Incorporating Network RAM and Flash into Fast Backing Store for Clusters

Tia Newhall and Douglas Woos

Computer Science Department
Swarthmore College
Swarthmore, PA 19081, USA
{newhall,dwoos1}@cs.swarthmore.edu

***Abstract*—We present Nswap2L, a fast backing storage system for general purpose clusters. Nswap2L implements a single device interface on top of multiple heterogeneous physical storage devices, particularly targeting fast random access devices such as Network RAM and flash SSDs. A key design feature of Nswap2L is the separation of the interface from the underlying physical storage; data that are read and written to our "device" are managed by our underlying system and may be stored in local RAM, remote RAM, flash, local disk or any other cluster–wide storage. Nswap2L chooses which physical device will store data based on cluster resource usage and the characteristics of various storage media. In addition, it migrates data from one physical device to another in response to changes in capacity and to take advantage of the strengths of different types of physical media, such as fast writes over the network and fast reads from flash. Performance results of our prototype implementation of Nswap2L added as a swap device on a 12 node Linux cluster show speed–ups of over 30 times versus swapping to disk and over 1.7 times versus swapping to flash. In addition, we show that for parallel benchmarks, Nswap2L using Network RAM and a flash device that is slower than Network RAM can perform better than Network RAM alone.**

I. Introduction

"Science has entered a data–intensive era, driven by a deluge of data being generated by digitally based instruments, sensor networks, and simulation devices." [1] As a result, designing systems that efficiently support data–intensive computing is increasingly important. As the disparity between the speeds of magnetic disk and other hardware such as RAM, inter–connection networks, and flash continues to grow, the cost of accessing disk will increasingly become the bottleneck to system performance. It is almost certain that this disparity will eventually make magnetic disk obsolete. In the meantime, it will be increasingly important to develop systems that can avoid using disk as much as possible and can make the best use of emerging fast storage technologies in clusters. It is also likely that cluster storage will be more heterogeneous in the future; at the very least, Network RAM will continue to rival solid state drives. In addition, swap and local temporary file system storage may not be managed entirely by the OS running on individual cluster nodes, particularly as network–shared storage and Network RAM become more common. Replacing magnetic disk with a heterogeneous set of fast, random access storage devices will require changes to OS subsystems that are designed assuming that swap and local temporary file data are stored on local disk and that this storage is managed solely by the OS running on individual cluster nodes.

In general purpose clusters (clusters that support multiple users and run a wide range of program workloads), resource utilization varies due to dynamic workloads. Because resource scheduling is difficult in this environment, there will often be imbalances in resource utilization 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 allows individual cluster nodes with over–committed memory to swap their pages over the network and store them in the idle RAM of other nodes.

Data intensive applications running on general purpose clusters, such as parallel scientific or multimedia applications, often perform large amounts of I/O either indirectly due to swapping or directly due to temporary file accesses. If swap and temporary file system data can be stored in Network RAM or solid state storage, these applications will run much faster than when swap and temporary file system data are stored on magnetic disk.

Nswap2L is an extension of Nswap, our Network RAM system for Linux clusters. Nswap2L is a scalable, adaptable system that is a solution to the problem of supporting a het–erogeneous set of backing storage devices in general purpose cluster systems, some of which may have changing storage capacities (Network RAM) and some of which may not be completely under the control of the local OS (Network RAM and other shared network storage.)

Nswap2L implements a novel two–level device design. At the top level is a simple interface presented to cluster operating systems and user–level programs. The top level manages the set of heterogeneous low–level physical storage devices, choosing initial data placement and migrating data between devices in response to changes in cluster–wide resource utilization and storage capacity, and to take advantage of strengths of different media, with a goal of making out–of–core data access as fast as possible. Higher level system services, such as temporary file systems or swapping systems, interact with our top–level single

[1] Michael Norman, Interim Director of SDSC from "SDCS to Host 'Grand Challenges in Data–Intensive Discovery' Conference", HPCwire, August 3, 2010.

978-0-7695-4516-5/11 $26.00 © 2011 IEEE
DOI 10.1109/CLUSTER.2011.22

IEEE computer society

device interface to take advantage of heterogeneous, adaptive, fast, cluster-wide storage. By moving most device-specific management into Nswap2L, our system frees cluster operating systems from needing specialized policies in swapping and file subsystems that are tuned for every different type of physical storage device. Our current implementation of Nswap2L can be added as a swap device on individual cluster nodes. Our future work includes extending its use, particularly for temporary local file system storage.

The rest of the paper is organized as follows: Section II discusses related work in fast random access storage; Section III presents background on Nswap's Network RAM implementation; Section IV presents experimental studies motivating our two-level design; Section V discusses the design and implementation of Nswap2L; Section VI presents results evaluating Nswap2L; and Section VII concludes and discusses future directions.

II. Related Work

The work most related to ours includes other work in Network RAM, and work in incorporating solid state storage into systems. Network RAM uses remote idle memory as fast backing store in networked and cluster systems. This idea is motivated by the observation that network speeds are increasing more rapidly than disk speeds. In addition, because disk speeds are limited by mechanical disk arm movements and rotational latencies, the disparity will likely continue to grow. As a result, accesses to local disk will be slower than using remote idle memory as backing store and transferring blocks over the faster network. Further motivation for Network RAM is supported by several studies [1], [2], [3] showing that large amounts of idle cluster memory are almost always available.

There have been several previous projects examining the use of remote idle memory as backing store for nodes in networks of workstations [4], [3], [5], [6], [7], [8], [9], [10], [11]. Some incorporate Network RAM into an OS's paging system, some into an OS's swapping system, and others into an OS's file system as a cooperative cache for file data. Most use a central server model, wherein a single node is responsible for managing the Network RAM resource and clients make memory requests to this central server. A few, including ours, are completely distributed, where peers running on individual cluster nodes can make Network RAM allocation and deallocation decisions without having to contact a central authority. Our system has the unique quality of adapting to changes in cluster RAM usage by migrating remotely swapped pages between nodes in response to these changes. This allows our system to be persistent on a cluster and ensures that it will not interfere with cluster application RAM use.

There has also been much recent work that examines incorporating emerging solid-state storage into systems [12], [13], [14], [15], [16]. Different uses include incorporating flash into the memory hierarchy and using it as a fast out of core buffer cache in database management systems. Some work has examined how flash can be used in high-performance computing. In [15], the authors compare scientific workload run times for different backing storage devices: two 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 [16], the authors model several emerging storage technologies, including phase-change memory and spin-torque transfer memory. They find that these newer technologies can lead to significant performance improvements, especially if the OS is modified to remove the classical assumption that I/O is very slow compared to computation.

The FlashVM project [14] examines using flash as the virtual memory system's paging and swapping device. They show that many changes need to be made to the operating system in order for flash to be used efficiently by the VM system. These include changes to device scheduling policies that are are optimized for magnetic disk, zero-page sharing to avoid some flash wear-out, and call-backs to the flash device when blocks are freed so that erasure blocks can be cleaned.

We anticipate that general purpose clusters will increasingly incorporate more flash memory, but will also continue to use disk until the cost of flash, or other fast backing store, significantly decreases. In addition, issues with flash wear-out and write degradation require solutions before flash SSD completely replaces magnetic disk. We also anticipate that Network RAM will continue to rival fast solid state storage.

III. Nswap's Adaptable Network RAM

Nswap [17], [18] is our Network RAM system for Linux clusters. It is implemented as a loadable kernel module that is easily added as a swap device on cluster nodes. Nswap runs entirely in kernel space on an unmodified [2] Linux 2.6 kernel; it transparently provides network swapping to cluster applications. Nswap is designed to be efficient, to adapt to changes in nodes' RAM use, and to scale to large-sized clusters.

Each Nswap node is an equal peer running both the Nswap Client and the Nswap Server (shown in Figure 1). The client is active when a node is swapping. The server is active when a node has idle RAM space that is available for storing page data swapped to it from remote nodes. At any point in time a node is acting either as a client or a server, but typically not both simultaneously; its role changes based on the current RAM needs of its local processes.

Nswap is designed to scale to large clusters using an approach similar to the Mosix [19] design for scalability. To find available remote swap space, each node uses only its own local information about available idle RAM in the cluster (shown as the IPTable in Figure 1.) This information does not need to be complete nor completely accurate. The IPTable stores an estimate of the amount of available idle RAM on some other nodes. These values are updated when nodes

[2] Currently, we require a re-compile of the kernel to export two kernel symbols so that our module can read the kernel's swap map for our device, but no kernel code is modified

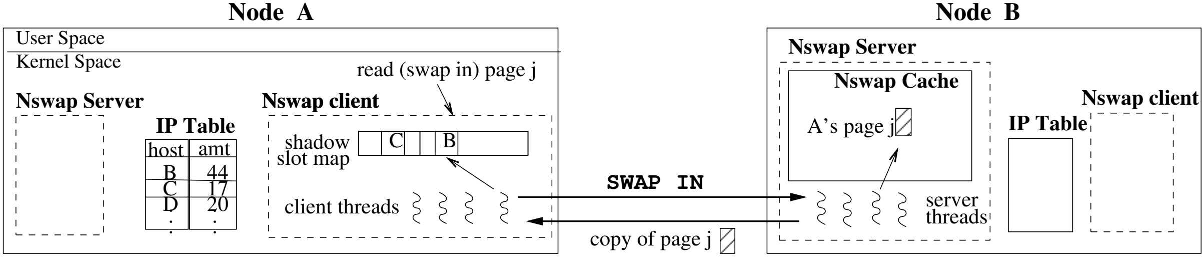

Fig. 1. Nswap System Architecture. *Node A shows the details of the client including the shadow slot map used to store information about which remote servers store A's pages. Node B shows the details of the server, including the Nswap Cache of remotely swapped pages. In response to the kernel swapping in (out) a page to our Nswap device, a client thread issues a SWAPIN (SWAPOUT) request to read (write) the page from a remote server.*

periodically broadcast their available idle RAM capacities. Each Nswap node is also solely responsible for managing just the portion of its local RAM that is currently available for storing remotely swapped pages (shown as the Nswap Cache in Figure 1.) Because there is no central authority managing network RAM allocation, Nswap can easily scale to large–sized clusters.

The multi–threaded Nswap Client is implemented as a device driver for our pseudo–swap device. A client thread is activated when the kernel makes a swap–in or swap–out request to our swap "device", just as it would to a driver for a swap partition on disk. For any swap device, the kernel has a data structure called a swap map used to keep track of each 4K page of allocated swap space on the device. The Nswap Client keeps additional information about each page of swap space in a data structure called the shadow slot map. There is one shadow slot map entry per kernel swap map entry. When a client thread receives a swap–in request from the kernel, it looks up the server ID in the corresponding shadow slot map entry and sends a swap–in request to the Nswap Server storing the page. For example, in Figure 1, the client thread handling a read request from the kernel for swap slot `j` looks up entry `j` in the shadow slot map to find that server `B` stores the page. When the client receives a swap–out request from the kernel, it finds a good Nswap Server candidate using IPTable data, updates its shadow slot map entry with this server's ID, and send the server a swap–out request and the page to store.

The multi–threaded Nswap Server is responsible for managing the portion of its RAM currently allocated for storing remotely swapped page data (the Nswap Cache). Server threads receive swap–in and swap–out messages from Nswap Client nodes. On a swap–in request, a server thread does a fast look–up of the page in its Nswap Cache and sends a copy of the page to the requesting client.

A novel feature of Nswap is its adaptability to changes in cluster–wide RAM usage. The amount of RAM Nswap makes available on each node for storing remotely swapped page data changes with the RAM needs of the workload. The Nswap Server on each node is responsible for growing and shrinking the amount of RAM it makes available for storing remotely swapped page data (its Nswap Cache capacity). It changes its Nswap Cache capacity in response to local memory use: when local processes need more RAM space, the Nswap Server releases pages from its Nswap Cache back to the local paging system; when idle RAM becomes available, the Nswap Server allocates some of it, increasing the size of its Nswap Cache. When an Nswap Server gives RAM back to the paging system, remotely swapped page data stored in that RAM are migrated to other Nswap Servers that currently have available Nswap Cache space. If no available Nswap Cache space exists, pages are migrated back to their owner's node and written to swap space on local disk. Nswap's adaptability is key; it allows Nswap to be persistent on clusters and not interfere with the RAM needs of cluster applications.

TABLE I
READ AND WRITE ACCESS TIMES TO FLASH AND NETWORK RAM. *The data show the time in seconds to read/write 500,000 4KB pages to each device via /dev. Each value is the average of 10 runs.*

Operation	Flash SATAI	Network RAM 1Gb Ethernet
Read	23.5 secs	21.7 secs
Write	32.7 secs	20.2 secs

IV. MOTIVATION FOR TWO LEVEL DRIVER SYSTEM

In support of the two–level design of Nswap2L, we conducted experiments comparing Network RAM and flash speeds. All experiments were run on a 12 node cluster, each node running a 2.6.30 Linux kernel and connected by a 1 Gigabit Ethernet switch [3]. Our first experiment compares direct reads and writes through `/dev` to flash and to Nswap's Network RAM. We measured the total time to perform a large sequential write to the device followed by a large sequential read. The amount of data transfered is larger than physical RAM, thus all reads should require physical device I/O and will not be satisfied by the file cache.

The results, in Table I, show that Nswap outperforms flash devices for reads and writes (21.7 vs. 23.5 seconds and 20.2 vs. 32.7 seconds.) However, the read speeds from flash are close to those from Nswap. The relative read performance will depend on the particular devices; however, based on this experiment as well as other studies of flash performance [16], [14], [20], we anticipate that reads to flash will rival, and may outperform,

[3]Nodes have Pentium4 processors, 80GB Seagate Barracuda7200 IDE disk drives, and Intel X25–M SATAI 80GB Flash SSD drives

TABLE II
KERNEL BENCHMARKS COMPARING SWAPPING TO FLASH VERSUS SWAPPING TO NSWAP'S NETWORK RAM.

Workload	Flash	Nswap
WL1: sequential writes and reads	253.34 secs	232.50 secs
WL2: random writes and reads	181.60 secs	119.49 secs
WL3: WL1 plus disk file system I/O	208.63 secs	147.31 secs
WL4: WL2 plus disk file system I/O	259.24 secs	120.06 secs

reads to Network RAM and that writes to Network RAM will outperform writes to flash. This experiment motivates choosing Network RAM as the initial target of written pages, and then migrating pages from Network RAM to flash so that some subsequent read requests can be satisfied by faster flash; or, in the case when flash and Network RAM are equally good, more reads can be handled in parallel by distributing them over both Network RAM and flash. We do not want to write to both flash and Network RAM simultaneously since the slowest device will determine the time it takes to satisfy the write request. Therefore, prefetching and migration will be a better way to take advantage of the strengths of each device.

Our second experiment compares using flash to using Nswap's Network RAM as a swap device. The experiment measures the runtime of four memory-intensive kernel benchmark programs, each designed to stress different cases when disk I/O should be particularly good or bad: WL1 consists of iterations of a large sequential write followed by a large sequential read to virtual memory and is the best case for swapping to disk because the memory access patterns match the swap allocation patterns, and disk seek time is minimized; WL2 writes and reads to random memory locations and triggers random read and write access to the swap partition, increasing disk head movement within the swap partition; finally, WL3 and WL4 consist of one WL1 or WL2 process and another process that reads and writes to a local file partition, further stressing disk arm movement between the swap and file partitions.

The results, in Table II, show that all four benchmarks perform better when Network RAM is used as the swap partition. However, the results for flash are comparable. Because the flash execution times include both reads and writes to the flash drive, our proposed two-level design that makes use of both flash and Network RAM has the promise to outperform either flash or Network RAM alone. Even if Network RAM is always faster than flash, our system will allow for data stored in Network RAM to be moved to flash when there is not enough cluster-wide idle RAM available for Network RAM.

Finally, we evaluate prefetching opportunities in real workloads by examining swap access patterns for several parallel benchmark programs: Radix from SPLASH-2 [21], [22], IS from NAS Parallel [23], and the Linpack HPL [24] benchmark. Figure 2 shows read and write accesses to swap slots over the run of Radix [4]. The results show locality in swap slot accesses, particularly a clear pattern of sequential writes, which match

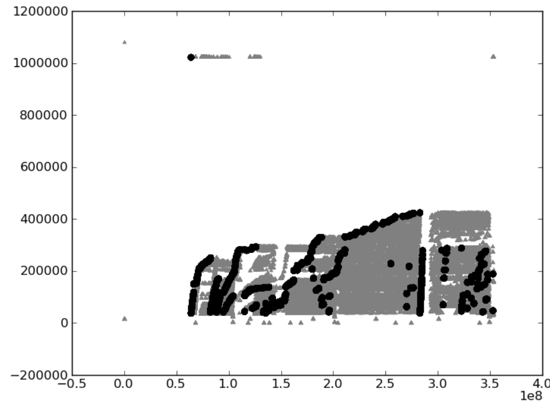

Fig. 2. Read and Write accesses to Swap Slots over the execution of the Radix SPLASH-2 benchmark. *The x-axis is time and the y-axis is swap slot number. Reads are shown in grey, writes in black.*

the OS's swap allocation policy. The data also show that writes are much less frequent than reads; in fact, analysis of the raw data indicates that almost all swap slots are written to only one time and read from multiple times. These results support prefetching pages from Network RAM to flash. The locality of swap access patterns means that prefetching policies could be developed to make good guesses at which pages to prefetch. In addition, because swapped data are likely to be written once but read multiple times, prefetching may be cost effective by prefetching a page just once into flash to obtain multiple subsequent fast reads of the page.

V. NSWAP2L DESIGN AND IMPLEMENTATION

Figure 3 shows Nswap2L's system architecture. It is a multi-layered system which separates the interface, policy, and mechanism components. At the top is the Interface Layer with which the OS and user-level programs interact. Currently, it implements an interface of a single, fast random access block device that can be added as a swap partition on cluster nodes. In the future we plan to extend Nswap2L functionality so that it can be used as backing store for other kernel and user-level services such as temporary file systems, and we plan to add new interfaces, including a programmable API. Currently, the Interface layer contains functions to read and write to our "device". In the future, we plan to add Interface functions to free and allocate blocks and to force persistent storage of some blocks. The Interface layer maintains a mapping of where blocks written to our top-level "device" are stored on the underlying physical devices.

The Policy Layer implements policies for choosing underlying placement of blocks written to our top-level "device", for prefetching blocks from one physical device to another, and for migrating blocks between different low-level devices.

The Mechanism Layer implements functionality to read and write blocks to different low-level physical devices, to move a block stored on one device to another, and to free blocks

[4]Swap access patterns for the IS and HPL benchmarks are very similar to Radix and are not included here due to space restrictions.

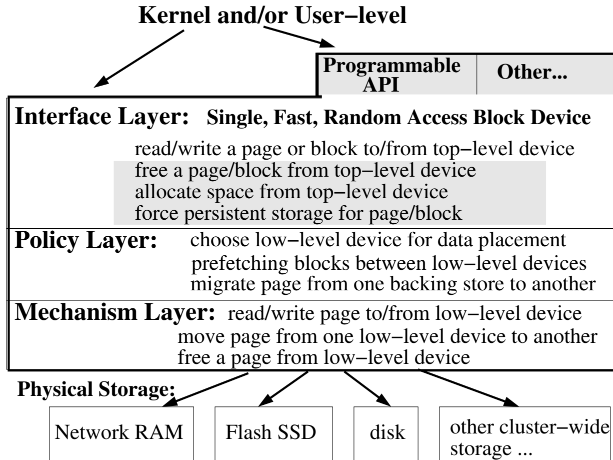

Fig. 3. Nswap2L System Architecture. *It supports a set of interfaces on top of multiple physical storage. Current functionality is shown in black and white. Future functionality is shown in grey boxes.*

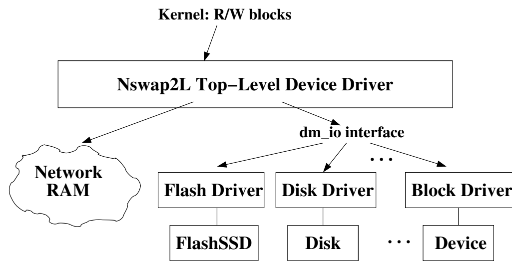

Fig. 4. *Nswap2L Implementation. When used as backing store for swap, the OS sends R/W requests to the top-level Nswap2L driver. The top-level driver directly manages all parts of Nswap's Network RAM backing storage and uses dm_io to pass I/O requests to the other low-level device drivers.*

stored on physical devices. The Policy Layer makes calls to the Mechanism Layer layer to place, prefetch, and migrate blocks between underlying physical devices.

Conceptually, Nswap2L is implemented as two levels of device drivers; the top level is a single pseudo–device driver that sits on top of multiple low–level device drivers, one for each physical storage device. The top level receives I/O requests from the OS's swap system and keeps track of where data are stored on underlying physical devices so that it can send I/O requests to the appropriate low–level physical storage device. The top–level driver implements all the layers of our system architecture. For example, when it receives a write request from the kernel, it runs code in the Policy Layer that chooses a physical device on which to place the blocks, updates mapping information about where the block is placed, and invokes functions in the Mechanism Layer for writing to an underlying physical device.

Our prototype implementation (in Figure 4) closely fits our conceptual two–level driver model. We appear to Linux as a single, fast, block device, and receive read and write requests from the Linux swap system. The top–level driver directly manages the Network RAM storage and passes read and write requests to lower–level device drivers for flash, disk, or other storage devices. We use Red Hat's Device Mapper module (dmio) [25] to pass I/O requests from the top–level driver to the low–level drivers.

The top–level driver uses the shadow slot map to track which underlying low–level device stores each block. When a read request to the top–level "device" is made, Nswap2L looks up the location of the page in its shadow slot map. If the page is stored in Network RAM, it handles all the low–level messaging to request a copy of the page from the Nswap Server storing the page. If, however, the page is stored on a different physical device, Nswap uses the dmio interface to read in the page from the underlying device. On a write request, the top–level driver chooses a low–level device for placement and encodes the device or Nswap Server in the shadow slot map entry for the page.

Nswap2L's current page placement policy always chooses Network RAM first, choosing flash only when no Network RAM space is available. This policy is designed to work well in systems where writes to Network RAM are faster than writes to flash. We anticipate that, in general, data placement policies will be simple and will likely consist of a fixed ranking of underlying devices. However, it is possible that policies that take into account resource usage such as network load may result in better placement decisions. In the future, we plan to investigate dynamic policies that may choose flash as the target even when there is Network RAM space available. For example, if there is a large burst of writes, it may be advantageous to distribute writes over flash and Network RAM to handle more in parallel.

Prefetching is one way to improve performance, taking advantage of the strengths of different underlying media by targeting different devices for reads and writes and distribut–ing storage across equally good devices to improve the I/O bandwidth of our "device". For example, writes to flash are generally slower than reads, primarily due to erasure. Our performance studies from Section IV support initially writing pages to Network RAM and later prefetching some pages from Network RAM to flash so that some subsequent reads can be satisfied by flash. If flash reads are faster than Network RAM reads, then this will improve the performance of reads to our "device". If flash reads remain comparable to Network RAM reads, then prefetching can improve I/O bandwidth.

It is important to develop good prefetching policies that will pick the best pages to move from Network RAM to flash. However, the cost of a bad prefetching choice in our system is much less than that of a bad prefetching choice in paging and file systems that prefetch from backing store into main memory; in our system, a bad prefetching choice will never result in more paging or swapping. Therefore, we may be able to implement more aggressive prefetching algorithms than those used in paging and file systems that prefetch pages from backing store into RAM.

Nswap2L's prefetching policy implementation is divided into sub–policies that answer three questions: (Q1) "When should prefetching occur?"; (Q2) "How many pages should be

prefetched?"; and (Q3) "Which pages should be prefetched?" Functions that implement answers to these three questions can be combined to create different prefetching policies. Our current focus is on prefetching from Network RAM to flash; however, to support prefetching between any two devices, a fourth policy question ("From which device should pages be prefetched?") would be added.

Our implementation uses a prefetching thread that periodically wakes up and runs policy functions associated with the three questions. The implementation is designed so that new policies can be easily added to our system. We also added a `/proc` interface to the prefetching subsystem that allows prefetching to be enabled or disabled, and allows changing the particular sub-policy functions on the fly.

The first question (when should prefetching occur) is determined by both the amount of time the prefetch thread sleeps between checks, and by the particular Q1 policy function. Q1 policies could be based on current swapping activity. For example, it may be advantageous to prefetch only when a node is in a swapping phase because pages have more potential to be prefetched before being swapped in again. On the other hand, prefetching during swapping activity may lead to a slow down of the application's performance as prefetching I/O could interfere with swapping I/O.

Q2 policies (how much should be prefetched) can be based on a fixed percent of total swap space in use, or on a percentage of the number of pages swapped out the last time the prefetch thread woke up. We have both types of policies implemented. The second type requires adding a counter to keep track of the number of swap outs between prefetch thread activations.

Q3 policies (which pages should be prefetched) could be quite simple, such as a round-robin selection of swap slots, or they could be based on swap slot access patterns in an attempt to make better prefetching choices by trying to chose pages to prefetch that are likely to be read soon. Currently, we have four Q3 policies implemented: round-robin of swap slots; randomly selected swap slots; selecting the Least Recently Swapped to (LRS) slots; and selecting the Most Recently Swapped to (MRS) slots. If there is locality in swap slot accesses, then MRS should prefetch pages that are most likely to be swapped in soon. LRS and MRS are implemented using a clock approximation algorithm. We added a reference bit to the shadow slot map that is set when a slot is swapped in or out, and is cleared by the clock hand when MRS or LRS Q3 policy functions run. LRS chooses slots with clear reference bits to prefetch (an approximation of the least recently swapped to), and MRS chooses slots with set reference bits.

VI. Experimental Results

In this section, we present results of several experiments evaluating Nswap2L. Our first experiment compares the runtime of a set of sequential and parallel benchmarks (described in Section IV) for different swap devices: Nswap2L; flash SSD; Nswap Network RAM; and disk. For this experiment, Nswap2L swapped only to Network RAM; prefetching to flash was not enabled. Table III shows the total runtime of the benchmarks for the four different swap devices. The results show that all benchmarks perform best when Nswap2L or Nswap Network RAM are used as the swap device. Disk is much slower, even for WL1 which is the best possible case for disk swapping. The results for flash are comparable to Network RAM. Since the flash execution times include both reads and writes to the flash drive, our proposed two-level system that makes use of both flash and Network RAM has the promise to outperform both flash and Network RAM alone. Even if Network RAM is always faster than flash, our system will allow for data stored in Network RAM to be moved to flash when there is not enough cluster-wide idle RAM available for Network RAM, and will allow for increased parallel reads by distributing them over both flash and Network RAM.

TABLE III

Comparison of different swap devices. *The benchmark total run time (in seconds) when run using Nswap2L, Nswap Network RAM, flash or Disk as the swap partition. Bold entries show the best time. Nswap2L speedups over disk are in parentheses.*

Benchmark	Nswap2L		Nswap	Flash	Disk
WL1	**443.0**	(3.5 speedup)	471.8	574.2	1547.4
WL2	**591.6**	(30.0)	609.7	883.1	17754.8
WL3	**503.3**	(3.4)	526.4	514.1	1701.3
WL4	**578.9**	(30.9)	591.7	978.4	17881.2
Radix	**110.7**	(2.3)	113.7	147.4	255.5
IS	**94.4**	(2.4)	95.1	107.6	224.4
HPL	536.1	(1.5)	**529.7**	598.7	815.3

These results also show no additional overhead of Nswap2L over Nswap Network RAM when Nswap2L swaps to Network RAM only. Given that in our implementation of Nswap2L the top-level driver directly manages Network RAM, we expected that Nswap2L would not add additional overheads to Nswap when all pages are swapped to Network RAM, and these data confirm our expectation.

Our second set of experiments evaluate Nswap2L's prefetching policies. For these experiments, the placement policy always chooses Network RAM for swapped-out page data, the prefetch thread then periodically runs and prefetches some pages from Network RAM to flash, and subsequent swap-in requests from the kernel are satisfied by Network RAM or flash depending on whether the page has been prefetched or not. All prefetching experiments used a Q2 policy (how many pages to prefetch) that tries to prefetch a number of pages up to 10% of the number of swap outs since the last activation of the prefetching thread. We compared runs with no prefetching (Control) to four different Q3 prefetching policies: Round-robin (RR); Random; Least Recently Swapped (LRS); and Most Recently Swapped (MRS).

Assuming that increasing the number of reads from flash is desirable, the most effective prefetching policy is the one that has the most reads per prefetched page—the policy that maximizes the likelihood of a page being swapped in from flash. Table IV shows the ratio of the number of reads from flash to the number of prefetches to flash for the different prefetching policies for each of the benchmark programs. A ratio value

TABLE IV
FLASH READ TO PREFETCHING RATIOS. *The rows are prefetching algorithms, the columns benchmark programs, and the values are the ratio of the number of reads from flash to the number of prefetches to flash.*

	WL1	WL2	IS	Radix	HPL
RR	1.1	3.0	1.7	0.5	0.9
RAND	1.2	3.2	1.4	0.5	0.8
LRS	1.1	2.7	1.9	0.2	0.8
MRS	1.2	3.0	1.6	0.4	0.8

TABLE V
AVERAGE DEGREE OF READ PARALLELISM.

	WL1	WL2	IS	Radix	HPL
No Prefetching	5.5	5.7	5.6	5.4	5.2
Prefetching	3.8	5.3	6.1	13.7	13.1

greater than one is an indication that prefetched pages are, on average, being read multiple times from flash before being swapped out again, and indicates a better prefetching policy.

For WL1 and WL2 the data show that RAND performs best (1.2 and 3.2 ratio values) and LRS performs worst (1.1 and 2.7). For WL2, RAND is likely to work just as well as policies that account for usage due to WL2's random memory access patterns. For WL1, we expected LRS to perform well because of WL1's large sequential access patterns, however, the data show that LRS does not perform any better than the other policies for WL1. This result is due to WL1's pattern of alternating large reads and writes that means that a prefetched page read in from flash has a 50% chance of being modified before being swapped out again. Thus, the best ratio we would expect for WL1 would be about 1.5.

The parallel benchmark results display more variance: LRS performs best on IS (ratio of 1.9), but has the worst performance on Radix (ratio of 0.2); RAND performs best on Radix, but has the worst performance on IS and HPL; and, RR performs best on HPL. It is surprising that MRS does not do better given that the swap trace results from Section IV indicate that recently swapped pages are often accessed again. It does, however, do reasonably well across all of the benchmarks, so it may be a good general policy. These results also may indicate that different policies perform better for different workloads, thus a system that is tunable like ours is likely to be best for handling the variable workloads of general purpose clusters.

To test the hypothesis that prefetching leads to more parallelism, we ran the benchmarks with Nswap2L's profiler thread enabled. The profiler thread attempts to get a picture of the amount of concurrency in the system by recording the number of Nswap Client threads simultaneously handling reads and writes to our device. The profiler thread wakes up twice every second and samples global counters that are incremented by client threads when they are actively handling swap-in or swap-out requests from the kernel. Over time, the profiler thread builds histograms of read and write concurrency. The average degree of read and write parallelism is obtained from these histogram data.

Table V shows the average degree of read parallelism of the sequential and parallel benchmarks. These data show that Nswap2L with prefetching leads to increased read parallelism for the parallel benchmarks (the best case being Radix with 13.7.) For the sequential benchmarks there is no improvement in average read parallelism primarily because there is only one sequential process running on the node for these benchmarks. The results for the parallel benchmarks show that prefetching increases parallelism (for example 5.2 for HPL with no prefetching vs. 13.1 for HPL with prefetching.) Because the parallel benchmarks are typical of the types of cluster workloads that our system is targeting, the parallel benchmark results show that Nswap2L with prefetching has the promise to improve the performance of applications running on general purpose clusters; these data show an increase in read parallelism by distributing reads across Network RAM and flash, so when flash and Network RAM speeds are about the same, the result should be improvement in total runtime.

TABLE VI
COMPARISON OF PREFETCHING POLICIES. *Average runtime is shown in seconds. Workloads in are columns, policies in rows.*

Policy	WL1	WL2	WL3	WL4	IS	Radix	HPL
Control	443.0	591.6	503.6	578.9	113.2	97.2	550.2
RR	905.7	832.3	694.7	835.5	179.5	114.9	619.1
RAND	650.3	819.2	621.9	818.0	146.2	108.2	607.3
LRS	924.1	815.2	678.7	802.0	172.1	105.6	580.5
MRS	884.5	829.9	685.0	815.6	180.8	114.0	622.6

Table VI lists the total execution time of the benchmark programs for the different prefetching polices. The results show that any prefetching to flash hurts performance, the best case for prefetching being HPL-LRS with a slow down of only 1.05 over the Control run (580.5 vs. 550.2 seconds), the worst case being WL1-LRS with a slow down of 2.08 (924.1 vs. 443.0 seconds). Based on our studies in Section IV, our particular flash device has slightly slower read performance than Network RAM, so we anticipated that the runs with prefetching might be slightly slower than the Control runs that use Network RAM alone. We also anticipated that we would see some improvements in run times even though flash is slightly slower than Network RAM due to increased parallelism in simultaneous reads from flash and Network RAM. However, we did not anticipate the larger slow downs.

The reason for the slow down in run times when prefetching is enabled is the high dmio overheads of our current implementation of Nswap2L. To quantify the overhead imposed by dmio, we ran the sequential benchmarks with Nswap2L using only flash as the underlying devices and then using only Network RAM as the underlying devices. We found that dmio adds up to 700% overhead on reads and writes to flash.

Although prefetching does not lead to runtime performance improvements under Nswap2L's current prototype implementation, this result is only an artifact of our current implementation's use of dmio, and is not fundamental to Nswap2L's design. It is therefore worthwhile to consider which prefetching policy would be most effective, given a different implementation of Nswap2L—an implementation that removes the dmio

TABLE VII
PARTS OF BENCHMARKS' RUN TIMES USED FOR CALCULATING IDEAL RUN TIMES. *All values are in seconds unless labeled otherwise. TT is the total execution time, NWRsp is average time to perform a single page read from Nswap2L's Network RAM, FRCtm is the cumulative time of all the reads from flash, NWRCtm is the cumulative time of all the reads from Network RAM, and NFreads is the total number of reads from flash.*

	TT	NWRsp	FRCtm	NWRCtm	NFreads
WL1 Control	455.8	177.5 μs	N/A	840.5	N/A
WL1 RAND	616.8	141.5 μs	349.0	655.3	413,713
HPL Control	628.4	153.1 μs	N/A	96.1	N/A
HPL LRS	708.5	152.7 μs	68.9	93.0	24,186

overhead for accessing the flash device.

We estimate the performance of Nswap2L without the dmio overhead to flash I/O, using our experimental results with dmio and our measured times of the low-level devices to remove dmio overheads. We calculate the ideal execution time (the time of Nswap2L without dmio overhead) as:

(1) $ideal = ((pctNS)*TT)+((pctS)*(TT-FRtm+IFRtm))$

The ($ideal$) runtime is for flash reads with no dmio overhead, $pctNS$ and $pctS$ are the proportion of the total runtime due to non-swapping and swapping, TT is total measured runtime, $FRtm$ is the time for flash reads with dmio, $IFRtm$ is the time for flash reads with no dmio overhead.

To compute the ideal runtime, we ran the benchmarks with timers enabled to extract the portion of the run time due to reading from flash and reading from Network RAM (shown in Table VII.) We also measured the proportion of each application's execution due to swapping by comparing run times using two different amounts of physical RAM, one that results in swapping and one that does not. We found that 99% of WL1's execution time is due to swapping, and 40% of HPL's execution is due to swapping.

The cumulative read times in Table VII do not account for the fact that reads are concurrent, so we estimate the part of the execution due to flash reads ($FRtm$) based on the proportion of the measured flash and network cumulative read times ($FRCtm$ and $NWRCtm$) multiplied by the total execution time (TT):

(2) $FRtm = (TT) * ((FRCtm)/(FRCtm + NWRCtm))$

Next, we calculate the ideal flash read speed with no dmio overhead ($IFsp$) as the ratio of measured direct flash ($DirFRtm$) read time to direct network read time ($DirNWRtm$), from Table I, multiplied by the average network read time for the run ($NWRsp$), from Table VII:

(3) $IFsp = ((DirFRtm)/(DirNWRtm)) * (NWRsp)$

From this we obtain an estimate of the the cumulative ideal flash read time ($IFCtm$) and portion of the execution time due to ideal flash read time ($IFRtm$):

(4) $IFCtm = IFsp * (total_num_flash_reads)$

(5) $IFRtm = TT * (IFCtm/(IFCtm + NWRCtm))$

With values for (2) and (5), we compute the ideal runtime of the prefetching experiments without dmio overhead (1). For example, for WL1-RAND we compute:

(2) $FRtm = (616.8) * ((349)/(349 + 665.3)) = 214.3$

(3) $IFsp = (23.62/20.43) * (141.5\mu s) = 163.6\mu s$

TABLE VIII
BENCHMARK COMPUTED PREFETCHING RUN TIMES. *Control is the measured non-prefetching time. Ideal is the computed prefetching runtime without added dmio overhead using our system's measured flash device speed. Flash 10% (and Flash 20%) are computed prefetching run times for a flash device that is 10% (and 20%) faster than the network.*

	Control	Ideal (no dmio)	Flash 10% $<$ network	Flash 20% $<$ network
WL1 Random	455.8	461.8	**450.1**	**445.3**
HPL LRS	628.4	**600.3**	**597.0**	**595.9**

(4) $IFCtm = (163.6\mu s) * 413713 = 67.7$

(5) $IFRtm = (616.8) * ((67.7)/(67.7 + 655.3)) = 57.7$

(1) $ideal = (.01)(616.8) + (.99)(616.8 - 214.3 + 57.8) = 461.8$

The ideal run time of WL1-RANDOM, 461.8, is much closer to the control WL1 run time of 455.77 seconds and better fits our expectations based on the measured speeds of our flash and network devices. Since there is no increase in parallel reads for the prefetching runs of WL1, the computed ideal run time for the prefetching run is slightly slower than the run time with no prefetching because reads from our flash device are slightly slower than reads from Network RAM.

In Table VIII we show calculated run times of WL1, and HPL benchmarks for our ideal measured flash times (no dmio overhead), and for flash devices that are 10% and 20% faster than the network. We chose WL1 and HPL because they both run long enough to do a fair amount of prefetching, and they illustrate two extremes in read parallelism during the runs with prefetching (WL1 shows no increase in parallelism, and HPL shows a significant increase.)

The run times for faster flash devices were calculated starting with a computed flash speed for function (3) and then applying functions (4) and (5) to calculate the total runtime (1). For example, to estimate a flash read speed that is 10% faster than the network we used:

$(3) IFsp = .9 * NWRsp$

The results in Table VIII show that when flash is 10% or 20% faster than the network, Nswap2L with prefetching outperforms swapping to Network RAM alone. In addition, the HPL-LRS ideal runtime (using our system's measured flash speed that is slightly slower than the network) is faster than using Nswap2L with Network RAM alone (600.3 vs. 628.4 seconds). HPL's faster ideal run time is due to the increase in the average degree of parallel reads when prefetching is enabled (5.2 for Control vs. 13.1 for LRS). This result supports our two-level design even when flash is slightly slower than Network RAM; here prefetching results in an increase in parallel reads that lead to a faster run time.

VII. CONCLUSION AND FUTURE WORK

Nswap2L is our novel two-level device design that provides a high-level interface of a single, fast, random storage device on top of multiple fast random access storage media, namely flash and Network RAM. By moving device-specific knowledge into the top level of our system, OS subsystems and policies do not need to be specialized for the heterogeneous set of backing storage that is emerging in clusters. Our

experimental results support our design, and they show that Nswap2L provides a fast swapping device for clusters. Even in systems where flash is slower than Network RAM, we show that when Nswap2L prefetches pages from Network RAM to flash, there is an increase in parallel reads to our "device" as reads are simultaneously handled by the underlying flash and network devices. Our current prototype implementation prevents us from achieving the performance improvements that we anticipated from prefetching, but we show that with a better implementation, Nswap2L will outperform flash or Network RAM alone. Nswap2L with prefetching performs particularly well for the parallel benchmarks, which match the typical cluster workload better than our sequential benchmarks.

Our future work includes developing and testing a new implementation of Nswap2L that removes the extremely high overhead of the dmio interface between our top-level driver and the flash driver. With a new implementation, we will be able to obtain experimental results that are not hindered by an implementation artifact. We also plan to further investigate prefetching policies, in particular, examining on-line tuning of prefetching polices based on current workload behavior, and comparing dynamic and fixed policies for different workloads.

Finally, we plan to examine other system abstractions that can use Nswap2L as backing store. In particular we will extend Nswap2L so that it can be used as backing store for local temporary file systems, and we will implement a programmable API that can be used to support other abstractions on top of Nswap. Because data persistence is a requirement of general purpose file systems, and because data stored in volatile Network RAM cannot implicitly meet these persistence requirements, we do not foresee adding support for general purpose file systems in the near future. However, by extending Nswap2L so that it can be used to store temporary files, Nswap2L could support a larger class of data intensive cluster applications. In particular, applications that process large amounts of data and store partial results in temporary files, such as database query processing and libraries for accessing large data sets [26], [27], will perform better using an Nswap2L-backed temporary file system.

REFERENCES

[1] 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*, May 1999.

[2] 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.

[3] E. P. Markatos and G. Dramitinos, "Implementation of a Reliable Remote Memory Pager," in *USENIX 1996 Annual Technical Conference*, 1996.

[4] T. Anderson, D. E. Culler, D. A. Patterson, and the NOW Team, "A case for NOW (networks of workstations)," IEEE Micro, Febuary 1999.

[5] L. Iftode, K. Petersen, and K. Li, "Memory Servers for Multicomputers," in *IEEE COMPCON'93 Conference*, Febuary 1993.

[6] G. Bernard and S. Hamma, "Remote Memory Paging in Networks of Workstations," in *SUUG'94 Conference*, April 1994.

[7] Michael J. Feeley and William E. Morgan and Frederic H. Pighinand 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*, Dec 1995.

[8] L. Xiao, X. Zhang, and S. A. Kubricht, "Incorporating Job Migration and Network RAM to Share Cluster Memory Resources," in *Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*, 2000.

[9] 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.

[10] S. Liang, R. Noronha, and D. K. Panda, "Swapping to remote memory over infiniband: an approach using a high performance network block device," in *IEEE Cluster Computing*, 2005.

[11] M. Dahlin, R. Wang, T. E. Anderson, and D. A. Patterson, "Cooperative caching: Using remote client memory to improve file system performance," in *Operating Systems Design and Implementation*, 1994.

[12] D. Roberts, T. Kgil, and T. Mudge, "Integrating nand flash devices onto servers," *Commun. ACM*, vol. 52, pp. 98–103, April 2009.

[13] R. Weiss, "Exadata smart flash cache and the sun oracle database machine," Oracle White Paper, http://www.oracle.com/database/exadata.html, October 2009.

[14] M. Saxena and M. M. Swift, "FlashVM: virtual memory management on flash," in *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, 2010.

[15] S. Park and K. Shen, "A performance evaluation of scientific I/O workloads on flash-based SSDs," in *Worshop on Interfaces and Architectures for Scientific Data Storage*, 2009.

[16] A. M. Caulfield, J. Coburn, T. Mollov, A. De, A. Akel, J. He, A. Jagatheesan, R. K. Gupta, A. Snavely, 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*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010.

[17] 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 Disributed Computing.

[18] T. Newhall, D. Amato, and A. Pshenichkin, "Reliable adaptable network ram," in *Proceedings of IEEE Cluster'08*, 2008.

[19] B. A., L. O., and S. A., "Scalable Cluster Computing with MOSIX for Linux," in *Proceedings of Linux Expo '99*, Raleigh, N.C., May 1999, pp. 95–100.

[20] S. Ko, S. Jun, Y. Ryu, O. Kwon, and K. Koh, "A new linux swap system for flash memory storage devices," in *Proceedings of the 2008 International Conference on Computational Sciences and Its Applications*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 151–156.

[21] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," Proc. the 22nd International Symposium on Computer Architecture, June 1995.

[22] Hongzhang Shan, "MPI port of SPLASH2 benchmarks."

[23] Van der Wijngaart, R. F., "NAS parallel benchmarks version 2.4," NASA Advanced Supercomputing (NAS) Division Technical Report NAS-02-007, October 2000.

[24] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary, "HPL – a portable implementation of the high-performance linpack benchmark for distributed-memory computers," http://www.netlib.org/benchmark/hpl/, January 2004.

[25] R. Hat, "device mapper," http://sources.redhat.com/dm/, 2009.

[26] D. E. Vengroff and J. Scott Vitter, "Supporting i/o-efficient scientific computation in tpie," in *Proceedings of the 7th IEEE Symposium on Parallel and Distribueed Processing*, ser. SPDP '95. IEEE Computer Society, 1995.

[27] R. Dementiev, L. Kettner, and P. Sanders, "Stxxl: standard template library for xxl data sets," *Softw. Pract. Exper.*, vol. 38, pp. 589–637, May 2008.