Performance Analysis of a User-level Memory Server

Scott Pakin #1, Greg Johnson #2

#Performance and Architecture Lab (PAL)
Los Alamos National Laboratory
Los Alamos, New Mexico, USA

1pakin@lanl.gov
2gjohnson@lanl.gov

***Abstract*—Large-scale parallel applications often produce immense quantities of data that need to be analyzed. To avoid performing repeated, costly disk accesses, analysis of large data sets generally requires a commensurately large amount of memory. While some data-analysis tools can easily be parallelized to distribute memory across a cluster, other tools are either difficult to parallelize or, in the case of simple data-analysis scripts with short lifespans, not worth the effort to parallelize. In this work, we present and analyze the performance of JumboMem, a simple, entirely user-level parallel program that enables unmodified sequential applications to access all of the memory in a cluster. Although there are many implementations of memory servers, all require either administrative privileges or program modifications. More importantly, no existing memory server has been evaluated on modern workstation clusters with high-speed networks, many nodes, and significant quantities of memory. This paper represents the first study of memory-server performance at supercomputing scales.**

I. Introduction

Many high-performance parallel applications have an insatiable need for memory and CPU cycles and can readily exploit the largest workstation clusters. Such applications commonly produce massive amounts of output that need to be analyzed. When no appropriate data-analysis software is available, users turn to scripting languages to rapidly develop customized data-analysis tools. These scripts—which may be as simple as a few commands piped together on the command line—need enough memory to process an application's output. While the total amount of memory in large-scale systems has increased exponentially over time, the total number of CPUs has increased proportionally. In other words, the memory available to each CPU—or even each node, as the number of CPUs per node has not been growing exponentially—has changed comparatively little over time (Figure 1).

When more memory is needed for a data-analysis script to run (without paging to disk), a user has three basic alternatives:

1) Buy more memory, which can be expensive and, for very large amounts of memory, may not be possible to add to an existing workstation.
2) Parallelize the script and run it on a cluster to divide the memory requirements among a large number of nodes. The script needs to be important enough to justify the time needed to rewrite and debug the code. Throwaway scripts, an increasingly common case, are seldom worth the effort—not to mention that there may be little opportunity for parallelism in many types of data analysis.
3) Use a *memory server* [1] (a.k.a. Network RAM [2]), in which an application pages out memory not to disk but over a high-speed network to idle RAM located on other nodes in a cluster. Memory servers generally require administrative privileges (to load and unload kernel modules), which are generally not available to users on production clusters.

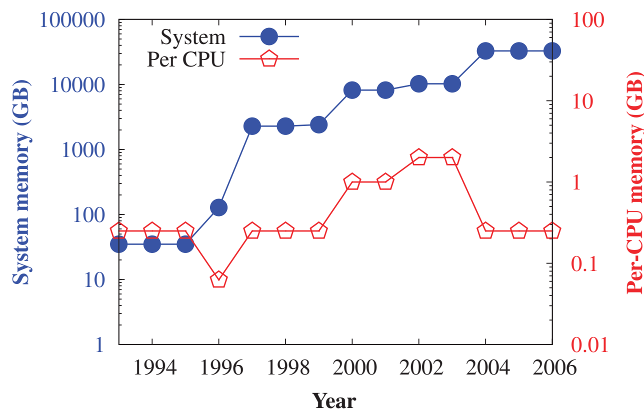

Fig. 1. Memory in the world's fastest supercomputer (cf. http://www.top500.org/)

In this paper we focus on the memory-server alternative. Our assumptions are as follows:

- Users need to analyze large volumes of data.
- Scripting languages are convenient for many forms of data analysis.
- One-shot programs need immense amounts of memory but are not worth parallelizing just so they fit in the available per-node RAM.
- Access to large, high-performance clusters is commonplace.
- Administrative access to high-performance clusters is not.

Based on those assumptions we propose employing a user-

level memory server to exploit the convenience of scripting languages for rapid analysis-tool development while still being able to access memory located throughout a cluster. Although user-level memory servers have previously been discussed in the literature [2], [3], no memory server implementation—user-level or not—has ever been evaluated at scale. In this paper we describe JumboMem, the first user-level memory server that enables unmodified applications to page to remote memory, and then evaluate JumboMem's performance on a cluster containing an order of magnitude more nodes and two orders of magnitude more memory than any previously implemented memory server. Our goal is to determine if memory servers in general and JumboMem in particular perform well when applications manipulate hundreds of gigabytes of memory spread over hundreds of cluster nodes.

The rest of this paper is organized as follows. Section II presents some background information about memory servers and some of the challenges of running a memory server on a large-scale cluster. Because there have been many prior memory-server projects, Section III contrasts our work to that of other projects in the literature. JumboMem's implementation is presented in Section IV. The core of this paper is Section V in which we analyze JumboMem's performance using a variety of microbenchmarks through complete applications. Section VI describes some opportunities for future work that leverage JumboMem as a research vehicle. Finally, we draw some conclusions from our work and present these in Section VII.

II. Background

In a memory-server environment, cluster nodes are designated as *masters*, which run applications, and *slaves*, which serve memory over the network to the master. (In some implementations a node can be both a master and a slave.) Essentially, a master treats the slaves as a large paging device.

Running a memory server on a large number of nodes and with large amounts of per-node memory is a qualitatively different problem from running a memory server with more modest node counts and memory capacities. We now examine three challenges that appear only at large scale: data-structure size, page-map capacity, and communication-subsystem resource utilization.

First, a large-scale memory server cannot practically utilize any data structure whose size is proportional to the total number of pages as this number can easily grow exceedingly large. For example, our cluster has a total of 1 TB of RAM, which corresponds to 268,435,456 four-kilobyte pages. Assuming a memory server needs to store four bytes of information per page, this represents a quarter of the master's physical memory and leaves only three quarters available to applications. The expected consequence is increased paging activity and therefore decreased performance. If we were to expand our cluster from 256 nodes to 1024 nodes (4 TB of memory) there would be no memory available on the master for applications to use.

Because user-level memory servers such as JumboMem explicitly map pages into (and unmap pages from) a process's address space a second issue concerning memory servers is the maximum number of per-process page mappings that the operating system supports. In Linux 2.6, the default maximum is 65,536 (although this can be increased by a user with administrative privileges). Although Linux automatically coalesces adjacent mappings, 65,536 mappings corresponds to a worst-case total (i.e., with alternating mapped and unmapped regions) of 512 MB of address space—a small number relative to the 4 GB of memory per node and 1 TB of total memory in our cluster but a large number relative to the node- and system-memory sizes that were commonplace just a few years ago.

A final issue regarding memory servers at scale is the memory required for the communication subsystem. Messaging layers that target high-speed networks such as InfiniBand [4] generally need to reserve more memory (e.g., for pinned, per-connection communication buffers) to achieve better performance. While the previously mentioned issues can be addressed by using large logical pages, doing so implies greater memory demands from the communication subsystem. For example, 255 connections with only a single outstanding page-sized message (4 KB) allocated per connection corresponds to over 1.5% of the total node memory.

III. Related Work

Paging to remote memory instead of local disk is not a new idea. Almost 25 years ago Garcia-Molina et al. proposed a "massive memory machine" in which a collection of nodes interconnected with a low-latency broadcast network collaborate to provide a large global address space to sequential applications [5]. The first actual implementation of the remote-memory concept is arguably the Apollo DOMAIN system, for which Leach et al. quantified the costs of page faults satisfied by remote memory in a paper dated 1983 [6].

What makes our work unique is primarily the scale at which we evalute it. Table I lists, in chronological order of publication date, a wealth of previously implemented memory servers. (Simulated memory servers are excluded from the list.) While we present measurements taken on hundreds of nodes on a cluster containing a terabyte of RAM, no prior work of which we are aware has been run on more than tens of nodes or tens of gigabytes of memory.

A second unique feature of our work is that our software, JumboMem, represents the first implementation of a memory server that works without application modifications yet does not need administrative access to install or run. Any user with access to a large cluster can run applications using JumboMem.

Dodo [3] and Network RAM [2] are among the few published examples of a user-level memory server. Dodo provides a library that exports a file-like interface (open, read, write, close, etc.) to applications, which must explicitly manage remote memory. Network RAM provides functions for initializing memory servers and for allocating remote memory

TABLE I
COMPARISON TO PREVIOUS WORK IN TERMS OF TOTAL NODES AND TOTAL MEMORY

Project	Nodes	Memory (MB)
Memory Server [7]	4	28
GMS [8]	9	576
RRMP [9]	6	192
PGMS [10]	5	320
Dodo [3]	14	1,536
SAMSON [11]	10	24,576
Nswap [12]	4	2,048
HPBD [13]	17	34,816
dRamDisk [14]	5	10,240
Anemone [15]	10	9,216
JumboMem	256	1,048,576

but is afterwards transparent. With JumboMem, in contrast, applications are provided an illusion that they have direct access to all of the memory in a cluster. Existing scripts and binaries can be run directly, which is important for our goal of being able to rapidly develop data-analysis programs. We consider it awkward to have to maintain separate small-memory and large-memory variants of a program.

Kernel-level memory servers are a far more common implementation approach. The Global Memory Service (GMS) [8], Anemone [15], the High-Performance Networking Block Device (HPBD) [13], Nswap [12], dRamDisk [14] SAMSON [11], the Prefetching Global Memory System (PGMS) [10], and the works of Iftode et al. [7] and Markatos and Dramitinos [9] are all kernel-level implementations, many of which appear to the operating system as a swap device. The key advantages of a kernel-level memory-server implementation are transparency (no application modifications are necessary) and performance (paging is possible with a single transition into and out of kernel space). The key disadvantages are portability (the code is tied to a particular kernel) and applicability (ordinary users cannot install a kernel-level memory server). JumboMem strives to mimic as best as possible the advantages of kernel-level memory servers while suffering from none of the disadvantages.

JumboMem and the previously mentioned memory servers are related to but differ from projects such as the Network RamDisk [16], which appears to applications (and the operating system) as a disk but is in fact an interface to remote memory. JumboMem and its related projects also differ from software distributed shared memory systems such as TreadMarks [17]. While software distributed shared memory provides additional functionality over a memory server—separate processes can concurrently access the same block of memory over the network—maintaining a coherent view of memory can be costly and is not needed by all applications.

IV. IMPLEMENTATION

To use JumboMem a user simply runs a program with "`jumbomem -np` ⟨*nodes*⟩ ⟨*program*⟩ ⟨*arguments*⟩". JumboMem is implemented as a shared object (`libjumbomem.so`) and a wrapper script (`jumbomem`). The wrapper script points the `LD_PRELOAD` environment variable at `libjumbomem.so` so the dynamic linker will load JumboMem's symbols before loading ⟨*program*⟩'s symbols. `jumbomem` then launches ⟨*program*⟩ on the cluster.

A. Overview

JumboMem's implementation bears a lot in common with that of Network RAM [2]. Both systems implement a SIGSEGV handler that is invoked automatically whenever a process accesses a page of memory that is not locally resident. The SIGSEGV handler evicts a locally resident page from the master node by sending its contents over the network to a slave node and unmapping the page from the process's address space with the `munmap()` system call. Then, the SIGSEGV handler uses `mmap()` to map the page that faulted into the process's address space and acquires the page's contents from a slave node.

We opted to emphasize speed and simplicity over extensive features. First, in the current implementation the master's memory is considered to be exclusively a cache of the global memory spread across the slaves. Hence, the master avoids the complexity of having to remember which slave holds each of its pages. Virtual addresses map to slaves and slave memory addresses in a simple, uniform manner (round robin by default). Second, we assume that cluster nodes are homogeneous. Again, this supports a simple mapping of the global address space to slaves and slave memory addresses. Third, JumboMem does not provide support for fault tolerance, memory harvesting, job migration, multiple master processes, nodes that serve as both masters and slaves, or many other features that related memory-server projects provide.

JumboMem provides its own memory allocator (due to Doug Lea [18]) that allocates memory from one of two pools. Calls to `malloc()`, `free()`, etc. made by the application allocate memory from the global address-space pool that spans the cluster. Calls made by JumboMem and its libraries allocate memory from the operating system in the normal manner.

JumboMem's code is fairly modular and various pieces can easily be parameterized or replaced. Except where indicated we ran JumboMem using MPI [19] as the communication interface and a pseudo-NRU (not recently used) replacement scheme. We say *pseudo* NRU because, although recently fetched pages are retained over not recently fetched pages, the scheme by default does not additionally favor retaining modified pages over unmodified pages. Furthermore, JumboMem—in fact, any fully user-level memory server—cannot determine if a previously fetched page has since been used.

B. Limitations

Although our goal for JumboMem's is for it to be completely transparent to applications this goal is not perfectly realizable by a user-level memory server. For example, JumboMem does not currently support programs that `fork()` child processes because of the complications of coordinating accesses to remote memory across multiple processes. Furthermore, JumboMem would need to locate additional cluster

nodes to use and somehow manage to migrate the child processes to it.

JumboMem also does not currently support multithreaded programs. The difficulty in doing so is in preventing a race condition between one thread accessing a page while another thread is faulting it in. Consider, for example, two threads concurrently reading values from a large array. If thread A faults on a particular page, the JumboMem SIGSEGV handler must first `mmap()` the page then fill the page with data received from a remote node. Between the `mmap()` making the page available and the data arriving, thread B might read the page. Thread B will not fault (because the page is resident) but will see invalid data (because the valid data has not yet arrived). For multithreaded programs to work correctly with a user-level memory server a mechanism is needed to receive data into a buffer page then atomically map the buffer page into the target address. This may be possible by writing to a file which is then memory-mapped into the correct address but the costs in doing so are expected to be prohibitively expensive.

Finally, JumboMem "sees" only dynamically allocated memory. Large static arrays such as those commonly used in older Fortran programs cannot be distributed across the cluster.

C. Challenges

Although JumboMem has its limitations there were a number of challenges to producing a transparent user-level memory server that we managed to overcome. Generally, the solutions involve overriding various problematic functions in the standard C library.

As the first example of a challenge, GNU Octave [20] is written in C++ and includes a number constructors spread across multiple Unix shared objects. These constructors dynamically allocate memory before `main()` is called. The problem is that this allocation may occur before JumboMem has had a chance to initialize. `LD_PRELOAD` guarantees only that JumboMem's symbols have been loaded before GNU Octave's; it does not guarantee that JumboMem's initialization function will be called before any of GNU Octave's constructors. The solution was to have JumboMem's memory-allocation routines detect if JumboMem had been initialized and explicitly initialize it if not.

A second challenge involves file access. Normally, the operating system knows how to fault in buffers passed to system calls such as `read()` and `write()`. However, if a page in a buffer is not mapped (i.e., it is a remote JumboMem page) the operating system neither knows how to fault it in nor raises a SIGSEGV signal that JumboMem could catch; the operation simply fails. The solution was to trap `read()`, `write()`, etc. and force them to touch all of the pages in the buffer before passing control to the operating system.

Explicit `mmap()` calls were initially problematic for JumboMem because the master's global address space is sparse—only locally cached pages are resident—so an application's `mmap()` invocation can return memory mapped into the middle of the JumboMem-controlled region, thereby denying JumboMem access to that part of the address space. The solution was to trap `mmap()` and specify a target address outside of the JumboMem-controlled region.

Finally, the bane of any user-level memory server is that the operating system can decide at any time to evict some of a process's memory to disk to free up physical memory for other purposes (e.g., the buffer cache or other processes' virtual memory). While remote memory is expected to be faster than local disk, remote disk is expected to be the worst case for performance. Although JumboMem cannot prevent the operating system from evicting any of its memory to disk (`mlockall()` is a privileged call and we are assuming no administrative access) we found it effective to have the slave processes cycle through their memory when otherwise idle and touch every page. Doing so hints to the operating system that the memory is in active use and should not be reassigned to other tasks.

V. Analysis

Having described JumboMem's implementation we now analyze JumboMem's performance to determine if the implementation is reasonable. We begin by presenting the results of some microbenchmarks (Section V-A) in order to quantify the primitive costs of page replacement in JumboMem. Section V-B presents the page-replacement time in a slightly larger context by utilizing the CacheBench memory-performance benchmark [21]. To help support comparisons with other memory servers described in the literature Section V-C showcases JumboMem's performance when sorting the lines of a large file. Finally, JumboMem's performance in the context of a full, unmodified, application (an interactive run of GNU Octave) is presented in Section V-D.

Except where indicated otherwise all experiments in were performed on the PAL cluster at Los Alamos National Laboratory. PAL is a 256-node (1024-core) cluster with a total of 1 TB of memory and is described in detail in Table II. When PAL was first introduced (2005) it was the world's 132nd fastest supercomputer according to the November 2005 Top500 list (http://www.top500.org/) but dropped off the Top500 list in June 2007. PAL is an ideal testbed for evaluating the performance of a memory server because of its relatively large node count and memory capacity.

We utilize a JumboMem page size of 256 KB for all of the experiments in this section except where noted otherwise. While different memory-access patterns favor different page sizes—we quantify this in Section V-A—we determined empirically that 256 KB gives reasonable peformance across the suite of applications that we consider for this paper.

A. Page-Replacement Time

We begin by comparing the cost of page replacement in JumboMem to the cost of ordinary, kernel-based page replacement. To measure kernel-based page replacement we wrote a simple microbenchmark that allocates a region of memory that is significantly larger than the amount of available physical memory, accesses data throughout that region, and records a histogram of the access times. The results are fairly

TABLE II
PAL CLUSTER CHARACTERISTICS

Category	Item	Value
CPU	Type	AMD Opteron 270
	Cores	2
	Clock rate	2 GHz
Node	CPU sockets	2
	Count	256
	Motherboard	Tyan Thunder K8SRE (S2891)
	BIOS	LinuxBIOS
Memory	Capacity/node	4 GB
	Type	DDR400 (PC3200)
Local disk	Capacity	120 GB
	Type	Western Digital Caviar 120GB RE (WD1200SD)
	Cache size	8 MB
Network	Type	InfiniBand
	Interface	Mellanox Infinihost III Ex (25218) HCAs with MemFree firmware v5.2.0
	Switch	Voltaire ISR9288 288-port
Software	Operating system	Linux 2.6.18
	OS distribution	Debian 4.0 (Etch)
	Messaging layer	Open MPI 1.2
	Job launch	Slurm

noisy. Discarding "fast" data points (indicating a page fault that was satisfied by memory or the disk cache) left a probability distribution with a sample mean of 8.3 ms and standard deviation of 3.1 ms. A fault time of 8.3 ms is corroborated by a run of the Bonnie++ benchmark [22], which measures 175 disk accesses per second or 5.7 ms apiece. Because a page fault comprises both a read (fetch) and a write (evict)—in fact, Linux 2.6.18 replaces clusters of eight adjacent pages at once—this corresponds to a total of 11.4 ms, which is close to our measurement of 8.3 ms.

We instrumented JumboMem to break down the cost of page replacement into the following time components: transitioning from the application to JumboMem's fault handler via delivery of a SIGSEGV signal, selecting a page frame to replace, allocating backing store for the new page, requesting the faulted page from a slave node, sending the corresponding victim page to a slave node, deallocating backing store from the victim page, and returning control to the application. Table III presents the results of our measurements taken from 49 runs representing a variety of JumboMem page sizes.

TABLE III
BREAKDOWN OF JUMBOMEM PAGE-REPLACEMENT COSTS

Operation	Cost (µs)
Transition from the application to JumboMem	13.1 ± 6.5
Select a replacement page frame	2.4 ± 1.1
Allocate backing store (`mmap()`)	7.1 ± 2.6
Communication (evict + fetch) of b bytes	$\sim(2.45\times10^{-3})b+81.3$
Deallocate backing store (`munmap()`)	24.5 ± 21.1
Other, unaccounted time spent in the fault handler	5.3 ± 4.7
Transition from JumboMem to the application	2.3 ± 1.0
Total fixed costs	60.4 ± 32.8

One observation to make from Table III is that the data is extremely noisy. The standard error tends to be approximately 50% for each measurement. Still, we can see that the expected fixed cost of a user-level page-replacement scheme is on the order of 30–90 µs (55,000–187,000 CPU cycles) on current hardware.

The cost of communication is naturally a function of the JumboMem page size. JumboMem communication comprises sending a short page-request message to a slave, sending a page to a (typically different) slave, and receiving a page from the first slave. JumboMem utilizes nonblocking primitives (`MPI_Isend()` and `MPI_Irecv()` when using MPI [19]) so sends and receives can overlap if overlapping communication is supported by the underlying messaging layer (not the case with Open MPI [23] over InfiniBand [4]). The regression listed in Table III models the cost of communication as 2.45 ns per byte plus 81.3 µs. Hence, it can be expected to take 152 µs to replace a 4 KB page or 2.7 ms to replace a 1 MB page. (Figure 2 illustrates the accuracy of that calculation.) In short, the dominant cost of page replacement is communication time; our fastest runs indicate that 78.9% of the total time for 4 KB pages and 95.3% for 1 MB pages is spent in communication. The implication is that a user-level memory server can be expected to observe similar performance to a kernel-level memory server because both need to pay the same communication costs.

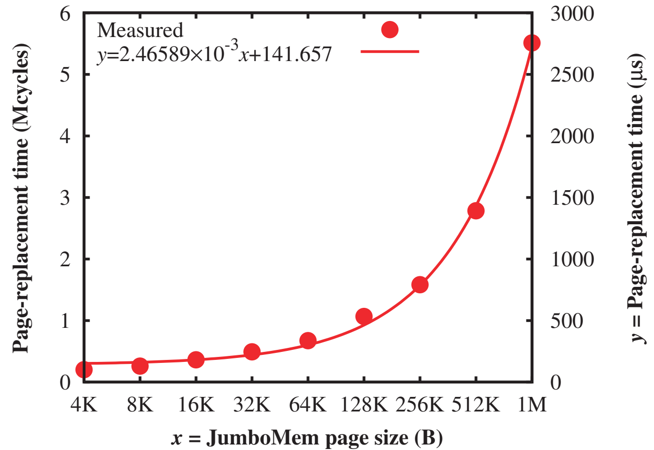

Fig. 2. Total cost of page replacement as a function of page size

JumboMem lets the user select a page size (any multiple of the operating-system page size) at job-launch time. The choice of page size is important for two reasons. First, given a fixed number of per-process page mappings, more memory can be mapped—and therefore cached at the master node—with large pages than with small pages. As stated in Section II, Linux allows by default only 65,536 page mappings per process. Because adjacent mappings are merged the worst case is represented by alternating mapped and unmapped regions. This case corresponds to 128 GB of locally cached memory when using 1 MB JumboMem pages, for example, but only 512 MB when using 4 KB JumboMem pages—much less than

the available RAM is a modern cluster node.

The second reason that the selection of JumboMem page size is important is performance. Figure 2 on the preceding page shows that it takes less time to replace a small page than a large page. However, more bytes are replaced per unit time when using large pages than small pages because more of the fixed costs can be amortized, leading to a lower effective per-byte cost. Using only the measurements from Figure 2 on the previous page we can compute analytically the regime in which one page size should outperform another. Figure 3 plots the time spent replacing pages as a function of the sparseness of the data accesses when using either 4 KB or 1 MB sized pages. On the left of the graph, where only one out of every million bytes is accessed, it makes more sense to use 4 KB pages because each access requires a page fetch and a 1 MB page takes 27.2 times as long to fetch as a 4 KB page. On the right of the graph, where one of out of every thousand bytes is accessed, it makes more sense to use 1 MB pages because the higher spatial locality favors the larger page's lower per-byte cost (a factor of 9.4 improvement). The crossover point between the two pages is at 2.59×10^{-5}, or one byte accessed out of every 38,536. Analogous calculations can be performed for other page sizes. The ability to select a page size at job-launch time is an important performance optimization available to user-level memory servers. Kernel-level memory servers, which are bound to the operating system's page size, can utilize page prefetching to exploit spatial locality but must still pay a large fixed cost for every faulted page.

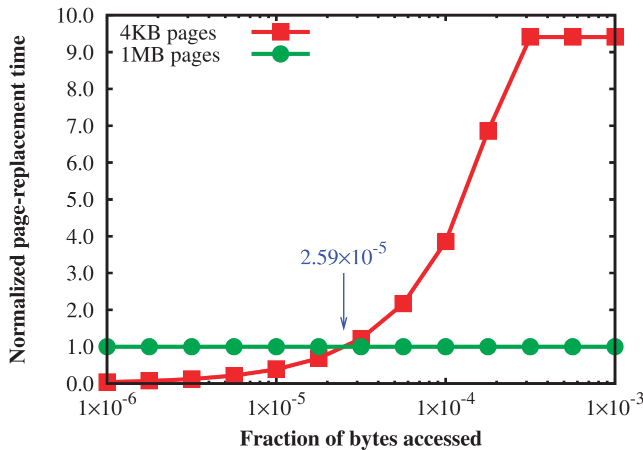

Fig. 3. Page-replacement time as a function of memory usage

B. *CacheBench*

CacheBench is a benchmark that measures bandwidth across the various levels of a computer's memory hierarchy [21]. Figure 4 plots the bandwidth measured by CacheBench's read/modify/write test, in which each word in a vector is read, modified, and written in place. We ran CacheBench out to a maximum vector size of 512 GB (2^{39} bytes) with "`cachebench -m 39 -e 1 -x 0 -d 2 -b`". Figure 4 shows that JumboMem over InfiniBand (the *JumboMem/IB* curve) introduces no noticeable overhead over the non-JumboMem case (the *Local swap* curve) when the vector fits in the level 1 cache, level 2 cache, or main memory. Once the vector no longer fits in main memory, the non-JumboMem performance plummets because the bandwidth is limited by the disk's random-seek latency of 175 seeks/s or 5.5 MB/s for 32 KB accesses (4 KB bytes/page × 8 pages/cluster). With JumboMem, the bandwidth is limited by network and messaging-layer bandwidth (707 MB/s for a 256 KB message). The fact that the JumboMem run realizes more bandwidth than the network's physical limit implies that the NRU page-replacement algorithm is still observing some page reuse, even at large scale.

The curve in Figure 4 labeled *Altix* represents a run of CacheBench on an SGI Altix 3700 containing 256 1.3 GHz Itanium II CPUs and a total of 1 TB of physical memory. Memory in the Altix is cache-coherent so a single process can access all of the memory in the system. As a point of reference, the world's largest shared-memory system at the time of this writing is an Altix 3700 at the Japan Atomic Energy Agency (formerly the Japan Atomic Energy Research Institute) with 13 TB of RAM in a single, cache-coherent domain [24]. Although the Altix and our cluster have different CPUs, clock rates, memory bandwidths, and networks (which implies that the *L1*, *L2*, and *Main* lines in Figure 4 are not relevant to the *Altix* curve), comparing the CacheBench performance of the two systems shows how JumboMem's software-only approach compares to the Altix's full-hardware implementation. In short, JumboMem performs quite admirably; the Altix's read/modify/write bandwidth over a quarter-terabyte vector is only 2.9 times what JumboMem can achieve on a commodity cluster with no special hardware for rapid, cross-node memory movement.

C. *Sorting*

Many papers on memory servers analyze memory-server performance with a sorting benchmark (typically using a quicksort algorithm) and generally observe a substantial performance improvement over paging to disk [7], [9], [13], [15]. To help compare JumboMem to prior memory-server work we too present sorting performance. We measured the time needed for the unmodified GNU *sort* program to sort a file containing 2^{25} ($\sim 32 \times 10^6$) lines of 64 characters apiece, for a total of 2 GB. Unlike most related projects we test our memory server with both internal and external sorting routines (i.e., fitting entirely within virtual memory versus explicit staging of partial results to and from disk). Figure 5 presents the time needed to sort the 2 GB file with either JumboMem or a local swap device and with internal or external sorting.

To measure the performance of an internal sort we specified "`-S 32G`" to the *sort* program to allow it to use up to 32 GB of address space to hold the input file and various intermediate data structures before switching to an external sort. (In fact, significantly less space was actually needed.) For the internal-sort curves in Figure 5 (*Local swap internal* and *JumboMem internal*) the x axis in represents the available memory on

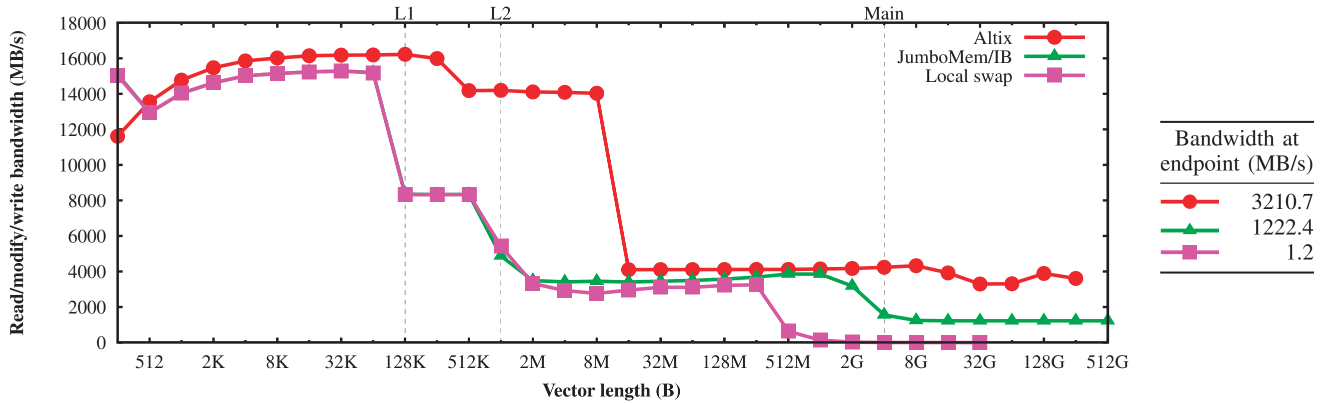

Fig. 4. CacheBench performance of JumboMem vs. alternatives

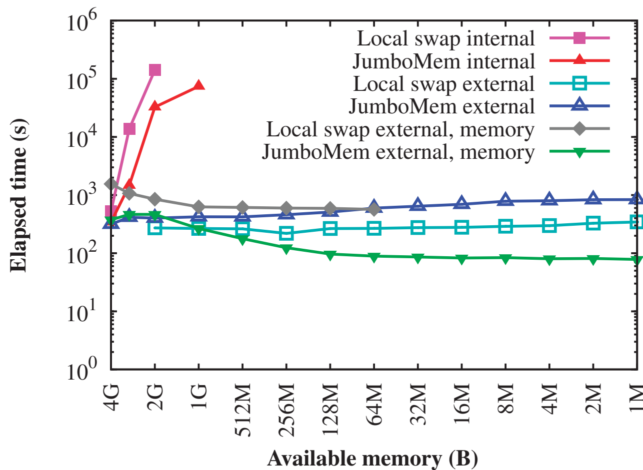

Fig. 5. Time for GNU *sort* to sort a 2 GB file

the master node, which we artificially limited using another process that allocated and locked 4 GB minus the x-axis value. The data show that with 4 GB available to *sort* (plus the operating system and various dæmon processes running on the node), JumboMem sorted the input file in 65.4% of the time needed when using local swap. With only 2 GB available, JumboMem sorted the input file in only 23.0% of the time needed when using local swap.

To measure the performance of an external sort we specified "`-S` $\langle x \rangle$" to the *sort* program for each value of x on Figure 5's x axis. Doing so limits the fraction of the data that *sort* is allowed to keep resident. The first thing to notice about the *Local swap external* and *JumboMem external* curves in Figure 5 is that the performance is significantly better than the corresponding *Local swap internal* and *JumboMem internal* curves. This is to be expected because the reduction in the number of page faults improves performance more than the additional, explicit disk activity degrades it. What is surprising is that JumboMem performs *worse* than local swap on the external sort. One would expect the two to perform equivalently because paging activity is minimal in both cases. Additional measurements indicate that the source of the performance difference is an artifact of an extra cost JumboMem incurs for each file operation. Specifically, JumboMem intercepts the C library's `read()` and `write()` calls (also `fread()` and `fwrite()`) and attempts to fault in all pages of the read/write buffer before allowing the call to proceed. Doing so prevents the call from terminating prematurely upon accessing an unmapped page—the operating system does not invoke JumboMem's SIGSEGV handler in this case—but incurs a cost we measured empirically to average 3.5 µs per file operation when running *sort*. Multiplying 3.5 µs by the number of file operations performed for each data point (which range from 67,126,478 to 142,937,415 across the *Local swap external* curve) is 95.3% correlated with the difference between the *JumboMem internal* and *Local swap internal* curves, which strongly suggests that JumboMem's extra file-operation overhead explains the performance difference.

Figure 5 contains two additional curves, *Local swap external, memory* and *JumboMem external, memory*. These represents runs of a *modified* version of GNU *sort* in which reads and writes of temporary files are replaced by copies in and out of a memory buffer. The idea is to see if JumboMem can be used to exploit the improved locality of an external sort without incurring the extra costs associated with the additional disk activity. As is evident from Figure 5, the extra locality greatly benefits JumboMem, making it the fastest sort overall and 77.3% faster than the next-fastest sort in the figure. However, the extra memory pressure greatly degrades *sort*'s performance when run with local swap, making the memory-based external sort perform worse than the disk-based external sort.

The primary conclusion to draw from the preceding study of sorting performance is that JumboMem can outperform kernel-based paging even for programs such as GNU *sort* that exhibit comparatively little spatial locality. Although JumboMem incurs only a small performance penalty for each file operation, *sort* executes an unusually large number of small file operations—over a hundred million 64-byte writes when sorting our input file. We anticipate that typical applications that will be run with JumboMem place less demand on the

Listing 1. A matrix-vector multiplication benchmark written in GNU Octave

```
function matvecmult (maxvectlen)
  for vectlen = 2 .^ (0:log2(maxvectlen))
    printf("%6d␣", vectlen);
    iters = 0;
    tic();
    do
      A = rand(vectlen, vectlen);
      x = rand(vectlen, 1);
      b = A * x;
      iters = iters + 1;
      t1 = toc();
    until (t1 > 5);
    printf("%.10f\n", t1/iters);
    fflush(1);
  endfor
endfunction
```

filesystem and will therefore benefit more from JumboMem's faster paging than suffer from the additional overhead of intercepted file operations.

D. GNU Octave

One anticipated use of JumboMem is to process large data sets (e.g., an instruction trace produced by a cycle-accurate processor simulator) from *interactive* data-analysis applications. For example, a user may want to perform model selection on a large data set. Because model selection is an iterative process—a user repeadedly selects a function, fits parameters to it, graphs the result, and refines the function—the data must be processed repeatedly. Replacing the multiple file accesses with a single file access plus multiple remote-memory accesses can yield a substantial performance benefit.

As a simple test of JumboMem's ability to work with unmodified, interactive data-analysis applications we launched an interactive GNU Octave [20] session using JumboMem. At the GNU Octave prompt, we entered and invoked the function shown in Listing 1, which measures the time to perform matrix-vector multiplication on increasingly large matrices and vectors. Each doubling of the matrix and vector dimensions quadruples the number of multiplications the function performs. Figure 6 shows that the run time scales perfectly even up to a massive matrix containing 262,144 elements in each dimension, requiring $(262{,}144^2 + 262{,}144) \times 8$ bytes = 512 GB of memory to hold both the matrix and the vector. For comparison, Figure 6 shows the performance of the same interaction with GNU Octave but run with paging to local disk instead of with JumboMem. Although JumboMem and local swap exhibit nearly equal performance for small problem sizes, as soon as paging is required the performance of local swap drops significantly while JumboMem's performance continues to scale smoothly. At the 32 GB data point (the final data point in the local-swap case) JumboMem multiplies a $65{,}536 \times 65{,}536$ matrix by a 65,536-element vector in only 6 minutes versus 1 hour, 43 minutes when using local swap. Although matrix-vector multiplication per se is not a realistic use of a memory server—large, dense matrices are rarely encountered in practice and matrix-vector multiplication is an easily parallelizable algorithm—the significance of this study is the demonstration of JumboMem's utility for interactive data analysis. With JumboMem, users can spend their time analyzing results, not wasting it parallelizing data-analysis scripts that may see limited use.

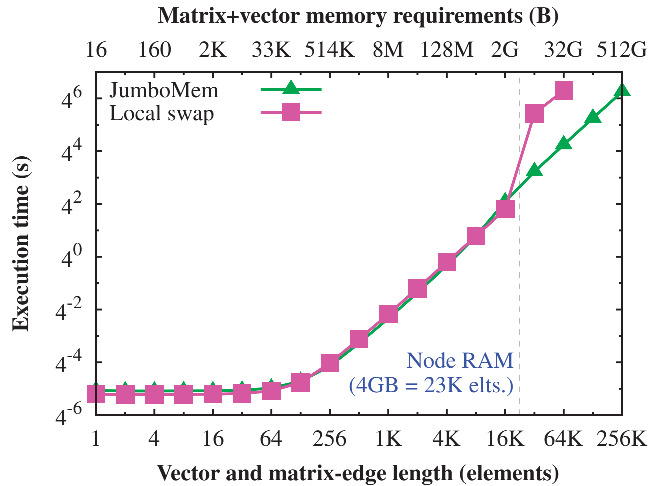

Fig. 6. Performance of the GNU Octave script presented in Listing 1

VI. Future Work

JumboMem is a highly parameterized, modular piece of software that can readily be modified to perform a variety of additional studies in the context of user-level memory servers. The following are some open questions that may be interesting avenues for future research:

- Can performance be improved by dynamically altering the JumboMem page size based on the sparseness of the data accesses (cf. Table III on page 5 and Figure 3 on page 6)?
- According to Figure 4 on the preceding page, main-memory bandwidth is approximately 3 GB/s, which is much higher than the network's peak bandwidth of approximately 1 GB/s. However, quad data rate (QDR) InfiniBand signaling, expected to be available relatively soon, has a peak data rate of 4 GB/s with 4X links and 12 GB/s with 12X links, both faster than current memory bandwidths and likely to equal memory bandwidths that will be available in the near future. With network bandwidth no longer being the performance limiter, will there be a qualitative change in the set of applications that a memory server can run effectively?
- Although JumboMem is more transparent than any other user-level memory server there are still some constructs that it cannot handle (Section IV-B). Can these limitations be overcome? What would be the impact on performance?
- Can one design a page-replacement algorithm that is specifically suited to memory servers? Are there any

aspects of the network topology, performance characteristics of the messaging layer, or features of the network interface that can be exploited to achieve higher-performance page replacement? One issue is that a user-level memory server has access to much less information than does the operating system's page-replacement algorithm. For example, on our cluster, Linux's page-replacement algorithm takes advantage of the Opteron TLB's "accessed" and "dirty" bits [25]. In contrast, a user-level memory server such as JumboMem cannot determine if a page was accessed since it was last checked and cannot quickly determine if a page has been modified.
- User-level memory servers are inherently more limited than kernel-level memory servers. Can one design a memory server in which kernel-level components are available for increased flexibility and/or performance but are not required for basic operation?

VII. Conclusions

Our first conclusion is that it is possible to implement a memory server that runs entirely in user mode and needs no administrative privileges to install yet is transparent to applications. The majority of memory servers described in the literature, including Memory Server [7], GMS [8], RRMP [9], PGMS [10], SAMSON [11], Nswap [12], HPBD [13], dRamDisk [14], and Anemone [15], run in kernel mode and generally replace the swap-device driver or page-replacement code with code to distribute pages across the cluster. Network RAM [2] and Dodo [3], representing some of the only published examples of user-level memory servers, require application modifications in order to exploit remote memory. JumboMem, introduced in this paper, is the first entirely user-level memory server that can grant unmodified binaries access to all of the RAM in a cluster.

The second conclusion that one should draw from this work is that there is no significant or inherent performance advantage to a kernel-level memory-server implementation over a user-level implementation (except perhaps in pathological cases such as the *sort* runs in Section V-C in which there is virtually no paging activity but an excessive number of small file operations). Our analysis of page-replacement costs in JumboMem indicates that the majority of the page-replacement time is spent in the communication subsystem (78.9% of the time for 4 KB pages and 95.3% for 1 MB pages). The same communication costs necessarily apply both to user-level and kernel-level memory servers. Assuming a 4 KB page size and infinitely fast kernel-mode execution, a kernel-level memory server would run only $\sim$21.1% faster than the user-level JumboMem.

Although we have demonstrated that there is no *performance* advantage to a kernel-level over a user-level memory server, each approach has its respective advantages. Kernel-level memory servers work with *all* applications, including those with large, static memory regions, those that spawn child processes, and those that make use of threads. In addition, kernel-level memory servers can handle workloads of multiple, independent, large-memory applications. User-level memory servers, however, provide greater flexibility—the ability to alter the page size or page-replacement algorithm on a per-application basis, for example—which can lead to improved performance.

Our third conclusion is that a memory server can be made to scale to hundreds of nodes and nearly a terabyte of available memory. Never before has anyone published memory-server performance data for more than a few tens of nodes or a few tens of gigabytes. There are numerous challenges in running a memory server at large scale including nonlinear memory requirements for communication buffers and limited available per-process page mappings. (Futhermore, not all applications are 64-bit clean. In the process of writing this paper we had to submit a CacheBench patch to get it to support 4 GB of memory [cf. http://lists.cs.utk.edu/pipermail/llcbench/2007-March/000107.html] and a GNU Octave patch to get it to support vector lengths of 2^{31} or more elements [cf. http://lists.cs.utk.edu/pipermail/llcbench/2007-March/000107.html].) However, JumboMem proves that these challenges can be overcome and that it is possible today to provide nearly a terabyte of memory to an unmodified, sequential application.

References

[1] D. Comer and J. Griffioen, "A new design for distributed systems: The remote memory model," in *USENIX Summer 1990 Technical Conference*, Anaheim, California, Jun. 11–15, 1990, pp. 127–136. [Online]. Available: http://protocols.netlab.uky.edu/~griff/papers/usenix90.pdf

[2] E. A. Anderson and J. M. Neefe, "An exploration of network RAM," EECS Department, University of California, Berkeley, Tech. Rep. UCB/CSD-98-1000, Dec. 9, 1994. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/1998/5443.html

[3] S. Koussih, A. Acharya, and S. Setia, "Dodo: a user-level system for exploiting idle memory in workstation clusters," in *The Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC'99)*, Redondo Beach, California, Aug. 3–6, 1999, pp. 301–308. [Online]. Available: http://www.cs.gmu.edu/~setia/papers/dodo.pdf

[4] *InfiniBand Architecture Specification Release 1.2*, InfiniBand Trade Association, Oct. 2004. [Online]. Available: http://www.infinibandta.org/specs/

[5] H. Garcia-Molina, R. J. Lipton, and J. Valdes, "A massive memory machine," *IEEE Transactions on Computers*, vol. C-33, no. 5, pp. 391–399, May 1984.

[6] P. J. Leach, P. H. Levine, B. P. Douros, J. A. Hamilton, D. L. Nelson, and B. L. Stumpf, "The architecture of an integrated local network," *IEEE Journal on Selected Areas in Communications*, vol. SAC-1, no. 5, pp. 842–857, Nov. 1983.

[7] L. Iftode, K. Li, and K. Petersen, "Memory servers for multicomputers," in *Compcon Spring '93, Digest of Papers*, San Francisco, California, Feb. 22–26, 1993, pp. 538–547. [Online]. Available: http://www.cs.princeton.edu/~liv/papers/compcon93.ps

[8] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath, "Implementing global memory management in a workstation cluster," in *15th ACM Symposium on Operating Systems Principles (SOSP 1995)*, ser. Operating System Review, vol. 29(5), Copper Mountain, Colorado, Dec. 3–6, 1995, pp. 201–212. [Online]. Available: http://www.cs.washington.edu/homes/levy/opal/sosp.ps

[9] E. P. Markatos and G. Dramitinos, "Implementation of a reliable remote memory pager," in *USENIX Annual Technical Conference*. San Diego, California: USENIX Association, Jan. 22–26, 1996, pp. 177–190. [Online]. Available: http://www.usenix.org/publications/library/proceedings/sd96/full_papers/markatos.ps

[10] G. M. Voelker, E. Anderson, T. Kimbrel, M. J. Feeley, J. Chase, A. Karlin, and H. M. Levy, "Implementing cooperative prefetching and caching in a globally-managed memory system," in *1998 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'98/PERFORMANCE'98*, Madison, Wisconsin, Jun. 24–26, 1998, pp. 33–43. [Online]. Available: http://www.cs.ucsd.edu/~voelker/pubs/gms-sigmet98.pdf

[11] E. W. Stark, "SAMSON network memory server project," http://bsd7.cs.sunysb.edu/~samson/, Aug. 29, 2003.

[12] T. Newhall, S. Finney, K. Ganchev, and M. Spiegel, "Nswap: A network swapping module for Linux clusters," in *9th International Euro-Par Conference*, ser. Lecture Notes in Computer Science, H. Kosch, L. Böszörményi, and H. Hellwagner, Eds., vol. 2790. Klagenfurt, Austria: Springer, Aug. 26–29, 2003, pp. 1160–1169. [Online]. Available: http://www.cs.swarthmore.edu/~newhall/europar03.pdf

[13] 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 International Conference on Cluster Computing (Cluster 2005)*, Boston, Massachusetts, Sep. 27–30, 2005. [Online]. Available: http://www.cse.ohio-state.edu/~liangs/paper/liang-cluster05.pdf

[14] V. Roussev, G. G. Richard III, and D. Tingstrom, "dRamDisk: Efficient RAM sharing on a commodity cluster," in *25th IEEE International Performance, Computing, and Communications Conference (IPCCC 2006)*, Phoenix, Arizona, Apr. 10–12, 2006, pp. 193–198. [Online]. Available: http://www.cs.uno.edu/~golden/Stuff/ipc1568975346.pdf

[15] M. R. Hines, J. Wang, and K. Gopalan, "Distributed Anemone: Transparent low-latency access to remote memory," in *13th International Conference on High Performance Computing (HiPC 2006)*, ser. Lecture Notes in Computer Science, Y. Robert, M. Parashar, R. Badrinath, and V. K. Prasanna, Eds., vol. 4297. Bangalore, India: Springer, Dec. 18–21, 2006, pp. 509–521. [Online]. Available: http://www.cs.binghamton.edu/~mhines/papers/hipc06.pdf

[16] M. D. Flouris and E. P. Markatos, "The Network RamDisk: Using remote memory on heterogeneous NOWs," *Cluster Computing*, vol. 2, no. 4, pp. 281–293, Dec. 1999. [Online]. Available: http://www.ics.forth.gr/carv/r-d-activities/networkMem/TR226/nrd.html

[17] C. Amza, C. Alan L, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "TreadMarks: Shared memory computing on networks of workstations," *IEEE Computer*, vol. 29, no. 2, pp. 18–28, Feb. 1996. [Online]. Available: http://csalpha.ist.unomaha.edu/~stanw/papers/csci8550/96-treadmarks.pdf

[18] D. Lea. (2000, Apr. 4,) A memory allocator. [Online]. Available: http://g.oswego.edu/dl/html/malloc.html

[19] *MPI: A Message-Passing Interface Standard*, Message Passing Interface Forum, Jun. 12, 1995. [Online]. Available: http://www.mpi-forum.org/docs/mpi-11.ps

[20] J. W. Eaton, *GNU Octave Manual*. Network Theory, Mar. 2002. [Online]. Available: http://www.gnu.org/software/octave/doc/interpreter/

[21] P. J. Mucci, K. London, and J. Thurman, *The CacheBench Report*, University of Tennessee, Knoxville, Nov. 1998. [Online]. Available: http://icl.cs.utk.edu/projects/llcbench/cachebench.pdf

[22] R. Coker. Bonnie++ now at 1.03a (very stable). [Online]. Available: http://www.coker.com.au/bonnie++/

[23] R. L. Graham, G. M. Shipman, B. W. Barrett, R. H. Castain, G. Bosilca, and A. Lumsdaine, "Open MPI: A high-performance, heterogeneous MPI," in *Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (HeteroPar'06)*, Barcelona, Spain, Sep. 25–28, 2006, pp. 1–9. [Online]. Available: http://www.open-mpi.org/papers/heteropar-2006/heteropar-2006-paper.pdf

[24] Silicon Graphics, Inc., "Japan Atomic Energy Research Institute will introduce 2,048 processor Linux supercomputer," Nov. 2004. [Online]. Available: http://www.sgi.com/company_info/newsroom/press_releases/2004/november/jaeri.html

[25] *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, Advanced Micro Devices, Sep. 5, 2006, publication number 24593, revision 3.12. [Online]. Available: http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24593.pdf