dRamDisk: Efficient RAM Sharing on a Commodity Cluster

Vassil Roussev, Golden G. Richard, III
University of New Orleans
Department of Computer Science
New Orleans, LA 70148
<vassil@roussev.net>,<golden@cs.uno.edu>

Daniel Tingstrom
ATC-NY
33 Thornwood Drive, Suite 500
Ithaca, NY 14850-1250
<dtingstrom@atc-nycorp.com>

Abstract

Recent work on distributed RAM sharing has largely focused on leveraging low-latency networking technologies to optimize remote memory access. In contrast, we revisit the idea of RAM sharing on a commodity cluster with an emphasis on the prevalent Gigabit Ethernet technology. The main point of the paper is to present a practical solution—a distributed RAM disk (dRamDisk) with an adaptive read-ahead scheme—which demonstrates that spare RAM capacity can greatly benefit I/O-constrained applications. Specifically, our experiments show that sequential read/write operations can be sped up approximately 3.5 times relative to a commodity hard drive and that, for more random access patterns, such as the ones experienced on a server, the speedup can be much higher. Our experiments demonstrate that this speedup is approximately 90% of what is practically achievable for the tested system.

1. Introduction

The problem of pooling the RAM resources of a cluster has been approached from a variety angles depending on the researchers' goals and the assumed characteristics of the network. The specific impetus behind our work was born out of frustration with the performance of some commercial digital forensics tools. The main challenge for digital forensics tools is to process targets (i.e. hard drives, RAIDs, etc.) that are growing in size much faster than the ability of a single machine to handle them. As the system runs quickly out of memory, it starts thrashing, leading to multi-day processing. Normally, this would be solved by parallelizing the software as demonstrated in [19], however, this decision is at the discretion of the vendor. At the same time, all digital forensics processing is inherently I/O-bound—all data must be read at least once and, often, many more times to answer interactive queries. Therefore, having more CPU cycles may not yield any notable performance gains, however, having more RAM for caching most likely will.

Thus, we are interested in a practical solution that can be easily be deployed to opportunistically utilize idle RAM resources (we have dozens of computers in instructional labs that are idle most of the time). Generally, the existence of idle RAM is a well-documented fact (e.g. [1]). Therefore, we expected to find a readily-available solution that we could simply deploy for our purposes. After surveying the available research/open-source systems, we could not find one that fit the bill so we set out to build our own.

1.1 Goals

Our main research goal is to develop a RAM-sharing system that can boost the performance of an I/O-bound process by utilizing a commodity gigabit cluster. In particular, we need a solution with the following properties:

- *Better performance*—using remote RAM should yield non-trivial performance gains.
- *Transparency*—RAM sharing should be transparent to the process.
- *Efficiency*—this requirement has two sides: the system should utilize as much as possible of the available network resources; and, a CPU-bound process should see no performance degradation (due to networking overhead).
- *Ease of use*—the systems should be easy to deploy and administer.

1.2 Related Work

Distributed RAM sharing is a well-established idea, and a number of implementation approaches have been developed over the years. Generally, they fall into two broad categories depending on their interaction with the user process. The first approach is to hide the fact that the sharing takes place and by tricking the application into believing that there is more RAM available than there actually is. This is very similar to what the virtual memory system does for a living. The difference is that, instead of coming from the hard drive, the extra memory is

physical RAM on another machine. The second approach is to expose the sharing and give the application some means to control the sharing process.

Before summarizing some of the practical results of previous systems, we should mention that a number of simulation studies have been performed to explore the viability of different ways of distributed RAM sharing. For example, Dahlin et al. [3] used a trace-driven simulation to study the performance benefits of cooperative file caching using several cooperative caching algorithms. In [20], and later [14], Oleszkiewicz and Xiao studied the impact of combining network memory and job migration for system scalability and throughput improvement. A *Parallel Network RAM* solution, based on global management was proposed for scientific applications.

In terms of practical solutions, a major line of research has been clustered around the idea of shared virtual memory, first proposed almost 20 years ago [9]. The essential view in this approach, surveyed in [5], is that strong consistency models are the root cause for underperformance. Thus, the focus has been on improving protocol implementation of ever more relaxed consistency models. This implies that, in the general case, the application needs to get involved in the sharing process to achieve better performance so the transparency of the memory sharing is compromised.

Recently, a number of systems have been designed to take advantage of low-latency technologies, such as RDMA over *InfiniBand*, to present the application with extra remote physical memory at a cost similar to the local memory. In [11], for example, an RDMA-optimized implementation of the MPI library is used to provide the transparent use of remote memory. In [5], a high-performance block-level device is used to provide remote swap memory for the paging system. Due the low latency of the underlying technology, the authors were able to achieve, for memory-intensive applications, performance comparable to that of local memory and over 20 times faster than traditional disk paging. Such solutions are certainly viable on a high-performance cluster, however, they rely on hardware that is hardly a commodity.

A completely different approach that tries to get around the RAM shortage is process migration. MOSIX [2], for example, uses a *memory ushering* algorithm to move the computation to a node with available memory rather than find extra memory. This, indeed, would solve the problem if there is a suitable node with enough RAM, however, this may well not be the case.

Another approach, most closely related to our own is the use of RAM block-level device. Typically, this is used as a means to present to the file system a remote hard disk as local one. This realized by communicating block-level operations over the network, as it is done in *NBD* 0[12] and its derivative versions. In [8], Kim et al. present an optimized version, which achieves remote throughput that matches the throughput of the physical drive. The *Network RamDisk* (NRD) presented in [7] is an extension of the network block device idea. NRD allows the RAM resources of a cluster to be presented as a single block-level device and used in lieu of a hard drive.

2. Design rationale

Consistent with our goals for a simple system that can easily be deployed, we have opted for a design that is minimally invasive to the system software and is fairly portable. The basic architecture consists of a set of user-level RAM server processes that provide local RAM access to a central RAM client (Figure 1).

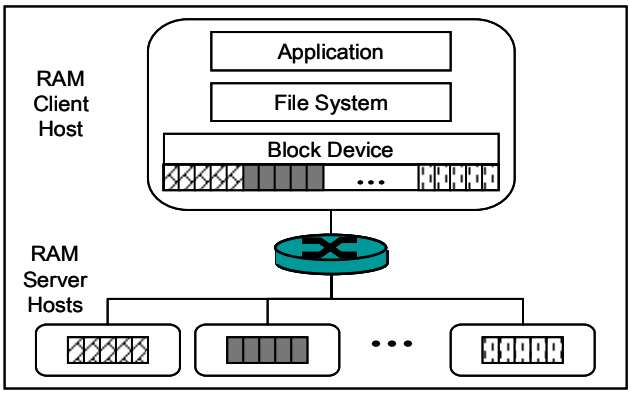

Figure 1 *dRamDisk* Architecture

The client is implemented as a kernel block-level device driver. We use an application-level protocol to perform service discovery and to exchange block read/write operations over TCP socket connections. We recognize that all of these design choices have certain drawbacks. Nonetheless, for our purposes, we believe that the advantages outweigh the shortcomings:

- *User-level RAM server*. One obvious drawback is that allocated memory cannot be locked into RAM so there is the potential for it to be swapped out by the paging system. Our rationale (supported by our experience) is that, for an idle system, the issue is moot—the user process will be able to use up all available memory simply because there is no competition. On the other hand, if the system has some other non-trivial job to do (albeit not memory-hungry), it should not be volunteering RAM in the first place since the RAM server will not be responsive enough. In other words, we expect the RAM server host to be otherwise idle and that there is no real resource contention from other processes.
- *Block*-level device driver. There are two obvious choices here. The first one is to implement file sys-

tem-level caching, which would allow optimizations based on the logical structure of the file system (FS) but would also make it FS-dependent. The alternative is to implement a (block-level) device that does not have the benefit of filesystem knowledge (and likely access patterns) but would work with any filesystem.

Our decision was based on two factors. The first is that for our specific application domain (digital forensics), unlike in most other domains, the applications do care about unallocated space and preserving the original block-level layout of the FS is necessary.

The other half of our reasoning is that modern operating systems already do a very good job of laying out files sequentially so simple read-ahead optimizations may well be enough to achieve good performance. Patterson [14] has empirically documented the secular trend in hardware that latency improvements consistently lag bandwidth and capacity.

This, of course, has not gone unnoticed by FS designers and they have gone to great lengths to ensure sequential file layout. We could not find an empirical study documenting the actual success of these efforts. However, from our unique (digital) forensics perspective we have seen enough evidence of that. Specifically, one of the basic tasks in forensics is data recovery of corrupted file systems. A standard approach is to use a *file carving* tool, such as `scalpel` [17], to extract sequentially laid out files based on header and footer signatures of different file formats. While this is arguably an inexact process (many file formats do not have well-defined headers/footers) our success has been very high.

- *TCP*-based communication. TCP/IP processing overhead is a well-known source of inefficiency, especially for high-speed communication networks. For a commodity solution seeking the lowest common denominator, the only other realistic options are UDP or Ethernet frames. Evidently these would need separate reliable transmission mechanisms, very similar to the one already provided by TCP. In initial testing, we did not find any appreciable difference between a TCP version and a UDP one (without a reliability mechanism). For our first version, presented here, we decided to go with the baseline TCP solution and revisit the issue, if the performance is unsatisfactory.

  Another argument supporting TCP/IP is that IT users have been very reluctant to adopt more efficient (but less widely accepted as standards) solutions designed to take advantage of more efficient communication technologies (e.g., *InfiniBand*). As a result, many vendors are providing TCP/IP emulation that enables users to take advantage of (most of) the optimization without parting with the "good old" TCP/IP sockets. Specialized Ethernet "accelerators" are emerging with TCP/IP implementations on a chip. Even SMP machines (e.g. from IBM) come with TCP/IP emulation so that the same code could be run on a cluster and on an SMP machine with shared memory. In other words, we have good reason to believe that TCP/IP is not going away anytime soon even for high-performance computing and that, in many cases, TCP-based solution would be able to directly benefit from hardware improvements.

3. Implementation issues

As Section 4.1 below shows, in terms of raw throughput, the network in our setup enjoys a 4:1 advantage at the outset. Therefore, we expected that a first-cut, un-optimized implementation that would satisfy request on a block-by-block basis would show *some* non-trivial improvement. Thus, 1KB block transfers that comfortably fit into a single Ethernet frame would seem logical. However, the real numbers showed a less than 5% improvement. Apparently, the network overhead paid for a small request/response data transfer is not worth it.

Subsequent experiments confirmed that, for large files, simply pushing the transfer (read-ahead) unit to over 100KB yielded substantial improvements. For small files that is clearly too expensive. Therefore, we implemented an adaptive read-ahead scheme to accommodate the conflicting requirements of small and large files:

- The initial transfer unit is set to the minimum (4KB).
- If a successive block request is adjacent to the previous block transferred, the size of the transfer unit is doubled subject to a maximum parameter (128KB).
- Otherwise, the transfer unit is set to the minimum.

The above scheme is quite similar in spirit to the TCP slow-start algorithm—every adjacent block request is treated as a "success" leading to the doubling of the transfer window, while every non-adjacent one is treated as a failure (akin to a packet loss) and the window is shrunk. The minimum of 4KB was picked because it is often used by operating systems as a minimum allocation unit. The maximum was picked after testing identified it as the point of diminishing returns—further increases yielded only marginal improvements in performance.

The presented adaptive scheme is somewhat similar to the one used by the *Linux* kernel in version 2.6 (which has its own issues [14]). There are at least two notable differences: a) at the block-device level we simply do not know about files so file-based optimization is not possible; b) our read-ahead is more aggressive and works along the file system read-ahead.

4. Evaluation

In this section, we present our experimental results and discuss their significance.

4.1 Experimental setup

For our tests, we used a mix of benchmark results and completion times from applications we use frequently. All experiments were performed on the following hardware:

- 5 x Dell Pentium 4 @ 3GHz/2GB RAM
- 1 x Gigabit 8-port Linksys Workgroup Switch.

All machines were running Linux 2.4 kernel. One host was dedicated to running the application/benchmark, while the other four were providing the distributed block device (i.e., all I/O requests/results cross the network).

The HDD used for comparison was a randomly-picked 60 GB Hitachi IDE drive from our lab and was directly attached to the host executing the applications. For testing, we used the complete content of two randomly chosen hard disks from our general purpose lab, 4GB and 6GB, respectively. For the network experiments, the test images were preloaded onto the distributed RAM drive. Before the running the tests, we benchmarked both the HDD and the network as follows:

- **Network:** End-to-end sustained bulk IP network transfer observed by processes: 100MB/s. (This was higher than our expectations so we performed the same experiment with two other switches—bigger and much more expensive—with similar results.)
- **HDD:** Sustained file system level bulk transfer (mass sequential copy): 24MB/s.

These baseline results show that the commodity network has the clear potential to beat the commodity HDD for bulk transfers, which is the strong suit of the hard drive. For random access, we would expect the performance gap to widen considerably.

4.2 Test results

Throughout our design and implementation process, we have targeted the development of a practical solution that can benefit users. Therefore, a principal question for our testing methodology was the selection of test cases that best represent typical access patterns.

After considering our goals, we concluded that the main measure of success is the ability to speedup sequential access patterns. The rationale here is twofold: a) it is the best side of hard drive performance—randomized patterns clearly kill HDD performance and play to our strengths; b) today, non-sequential access patterns are not the norm, but the aberration. For example, *Google* FS [5] does not even attempt to optimize for non-sequential access. Applications that do need to access large amounts of data with potentially randomized patterns explicitly manage their I/O requests to improve performance (DBMS are an obvious example).

A common exception from the above cases are file servers: due to concurrent independent requests the block requests could become really scattered. This, however, should naturally favor solution over a mechanical drive.

Benchmark results

For our benchmark testing, we used the *IOzone* file system benchmark tool (`http://www.iozone.org`) and ran identical tests for both the local and the network drives. Below, we summarize the numeric results (Table 1) and provide a brief description of the different *IOzone* performance measurements.

Table 1 IOzone benchmark results

Test	Performance (KB/s)		Relative Speedup
	HDD	RAM disk	
Write	24,026	92,309	284%
Rewrite	25,788	92,628	259%
Read	26,568	88,768	234%
Reread	26,487	88,357	234%
Random Read	396	9,065	2189%
Random Write	495	10,501	2021%
Backwards Read	5,071	18,216	259%
Strided Read	5,243	7,834	49%

- **Write**: Sequential writing to a new file.
- **Re-write**: Sequential writing to an existing file.
- **Read**: Sequential reading of an existing file.
- **Re-read**: Sequential reading of a file that has already been read.
- **Random Read**: Reading from random locations within a file.
- **Random Write**: Writing to random locations within a file.
- **Backwards Read**: Sequential backwards reading of a file.
- **Strided Read**: Reading a file with a strided access, e.g., a 4KB read followed by 200KB sequential seek, another 4KB read, and so on.
- **Record Re-write**: Repeated writing and re-writing of a particular spot in a file.

Several points are worth noting from the above table:

- For sequential read/write operations, the distributed RAM disk was able to achieve 88-92% of the sustainable IP bulk transfer rate over our network. Clearly, the write performance is not a function of any optimizations on our part but is an artifact of the ability of TCP to sustain the measured rate. In the

other hand, the read performance demonstrates that our aggressive adaptive scheme is able to feed enough data to keep TCP busy at close to peak rate.

- For sequential read/write operations, the distributed RAM disk performed about 3.5 times faster than the IDE drive. Recall that, from the initial benchmarking, the raw network transfer rate (our practical limit) was 4 times the HDD one.
- For randomized access, the ram disk showed an average improvement of 22 times over the mechanical drive. At the same time, the observed transfer rate (~10MB/s) was 10% of the maximum, whereas for the hard drive that number is well under 2%.
- The 100% improvement of backwards read over random read for the ram disk is entirely due to the read-ahead policy of the kernel—we did not tweak our read-ahead algorithm to handle this the way we handle forward read for the sake of the test. We find the result interesting as it gives an idea of the relative effects of file system read-ahead and block device read-ahead policies.
- One relatively minor discrepancy are the stride read results for the ram drive—they are somewhat lower than the random access results, which we would expect that to be the absolute floor of performance. Since the block device does not do anything differently, our best guess is that the file system issues read-ahead requests that are eventually not used and not counted by the benchmark application.

Application results

The pitfalls of using benchmark results to predict application performance are well known. Therefore, we present some test result for known I/O-intensive applications. We used three different command-line tools:

- `tar`—the standard Unix archiving utility;
- `md5sum`—a tool for computing MD5 file hashes;
- `scalpel`—an optimized tool for carving files out of a disk image [17]. It performs two block-level passes over the target (the file system is assumed to be corrupted). During the first one it uses known header/footer signatures to identify the file locations. During the second pass it performs the actual carving by reading the identified block ranges and writing them as separate files.

The tests were performed on both 4GB and 6GB NTFS targets described earlier. For `tar` and `scalpel`, both the read and writing were performed in the RAM device. Thus, the `tar` test was not performed on the 6GB target as our setup provided only an 8GB device, while `tar` needed 12 GB to complete the task.

Table 2 Application test results: 4GB target

Application	Completion time (s)		Relative Speedup
	HDD	RAM disk	
`tar`	337	92	266%
`md5sum`	155	46	237%
`scalpel`	846	331	156%

Table 3 Application test results: 6GB target

Application	Completion time (s)		Relative Speedup
	HDD	RAM disk	
`md5sum`	272	73	273%
`scalpel`	1,541	861	79%

The `tar`/`md5sum` results correlate very well with the benchmark results with speedups falling well within the 234-284% range observed earlier.

The `scalpel` results for the two targets are not directly comparable—the execution time depends on the number and size of the identified files. The reason is that the performance is dominated by the number and size of files carved out during the second pass.

4.3 Discussion

To place our results in the context of previous work we compare them with respect to NRD [7], which has the closest goals and performance metrics to ours. A direct head-to-head comparison is not possible due to the varying technologies used, so our main basis for comparison is *efficiency*. One way to measure efficiency it to compare how well does each of the two implementations realize the available network bandwidth.

For sequential read/write operations, *dRamDisk* utilizes 71 and 74%, respectively, of the theoretical maximum of 1GB/s (Table 1). On the other hand, the respective numbers for NRD we derive to be 22 and 25%, for the 10 Mb/s Ethernet quoted. We deduce the sequential NRD read throughput from the performance for `find` presented in Table 2—28 MB read is completed in 104 seconds (~276KB/s). The sequential write performance comes from Figure 11, which shows a 30 MB sequential *IOzone* write to take about 100 seconds (~307 KB/s).

Clearly, other factors play into these end-to-end performance measurements—quality of hardware, NIC drivers, TCP/IP stack, etc. However, they cannot account for the *threefold* improvement in efficiency. Further proof can be found in the fact that NRD achieved only 25% improvement in sequential read performance relative to the hard drive. If we extrapolate for a network that is 4 times faster than the HDD (our setup) the speedup would not exceed 100%, whereas ours stands at 234%.

5. Conclusions and Future Work

In this paper, we presented a practical solution for sharing of RAM resources on a commodity gigabit cluster. The solution is based on a distributed block-level device called *dRamDisk*. Unlike previous work, our solution is targeted at improving sequential read/write operations, which are the dominant disk access pattern. Our experiments show that sequential read/write operations can be sped up approximately 3.5 times relative to a commodity IDE hard drive. Furthermore, this speedup is approximately 90% of what is practically achievable for the tested system. To achieve this performance, we employ an adaptive read-ahead scheme which exponentially expands the read-ahead window during sequential reads.

Relative to previous work, our system is approximately 3 times more efficient in its ability to use available network bandwidth and is able to utilize 71-74% of the theoretical LAN capacity.

For random access patterns, the measured speedup is over 20 times. Thus, for mixed loads, such as the ones experienced on a server, the speedup can significantly exceed the baseline 3.5 factor.

This is still early work and there are a number of features we plan to implement and test experiments we would like to perform: a) an optimized multi-threaded read-ahead implementation to improve performance by reducing the penalty for unsuccessful speculation; b) file-aware block allocation, i.e., use file system information to both ensure that all data blocks belonging to a file are stored on the same node, and to improve read-ahead success rate; c) multi-client access; d) perform large-scale experiments on our 64-node cluster; e) study the effects of "overbooking" the RAM server, i.e., relying on the VM to promise more RAM than physically available.

6. References

[1] A. Acharya and S. Setia. "Availability and utility of idle memory in workstation clusters". In *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 1999.

[2] A. Barak and A. Braverman. "Memory ushering in a scalable computing cluster". In *Proceedings of IEEE Third International Conference on Algorithms and Architecture for Parallel Processing*, 1997.

[3] M. Dahlin et al. "Cooperative caching: Using remote client memory to improve file system performance". In *Proceeding of the First Symposium on Operating Systems Design and Implementation*, 1994.

[4] C. Dubnicki et al. "Software Support for Virtual Memory Mapped Communication". In Proceedings of the 10th International Parallel Processing Symposium, 1996.

[5] S. Ghemawat, H. Gobioff, and S. Leung. "The Google File System", In *Proceedings of 19th ACM Symposium on Operating Systems Principles*, 2003

[6] L. Iftode and J. Singh. "Shared Virtual Memory: Progress and Challenges", In *Proceedings of the IEEE*, Vol 87(3), 1999.

[7] M. Flouris and E. Markatos. "The Network RamDisk: Using remote memory on heterogeneous NOWs". *Journal of Cluster Computing*, 2(4):281–293, 1999.

[8] Kangho Kim, Jin-Soo Kim, and Sung-In Jung. "GNBD/VIA: A Network Block Device over Virtual Interface Architecture on Linux". *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, 2002.

[9] K. Li. "Shared Virtual Memory on Loosely-coupled Multiprocessors". *PhD thesis*, Yale University, 1986. Tech Report YALEU-RR-492.

[10] S. Liang, R. Noronha and D. K. Panda. "Swapping to remote memory over InfiniBand: An Approach using a High Performance Network Block Device", *Proceedings of the IEEE Cluster Computing*, 2005.

[11] J. Liu, J. Wu, and D. K. Panda. "High Performance RDMA-Based MPI Implementation over InfiniBand". *International Journal of Parallel Programming*, 32(3), 2004.

[12] P. Machek. Network Block Device (TCP version). `http://nbd.sourceforge.net/`.

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

[14] J. Oleszkiewicz, L. Xiao, and Y. Liu. "Parallel Network RAM: Effectively Utilizing Global Cluster Memory RAM: Effectively Utilizing Global Cluster Memory", In *Proceedings of the 33rd International Conference on Parallel Processing*, 2004.

[15] R. Pai, B.Pulavarty, and M.Cao. "Linux 2.6 performance improvement through readahead optimization", In *Proceedings of Proceedings of the Linux Symposium*, 2004.

[16] D. Patterson, "Latency Lags Bandwidth", *Communications of the ACM*, 47(10), 2004.

[17] Philipp Reisner. "DRBD—Distributed Replicated Block Device". *9th International Linux System Technology Conference*, 2002.

[18] G. Richard and V. Roussev. "Scalpel: A Frugal, High Performance File Carver", In *Proceedings of the Fifth Digital Forensics Research Workshop*, (DFRWS) 2005.

[19] V. Roussev and G. Richard. "Breaking the Performance Wall: The Case for Distributed Digital Forensics". In *Proceedings of the Fourth DFRWS*, 2004.

[20] L. Xiao, X. Zhang, and S. A. Kubricht. "Incorporating Job Migration and Network RAM to Share Cluster Memory Resources". In *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing*, 2000.