap2L adds 6.7% to the total runtime vs. directly swapping to flash. Even with this extra cost, swapping to Nswap2L’s Network RAM significantly outperforms swapping directly

Figure 5: Average Runtime (in seconds) of the HPL Benchmark comparing different underlying swap devices. *Nswap2L* to Network RAM, flash and disk are shown in the left (blue) bars, and directly to flash and disk on the right (orange).

Figure 6: Average Runtime (in seconds) of the MPI Odd-Even Sorting Benchmark comparing different underlying swap devices: *Nswap2L* to Network RAM, *Nswap2L* to flash and *Nswap2L* to disk.

Placement Policy	Normalized Runtime
Network RAM	1.0
Alternating	1.1
Flash	1.3

Table 2: Normalized Runtime (relative to Network RAM) of three Placement Policies: to flash, to Network RAM, and alternating between the two. Performance for the alternating policy is between performance flash only and Network RAM only. Run with 64 worker threads on 10 Gbit Ethernet.

to flash or disk.

Figure 6 shows the results for the MPI Odd-Even Sorting benchmark. The results show a speed-up of 2.0 times over swapping to flash and a speed-up of 5.8 over swapping to disk. The higher speed-up values for Odd-Even sort vs. HPL reflect differences in the total amount of swapping each benchmark incurs. The sorting benchmark has more swapping I/O than HPL, and thus can benefit more from swapping to the faster Nswap2L backing store device.

4.3 Evaluating Nswap2L Policies

A main feature of Nswap2L is that it offers the opportunity to develop and test policies for swapping decisions. Ideally, the benefit added by effectively using a heterogeneous set of backing store devices outweighs the added overhead of Nswap2L’s two-layer device design.

In addition to testing placement policies that direct all writes to a single underlying device, we tested a simple policy that alternates sending writes to Network RAM and flash, striping requests between the two underlying devices.

Table 4.3 shows the average run-times (normalized to Network RAM placement) for runs of the sequential kernel benchmark under the three placement policies: to Network RAM only; to flash only; and alternating between Network RAM and flash. The results show what we expected – the performance of the alternating placement policy is about halfway between the performance of the to flash alone and to Network RAM alone policies. The alternating policy is too simple to result in an overall improvement in runtime over the placement to Network RAM alone policy; any benefit of increased read parallelism from the alternating policy is outweighed by the slower flash writes, and this simple alternating policy directs half of the writes to flash. The results do, however, suggest that effective placement policies that incorporate information about the speeds of reads and writes of underlying physical devices and the current available capacity of underlying physical storage may result in improved performance over a single underlying device. Because Nswap2L collects these types of data, exports them via `/sys`, and also exports changing placement policies vis `/sys`, it has the capability to implement more adaptable data placement policies that achieve better performance.

These data also provide some motivation for Nswap2L’s prefetching policies. We ran an experiment of the sorting benchmark with Nswap2L’s placement policy choosing Network RAM, due to its significantly faster writes than other devices, and with a Prefetching Policy that prefetches some pages from Network RAM to flash. Our conjecture was that even though flash reads are slower than reads from Network RAM, by distributing reads across multiple underlying de-

Figure 7: Average Runtime (in seconds) of the MPI Sorting Benchmark comparing swapping to underlying flash only, to underlying Network RAM only, and a Placement Policy of writing to Network RAM with Prefetching from Network RAM to flash (subsequent reads of page data may be to Network RAM or to flash in the Prefetching result).

vices, an increase in read parallelism could result in faster performance then directing reads to the fastest underlying device. Because flash writes are slower than flash reads, choosing a placement policy that always writes to Network RAM may lead to overall performance improvements (vs. our more naive alternating placement policy).

Figure 7 shows results comparing the MPI Sorting Benchmark run on Nswap2L with placement policy choosing flash with no prefetching, choosing Network RAM with no prefetching, and choosing Network RAM with prefetching from Network RAM to flash.² The results show a slight improvement in total runtime for the run with prefetching from Network RAM to flash (162 seconds vs. 159 seconds). These differences are not statistically significant. However, it should be noted that in the runs with prefetching, some page data are being read in from slower flash, whereas in the run without prefetching all page data are being read in from faster Network RAM. This result indicates that there may be a benefit from distributing reads over multiple devices, even in cases like this where data are being prefetched from a device with faster reads to a device with slower reads.

Another potential advantage of Nswap2L’s prefetching support is to move data that were originally placed on a slower underlying device to a faster device that becomes available later. For example, if Network RAM is full at the time of a write, Nswap2L’s placement policy will choose to send the write to flash or disk. Later, if Network RAM space becomes available, Nswap2L’s prefetching policy may move

²These results were obtained for different problem sizes than the previous sorting results. Thus, the runtime cannot be compared across the two graphs.

Figure 8: Average Runtime (in seconds) of the MPI Sorting Benchmark comparing swapping to underlying flash only, to underlying Network RAM only, and to flash for first 300 seconds of run, and then to Network RAM after that as it becomes available during the execution.

some page data stored on the slower underlying device to Network RAM to improve subsequent swap-in requests from the OS, and its placement policy will choose Network RAM for new writes.

Figure 8 shows results from an experiment running the MPI Sorting benchmark on Nswap2L with adaptable policies that take advantage of Network RAM when it becomes available. At the beginning of the sorting application’s execution, there is no available Network RAM space and Nswap2L’s placement policy writes to underlying flash. About 5 minutes into its execution, Network RAM space becomes available and Nswap2L’s policies make use of this faster underlying device to place swapped out page data.

The results show a 1.6 times speed-up of our adaptive policy run (middle bar) vs. the runs that swap only to flash without using prefetching and placement policies that adapt to changes in underlying storage capacity (right bar). For this experiment, the exact runtime values are not important (we could have made Network RAM space available earlier than 5 minutes into the sorting application’s execution, and achieved a better speed-up result). Instead, what is important from this experiment is its demonstration of how Nswap2L can adapt on-the-fly to changes in underlying device capacity to make better placement policy decisions that result in improvements in application performance.

Our experimental results support our design of Nswap2L. They show that its two-level design adds minimal overhead to directly swapping to the underlying devices. Further, these overheads are reclaimed by the performance gains of its implementation of fast Network RAM and by its placement and prefetching policies that dynamically manage a heterogeneous set of underlying physical storage devices to

improve data-intensive application performance.

5. CONCLUSION

With the increasing prevalence of cluster computing, there is strong motivation to manage cluster memory in a holistic way. Jobs running on cluster computers often cause an imbalance in cluster memory availability, which motivates the usage of Network RAM. At the same time, each of these nodes typically has local secondary storage devices. Ideally, local storage devices and Network RAM could be combined in a way that provides better overall performance.

Nswap2L is our novel two-level Linux device driver for cluster computers, developed with the goal of taking advantage of both Network RAM and other physical storage devices. Nswap2L presents itself to the OS as a single storage device, which isolates the complexity of effectively managing these storage options from the operating system. Internally, Nswap2L manages these devices using interchangeable policies which dictate where data should be placed, and potentially relocated, in order to achieve optimal performance.

Our results demonstrate Nswap2L's potential to act as a fast backing storage device, and support its adaptive management of underlying physical storage. Nswap2L adds minimal overhead to the latency of the local storage devices, and its Network RAM has the capability to outperform flash SSD.

One direction for future work is to continue to develop tunable policies that will make the best use of cluster resources. We plan to investigate more adaptable policies that may incorporate real-time performance data when choosing how to place and prefetch pages. We also are working on implementing better heuristics for dynamically changing the amount of RAM reserved on each node for Network RAM space. We additionally plan to investigate other uses for the Nswap2L driver. In particular, we are currently investigating its use as backing store for file systems, with an initial focus on storing temporary file system data. This feature would allow data intensive applications that write partial results to temporary files to take advantage of Nswap2L's fast, adaptable, backing store.

Acknowledgements

This work is supported by the National Science Foundation grant NSF:CNS:CSR-1116224. We thank our anonymous reviewers for their insightful feedback on our paper. We thank Alec Pillsbury for his help with the HPL benchmark and with testing prefetching policies, and Sam White for his help with early feasibility studies related to this project. Finally, we thank Jeff Knerr for administering and managing the cluster system that we used for Nswap2L development and evaluation.

6. REFERENCES

- [1] Device mapper, red hat inc. <http://sources.redhat.com/dm>.
- [2] A. Acharya and S. Setia. Availability and Utility of Idle Memory on Workstation Clusters. In *ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, pages 35–46, May 1999.
- [3] T. Anderson, D. E. Culler, D. A. Patterson, and the NOW Team. A case for NOW (networks of workstations). *IEEE Micro*, February 1999.
- [4] R. H. Arpaci, A. C. Dusseau, A. M. Vahdat, L. T. Liu, T. E. Anderson, and D. A. Patterson. The Interaction of Parallel and Sequential Workloads on a Network of Workstations. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1995.
- [5] A. M. Caulfield, J. Coburn, T. Mollov, A. De, A. Akel, J. He, A. Jagatheesan, R. K. Gupta, A. Snively, and S. Swanson. Understanding the impact of emerging non-volatile memories on high-performance, IO-intensive computing. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '10*, Washington, DC, USA, 2010. IEEE Computer Society.
- [6] 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*, page 19. USENIX Association, 1994.
- [7] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
- [8] J. Dongarra. The linpack benchmark: An explanation. In *Proceedings of the 1st International Conference on Supercomputing*, 1988.
- [9] M. D. Flouris and E. P. Markatos. *Network RAM, from High Performance Cluster Computing: Architectures and Systems*, chapter 16. 1999.
- [10] 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*, page 6. ACM, 2007.
- [11] <http://stxxl.sourceforge.net/>. Stxxl: Stxxl: Standard template library for extra large data sets.
- [12] L. Iftode, K. Li, and K. Petersen. Memory servers for multicomputers. In *Compcon Spring'93, Digest of Papers.*, pages 538–547. IEEE, 1993.
- [13] 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*, pages 101–110. IEEE, 2015.
- [14] 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*, 86(3):744–750, 2013.
- [15] P. A. Marin Lopez, Arturo Garcia Ares. The network block device. *Linux J.*, 2000(73es), May 2000.
- [16] E. P. Markatos and G. Dramitinos. Implementation of a Reliable Remote Memory Pager. In *USENIX 1996 Annual Technical Conference*, 1996.
- [17] Michael J. Feeley and William E. Morgan and Frederic H. Pighin and Anna R. Karlin and Henry M. Levy and Chandramohan A. Thekkath. Implementing Global Memory Management in a Workstation Cluster. In *15th ACM Symposium on Operating Systems Principles*, December 1995.

- [18] T. Newhall, S. Finney, K. Ganchev, and M. Spiegel. Nswap: a network swapping module for linux clusters. In *Lectures in Computer Science*, 2003. Proceedings of Euro-Par'03 International Conference on Parallel and Distributed Computing.
- [19] T. Newhall and D. Woos. Incorporating network ram and flash into fast backing store for clusters. In *Proceedings of 2011 IEEE International Conference on Cluster Computing*, 2011.
- [20] 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)*, pages 385–398, 2013.
- [21] J. Oleszkiewicz, L. Ziao, and Y. Liu. Parallel network RAM: Effectively utilizing global cluster memory for large data-intensive parallel programs. In *IEEE 2004 International Conference on Parallel Processing (ICPP'04)*, 2004.
- [22] S. Park and K. Shen. A performance evaluation of scientific I/O workloads on flash-based SSDs. In *Workshop on Interfaces and Architectures for Scientific Data Storage*, 2009.
- [23] S. Robbins. Ram is the new disk. . . . *InfoQ News*, Jun, 2008.
- [24] D. Roberts, T. Kgil, and T. Mudge. Integrating nand flash devices onto servers. *Commun. ACM*, 52:98–103, April 2009.
- [25] 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*, pages 4–12. ACM, 2011.
- [26] M. Saxena and M. M. Swift. FlashVM: virtual memory management on flash. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 14–14. USENIX Association, 2010.
- [27] M. Saxena, M. M. Swift, and Y. Zhang. Flashtier: A lightweight, consistent and durable storage cache. In *EuroSys'12*. ACM, 2012.
- [28] 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*, pages 465–474. IEEE, 2015.
- [29] R. Weiss. Exadata smart flash cache and the sun oracle database machine. Oracle White Paper, <http://www.oracle.com/database/exadata.html>, October 2009.
- [30] P. Werstein, X. Jia, and Z. Huang. A Remote Memory Swapping System for Cluster Computers. In *pdcats*, pages 75–81. IEEE, 2007.