Evaluating Dynamics and Bottlenecks of Memory Collaboration in Cluster Systems

Ahmad Samih†, Ren Wang‡, Christian Maciocco‡, Tsung-Yuan Charlie Tai†, Ronghui Duan‡, Jiangang Duan‡ and Yan Solihin†

†Dept. of Electrical and Computer Engineering North Carolina State University Raleigh, NC, USA {aasamih, solihin}@ncsu.edu
‡System Architecture Lab Intel Labs Hillsboro, OR, USA {ren.wang, christian.maciocco, charlie.tai}@intel.com
‡Software and Services Group Intel Corp. Shanghai, China {ronghui.duan, jiangang.duan}@intel.com

Abstract—With the fast development of highly-integrated distributed systems (cluster systems), designers face interesting memory hierarchy design choices while attempting to avoid the notorious disk swapping. Swapping to the free remote memory through Memory Collaboration has demonstrated its cost-effectiveness compared to overprovisioning the cluster for peak load requirements. Recent memory collaboration studies propose several ways on accessing the under-utilized remote memory in static system configurations, without detailed exploration of the dynamic memory collaboration. Dynamic collaboration is an important aspect given the run-time memory usage fluctuations in clustered systems. Further, as the interest in memory collaboration grows, it is crucial to understand the existing performance bottlenecks, overheads, and potential optimization.

In this paper we address these two issues. First, we propose an Autonomous Collaborative Memory System (ACMS) that manages memory resources dynamically at run time to optimize performance. 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. Second, we analyze, in depth, the end-to-end memory collaboration overhead and pinpoint the corresponding bottlenecks.

I. INTRODUCTION

With every new software generation, applications’ memory footprints grow exponentially in two dimensions—horizontally due to an increase in their pure 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) [13]. In cases where the system runs short on local memory, the OS resorts to swapping devices, e.g., Hard Disk Drives (HDDs) and Solid State Drives (SSDs). Swapping devices such as HDDs or even SSDs operate at several orders of magnitude slower than main memory modules [22]. 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, and highly-integrated distributed systems such as clustered systems (e.g., Seamicro’s SM1000-64HD[24], Intel’s microServer [6]). 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 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 [22], [23], or even adding a dedicated storage servers that stores all data on their main memory (RAMs); namely the RAMCloud [19]. While over provisioning the system to accommodate memory needs solves the problem, it, unfortunately, 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 grouped into two main categories: remote memory swapping [12], [18], [14], [34], and remote memory mapping [16], [34].

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. Further, some remote memory mapping techniques such as [16] require user intervention to explicitly define the availability of memory space at remote nodes.

In this paper, we focus on the other approach, remote memory swapping, which considers remote memory as swap space. Approaches that fall in this category have demonstrated the ability to be deployed transparently with little/no modification to 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. However, these proposals 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 for the whole cluster. Furthermore, as remote memory swapping becomes an appealing approach, it is critically important to understand the performance bottlenecks that exist in current OSes and how such bottlenecks could be potentially mitigated or even removed.
In this paper, we address these two concerns and make the following major 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 local memory requirements. We demonstrate the feasibility and benefit of the proposed ACMS by developing a prototype and evaluating real-world applications. Our prototype 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.

2) We pinpoint and analyze the major performance bottlenecks and overheads during the lifetime of the remote memory swapping approach. We further study several optimizations to improve remote memory swapping performance. Our investigation shows that network stack, and Linux kernel swapping process are the major bottlenecks.

II. RELATED WORK

There is a rich body of work that has studied the problem of managing capacity at different levels of the memory hierarchy [25], [26], [27], [4], [20]. 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 [22], [23], or add additional resources to the cluster in order to avoid prohibitive disk activity. In particular, a recent proposal: the RAMCloud [19], 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.

Lim et al., [13] avoid to over-provision 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) [31], 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 [3] issues due to their shared memory architecture. Further, as reported in [3], 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 [12], [18], [14], [34], [5], 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. To address this concern, we design a run-time mechanism to manage the memory resources across collaborating nodes within the cluster and we further provide QoS measures for individual nodes.

III. MOTIVATION FOR DYNAMIC MEMORY COLLABORATION

In Section I, 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 [24], [6], 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 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, local storage device.

![Logical Grid Memory last day](image-url)
IV. 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.

A. 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) [8], Infiniband [12], 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 discussion in Section IV-B)

3) Collaborative Memory Service Manager: The manager, with the proposed protocol and algorithm, is responsible for memory discovery, memory allocation and memory 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.

B. Node Classification Algorithm

As mentioned in Section III, 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 a 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 that 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 its 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 measured based on empirical evaluations taking into consideration the system memory and the expected running workloads behavior and their stability. Hence, this gives designers the flexibility of fine tuning their systems.

C. 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 primitives 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 the available memory is relatively stable, 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.\(^1\)

\(^1\)In our implementation we consider FCFS arbitration scheme. However, optimizations based on other arbitration schemes, topology, or real-time network traffic are left as future work.
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 as 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 to reclaim the granted memory. Once the remote client receives this message, it starts migrating its data back from the remote server. Note that, the time it takes for a client to migrate its data back depends on the network speed (e.g., 1Gbps, 10Gbps, etc.) and the amount of data that resides on the remote swap space.

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](image_url)

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

**D. Discussion**

**Memory Usage Monitoring.** Memory usage can be monitored by either software or hardware approaches. In our prototype (described in next Section), we use OS counters to monitor the memory usage such as MemTotal, MemFree, Buffers, Cached, etc. Further, for hardware-based memory collaboration, memory monitoring could be done via hardware techniques such as Memory Monitoring Circuit (MMON) [21]. MMON uses the classical stack distance histogram (SDH) analysis [30] to estimate the memory requirement (i.e., page miss rate) at run time. Discussing hardware-based memory collaboration is outside the scope of this paper.

**Memory Collaboration Scalability.** We discussed the broadcast-based memory acquisition protocol, which we implement and evaluate in a small-scale cluster prototype (5 nodes) in the coming sections. However, when the scale of the system grows to tens, hundreds, or even thousands of nodes [10], the scalability characteristics must be taken into consideration.

In this subsection, we discuss how the proposed protocol scales with larger cluster systems. We propose a Zone-Based Broadcast Protocol (ZBBP), which limits each node to only broadcast the messages to its n-hop neighbors. The broadcast and memory sharing scope is limited due to two main purposes: First, it reduces the broadcast overhead. A node can only broadcast to its n-hop neighbors instead of the whole network. Hence, the overhead of processing the broadcast messages and the burden on the network are greatly reduced. Due to this, the overall overhead of collaborative memory is reduced as well.

Second, it improves distance locality. Forcing a node to only share memory with its n-hop neighbors instead of sharing memory with nodes located far away is important for both performance and energy considerations. Accessing close-by nodes incurs less latency, both due to smaller number of hops and also less chance to encounter congestion.

The ZBBP operates as follows. When a node broadcasts its memory availability to its n-hop neighboring nodes, it adds the maximum hop count as a parameter to the message. The maximum hop count is initially set to be the radius of the zone. For example, a maximum hop count is set to 3 if a node is allowed to only share memory with other nodes at most 3 hops away. The nodes who receive the message will continue to process the message according to the discussion in Section IV-C, however, with a slight modification. The node will extract the hop count from the message, if the hop count is greater than zero, it decreases the hop count by one and forward the message to all its neighbors except the one from whom it received the message. If the hop count is zero, the node processes the message without forwarding it any further. If a node receives the same message for the second time, it will discard the message without processing/forwarding it to reduce broadcast overhead. Additional methods, such as, Smart Gossip [11] can be applied to further reduce broadcast overhead. However, discussing these works is outside the scope of this paper.

Next we will discuss the prototyping for our Autonomous Collaborative Memory System.

**V. SYSTEM IMPLEMENTATION AND PROTOTYPING**

In this section, we describe the system implementation of the proposed ACMS to conduct feasibility and benefit, and to quantify and evaluate the overhead of remote memory paging. For prototyping purposes, we make 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 II, is that swapping requires less system modification and provides a feasible and rapid implementation approach to study and analyze ACMS performance and bottlenecks VII.

(2) We choose Ethernet as the interconnect medium among the computing nodes and use TCP/IP suite as the communication protocol for inter-node communication. However, our protocols can be also implemented over other types of interconnects and communication.
protocols, for example, Remote DMA access (RDMA) over infiniband [12], lightpeak [8], and PCIe.

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

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) [17]. 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 a lower priority via swapon/swapoff system calls.

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

VI. SYSTEM PERFORMANCE EVALUATION

In this section, we evaluate the performance of the proposed ACMS comparing to a traditional system with Hard Disk Drive (HDD) 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 compared to HDD swapping.

Our experimental setup consists of multiple (up to 5) 2.6GHz Intel®Core™ i7-based machines [9], each having 4GB of RAM, and a 250GB, 7200RPM HDD. Machines are running a Fedora 14 [7] 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 that inflates and locks a user-specified amount of local memory. To test and analyze the system behavior, we use both microbenchmarks we developed for controlled environment analysis, as well as real-world applications such as SPEC CPU2006 [29], TPC-H [32] with PostgreSQL 9.0.4 DataBase, and Apache Hadoop [2].

Figure 5 shows the autonomous operation of our ACMS dynamics. The figure shows the network traffic, send (purble curve) and receive (blue curve), 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 run-time change causes A to become a memory client, and B a memory neutral. As a result, and as described in Section IV, 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 which lasts for about 10 seconds. Meanwhile, A starts collaborating with a third node C (not shown in the figure) with 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 6 shows the application performance (as measured by completion time in seconds) for various TPC-H queries, and for sorting various data structure 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 VI. The legends in the figures represent the completion time while running on a system with enough (4GBs 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.

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. If the system has limited local memory, TPC-H is optimized to reduce its memory footprint in order to limit disk swapping. On the other hand, Hadoop does not pay similar attention to the available local memory, thus resorting to swapping more frequently. This shows that having an efficient swap device...
could potentially reduce the programming effort needed to optimize workloads such as TPC-H.

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 VII).

Thus far, we have shown that remote swapping improves performance for memory clients. However, this performance improvement should not come at the expense of performance degradation for memory servers. Figure 7 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. Other workloads such as TPC-H, and Hadoop show similar trends to the SPECCPU2006 benchmarks. Due to space limitations, we omitted these figures.

Fig. 7. Impact on memory servers while running several SPECCPU2006 applications

We omit the results of SPECCPU2006 applications due to space limitations, since similar trends were observed. In summary, ACMS achieves an average speedup of 3x compared to a non-collaborative memory system, while falling short behind running with enough local memory.

VII. 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 timeline of remote swapping and potential overhead sources. The high level summary of the analysis is that the network stack and kernel swapping handling process are two major sources of low performance.

CPU load consideration. In our prototype, all processing, both on 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 for memory bound applications has shown that CPU is idling more than 70%-80% of the time waiting for I/O requests to complete. This shows that CPUs are not overloaded.

Network bandwidth Consideration. We conducted our experiments over 1Gbps Ethernet links between clients and servers. Our network monitoring tools confirmed that only about 50% of the network bandwidth is being utilized. In today’s data center and cluster system, usually 10Gbps Ethernet links are not uncommon. Other interconnects such as Lightpeak [8] has significantly higher physical bandwidth. Hence, network bandwidth is not a main bottleneck, at least before other bottlenecks are removed.

Network stack and swapping overhead. In our prototype, all communications between nodes go through the TCP/IP stack and an NBD device, making them potential major bottlenecks. In order to show the impact of network stack and the NBD device, we conducted the following experiment. We created a RAMDisk as a local 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 8 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.

Fig. 8. Completion time and CPU utilization for various swapping schemes.

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 micro-seconds), 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\(^2\). The CPU utilization is 100% when the application runs entirely 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 those modest CPU utilization numbers do not correspond to useful work all the time. Thus, kernel swapping proves to be another major bottleneck.

Next, we are going to discuss the network and kernel swapping overhead in details.

A. Network Overhead Analysis

In this subsection, we investigate the overhead induced by accessing remote memory through the network over TCP/IP stack. We provide an overview on the life cycle of bringing a page from remote memory into local memory in order to understand the cost of network related operations.

\(^2\)CPU utilization is measured as (CPU time executing user space code (userTime) + CPU time executing system code (systemTime))/Wall clock time.
Autonomous Collaborative Memory System (ACMS) that permits memory collaboration is necessary.

We summarize these issues into the following.

1) When a page fault occurs, an exception is raised followed by a CPU pipeline flush in order to service the page fault routine. Pipelining is a powerful technique to hide the long memory latency. Flushing the pipeline frequently reduces the effectiveness of latency hiding, hence rendering a low CPU utilization. Further, executing the page fault routine pollutes the data and instruction caches, TLBs, and key processor structures. Prior work [28] shows that if for example, SPEC-JBB throws an exception once every 1k instruction, its performance could be degraded by up to 48% due to the aforementioned reasons.

2) When a page fault occurs, the OS scheduler assumes that the page fault is going to 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.

3) If the memory pressure is very high, the OS blocks the running process until the kernel swap daemon (AKA kswapd) frees some memory. This scenario is known as congestion wait. Our kernel probing and profiling of the kswapd indicates that the function get_swap_page - which is used to find a potential contiguous space in the swap out device to allocate swapped-out pages, consumes more than 45% of the system CPU clock cycles, and more than 74% of the retired instructions.

4) Under high memory pressure, the kernel performs heavy page scanning to figure out which page is next to be replaced (or swapped).

5) 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 (remote memory, etc.), 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.

VIII. Conclusions and Future Work

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. Further, in order to optimize the performance of memory collaboration, detailed understanding of the major performance bottlenecks in the end-to-end memory collaboration is necessary.

To address these issues, in this paper, we have developed an Autonomous Collaborative Memory System (ACMS) that permits dynamic, run-time memory collaboration across nodes. We have implemented a prototype realizing our proposed ACMS and our results show up to 3x performance speedup compared to a non-collaborative memory system. In addition to improving the nodes performance significantly, ACMS also has safeguards to ensure that nodes whose memories are accepting pages from other nodes are not degraded by much. Further, we conduct a detailed end-to-end analysis to identify several memory collaboration bottlenecks. Our investigation shows that network stack and kernel swapper are the major performance bottlenecks.

We have considered two angles to expand our memory collaboration work to improve its performance and energy efficiency.
1) We are investigating a detailed implementation and evaluation of several overhead optimization approaches in an attempt to deliver better remote memory performance that gets close to performance of the local memory. For example, we will evaluate the usefulness of RDMA in terms of eliminating TCP/IP overhead in a memory collaboration environment.

2) We are investigating the scalability of our ACSM for large scale clusters, e.g., scemicro’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.

IX. 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 aspects of swapping activities; Arjan Van De Ven for helping understanding aspects of the kernel swapping mechanism; Eugene Gorbatov for providing several insights on data center provisioning; James Tsai, Mesut Ergin, and Sameh Gobriel for helping improving the presentation of the paper; Alexander Minse and Michael Mesnier for helping improving the presentation of the paper. Special thanks to: Feng Chen and Nezih Yigitbasi for facilitating feedback and suggestions. In particular, we would like to extend special thanks to: Feng Chen and Nezih Yigitbasi for facilitating feedback and suggestions.

REFERENCES