Determining and Correcting TPIE's Performance Loss Effects on NSWAP

Colin Schimmelfing
`cschimm1@cs.swarthmore.edu`

May 20, 2010

Abstract

Nswap, a Network Swapping System uses the idle memory of local network computers as a faster swap space. We extended this concept to present the network memory as a filesystem to the client, called NswapFS. This is ideal for applications which must repeatedly read or write data which is larger than the memory of the client node. Using this new functionality, we found that even disk-optimized algorithms are up to $\frac{1}{3}$ faster using Nswap as a filesystem - only in unrealistic conditions can disk outperform NswapFS. We also believe that by re-implementing Nswap functionality, we can attain speedups far greater than 1.6.

1 Introduction

One of the greatest disparities in computer systems has been the relative performance of disk and of memory (RAM). Today, memory access is often over one million times faster than disk [2]. Nswap [8] was created to offer swapping to the idle memory of workstations in a computer's LAN, and outperforms disk even on sequential reads and writes, in which the disk has the best performance due to prefetching and the physical nature of how data is laid out on disks. As network speeds have increased to gigabit ethernet (10 GB ethernet is on its way), disk speeds have not increased. Thus the performance gap between Nswap and disk will only increase in the near future, with the caveat that technologies like solid state hard drives may overtake hard drives.

We have created a new application for Nswap, in which a filesystem is overlaid on the network memory for semipermanent storage of data (this system is hereafter called NswapFS). Faster than a filesystem residing on disk, but less persistent (in the case of node failure, much data can be lost), this network filesystem would be ideal for applications which read and write temporary files as part of their computation. Examples of applications include external memory algorithms such as external merge sort, calculations in which data is returned to or modified multiple times such as in GIS applications, and even more mundane applications like web browser caching.

Testing NswapFS, we see that even on sequential reads and writes, which should be optimal for disk compared to NswapFS, our system is about 1/3 faster. This result suggests that with applications in which the disk does not benefit from prefetching and fewer r/w head seeks, NswapFS should outperform disk by a large margin. Similar tests on Nswap as a swap device and have found speedups from 1.9 for sequential I/O patterns up to 32 for more random access [7]. Future work includes determining if NswapFS can provide the same order of speedup, which is likely.

One application which is an ideal candidate for network speedup is TPIE, 'A Transparent Parallel I/O Environment' [2]. TPIE is a library of disk-optimized algorithms for data-intensive projects, and therefore should provide a good basis for how well we can do for real-world examples- TPIE should be able to sort, for example, using a disk as effectively as the disk can be used. Since many applications, especially scientific ones, have been optimized for the peculiarities of disks, it is necessary for us to test with a realistic case such as TPIE.

Another library which implements these data-intensive algorithms is STXXL. This project is an extension of the C++ Standard Template Library 'for XXL datasets', and is optimized for multiple disks. Even using one disk, however, STXXL is faster than TPIE [4]. Given this performance gain, and better documentation for STXXL, it is likely that average users will use STXXL instead of TPIE.

However disk-optimized libraries like STXXL and TPIE are, however, NswapFS should still perform better than disk. We can infer this by the superior performance of NswapFS compared to disk with sequential I/Os, the best I/O formation for disk. Unfortunately, when we tested NswapFS using the TPIE library, we found disk to be about a third faster. This result prompted our investigation to answer this question: "Why does NswapFS perform worse when real data is used?" To answer this question, we investigated a couple of possibilities. One possibility was that an odd implementation detail of TPIE was causing the slowdown. Another hypothesis suggested that the disk device driver was able to merge requests and improve efficiency, while Nswap was not able to do so. A final possibility was that little gain was possible due to most of the data remaining in a file cache.

Essentially all three of the hypotheses were correct. When the memory available to the file cache was artificially shrunk, TPIE's external merge sort started producing better results for NswapFS compared to disk, although still disk is faster. When STXXL's implementation of external merge sort was used, NswapFS outperformed disk as well. While the ability of the disk device driver to merge requests is still providing an advantage to disk, the effects of this inefficiency on NswapFS' part appear with both real data and simulation I/Os.

By implementing functionality in NswapFS for merging requests (which has been implemented for Nswap in the past) we can probably improve speedup beyond the 1.6 which NswapFS currently achieves.

2 Related Work

This work builds on the earlier contributions of Nswap, including [8], [6], and [7]. The idea is actually similar to RAM disks, which have been mostly ignored since the early 1990s.

The work by Newhall et. al [8] introduces Nswap and makes a case for the utility of shared network memory. They stress the benefits of Nswap, including ability to handle a heterogeneous network, and the implementation choice of a loadable kernel module. This is the system that my work is based on, the most important difference being the new, gigabit ethernet speeds which allow Nswap huge gains in performance compared to disk. Work by Newhall and other student researchers [7] investigates adding reliability to Nswap and testing the system on clusters with gigabit ethernet speeds. In fact, this group uses the same cluster to achieve a speedup of 1.9 for sequential page accesses and nearly 33 for random accesses.

The researchers in [9] try to decrease the performance issues disks have by making disk controllers smart enough to understand the meaning of what they are reading. By intelligently managing the cache and file placement, speedup can be achieved. Additionally, secure deletion and some journaling can be implemented, even if the filesystem was not designed to support these features.

The TPIE manual [2] details much (but not all) of the TPIE library. While it contains a large amount of information, often there are chunks of the implementation missing. It does contain a useful tutorial and discussion on some of the underlying structure of the implementation. Through discussions with a former project manager, we have found out that only the UFS interface is used. Overall, this reference can be useful but does not have the information necessary to understand the problems NswapFS is having with TPIE.

This introduction to STXXL [3] explains the justification behind creating the system and introduces key concepts. STXXL was created with a focus on exploiting multiple disks, but outperforms TPIE on single disks as well [4]. This is important since both libraries overlap in major ways. STXXL is meant to be more accessible to programmers, however, by extending the standard template library (STL). It has less of an algorithms background, however, so it does not support advanced structures like K-D-B Trees.

The researchers in [1] try to determine just how much memory is available in a cluster, how much is available in a particular node, and how long this available memory will be idle. They do this by collecting statistics on user activity for a cluster of workstations over a period of about two weeks. They find that, on average, 60-85% of memory is idle, ready for programs like Nswap to take advantage of. This work shows how idle network resources generally are, ready for a system like NswapFS to utilize.

3 Implementation

Nswap as a swap device already existed before our project. There have been several implementations with different features like parity and batching of requests, but all are built on a the same foundation. The basic model consists of both a client piece and a server piece running on all machines in the local area network that allow memory sharing. Nswap is a loadable kernel module, presenting itself as a swap device. Thus, it accepts pages of memory of size 4 KB, and presents itself as a block device to the kernel like swap devices. When a machine has plenty of idle memory, Nswap behaves as a server, accepting pages of memory from other local machines. It stores these in a variable sized memory chunk called the Nswap cache, which Nswap can shrink or grow based on machine memory usage. When a machine starts swapping, Nswap behaves as a client, sending swapped pages onto the network. When pages of memory are paged back in on the client, Nswap requests the needed pages from the servers which it had sent the pages previously. If the local network becomes busy, the Nswap client will accept the pages back and move them to disk on the local machine. Requests to different machines can be handled efficiently, as Nswap is a multi-threaded system.

At least three threads always exist- the listener thread, the memory thread, and the status thread. The status and memory threads handle information about the state of current memory in the local node and the state of all nodes in the system. Each node broadcasts small messages informing other nodes of how many pages it has free, and clients use this information to decide which servers to send their pages to. As the load on a node changes, Nswap will send status updates of how much memory is available, and will adjust its cache accordingly. These small messages organized in a decentralized topology allow Nswap to scale to dozens of nodes easily.

The listener thread makes and accepts connections to other nodes, spawning off new threads for each request. It is helpful to trace the path of a particular page of memory, starting in the memory of the client node. Once the memory is full, the operating system begins swapping and passes the page of memory to Nswap. Nswap decides, based on how much memory is in each node as provided by the status messages, which server to send the page to. In a data structure called the shadow slot map, the logical block which the kernel believes it is writing to is mapped to the server ID Nswap is sending the page to. A ‘putpage’ is then undertaken, sending the page across the network. When the client needs the page, it consults the shadow slot map and requests the page from the correct server with a ‘getpage’. If that memory was never touched again, we would like to free up the space for other data. In Nswap, this is done with a garbage collector which issues ‘invalidate’ messages for pages which have not been touched recently. When a server receives an ‘invalidate’ message, it removes the page from its cache and reports the new space in the next status message. Finally, there is also support for page migration- if one server node becomes busy but there is free space within the network, that server can migrate its pages to other nodes.

Implementing file system functionality with the existing version of Nswap was in fact not very difficult. There were a few changes that needed to be made, however. Nswap, configured as a swap device,

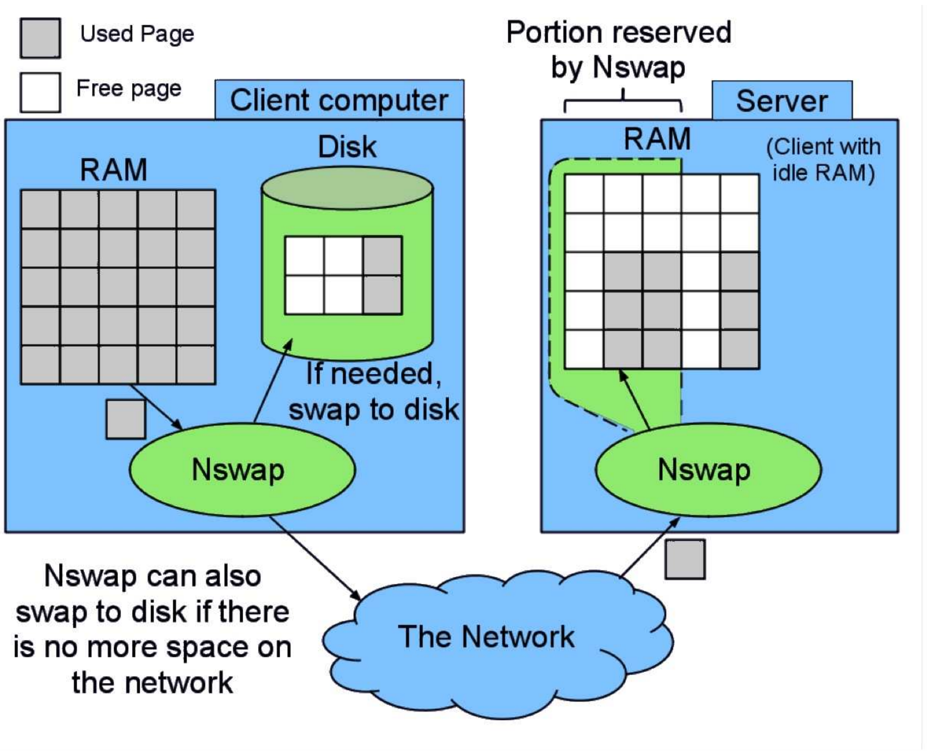

Figure 1: An overview of Nswap as a swap device.

garbage collects pages which were not recently used. For a filesystem, this is obviously unacceptable! Swap devices also treat the first block of the partition in a special way. Requests for this block return data about the device and its contents. For a filesystem, the first block is usually the most important, holding crucial metadata. Since block 0 is so important, it must be treated in a special way whether Nswap is a swap device or a filesystem. The most serious obstacle to NswapFS was the nature of filesystem requests. These can be as small as 512 bytes, while Nswap as a swapping device could only handle requests down to 4 KB.

To solve the first two problems, we merely turned off the functionality in Nswap for garbage collection and the special response for the first data block. This does not disable Nswap from also becoming a swap device, however, because there is a tool called `mkswap` which can take a regular partition and create that special first data block. Garbage collection is also quite easy to turn off and back on, although dynamically switching between the two is not complete just yet. Thus, Nswap can be used as a swap device and be switched over to be used as a filesystem, or vice-versa. Of course all data is lost, so it is not recommended to do this while swapping may be occurring or while data on the filesystem is still needed.

Handling requests of 512 bytes is more difficult. Nswap uses a 'shadow slot map', mapping pages (4 KB) to the remote server which contains them. This is adequate for a swapping device, but is too coarse-grained for filesystem requests which can be 1/8 the size. One solution would be to increase the size of the shadow slot map by a factor of 8, keeping track of where each 512 byte sector is located. However, we determined that this solution would draw too heavily on the memory of the client node.

An alternate solution to handle 512 byte requests without increasing the shadow slot map size is to translate each request into a set of pages and offsets into each page, using simple math . Each 4 KB page is logically partitioned into eight 512 byte 'slices'. The NswapFS architecture takes a request for a number of

512 byte sectors, determines which pages and offsets into the pages the request corresponds to, and requests those pages from remote servers. A diagram (Figure 2) is useful.

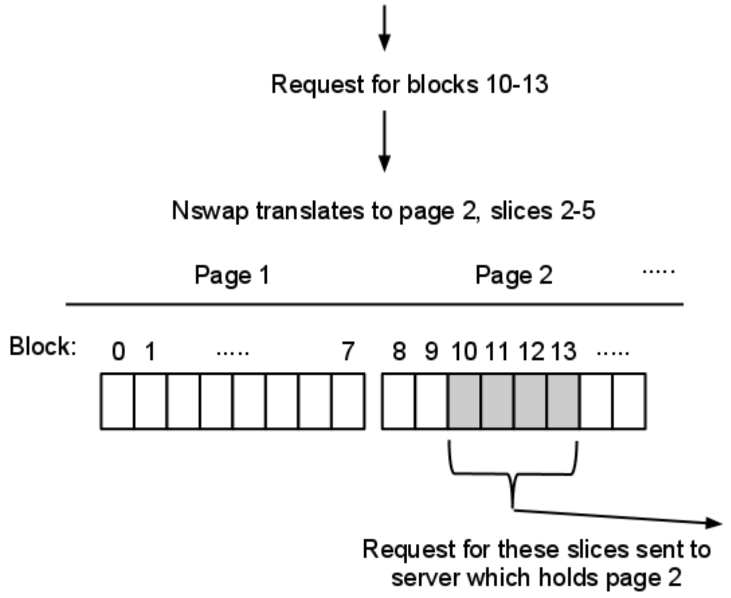
Request for blocks 10-13
Nswap translates to page 2, slices 2-5
Page 1
Page 2
.....
Block:
0 1 ..... 7 8 9 10 11 12 13 .....
Request for these slices sent to server which holds page 2

Figure 2: A diagram of a request using the 'slices' solution. The kernel sends a request which is mapped to a page and to an offset (in number of slices) into the page. The server which holds the page is looked up in the shadow slot map, and a request sent to that server for the data on that page, starting at a designated slice and continuing for a designated length.

Another solution is to inform the operating system that NswapFS has a 'hard sector size' of 4 KB. This means that the operating system behaves as though the smallest physical block on disk is 4 KB. This information is kept track of in the block device global variables structure, shown below:

```
static struct block_device_globals {
        unsigned long size;     // device size
        int blksize;            // block size
        int hardsect;           // sector size
        int rahead;             // read-ahead value
        int max_rahead;         // maximum count to read-ahead
        struct gendisk* gd;   // the gendisk structure describing this device
        spinlock_t req_lock;  // a spinlock for the device
                              // (related to the request queue)
        spinlock_t m_lock;      // a spinlock for our internal request queue
} BDglobs;
```

This structure is shared with the kernel, so if we set 'hardsect' to 4 KB, we can inform the kernel that we cannot handle requests smaller than that size.

While this solution works, it is not ideal: any requests which are smaller than 4 KB waste space. Fortunately, for our purposes this is not too large of a problem- most of the data stored on NswapFS probably will be at least megabytes in size, if not larger. As a semi-persistent filesystem, small files like configuration files are unlikely, and in fact modern operating systems like Mac OSX 10.5 have a smallest file size of 4 KB anyway. This tradeoff between internal fragmentation and a more efficient system has been discussed more generally at length by the creators of GFS [5].

While we implemented the solution 'slicing' the pages into eight parts, all of experiments run in this paper have been run using the 'hard sector size' solution. We found that the actual performance difference between the two implementations was not great, but that the 'slicing' solution still needed work to be as robust to special cases. We do not think that implementing the 'slicing' solution fully will improve performance, due to the large request sizes.

4 Experimental Results

As we concluded our implementation, we ran tests to determine how much of a speedup Nswap provided over disk. As a true test of NswapFS' abilities, we started with large sequential reads and writes, the optimal I/O pattern for disk. The major cause of the 1,000,000x slowdown of disk compared to memory comes from the physical movement of the r/w head. Mechanical movement is extremely expensive, so to counteract this effect the operating system 'prefetches' data blocks which are close by (and cheap time-wise to read) and may be needed later. Thus, disk performs best when reading large amounts of contiguous data, such as in sequential reads. The limitations of disks are more present in writes, as pre-writing is not possible! Instead, the operating system batches together many sequential writes to diminish the number of head seeks. Thus, sequential writes are also ideal for disk, compared to other I/O patterns. In contrast, NswapFS' performance should not change due to I/O pattern, an advantage of our system.

We began by determining a baseline of performance for NswapFS, that is, a minimum speedup we can expect compared to disk. We compared the implementation of NswapFS and disk for sequential I/Os, disk's best case. This experiment, and all subsequent experiments, was run with on `gbcluster`, a cluster of 7 nodes connected with Gigabit ethernet. The client node these experiments are run on has about 900 MB of memory, although the available memory can be tweaked using a loadable kernel module (LKM) called `use_ram`. Nswap uses only about 200MB of memory on each server node, and each node is guaranteed to be idle and to have at least 512MB of memory- thus we do not need to consider memory shortages on the server nodes.

We first ran a simple experiment which wrote a large chunk of data sequentially, and then read it back in sequentially. This experiment used about 390 MB of data, showing a speedup of about 1.6 for NswapFS (Figure 3). This result is not actually as ideal as we would have hoped- earlier versions of Nswap as a swap device were able to achieve a speedup of 1.9 for sequential I/O [7].

Taking advantage of prefetching, libraries like TPIE and STXXL implement algorithms using large chunks of contiguous data, each 'chunk' read in all at once. External merge sort is the quintessential example of this type of algorithm, optimized for data too large to fit into memory and assumed to fit onto disk. Many practical applications use external merge sort, and the simple interface of TPIE to direct which directory temporary files are written to is ideal for experimentation. There are some differences between the two implementations. TPIE writes multiple temporary files, while STXXL seeks within a large file, STXXL can support multiple disks, and the way the libraries deal with 'streams' or 'vectors' of data is different. Differences like these probably account for the STXXL performing about 1/3 faster than TPIE with a single disk [4], even though both sorting implementations performed essentially the same task. Each would write a large amount of random integers to a file, check to see that the data is not sorted, logically separate that file into chunks based on how much memory was available to the program, sort each chunk, combine these chunks in a large output file, and finally check that the data is sorted. Both use twice the 'disk' space as the

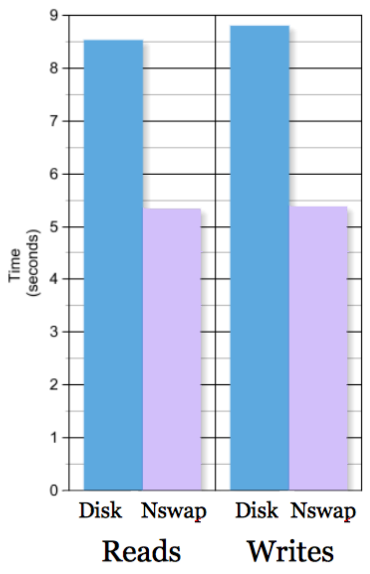

Figure 3: Sequential write and read of about 390 MB of data. NswapFS achieves about a 1.6 speedup.

size of the input data to be sorted.

Initially we ran an experiment using an external merge sort algorithm from the TPIE library, using 300 MB of data broken up into 30 MB chunks. This experiment produced a result (Figure 4) showing that NswapFS was about 1/4 *slower* than disk! This appeared to be impossible, since our best-case for disk still had NswapFS faster than disk. Searching for the cause of this discrepancy, we created a program to simulate the I/O of external merge sort. This test was able to use 1600 MB of data, with chunks of size 100 MB. NswapFS behaved as expected, showing a speedup of about 1.6 (Figure 5). This experiment suggested that either there was some inefficiency in TPIE, or that there was an issue with disk caching or memory usage.

To investigate disk caching and memory usage further, we used the LKM `use_ram` to restrict the available memory. This restricted the amount of disk cache space to virtually nothing- most of the available memory was occupied by TPIE in order to sort the current chunk. We can see the results in figure 6, showing NswapFS approaching the same performance as disk as we restrict the amount of memory available to be used for disk caching, and other memory needs, to almost nothing beyond the bare minimum to perform the calculation without swapping.

While we explored memory usage, we also ran an external merge sort program using the STXXL library, to determine if the behavior was due to erratic behavior on TPIE's part. Running the same data as in figure 4, we found a speedup of 1.3 for NswapFS compared to disk when using STXXL. This result is actually confusing, given the effect of file caching on TPIE's performance. One would expect STXXL to behave in the same way, as both TPIE and STXXL are logically at a level above the file system [3], [2]. We interpret this result to be a deficiency in TPIE's implementation. To see if STXXL exhibits the same performance over different ranges of free memory, we performed experiments similar to the TPIE tests. Shown in Figure 7, we see that STXXL is only slightly affected by the lack of free memory, in the opposite way that TPIE is. Both NswapFS and disk become slower as expected, but NswapFS loses ground to disk. This is actually what we might expect, as Nswap requires some more memory overhead, and just deepens the mystery of why TPIE acts the way it does.

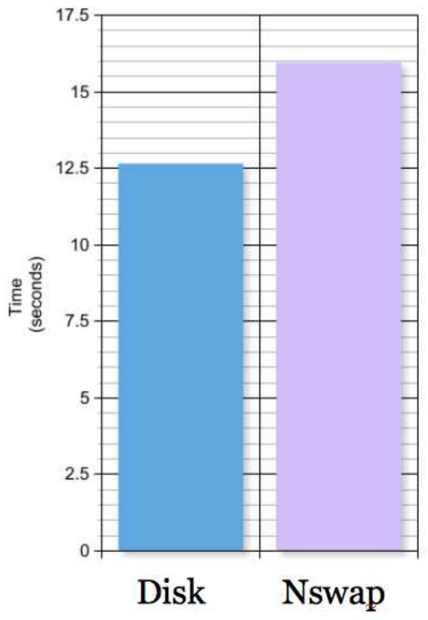

Figure 4: External merge sort using the TPIE library. Data size is 300 MB, in 30 MB chunks, available memory on the client node is about 800 KB. NswapFS is about 1/4 slower than disk.

This performance is encouraging, but previous experiments using Nswap suggest that speedups should be somewhat larger [8]. Sorting real data complicates the matter, but at least our simulation of external merge sort I/O should exhibit larger speedups than about 1.6. A final set of experiments suggests that a previous implementation of Nswap may be useful. This implementation is able to combine multiple requests into a larger request, saving the overhead of a new request for every 4 KB piece of data. This functionality is called PUTPAGES. We can see the efficiency loss by looking at the tool `/proc/diskstats`, which reports I/O statistics for all partitions. We looked in particular at the number of reads, number of reads merged, number of sectors read, as well as at the corresponding fields for writes. Additionally the fields that record overall I/O time for reads, writes, and total time were useful. In tables 1 and 2 are the results for our I/O external merge sort simulation using NswapFS and disk. We can see that NswapFS is much faster at writing, but much slower at reading. We can also see the large disparity in the actual numbers of reads and writes issued. While both NswapFS and disk read and write approximately the same number of sectors, the disk device driver is able to combine requests, increasing efficiency.

	Reads Issued	Reads Merged	Sectors Read	Time Reading (ms)
NswapFS	162405	0	1299240	402120
Disk	10019	2264	1317008	55020

Table 1: Diskstats for our I/O simulation merge sort reads. This experiment used 320MB of data, 'chunked' into 4 pieces.

These general trends of large amounts of writes and reads merged by disk at the expense of NswapFS continue for both TPIE and STXXL. Improving the performance of NswapFS by implementing PUTPAGES will aide these libraries as well.

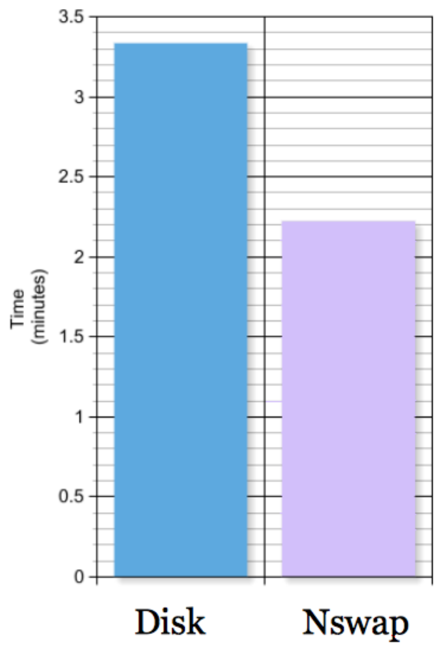

Figure 5: Simulated I/O of external merge sort, using 1600 MB of data and 100 MB chunks. NswapFS achieves about a 1.6 speedup.

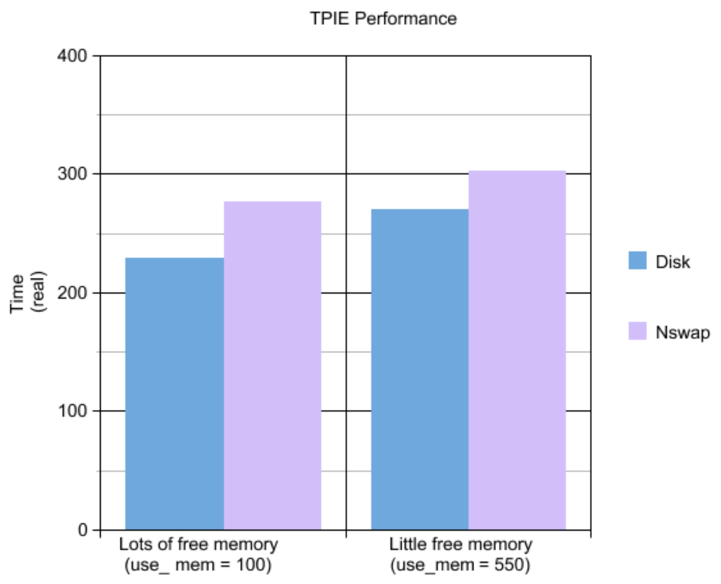

Figure 6: TPIE run with plenty of free memory (use_mem=100) and with very little free memory (use_mem=550). NswapFS' performance improves, but does not outperform disk.

5 Future Work

The first step in the future is to implement PUTPAGES functionality. With the requests combined, we can take advantage of the full power of our system. The overhead required for sending so many more

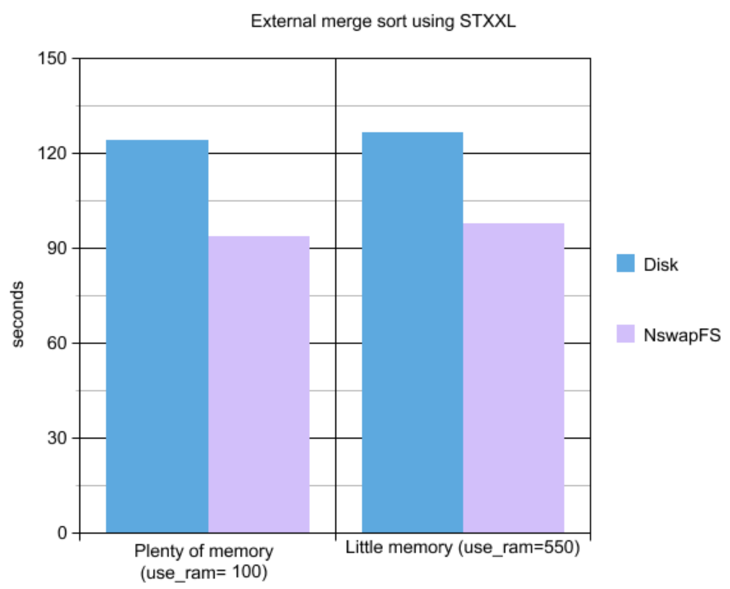

Figure 7: A comparison of STXXL performance for plenty of free memory (use_mem=100) and for very little free memory (use_mem=550). NswapFS performs better than disk, by about 1/4.

	Writes Issued	Writes Merged	Sectors Written	Time Writing (ms)
NswapFS	246349	0	1970792	2313536
Disk	8477	946046	1909272	9064544

Table 2: Diskstats for our I/O simulation merge sort writes. This experiment used 320MB of data, 'chunked' into 4 pieces.

thousands of requests is most likely what is causing NswapFS to perform less than ideally. Additionally, experiments with a larger set of machines would be advantageous. As the datasets NswapFS can aide the most can be tens or hundreds of gigabytes in size, a network of fewer than a dozen machines may not be an accurate measure of the actual power of NswapFS.

A strong criticism of the NswapFS system may be its lack of persistence. Client node failure is acceptably catastrophic, but in the current implementation any server node failure will cause an irreparable loss of data and the calculation must be restarted from the beginning. This issue has already been addressed for Nswap in the past through an addition of parity pages [7]. While there are some complications due to the structure of Nswap, parity has been implemented before and does not significantly affect runtime for Nswap as a swap device. It should be easy to re-implement parity pages for NswapFS, with only minor modifications.

Additionally, a speedup of 1.5 is not as impressive as we would like. While we are limited by the speed of the network, the I/O used by our applications is extremely disk-optimized. These types of I/O patterns are prevalent among the applications we are focused on precisely because they perform so well with hard disks. In the future, advancements such as solid-state technology may disrupt which algorithms are standard. So while some current algorithms using large datasets are less disk-optimized, in the future that trend will only continue. These more randomized I/O patterns are worth looking into, and previous papers have shown [7] that Nswap as a swap device can attain speedups of up to about 30 with random access, while for sequential I/O Nswap attained a speedup of 1.9. We believe that can expect the same improvement with our system.

6 Conclusions

We have created a filesystem overlaid on idle network memory which can outperform disk by a fair amount. This system can be mounted and accessed like any other partition, and is ideal for applications which use datasets too large to fit in memory. Initial tests of real-world applications showed NswapFS underperforming compared to disk, a confusing result given that sequential reads and writes are faster with NswapFS. Adjusting the conditions of the experiments to reflect real-world conditions shows NswapFS outperforming disk by about a third for external merge sort. We see better performance with the STXXL library compared to TPIE, a positive result as STXXL is generally faster than TPIE overall. Even so, NswapFS' performance gains are not as great as we expect. We believe that by consolidating requests as the disk is able to do, NswapFS can significantly improve upon the performance of disk. The superior performance of NswapFS will enable computation of large datasets in far less time, and will use the idle memory of local networks to their full potential.

References

[1] Anurag Acharya and Sanjeev Setia. Availability and utility of idle memory in workstation clusters. In *Proceedings of ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, pages 35–46, 1999.

[2] Lars Arge, Rakesh Barve, David Hutchinson, Darren Erik, Rajiv Wickeremesinghe, Octavian Procopiuc, Octavian Procopiuc, Laura Toma, Laura Toma, Darren Erik Vengroff, and Rajiv Wickeremesinghe. Tpie - user manual and reference, 1999.

[3] R. Dementiev, L. Kettner, and P. Sanders. Stxxl: standard template library for xxl data sets. *Softw. Pract. Exper.*, 38(6):589–637, 2008.

[4] Roman Dementiev and Peter Sanders. Asynchronous parallel disk sorting. In *IN 15TH ACM SYMPOSIUM ON PARALLELISM IN ALGORITHMS AND ARCHITECTURES*, pages 138–148. ACM, 2003.

[5] Sanjay Ghemawat Howard, Howard Gobioff, and Shun tak Leung. The google file system, 2004.

[6] B. Mitchell, J. Rosse, and T. Newhall. Reliability algorithms for network swapping systems with page migration. *Cluster Computing, IEEE International Conference on*, 0:490, 2004.

[7] Tia Newhall, Daniel Amato, and Alexandr Pshenichkin. Reliable adaptable network ram. In *CLUSTER*, pages 2–12. IEEE, 2008.

[8] Tia Newhall, Sean Finney, Kuzman Ganchev, and Michael Spiegel. Nswap: A network swapping module for linux clusters, 2003.

[9] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dussseau, and Remzi H. Arpaci-dusseau. Semantically-smart disk systems. pages 73–88, 2002.