A Collaborative Memory System for High-Performance and Cost-Effective Clustered Architectures

Ahmad Samih[†‡], Ren Wang[†], Christian Maciocco[†], Tsung-Yuan Charlie Tai[†] and Yan Solihin[‡]

[†]System Architecture Lab
Intel Research Labs
Hillsboro, OR, USA
{ahmad.samih, ren.wang,
christian.maciocco, charlie.tai}@intel.com

[‡] Dept. of Electical and Computer Engineering
North Carolina State University
Raleigh, NC, USA
{aasamih, solihin}@ncsu.edu

ABSTRACT

With the fast development of highly integrated distributed systems (cluster systems), especially those encapsulated within a single platform [28, 9], designers have to face interesting memory hierarchy design choices that attempt to avoid disk storage swapping. Disk swapping activities slow down application execution drastically. Leveraging remote free memory through *Memory Collaboration* has demonstrated its cost-effectiveness compared to overprovisioning for peak load requirements. Recent studies propose several ways on accessing the under-utilized remote memory in static system configurations, without detailed exploration on the dynamic memory collaboration. Dynamic collaboration is an important aspect given the run-time memory usage fluctuations in clustered systems.

In this paper, we propose an Autonomous Collaborative Memory System (ACMS) that manages memory resources dynamically at run time, to optimize performance, and provide QoS measures for nodes engaging in the system. We implement a prototype realizing the proposed ACMS, experiment with a wide range of real-world applications, and show up to 3x performance speedup compared to a non-collaborative memory system, without perceivable performance impact on nodes that provide memory. Based on our experiments, we conduct detailed analysis on the remote memory access overhead and provide insights for future optimizations.

1. INTRODUCTION

With every new software generation, applications' memory footprints are growing exponentially in two dimensions – horizontally due to an increase in their data set, and vertically due to additional software layers. This fast memory requirement growth outpaces the growth in the capacity of current memory modules (RAMs) [15]. This leads the OS's virtual memory manager to resort to swapping to storage devices, e.g., Hard Disk Drive (HDD) and Solid State Drive (SSD). Swapping devices such as HDDs or even SSDs operate at several orders of magnitude slower compared to main memory modules [25]. Excessive paging activity to and from the swapping device renders a system crawling as the CPU is mostly waiting for I/O activity. The performance degradation, in turn, poses serious power implications since the slow execution keeps the CPU and system in high power state longer than necessary.

Recently, we see the trend of the fast development of high density, low power, highly integrated distributed systems such as clustered systems (e.g., Seamicro's SM1000-64HD[28], Intel's microServer [9]). With these systems, hundreds or even thousands of independent computing nodes are encapsulated within a single platform. This, therefore, poses interesting challenges as to how designers could restructure the memory hierarchy to achieve optimal performance given a certain peak load requirement, with consideration of the cost and energy budgets.

There is a spectrum of solutions that attempt to bridge the vast performance gap between the local memory and the disk storage in clustered systems by avoiding swapping activity as much as possible. One end of the spectrum suggests over provisioning the system with more precious resources. Over provisioning may range from installing more physical memory, adding dedicated memcached servers [1], leveraging a hybrid memory system of PCM, PRAM, and DRAM [25, 27, 10], or even adding a dedicated storage servers that stores all data on their main memory (RAMs); namely the RAMCloud [23]. While over provisioning the system to accommodate all memory needs solves the problem, it comes with prohibitive costs and excessive power budget.

The other end of the spectrum, suggests a more cost-effective design by improving the aggregate cluster memory utilization. At high level, improving cluster utilization involves making use of idle memory located at remote nodes, namely Memory Collaboration. Memory Collaboration can be categorized into two approaches:*remote memory swapping* [14, 22, 16, 17, 38], and *remote memory mapping* [20, 38].

Remote memory mapping techniques deal with the remote memory as an extension to the local memory space. Such techniques usually require inflexible malloc-like APIs to manage local and remote memory resources, or recompilation of applications to distribute the statically defined memory structures (i.e., arrays) onto local and remote memories.

ASBD'11, October 10, 2011, Galveston Island, TX, USA.

Further, some remote memory mapping techniques such as [20] requires user intervention to explicitly define the availability of memory space at remote nodes.

In this paper, we focus on remote memory swapping techniques, which deal with remote memory as a swap device. Such approaches have demonstrated the ability to be deployed transparently without modifying the OS or the running applications, while at the same time partially filling the performance gap between local memory and hard disk with a cost-effective design.

Current remote memory swapping proposals, however, often focus on static system configurations and lack the detailed investigation, implementation and evaluation on the aspect of dynamically detecting, provisioning, and utilizing remote memory to optimize performance and energy for the whole cluster. Furthermore, QoS guarantees for nodes that donate part of their memory as a swap space are often not considered, which may lead to significant performance degradation for such nodes.

In this paper, we propose an Autonomous Collaborative Memory System (ACMS) for Clustered Architectures. In particular, we deliver the following contributions:

1. We propose a system architecture and a memory acquisition protocol to perform robust, efficient, and autonomous memory collaboration across multiple nodes within a cluster. The proposed protocol allows for multiple nodes to dynamically discover, allocate, and deallocate remote memory based on the local memory requirements.

2. We demonstrate the feasibility and benefit of the proposed ACMS by developing a prototype and evaluating real-world applications. Our results show that an ACMS-based system can *adapt* to workload dynamics and memory variations, and achieves up to 3x speedup compared to a non-collaborative memory system, which also leads to significant power savings due to shorter execution time.

3. We analyze various performance bottlenecks and overheads during the lifetime of the remote memory access, study several optimizations to improve memory collaboration performance; based on the analysis, we give insights into how to take advantage of software/hardware optimizations to significantly speed up the remote memory access.

The rest of the paper is organized as following. Section 2 reviews the related work. Section 3 motivates our dynamic approach for collaborative memories in clustered architectures. Section 4 describes the design of the proposed Autonomous Collaborative Memory System (ACMS). Section 5 describes the implementation details and prototyping of ACMS. Section 6 describes the evaluation methodology and provides the results from our evaluations and analyzes the findings. Section 7 concludes the work and discusses the future works. Section 8 extends special thanks to the paper reviewers and people who contributed to the work.

2. RELATED WORK

There is a rich body of work that has studied the problem of managing capacity at different levels of the memory hierarchy [30, 31, 6, 8, 24, 39, 5, 26, 34, 3]. However, in this work we focus on improving cluster throughput by managing the capacity at the main memory level. Prior art in this area can be divided into three main categories:

Modifying the memory hierarchy to hide/avoid disk activity. Several *high-cost* proposals argue for the need to redesign the memory hierarchy [25, 27, 10], or add additional resources to the cluster in order to avoid prohibitive disk activity. In particular, a recent proposal; the RAMCloud [23], motivates the need to replace disk storage with permanent RAM storage in order to reduce latency and improve throughput. The RAMCloud requires thousands of dedicated, interconnected commodity servers attached to the cluster to deliver its promise, which as the authors mention in their paper, comes at a high cost per bit, and a high energy usage per bit.

In [15], Lim *et. al.,* avoid to overprovision individual servers by encapsulating large portion of memory in remote dedicated memory blades which dynamically assigns memory to individual servers when needed. Although this scheme provides better utilization of aggregate memory resources, it is targeted for commodity blade servers and may require hardware changes to access the remote memory.

Management of memory resources under single OS image. In distributed systems with a single OS image (DOSes) [35], the entire address space is made visible to each process running on any node. This assumes a coherent underlining shared memory architecture. Several previous studies have shown that DOSes suffer performance and scalability [4] issues due to their shared memory architecture. Further, as reported in [4], DOSes are relatively expensive to maintain, and to deploy.

Management of memory resources under multiple OS images. Works that belong to this category are closest to our work in terms of the scope of the problem. In distributed systems with multiple OS images, each node in the system can leverage remote memory at another node by either paging to/from the remote memory [14, 22, 16, 17, 38, 7], or by extending its address space to encapsulate the remote memory. However, these schemes lack the ability to deal with the temporal/spatial node memory requirements fluctuation within the cluster to achieve optimized performance and energy-efficient memory collaboration. Further, prior proposals do not provide Quality-of-Service measures to protect nodes donating part of their memories from having their performance negatively impacted. To address these concerns, we design a run-time mechanism to manage the memory resources across collaborating nodes within the cluster, and we provide QoS measures for individual nodes.

3. MOTIVATION FOR DYNAMIC MEMORY COLLABORATION

In Section 1, we have discussed that limited memory resources lead to resorting to storage devices which has major implications on performance. For single-node systems, if over provisioning is not an option, the O/S has to start paging to and from the disk and therefore suffer the high latencies.

With multi-node clusters [28, 9], the overall picture is different. Some nodes in the cluster may over utilize their memory system, while other nodes may under-utilize them, and the dynamics often change over the time. This imbalance in memory usage across different nodes in a cluster has motivated our work to investigate techniques to make use of the under-utilized, and fast remote memory, over using the slow, storage device.

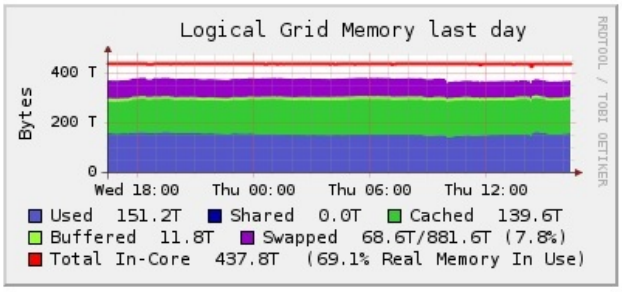

Figure 1: Example of Data Center Memory Usage Imbalance.

Figure 1 shows the memory usage during a typical workday in a typical data center clusters. The data is collected using Ganglia tool [18], which is a scalable monitoring system tool for high-performance computing systems such as clusters and Grids. As can be seen in the figure, the aggregate memory in the cluster reaches 437TB. However, only 69% of this aggregate memory is being utilized (i.e., used + cached + buffed / Total). Despite the fact that the aggregate utilization is far from 100%, there is about 68TB of data residing in swap devices. This demonstrates that some nodes are over utilizing their memories, while others have free memory space potentially for donation. Since memory nodes are physically private to each node, this free memory will not be utilized by default by other nodes in the cluster. *This motivates the need to have a collaborative memory framework to manage aggregate memory resources within the cluster in order to reduce storage device swapping activity and improve performance.*

Furthermore, studies have shown that local memory requirements can vary drastically [15] over time based on multiple factors such as workload variations, orthogonal execution phases, etc. Moreover, thread migration from one node to another (e.g., VMware's vMotion technology [37]), shifts the memory demand from the source node to the destination. Managing the drastic spatial/temporal variation of memory requirement in multi-node clusters is no easy task. It calls for a stable, run-time mechanism to *classify* and continually *reclassify* nodes based on their memory requirements, and to dynamically assign remote memory to achieve optimized performance and energy-efficient memory collaboration.

To address these concerns, we propose a *fully* autonomous memory acquisition protocol that can be implemented over different types of interconnects. The protocol facilitates memory detection, memory exchange, and memory reclaim across nodes within the cluster dynamically at run time. The dynamic control protocol for memory collaboration is, to the best of our knowledge, novel.

4. AUTONOMOUS COLLABORATIVE MEMORY SYSTEM: ARCHITECTURE, PROTOCOL AND ALGORITHM

In this section, we describe the proposed Autonomous Collaborative Memory System (ACMS), including ACMS architecture, protocol and algorithm. We adhere to the following design philosophies while designing our system: low operation overhead, high system stability and QoS guarantees for nodes that donate their memories.

4.1 ACMS Architecture

Figure 2 shows a high level ACMS architecture, which consists of the following components.

1. Interconnect. The interconnect medium used to link cluster nodes with each other. We do not specify strict requirements on the type of the interconnect. Although we conduct our prototype and analysis over Ethernet, the ACMS interconnect could be as well PCIe, Lightpeak (Thunderbolt) [12], Infiniband [14], etc.

2. Collaborating Nodes. These represent individual computing nodes comprising the cluster. The nodes may use remote memory (i.e., memory clients), provide memory for other nodes(i.e., memory servers), or neither (i.e., memory neutrals). (Detailed dissuasion in 4.2)

3. Collaborative Memory Service Manager. The manager, with the proposed protocol and algorithm, is responsible for memory discovery, memory allocation and release. The service manager could be a centralized manager responsible for managing all nodes in the cluster, or distributed across all nodes or a collection of nodes. In this paper, we propose a fully distributed memory acquisition protocol that does not require centralized control. Each node makes its decision of when, and with whom it shall collaborate.

It's worth noting that although we focus on remote memory swapping in this paper, the ACMS protocol and algorithm can also be applied to other remote memory leverage approaches such as remote memory mapping.

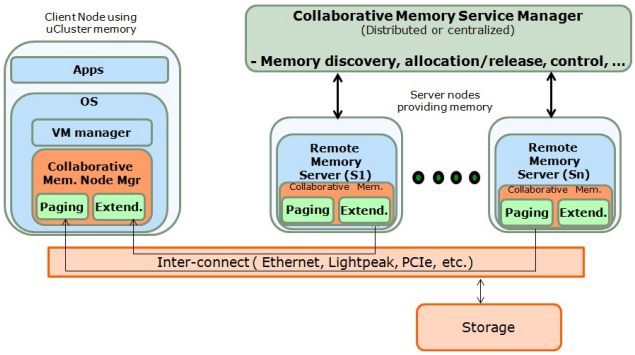

Figure 2: ACMS High-level Architecture

4.2 Node Classification Algorithm

As mentioned in Section 3, static memory collaboration lacks the desired performance with the typical cluster variations. It is important to dynamically discover, allocate and reclaim

remote memory adapting to the nodes condition, to optimize the whole cluster performance and energy efficiency.

To this end, first we classify nodes into three main categories according to their run-time memory usage:

1. A memory client node: a node that is running high demand application and needs extra memory space.
2. A memory server node: a node that possesses significant amount of free memory and can potentially donate part of its memory space to remote memory clients.
3. A memory neutral node: a self satisfied node that has mid-level memory usage that neither offers memory nor needs extra memory.

In general, when the memory usage is smaller than MEM_MIN (indicating very low local memory demand), the node is classified as a memory server; if memory usage is larger than MEM_MAX (indicating very high local memory demand), the node becomes a memory client; on the other hand, if memory usage stays between MEM_MIN and MEM_MAX, the node is a neutral node who is self satisfied. In our classification algorithm, guard bands are applied to both MEM_MIN and MEM_MAX to prevent system oscillation. This attribute is crucial for the stability of the system as it limits nodes oscillation from a memory client to a memory server. Specifically, four thresholds, MEM_MIN_LOW, MEM_MIN_HIGH, MEM_MAX_LOW, MEM_MAX_HIGH are used to decide when to change the node classification. When the memory usage is within the "no change" guard bands, no node class change is asserted, as illustrated in Figure 3. The memory thresholds are programmable parameters that give designers the flexibility of fine tuning based on workloads' characteristics.

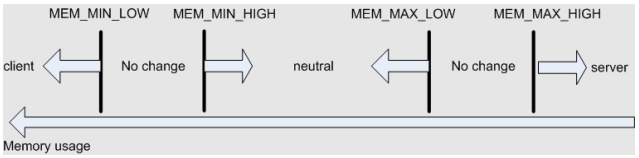

Figure 3: An illustration of the node classification algorithm showing how memory servers and memory clients are classified. Further, it shows the guard bands used to limit node oscillation.

4.3 Dynamic Memory Acquisition Protocol

During run-time, nodes are classified into their corresponding category, and engage in the ACMS using the memory acquisition protocol described in this section. The proposed protocol allows nodes to exchange information about their memory availability, and facilitate dynamic memory collaboration decision in a distributed fashion.

There are five messages defined for the protocol, as described below.

1. OFFERING_MEM: This message is periodically broadcast by a memory server to all other nodes in the system, to indicate its memory availability. The message includes the ID of the memory server, and the amount of available memory. In ACMS, we also monitor the variation of the available memory. If available memory stays relatively stable with little variation, the broadcast frequency is reduced accordingly to reduce the operation overhead without impacting the freshness of the information.
2. REQUESTING_MEM: This message, generated by a memory client, is either broadcast to all the other nodes, or sent out to one or more memory servers, responding to a previous OFFERING_MEM message. In this message, the client indicates that it requests free remote memory. In the case that a memory client has multiple potential memory servers to choose from, the client selects a subset of servers based on certain criteria and arbitration mechanism, for example, First Come First Serve (FCFS) for simplicity, Round Robin (RR) for fairness, Nearest Client First (NCF) for more energy efficient collaboration, etc. One interesting future direction is how to select appropriate memory servers to optimize whole cluster performance and energy efficiency considering node idle/active state.
3. GRANTING_MEM: This message is sent out by a memory server to a given memory client responding to a REQUESTING_MEM message. Note that, this does not bind a memory server with a memory client since the client may get multiple grant messages from multiple servers.
4. ACK_MEM: This message is sent by a memory client to one and only one memory server responding to a GRANTING_MEM message. This message binds a memory client with a memory server. The client may have to do some arbitration to select one of the servers that granted memory. ACK_MEM message indicates the end of a handshaking transaction to bind a certain memory server with a memory client.
5. RECLAIM_MEM: In order to provide guarantees that a memory server does not get hurt by much when it engages in memory collaboration, we give the memory server the opportunity to stop donating its memory when deemed necessary. To achieve that, when the memory server's memory needs change and gets classified as a memory neutral, it sends a reclaim message to the remote client using its memory to reclaim its granted memory. Once the remote client receives this message, it starts migrating its data back from the remote server.

In order to reduce message broadcasting overhead in the system, we monitor the ratio of memory clients to memory servers during run-time, and the responsibility of broadcasting could be offloaded to the group with the smaller number of nodes. For example, in a network environment heavy with memory servers, it is more appealing to let "few" memory clients broadcast their memory needs, instead of letting "many" memory servers broadcast their memory availability, which leads to higher operation overhead.

Depending on who initiates the broadcast message, the memory acquisition process consists of either a 3-way handshake protocol or a 4-way handshake protocol, as illustrated in Figure 4.

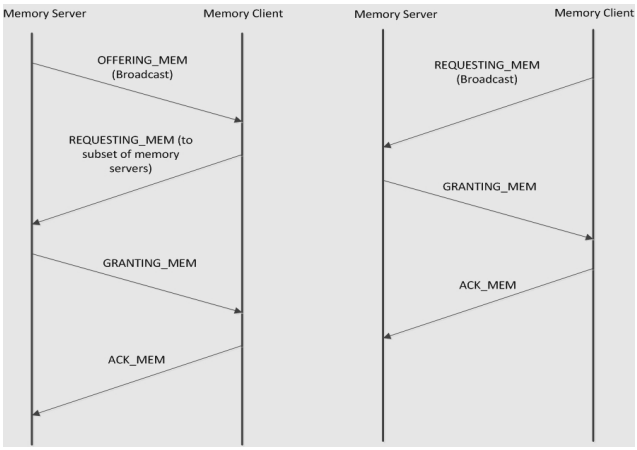

Figure 4: Protocol illustration: (Left) 4-way handshake if server initiates broadcast, (Right) 3-way handshake if client initiates broadcast

Next we will discuss the prototyping for our Autonomous Collaborative Memory System. Although in our exemplary implementation we consider FCFS arbitration scheme to select a remote client/server, one might utilize other schemes to deliver optimal energy efficiency or better fairness. Further optimizations based on topology, real-time network traffic and workloads for energy-efficiency are left as future work.

5. SYSTEM IMPLEMENTATION AND PROTOTYPING

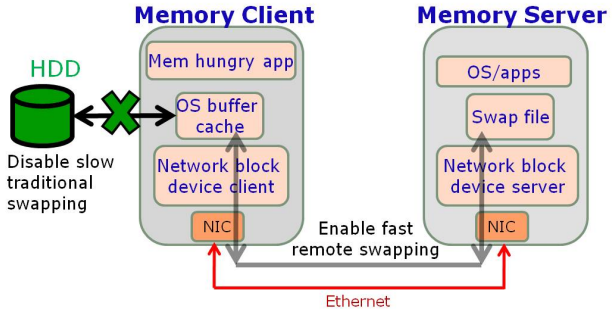

Figure 5: An example of a collaborative memory system consisting of two nodes, a memory client, and a memory server.

In this section, we describe the system implementation of the proposed ACMS to conduct feasibility, benefit, and quantify and evaluate the overhead. For prototyping purposes, we made the following three design choices. (1) We leverage remote memory by applying remote memory swapping (as opposed to remote memory mapping). One main reason, as we also mentioned in Section 2, is that swapping requires less system modification and provide a feasible and rapid implementation approach to study ACMS performance and dynamics. We understand that the remote swapping method suffers certain operation overheads, which we analyze in-depth in Section 6.2.

(2) We choose Ethernet as the interconnect among the computing nodes and use TCP/IP suite as the communication protocol for inter-node communication. However, the ACMS protocol can be also implemented over other types of interconnects and communication protocols, for example, Remote DMA access (RDMA) over infiniband [14], lightpeak [12], and PCIe.

(3) We implement the dynamic ACMS memory detecting, allocation and deallocation protocol as a process running in user space. As a result, no kernel or application modification is required.

A memory server node donates a portion of its memory space to a remote memory client. Once the memory client runs short on its local memory, it disables local, slow hard disk swapping, or assigns it a low priority. At the same time, it enables remote swapping, as shown in Figure 5.

In order to facilitate swapping over Ethernet, we have leveraged several extant features in current operating system kernels. Among them is an external kernel module called Network Block Device (NBD) [21].[1] Once setup over network, NBD allows the local file system at the memory client to access a remote file system at the memory server transparently, hence, adding the ability to swap remotely. Further, the local swap device (i.e., HDD) can be assigned lower priority via swapon/swapoff system calls.

The node classification algorithm, as well as the dynamic memory acquisition protocol (discussed in Section 4.2, Section 4.3), are implemented in user-space threads at each node. This allows each node to dynamically identify runtime memory usage and communicate information with other nodes to accomplish ACMS objectives.

6. SYSTEM EVALUATION AND ANALYSIS

In this section, we evaluate the performance of the proposed ACMS comparing to a traditional system with disk swapping, to a system with static memory collaboration, and to a system with enough local memory. The summary is that ACMS can significantly improve the memory client performance (up to 3x) without perceivable performance impact on memory servers. We also show the protocol dynamics at run time to illustrate the effectiveness of ACMS. Last but not least, we analyze the performance bottlenecks for remote swapping, and highlight insights into software/hardware optimizations to reduce them.

Our experimental setup consists of multiple (up to 5) 2.6GHz Intel®Core™ i7-based machines [13], each one comes with 4GB of RAM, and a 250GB 7200RPM HDD. Machines are running a Fedora 14 [11] OS with a Linux kernel version 2.6.35, and are connected via 1Gbps Ethernet NICs, Further, we are using a network block device version 2.9.23. In order to control the amount of available memory available at the local node to study system behavior under different memory provision and usage conditions, we have developed a memory balloon application that fills up and locks user-specified amount of local memory. To test the system and

[1]An NBD device consists of two components complementing each other, an NBD-client and an NBD-server. In order for our remote paging to operate through an NBD device, each memory server has to create a swap partition in its memory (i.e a regular swap file, or a RAM disk). Then it attaches the swap partition to its NBD-server. At the same time, each memory client establishes a connection between the NBD-client daemon at the memory client side, and the NBD-server at a memory server side.

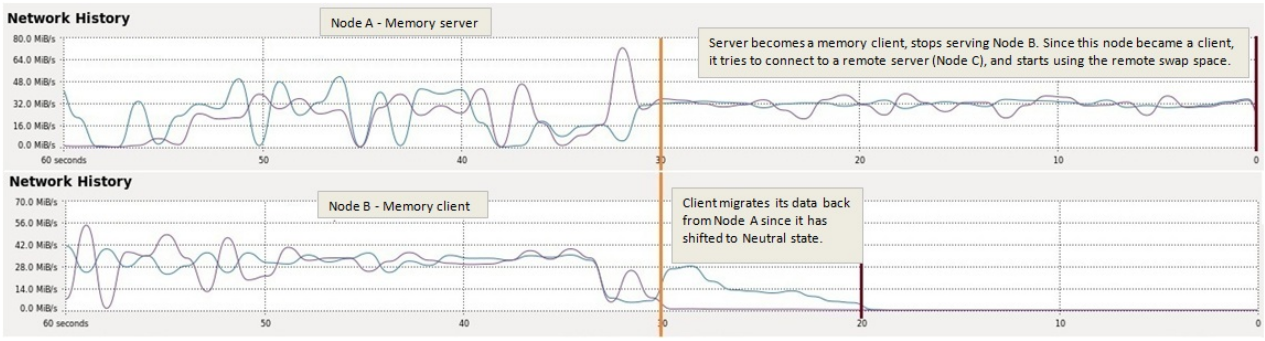

Figure 6: ACMS dynamic memory collaboration activity represented by network traffic monitoring. An injected run-time change at about the 30th second, shifts Node A to become a memory client, and Node B to become a memory Neutral.

study ACMS behavior, we use both microbenchmarks we developed for controlled environment analysis, as well as real-world applications such as SPECCPU2006 [33], TPC-H [36] with PostgreSQL 9.0.4 DataBase, and Hadoop [2].

6.1 Performance Evaluation

Figure 6 shows the autonomous operation of our ACMS dynamics. The figure shows the network traffic at two nodes, a memory server node A (top), and a memory client node B (bottom). The left half of both figures shows the traffic while nodes A and B collaborate with each other (i.e., A is servicing B). At around the 30th second, an application with large memory demand starts on node A, meanwhile memory demand on node B decreases gradually. This injected run-time change causes node A to become a memory client and node B to become a memory neutral. As a result, and as described in Section 4, node A sends out a reclaim message to node B to reclaim its memory back. Once B receives the message, and starts migrating its data back from the remote server's memory which lasts for about 10 seconds.[2] Meanwhile, node A starts collaborating with a third node C (not shown in the figure) with node A acting as a memory client. The traffic at the right portion of the figure shows the swapping activity being sent out to node C. At the same time, node B becomes a neutral that does not collaborate with remote nodes. This visual illustration shows the dynamics and elasticity of the ACMS protocol given the changing memory requirements for running workloads.

Figure 7 shows the application performance (as measured by completion time in seconds) for various TPC-H queries, and for sorting various datastructure sizes using Hadoop workloads. These experiments are conducted using two machines only with one acting as a memory server, and the other acting as a memory client, with the configurations mentioned in Section 6. The legends in the figures represent the completion time while running on a system with *enough* (3GBs free) local memory, a system with *limited* (less than 200MBs free) local memory and swapping to remote memory, and a system with *limited* (less than 200MBs) local memory and swapping to HDD, respectively. As shown in the figures, swapping to remote memory can improve the performance by an average speedup of 1.4x in TPC-H and 3.1x in Hadoop.

[2]The time it takes for a client to migrate its data back depends on the network speed and the amount of data that resides on the remote swap space.

The reason why TPC-H provides less performance improvement compared to Hadoop is that, TPC-H is optimized to operate within the available local memory. Hence, the more the available memory in the system, the more TPC-H makes use of it. Therefore, if the system has limited local memory, TPC-H tries to reduce its memory footprint to reduce disk swapping as much as possible. On the other hand, Hadoop does not pay similar attention to the available local memory, hence, it resorts to swapping more frequently.

The figures also show that running with enough local memory renders much better performance compared to remote swapping, which is expected. Accessing data in the local main memory is faster than accessing data in a remote memory space, due to both the network latency and the swapping overhead (discussed in more detail in section 6.2). We don't show the results of SPECCPU2006 applications due to space limitations. However, for the applications we experimented with, ACMS achieves an average speedup of 3x compared to a non-collaborative memory system.

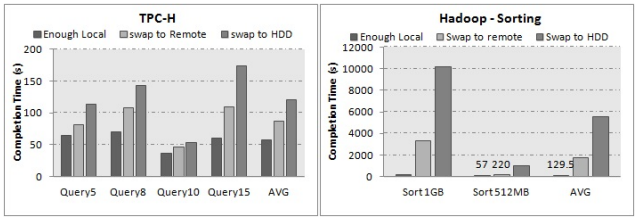

Figure 7: Performance of TPC-H and Hadoop while running with enough local memory, limited local memory/swapping to remote memory, and limited memory/swapping to HDD.

The above shows that remote swapping improves performance for memory clients. However, the performance improvement should not come at the expense of performance degradation for memory servers. Figure 8 shows the completion time for several SPECCPU2006 applications running on memory servers. The results show that the applications' performance degraded very little, confirming the resilience of memory servers to memory collaboration. This robust behavior is a result of the ACMS adaptive design. First, a node is classified as a memory server if it has enough free memory (Section 4.2). Second, if memory needs of a memory server increase beyond the threshold mentioned in Section 4.3, it

sends a reclaim message to the remote client which in turn will migrate its data back and stop using the memory at the server side. Other workloads such as TPC-H, and Hadoop show similar trends to the SPECCPU2006 benchmarks. Due to space limitations, we omitted these figures.

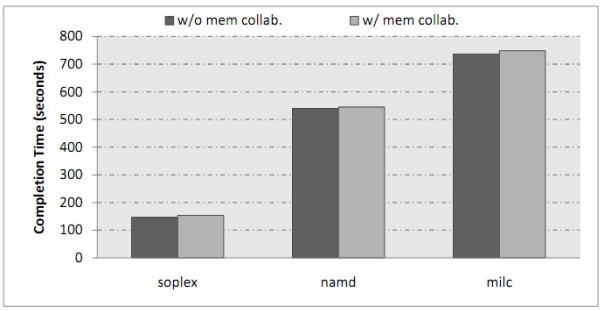

Figure 8: Impact on memory servers while running several SPECCPU2006 applications without servicing a remote client (left bar), and while servicing a remote client (right bar).

6.2 Remote Swapping Overhead Analysis

As shown earlier, remote memory swapping achieves significant speedups compared to traditional disk swapping. However, the performance of remote memory swapping also falls way short compared to running the application entirely on local memory, even with the consideration of interconnect propagation delay, which is a physical limitation. In this section, we investigate the life of the remote swapping and potential sources for overhead. The high level summary of the analysis is that the network stack and kernel swapping handling process are two major sources for the low performance. Based on the understanding, we provide insights on how to reduce the overhead and improve performance.

CPU load consideration. In our prototype, all processing, both the client and server side, is done by the host processor. There are no special hardware accelerators (e.g., remote DMA engine) that handle portions of the processing. However, our system profiling has shown that CPU is idling more than 70%-80% of the time waiting for I/O requests completion. This shows that *CPUs are not overloaded.*

Network bandwidth Consideration. We conducted our experiment using 1Gps Ethernet links between clients and servers. Our network profiling confimed that only about 50% of the network bandwidth is being utilized. In today's data center and cluster system, usually 10G Ethernet links are not uncommon. Other Interconnect, for example, Lightpeak, also has significantly higher physical bandwidth. *Hence, network bandwidth is not a main bottleneck,* at least before other bottlenecks are removed.

Network stack and NBD device overheads. In our prototype, all communications between nodes go through the TCP/IP stack and an NBD device, making them potential major bottleneck. In order to show the impact of network stack and the NBD device, we conducted the following experiment. We created a RAMDisk as a swap device on the local memory itself. When the system runs short on memory, it starts swapping to/from the local RAMDisk. This operation does not involve any TCP overhead or NBD device overhead since the swap device is located locally. Figure 9 shows the completion time for a microbenchmark application while running with enough local memory, limited local memory/swapping to local RAMDisk, limited local memory/swapping to remote machine over network, limited local memory/swapping to local disk. The figure shows two interesting observations.

First, avoiding the network delay, including TCP/IP stack, NBD device operation and propagation delay, can save almost 50% of the overhead (319sec to 160sec). Considering the very small propagation delay (on the order of a few usecs), the network stack proves to be a major bottleneck

Second, even though the RAMDisk is located locally (no network involved), swapping to RAMDisk still performs much worse than running with enough local memory. The reason for that will become clear if we look at the top curved line in the same figure which shows the CPU utilization of the running application.[3] The CPU utilization is 100% when the application entirely runs on local memory, 20% while swapping to RAMDisk, 8% while swapping to remote memory, and less than 1% while swapping to local disk. The bottom curved line represents the CPU utilization while executing user-space code only (i.e., excluding system CPU utilization), which shows that even these modest CPU utilization numbers do not correspond to useful work all the time. Further, it demonstrates that the time spent servicing system calls and other related system code could be a significant fraction.

Kernel Swapping process overheads. There are several reasons why the CPU utilization is low when the application starts swapping, among them are:

1. When a page fault occurs, the OS scheduler assumes that the page fault will take long time to finish, hence, it context switches the process out and adds it to the I/O waiting queue. This adds a fixed overhead to every page fault regardless of how fast it gets serviced.
2. If the memory pressure is very high, the OS blocks the running process until the kernel swap daemon frees some memory. This scenario is known as congestion wait.
3. When the system has to free pages, some clean pages get dropped from the page cache. These clean pages may correspond to the *program code* that is already running. In which case, the OS has to bring them back as the program continues execution.

Therefore, once the system resorts to swapping, regardless of how fast or optimized the swap device is, the system performance degrades significantly due to the inherent limitation in kernel swapping method which is designed for very slow devices such as the HDD.

6.3 Remote Swapping optimizations

We discuss several directions in both software and hardware space to reduce the remote swapping latency and improve

[3]CPU utilization is measured as (CPU time executing user space code (userTime) + CPU time executing system code (systemTime))/Wall clock time.

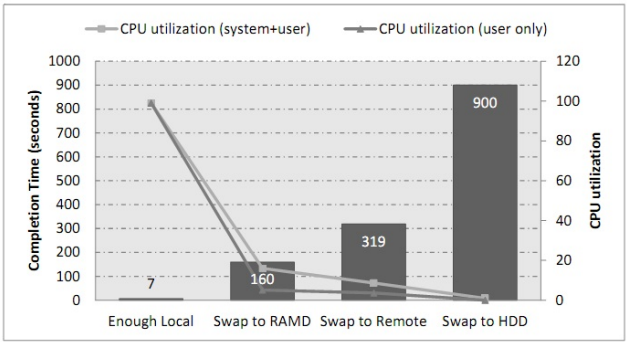

Figure 9: Completion time and CPU utilization for various swapping schemes.

the CPU utilization, thus improving the performance and cluster energy efficiency.

1. Reducing network overhead, including TCP/IP stack and various memory copying, by setting up a direct remote memory access channel between the memory client and the memory server. One existing technology, RDMA over infiniband [29, 14], or iWarp [19, 32] (RDMA over Ethernet) provide this capability. However, it requires special NIC with special interconnect that is not widely available in data center and cluster systems. Our ultimate objective is to investigate direct remote memory access method over general interconnect such as Ethernet.

2. The virtual management attributes can be manipulated based on the swap device characteristics. Such attributes includes: (1) the number of pages to fetch at a time when the swap device is accessed (i.e., prefetching). (2) the time quantum at which the OS starts freeing dirty pages from memory. (3) the threshold at which the OS consider its free memory to be low.

3. Having a hardware accelerator to intercept page fault interrupts and service them in hardware, thus effectively avoiding kernel context switches. The accelerator manages the page table for swapping and establishes a link with a remote memory server to facilitate efficient swapping. Using this approach, the page swapping does not have to be limited on page boundary. Instead, the size of swapped data can be adaptively adjusted based on application requirements. Although this optimization could potentially render the best performance, it also comes with the requirement for hardware modification, as well as limited OS modifications(for page table management).

Due to space limitation, we have only implemented and evaluated one optimization (i.e., prefetching), and applied it to our ACMS system. By default, the Linux kernel fetches 8 pages from the swap device when servicing a page fault, in an attempt to reduce swap device accesses. We modified this default parameter over a logarithmic range from 1 to 1024 pages. For our microbenchmark, we found that performance improves up to 512 pages, beyond which prefetching starts hurting the application. Figure 10 shows the performance of the same microbenchmark application before and after applying the 512-page prefetching optimization. This experiment shows that appropriate prefetching improves this microbenchmark performance by about 25% over unoptimized remote swapping. Note that, since we know the access pattern of the microbenchmark, which is sequential in our case, then we could expect to see benefit by applying the page prefetching technique. However, with more complicated or hard-to-predict access patterns, prefetching may not be a fruitful optimization. Demonstrating the benefit of the rest of the optimizations is left as a future work.

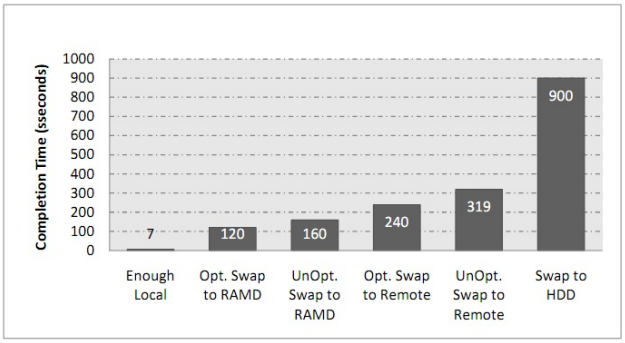

Figure 10: Completion time for various swapping schemes given the page prefetching optimization.

7. CONCLUSIONS

Memory collaboration reduces capacity fragmentation in clustered architectures; it allows nodes that need additional memory space to place their data in remote memories instead of slow storage. Current memory collaboration mechanisms lack the ability to provide autonomous memory collaboration and to adapt dynamically with oscillating memory needs by various applications. To address these issues, in this paper, we have developed an Autonomous Collaborative Memory System (ACMS) that permits dynamic, run time memory collaboration and provides QoS guarantees for memory servers, i.e., nodes that donate their memory for remote use. We have implemented the ACMS and our results show up to 3x performance speedup compared to non-memory-collaborative system. This would allow system to finish job earlier and potentially transit into low power state sooner for energy efficiency. Further, we conduct a detailed analysis to identify several memory collaboration bottlenecks, and provide insights as to how to overcome such bottlenecks to further improve ACMS performance.

We have considered two angles to expand our memory collaboration work to improve its performance and energy efficiency.

1. We are investigating the scalability of our ACMS for large scale clusters, e.g., seamicro's new system with more than 1000 nodes. In such systems, there are many interesting questions to be answered, such as; how nodes should communicate efficiently? how to choose memory severs/memory clients for optimized cluster energy consumption? etc.

2. We are investigating a detailed implementation and evaluation of the overhead optimization approaches we proposed in this paper in an attempt to deliver better

remote memory performance that gets close to performance of the local memory.

8. ACKNOWLEDGMENTS

We would like to thank all anonymous reviewers for their helpful feedback and suggestions. In particular, we would like to extend special thanks to: Feng Chen and Nezih Yigitbasi for facilitating TPC-H and Hadoop workload setups; Phil Cayton, Frank Berry, and Michael Mesnier for helping understanding storage and swapping activities; Arjan Van De Ven for helping understanding the kernel swapping mechanism; Eugene Gorbatov for providing several insights on data center provisioning; James Tsai, Mesut Ergin, Rajesh Sankaran, Sanjay Kumar, and Richard Uhlig for the insightful comments and discussions on memory collaboration; Alexander Min and Sameh Gobriel for helping improving the presentation of the paper.

9. REFERENCES

[1] A. Agarwal. Facebook: Science and the social graph. http://www.infoq.com/presentations/Facebook-Software-Stack, 2009. presented in QCon San Francisco.
[2] Apache. Hadoop. *http://hadoop.apache.org/*, 2011.
[3] M. Awasthi, K. Sudan, R. Balasubramonian, and J. Carter. Dynamic Hardware-Assisted Software-Controlled Page Placement to Manage Capacity Allocation and Sharing within Large Caches. In *HPCA '09: 2009 IEEE 15th Intl. Symp. on High Performance Computer Architecture*, 2009.
[4] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schuepbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *SOSP '09: 22nd ACM symposium on Operating systems principles*, New York, NY, USA, 2009. ACM Press.
[5] B. M. Beckmann, M. R. Marty, and D. A. Wood. ASR: Adaptive Selective Replication for CMP Caches. In *MICRO 39: 39th IEEE/ACM Intl. Symp. on Microarchitecture*, 2006.
[6] J. Chang and G. S. Sohi. Cooperative Caching for Chip Multiprocessors. In *Computer Architecture, 2006. ISCA '06. 33rd Intl. Symp. on*, 2006.
[7] H. Chen, Y. Luo, X. Wang, B. Zhang, Y. Sun, and Z. Wang. A transparent remote paging model for virtual machines, 2008.
[8] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Optimizing Replication, Communication and Capacity Allocation in CMPs,. In *In the 32th ISCA*, June 2005.
[9] I. Corp. Chip shot: Intel outlines low-power micro server strategy, 2011.
[10] G. Dhiman, R. Ayoub, and T. Rosing. PDRAM: a hybrid PRAM and DRAM main memory system. In *46th Design Automation Conf.*, DAC '09, pages 664–469, New York, NY, USA, 2009. ACM.
[11] Fedora Project . Intel. Core. i7-800 Processor Series. *http://fedoraproject.org/*, 2010.
[12] Intel Corp. Thunderbolt Technology. *http://www.intel.com/technology/io/thunderbolt/index.htm*, 2011.
[13] Intel Microarchitecture . Intel. Core. i7-800 Processor Series. *http://download.intel.com/products/processor/corei7/319724.pdf*, 2010.
[14] S. Liang, R. Noronha, and D. Panda. Swapping to remote memory over InfiniBand: An approach using a high performance network block device. In *Cluster Computing, 2005. IEEE Intl.*, pages 1 –10, 2005.
[15] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *36th annual international symposium on Computer architecture*, ISCA '09, pages 267–278, New York, NY, USA, 2009. ACM.
[16] E. Markatos, E. P. Markatos, G. Dramitinos, and G. Dramitinos. Implementation of a reliable remote memory pager. In *In USENIX Technical Conf.*, pages 177–190, 1996.
[17] E. P. Markatos and G. Dramitinos. Adding flexibility to a remote memory pager, 1996.
[18] M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: Design, implementation and experience, 2004.
[19] C. R. R. Maule. iwarp ethernet: key to driving ethernet into high performance environments. In *2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.
[20] H. Midorikawa, M. Kurokawa, R. Himeno, and M. Sato. DLM: A distributed large memory system using remote memory swapping over cluster nodes. In *Cluster Computing, 2008 IEEE Intl. Conf. on*, pages 268 –273, 2008.
[21] Network Block Device TCP version. NBD. *http://nbd.sourceforge.net/*, 2011.
[22] T. Newhall, S. Finney, K. Ganchev, and M. Spiegel. Nswap: A network swapping module for linux clusters, 2003.
[23] J. K. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for ramclouds: Scalable high-performance storage entirely in DRAM. In *SIGOPS OSR*, 2009.
[24] M. Qureshi. Adaptive Spill-Receive for Robust High-Performance Caching in CMPs. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th Intl. Symp. on*, 2009.
[25] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *36th annual international symposium on Computer architecture*, ISCA '09, pages 24–33, New York, NY, USA, 2009. ACM.
[26] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural support for operating system-driven CMP cache management. In *PACT '06: 15th international conference on Parallel architectures and compilation techniques*, 2006.
[27] L. E. Ramos, E. Gorbatov, and R. Bianchini. Page placement in hybrid memory systems. In *international conference on Supercomputing*, ICS '11, pages 85–95, New York, NY, USA, 2011. ACM.
[28] A. Rao. Seamicro technology overview, 2010.
[29] A. Romanow and S. Bailey. An overview of RDMA over IP. In *In 1st Intl. Workshop on Protocols for Fast Long-Distance Networks (PFLDnet*, 2003.
[30] A. Samih, A. Krishna, and Y. Solihin. Understanding the limits of capacity sharing in CMP Private Caches, in CMP-MSI, 2009.
[31] A. Samih, A. Krishna, and Y. Solihin. Evaluating Placement Policies for Managing Capacity Sharing in CMP Architectures with Private Caches. *ACM Trans. on Architecture and Code Optimization (TACO)*, 8(3), 2011.
[32] M. Schlansker, N. Chitlur, E. Oertli, P. M. Stillwell, Jr, L. Rankin, D. Bradford, R. J. Carter, J. Mudigonda, N. Binkert, and N. P. Jouppi. High-performance ethernet-based communications for future multi-core processors. In *2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 37:1–37:12, New York, NY, USA, 2007. ACM.
[33] Standard Performance Evaluation Corporation. *http://www.specbench.org*, 2006.
[34] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm. RapidMRC: Approximating L2 Miss Rate Curves on Commodity Systems for Online Optimizations. *SIGPLAN Not.*, 44(3), 2009.
[35] A. S. Tanenbaum and R. Van Renesse. Distributed operating systems. *ACM Comput. Surv.*, 17:419–470, 1985.
[36] Transaction Processing Performance Council. TPC-H 2.14.2. *http://www.tpc.org/tpch/*, 2011.
[37] vmware . experience game-changing virtual machine mobility. *http://www.vmware.com/products/vmotion/overview.html*, 2011.
[38] N. Wang, X. Liu, J. He, J. Han, L. Zhang, and Z. Xu. Collaborative memory pool in cluster system. In *Parallel Processing, 2007. ICPP 2007. Intl. Conf. on*, page 17, 2007.
[39] M. Zhang and K. Asanovic. Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors. In *ISCA '05: 32nd annual international symposium on Computer Architecture*, 2005.