A Case for Tracking and Exploiting Inter-node and Intra-node Memory Content Sharing in Virtualized Large-Scale Parallel Systems

Lei Xia   Peter Dinda

Department of Electrical Engineering and Computer Science
Northwestern University
{lxia,pdinda}@northwestern.edu

ABSTRACT

In virtualized large-scale parallel systems scientific workloads consist of numerous processes running across many virtual nodes. Their memory footprint is massive, and this has consequences for services that enhance performance, reliability, or power. We argue that a service that dynamically tracks the sharing of memory content, both within individual nodes, *and across nodes*, can simplify and enhance the implementation of such services. For example, leveraging content sharing could significantly reduce the size of a checkpoint of a group of nodes. As another example, it could speed VM migration by allowing the reconstruction of a VM's memory from multiple source VMs. Finally, a service that improves reliability by introducing memory redundancy could leverage existing content sharing to minimize the memory costs of any particular level of redundancy. We argue that both intra- and inter-node memory content sharing is common in parallel applications, supporting this claim by a detailed study of both kinds of sharing, at different scales, different granularities, and different times for a range of applications and application benchmarks. We then describe the high level approach we are taking to design and implement a distributed, VMM-based system that can efficiently and scalably identify and track such sharing with low overhead.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design

General Terms

High Performance Computing, Deduplication

Keywords

Content Sharing, Virtualization

This project is made possible by support from the United States National Science Foundation (NSF) via grant CNS-0709168 and the Department of Energy (DOE) via grant DE-SC0005343.

VTDC'12, June 18, 2012, Delft, The Netherlands.

1. INTRODUCTION

Virtualization has the potential to dramatically increase the usability and reliability of high performance computing (HPC) systems by maximizing system flexibility and utility to a wide range of users [16, 21, 18]. We argue here that the virtualization infrastructure of a large-scale HPC system should include a facility that continuously tracks memory content sharing across the machine. To support this claim, we consider the *impact* that such a facility would have, the *opportunity* that it could unveil, and the *feasibility* of creating it.

The impact of a memory content sharing tracking system would accrue from the services that it would enable or simplify. Our thinking in this regard, and the drives behind our design and our studies of sharing is driven by three examples: migration, checkpointing, and redundancy, on which we elaborate in Section 2. If our proposed facility existed, and significant memory content sharing existed at run time, each of these services could be transformed.

The core of this paper focuses on the question of opportunity: does significant inter-node and intra-node content sharing actually exist in parallel workloads? In Section 3, we describe a detailed experimental study of memory content sharing we have conducted on a range of parallel applications and application benchmarks. The study considers both forms of sharing, at different scales, and across time. We believe that the results support our case; both intra- and inter-node memory content sharing are common in parallel applications. Hence, there is significant opportunity.

It is important to point that we focus on inter-node sharing, which is the core contribution of our work. As we elaborate on in Section 6, there is considerable previous work on deduplicating memory content sharing within individual nodes (e.g., [28, 10, 11]) but little work that considers the opportunities of tracking and leveraging inter-node memory content sharing, and none that we are aware of does so in a parallel computing context. As we will demonstrate in the paper, there is substantial additional sharing across nodes beyond that of sharing within individual nodes, and capturing this sharing drives our proposed design.

We next consider the feasibility of building a facility that tracks memory content sharing across an HPC system with minimal performance impact. Based on our driving services and our study, we have begun the design and implementation of such a facility. We discuss the many challenges in Section 4, and then describe the envisioned interface of our system, its overall architecture, and our approach to building it in Section 5. Our discussion includes promising initial performance results for parts of the proposed system.

2. DRIVING SERVICES

We now describe how three services could leverage a facility that tracked intra-node and inter-node memory content sharing in an HPC system.

2.1 VM migration

VM migration [12, 26], our first driving service, is a useful feature provided by most virtualization systems. This capability is being increasingly employed in today's HPC systems to help provide fault tolerance [21]. Current VM migration services are mostly focused on migrating single virtual machine across hosts. A facility that tracked content sharing could speed single VM migrations by allowing the reconstruction of the VM's memory from multiple source VMs. Furthermore, there are many cases in which migrating a *set* of VMs that runs a parallel application has been found useful [23]. By leveraging intra-node and inter-node memory content sharing, each distinct memory page in the group of migrating VMs could be copied only once during migration, and each destination VM could reconstruct its memory from multiple source VMs to make its migration process faster. The total amount of data to be transfered could also be reduced.

2.2 Checkpointing

Harnessing peta- and exascale computational power presents a challenge as system reliability deteriorates with scale. Our second driving service, checkpointing [8, 20] with rollback, is a well-known technique for fault-tolerance in which the application save it state in stable storage, usually in a parallel file system (PFS), and rolls back in the event of a node failure.

Checkpointing results in high overheads due to often simultaneous writes of all nodes to the PFS, which reduces the productivity of such systems. For example, when a large parallel application is checkpointed, tens of thousands of nodes may write their memory content to the PFS, producing many terabytes of data. Furthermore, the I/O bandwidth of HPC systems generally does not increase at the same rate that computational capabilities and physical memory do. And the larger the system is, the more frequently checkpoints may be needed due to the lower mean time to failure. Even assuming storage is cheap, ever larger checkpoints at every higher rates will lead to an I/O bottleneck. This appears to be already occurring on petascale systems, and will certainly occur in exascale systems.

A facility that dynamically tracked the sharing of memory content could address this checkpointing problem. First, leveraging sharing would allow us to significantly reduce the size of a checkpoint by saving only a single copy of each distinct memory page. Second, it would reduce the total time and the I/O bandwidth needed to transfer and store the checkpoints.

2.3 Redundant computation

Our third driving service would improve reliability through redundancy. Redundant computation and process replication [22, 14] are employed to enhance the availability and reliability of high performance computing and mission critical systems. In these systems, a process's state is replicated and stored in its partner nodes. If the process fails, these available replicas can recover and assume the original process's role quickly. Process replication offers a different set of tradeoffs compared to rollback recovery techniques. It completely masks a large percentage of system faults, preventing them from causing application failures without the need for rollback. However, process replication can be costly in terms of the large amount of extra memory that is needed, which is a large budgetary and power consumption item. By leveraging memory content sharing, there is a potential to reduce these costs by avoiding explicitly creating memory page replicas when memory pages with the same content already exist elsewhere. That is, we can potentially use the applications' own redundancy instead of making more.

3. EXPERIMENTAL STUDY

The goal of our experimental study is to determine how much memory content sharing is likely to occur in practice and to characterize where and when it does occur. The study focuses on a set of parallel applications and benchmarks. We describe our methodology, the benchmarks, and our results.

The key observation from our experimental study is that intra- and inter-node memory content sharing is common in parallel applications. This suggests that there is opportunity for exploiting this memory content sharing to benefit many services in HPC systems as we have discussed before.

3.1 Methodology

We have investigated the memory content sharing in scientific applications by running a set of applications and benchmarks. For each application or benchmark, a number of processes are initiated across physical nodes by MPI, with one process running on each physical node. During the execution of the application, all processes are periodically[1] paused synchronously and their memory contents are dumped. An MD5 hash value is generated to describe the content of each dumped memory block. A hash table is thus generated for each process to hold all these different hash values. Hash collisions indicate intra-node sharing. The hash tables are then compared to find inter-node sharing.

In our study, we are interested in both intra-node memory content sharing and inter-node memory content sharing. We use intra-node and inter-node *distinct memory block ratio* to represent the level of memory content sharing. The metrics we use are defined here:

- *Total memory blocks* shows the total number of memory blocks of all processes.
- *Intra-node distinct blocks* gives the summation of the number of distinct memory blocks in each process.
- *Inter-node distinct blocks* gives the number of distinct memory blocks across all processes.
- *Intra-node distinct ratio* represents the ratio of *intra-node distinct blocks* over *total memory blocks*, The smaller of this ratio, the more intra-node memory content sharing. This ratio is labeled as *intra%* in the following graphs.
- *Inter-node distinct ratio* represents the level of both intra-node and inter-node content sharing, which is the ratio of *inter-node distinct blocks* over *total memory blocks*. The smaller of this ratio, the more memory content sharing exists across all nodes. This ratio is labeled as *inter%* in the graphs.

We report mainly the *intra-node* and *inter-node* distinct ratios of the tested parallel applications and benchmarks here. All the results are averaged from these numbers that are got periodically during the entire application execution. The memory block size we use is a single x86 page (4096 bytes), unless otherwise specified.

We performed our tests on a cluster with 12 nodes. Each node is equipped with two Dual Core Intel(R) Xeon(TM) 2.00GHz CPU, 1.5GBytes RAM and 32GB disk, a Broadcom NetXtreme BCM5703X

[1] At 0.5 Hz in this work.

Benchmark	Memory (MB)	Benchmark	Memory (MB)
bt.C.9	4364	cg.C.8	1292
ep.C.8	245	is.C.8	2377
lu.C.8	864	mg.C.8	3567
sp.C.9	2734	Moldy.8	5324
pHPCCG.8	3986	HPCCG.8	3987
miniFE.x.8	4350	miniMD.8	6313
Lammps.4	4265	HPCC.8	2778

Figure 1: Memory footprint size of tested applications. This shows that most of tested applications are memory-intensive.

1Gbps Ethernet card. The nodes are connected through a 1Gbps Ethernet switch.

Note that our measurement study is done using only the application portion of the address space. If we also considered the kernel memory contents, there will be more memory content sharing both inside and across nodes. Our measurement *underestimates* the amount of both kinds of memory content sharing that is available. That is, if we considered kernel memory, the opportunity would grow.

3.2 Benchmarks

We have run a set of applications and application benchmarks that are designed to run on large parallel systems to study their memory content sharing. These applications include:

- Moldy [5] is a general-purpose molecular dynamics simulation program. It is sufficiently flexible that it ought to be useful for a wide range of simulation calculations of atomic, ionic and molecular systems. We used MPI-based Moldy version in the experimental study.
- The NAS Parallel benchmarks (NPB) [6] is a set of benchmarks targeting performance evaluation of highly parallel supercomputers. In our test, NPB 2.4 version is used, which is MPI-based. 7 benchmarks from NPB are tested, which are BT, EP, LU, SP, CG, IS and MG.
- The HPC Challenge benchmarks (HPCC) [1] is a set of benchmarks targeting to test multiple attributes that can contribute substantially to the real-world performance of HPC systems. Version 1.4.1 is used in the test.
- The Sandia Miniapps [4], part of Sandia National Labs' Mantevo Project, is a set of small, self-contained programs that embody essential performance characteristics of key applications. We used miniFE, HPCCG, pHPCCG and miniMD in our test.
- Lammps [2] is a molecular dynamics simulator and an acronym for Large-scale Atomic/Molecular Massively Parallel Simulator, which is also distributed by Sandia National Laboratories.

3.3 Results

Most of the tested application benchmarks have significant memory footprints, as can be seen from Figure 1.

The intra-node and inter-node distinct ratio for all of the applications and benchmarks are shown in Figures 2 (where inter-node sharing is dominant) and 3 (where intra-node sharing is dominant). Each application and benchmark is run with at least two different problem sizes, on different numbers of nodes, where both are encoded in the name. For example, *bt.C.9* means the bt benchmark with problem size C on 9 nodes.

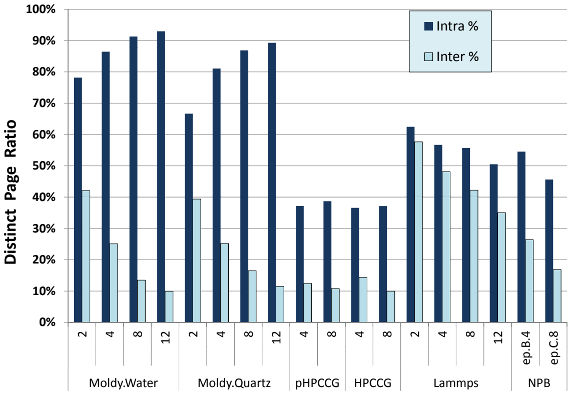


Figure 2: Parallel applications that have more inter-node sharing but less intra-node sharing.

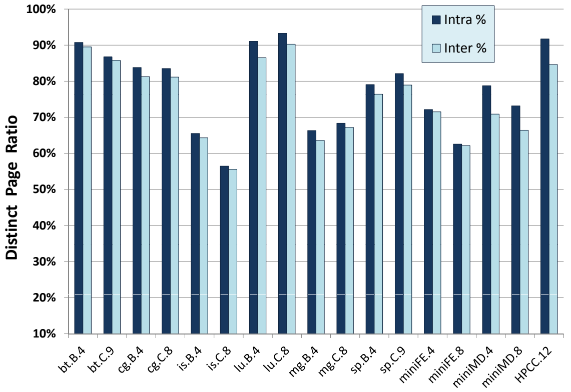


Figure 3: Parallel applications which have more intra-node sharing but less inter-node sharing.

The most important observation on Figures 2 and 3 is that memory content sharing of some form is common. This is the opportunity that we seek to take advantage of. Some VMM systems already employ memory-deduplication techniques to reduce memory pressure on a single node. We were hopeful that further deduplication is possible across nodes, and we have found the evidence to support that.

Figure 2 shows the applications that have a significant amount of inter-node memory content sharing. For example, Moldy can have up to a 10% inter-node distinct ratio, comparing to only a 78% intra-node distinct ratio. Also note that its inter-node distinct ratio keeps decreasing as the problem size and number of nodes increases.

If significant inter-node sharing exists, it could potentially be used to reduce the actual memory footprint by deduplicating pages based on this sharing. Figure 4 illustrates the potential memory footprint reductions for the Moldy application as a function of the problem size. The last column in the figure shows the size of memory the system could saved if all inter-node duplicated memory contents are removed. We can see for this application, (a) cap-

Problem Size	Number of Nodes	Total	Intra-Distinct	Inter-Distinct	Reduction (Intra)	Reduction (Inter)
128x128x256	2	29 MB	19 MB	11 MB	10 MB (34%)	18 MB (62%)
256x256x256	4	161 MB	131 MB	41 MB	30 MB (19%)	121 MB (75%)
512x512x256	6	489 MB	417 MB	91 MB	72 MB (15%)	398 MB (81%)
1024x1024x256	8	1337 MB	1161 MB	220 MB	176 MB (13%)	1116 MB (83%)
2048x2048x256	10	3057 MB	2706 MB	426 MB	351 MB (11%)	2631 MB (86%)
4096x4096x256	12	5324 MB	4753 MB	612 MB	571 MB (11%)	4713 MB (89%)

Figure 4: Memory that could potentially be reduced when memory contents are deduplicated using intra- and inter-node content sharing in the Moldy application (with quartz as input).

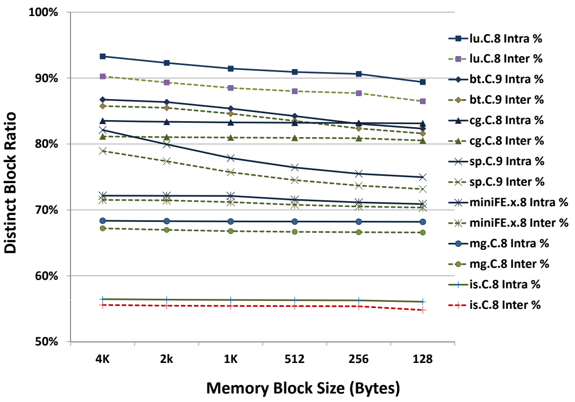


Figure 5: Memory content sharing using various memory block sizes.

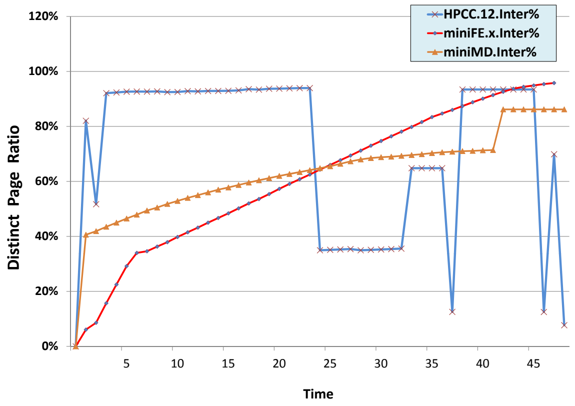


Figure 6: Applications which have different level of inter-node content sharing over the execution time.

turing inter-node sharing is critical, and (b) the size of saved space scales well with problem size. This behavior also occurs for the benchmarks of Figure 2.

Figure 3 shows applications that have some degree of intra-node content sharing but only very little inter-node sharing. A natural question to ask is whether the latter is due to the block size in use. Will a finer granularity expose more sharing? Figure 5 shows results in which the block size was varied down to 128 bytes. At least down to this level, granularity has little effect. This is unfortunate, but there is also a bright side to these results (and the results for the benchmarks that do have significant inter-node sharing): they suggest that page granularity is sufficient. This is important because we expect to implement our system in a virtualization context, where the page granularity is readily accessible, including with hardware assistance, while sub-page granularity is much more challenging to handle.

For some benchmarks, such as HPCC, miniFE and miniMD, the memory tracing results show that on average there are some but not much memory content sharing. However, if we look at sharing over time during execution, we find that there are phases in which the inter-node memory content sharing is considerable, as can be seen in Figure 6. This suggests that a facility for tracking memory content sharing must act dynamically.

4. CHALLENGES AND ASSUMPTIONS

The preceding sections have shown the opportunity and potential impact of a facility for detecting and tracking memory content sharing. We now consider the challenges that creating such a facility would face, and our assumptions.

4.1 Challenges

The detection system must be designed to fit into large-scale parallel systems. Scalability is one of the critical features. It should be able to scale from small clusters with hundreds of nodes to large-scale parallel systems with hundreds of thousands of nodes. The system should work effectively and maintain low overheads as the system size grows. A centralized model could not only prevent the system from being scalable but also present a single point of failure. Thus, there should be no central control node in the system which collects all content sharing status is allowed. Instead, the nodes must collectively form the system without any central coordination, while each node could has only partial knowledge of the entire system.

The system should be an online system that reports the current, or near-current memory content sharing in the system as the application changes its memory contents during its execution. However, some applications may have a very high memory update rate, and these applications could produce a huge burden on the system by requiring a very high update rate.

4.2 Assumptions

We assume that each node in the system fails independently, which implies that replicating computation and data can provide fault tolerance. We further assume that the network links between nodes have high throughput and low latency, which is very common for cluster network or the interconnect between nodes in supercomputers, typically these local area networks that can achieve 1Gbs to 10Gbps throughput with 10 to 100 μs latency. Also, the system should support efficient collective communication. We expect that the network's bisection bandwidth will scale as the system size increases and that its diameter will scale sublinearly as the system size increases. One example of such a network might be a mesh network such as is common in modern parallel machines. Overall, our assumptions boil down to the underlying system being a parallel system, not a distributed system. We expect it to be a carefully, purpose-built controlled environment.

These assumptions help with the design of the detection system. For example, the low latency and high bandwidth scalable interconnect means that the synchronization between all DHT nodes is not prohibitively expensive. Independent failure means we can an rely on data being available in more than one failure boundary (i.e., the physical memory of more than one node) while designing the recovery protocols.

5. TOWARDS A SYSTEM FOR MEMORY CONTENT SHARING DETECTION

We propose an online memory content sharing detection system that is able to track sharing and identify the specific shared content blocks across virtual nodes in large-scale parallel systems with minimal performance impact. In this section, we present the interface and architecture of our proposed system, then we describe how such a system can be implemented. We conclude with preliminary results that illustrate the potential overheads of the system.

5.1 API

The detection system exposes a set of interfaces that allow virtualization tools to easily query memory content sharing across a number of virtual nodes. The interface could be a set of function calls, which are integrated into the VMM. The main functions of the API are:

- *double degree_of_sharing(NodeSet nodes, ShareType type)*: This returns degree of content sharing of different types across the given set of nodes. The ShareType could be "intra-node", "inter-node" or "both".

- *NodeSet get_location(hash_value)*: Returns a list of nodes which each have at lease one memory block that hashes to the given hash value.

- *int get_copies(hash_value)*: Returns the number of copies of the memory blocks that exist in system that hash to the given hash value.

- *HashSet hashList_of_shared(NodeSet nodes, int k)*: Return a list of hash values which represent those memory blocks have at least *k* replicas across a set of nodes.

5.2 Architecture

Figure 7 presents a high-level overview of the architecture of the detection system. It comprises two components: a *memory tracer* that runs on each node and a distributed *memory content synchronizer* to collectively maintains the memory content sharing across nodes in the system. The memory tracer is deployed inside VMM for each node, which periodically collects the memory content of the virtual machines and updates them to the *Memory Contents Synchronizer* in the system through its *Update Interface*.

For illustration purposes, we also include the migration/checkpoint service in the VMM. This service would use the the detection system's *Content-sharing Query Interface* to issue queries about content sharing of different kinds.

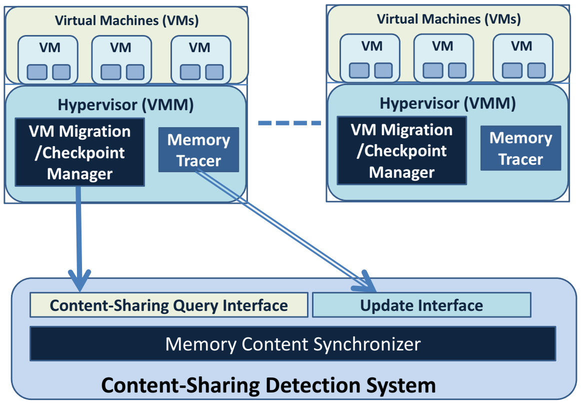

Virtual Machines (VMs)
VM
VM
VM
Hypervisor (VMM)
VM Migration /Checkpoint Manager
Memory Tracer
Virtual Machines (VMs)
VM
VM
VM
Hypervisor (VMM)
VM Migration /Checkpoint Manager
Memory Tracer
Content-Sharing Query Interface
Update Interface
Memory Content Synchronizer
Content-Sharing Detection System

Figure 7: Architecture of the proposed memory content sharing detection system within virtualized environments

5.3 Implementation

A hash-based approach is used to identify identical memory content. The memory blocks in each node will be hashed and compared with memory blocks on other nodes periodically. A system-wide distributed hash table (DHT) is built to keep track of the contents of memory blocks in the system and identify the shared content.

System-wide distributed hash table (DHT).

The DHT is partitioned and spread among all nodes. Each node is responsible for storing and maintaining some number of partitions of the table. When new nodes are added to the system, this partitioning is altered so that data is spread onto the new node. Similarly to classic DHTs, all DHT participant nodes form an overlay logically. Each node in the overlay is identified by a unique ID and is responsible for storing memory blocks with certain hash values in its partitions.

The DHT API provides services with *add()* and *remove()* for the memory tracer to update the DHT. In addition to the update operations, each node in the system can query the DHT to discover the degree of intra- and inter-node memory content sharing the system currently has, where these shared memory blocks are located, etc.

The basic structure of the DHT is the <hash, list(node id)> pair. In general, the DHT receives <hash, node ID> pair updates from memory tracers that are running on each physical node. The DHT collects all pairs of <hash, node ID>, sorts them and generates <hash, list(node ID)> from pairs with the same hash. In addition, the DHT computes and maintains the degree of inter-node content sharing, the number of content shared blocks and all other information from these stored hash pairs accordingly with the DHT update operations.

Memory tracer.

A memory tracer is run on each physical node. It works by periodically stepping through the full memory of the machine being traced, generating hashes for each of its memory blocks. Since a full memory scan is a costly operation, an alternative approach for the tracer is to monitor the memory updates in the system, and only rehash these memory blocks that have been modified since the last scan. This can be implemented by scanning the page table (shadow page table or nested page table for a virtual machine) to locate all updated pages (by checking the dirty bit in the page table entry) from last scan round. After each page table scan, the dirty bit is cleared in the entry.

In each round, the memory tracer scans the target VM's page table, rehashes all memory blocks that have been modified since the last scan and send the updates of these hashes to the DHT. Whenever there is a hash update to a memory block, the new hash value of the block is added to the DHT, while its old hash value is removed from DHT.

In addition, the memory tracer maintains a local hash table that allows the DHT to efficiently locate a memory page's content from its hash, whenever the DHT requests the memory page. Each entry in the hash table corresponds to a memory hash value, which can map to several memory blocks.

5.4 Discussion

We now discuss issues that arise in the design and implementation.

Hash function.

Since we use a hash value to represent the content of a memory block, there is always possibility of collisions, which means the different memory blocks are mapped to the same hash value by the system. However, when we use MD5, which generates 128bit hash value, if we consider a system with 1 Petabyte (2^{50} bytes) of 4KB memory pages hashed by our memory tracer using the MD5 hash function, the collision probability is around 10^{-14}. This is considered to be less than probability of physical memory corruption.

Since there is no critical security concern for our target parallel systems. We can potentially reduce compute overhead by using non-cryptographic hash functions, for example, we extend SuperHash [7] to generate 128bit hash as the same length as MD5 but need less computing efforts. We will compare the performance overheads by using the two hash functions in section 5.5.

It is important to point out that while we need a good hash function, we do not need a cryptographic hash. Furthermore, while it is true that there are attacks on, e.g., SHA-1, that allow for collisions to be engineered, we do not assume that the VMs are adversarial, which is a common assumption in HPC.

Memory update rate.

The memory tracer will compute the number of memory pages that have been updated since last scan and predict the possible computing overhead to rehash all of them. If the memory update rate becomes too high, the tracer will temporarily increase the scan interval to reduce the accumulated overheads to the system. It slows down memory hash updates to the DHT to prevent a burst flood to the network and the DHT. As we discuss in Section 5.5, increasing the scan interval will generally decrease the system overheads. Furthermore, the memory tracer could adaptively adjust its scan interval to control the system overheads.

DHT availability.

Communication failures or node failure can lead to part of the content-sharing information being not available. As the size of system grows, the likelihood of communication/node failures increases. The DHT itself has to be kept available in case of all these different kind of failures.

Given that the data in the DHT is spread across multiple nodes, if any of those nodes fail, then a portion of the hash table will become unavailable. For this reason, each partition of the DHT can be replicated on more than one node. The set of replicas from the same partition forms a group, while all replicas in the group are kept coherent with each other. Any replica in the group can be used when there are read operations to it. However,to maintain consistency among all replicas in the group, when the write operations to the partition, for example (*put*() and *remove*()), are issued, all replicas belonging to the same partition must be synchronously updated. If a node fails, the data from its partitions is still available from other surviving nodes which contain these partitions.

Memory block granularity.

Generally, the size of the memory block in the system is of page granularity, i.e, 4KB page in x86 machines. However, detecting memory content sharing with another block size is possible, but the smaller the memory block, the more extra CPU, memory and communication overhead is needed. In addition, from Figure 5, we can see that reducing the size of memory blocks does not increase the detected amount of memory content sharing for most of applications.

Another option is to dynamically change the granularity of detected memory block size at runtime, such as what has been applied on incremental checkpoint [8]. In this approach, the size of memory block to be hashed is dynamically changed through adaptively splitting/merging according to the memory access pattern and the working set of the detected workloads.

5.5 Initial performance evaluation

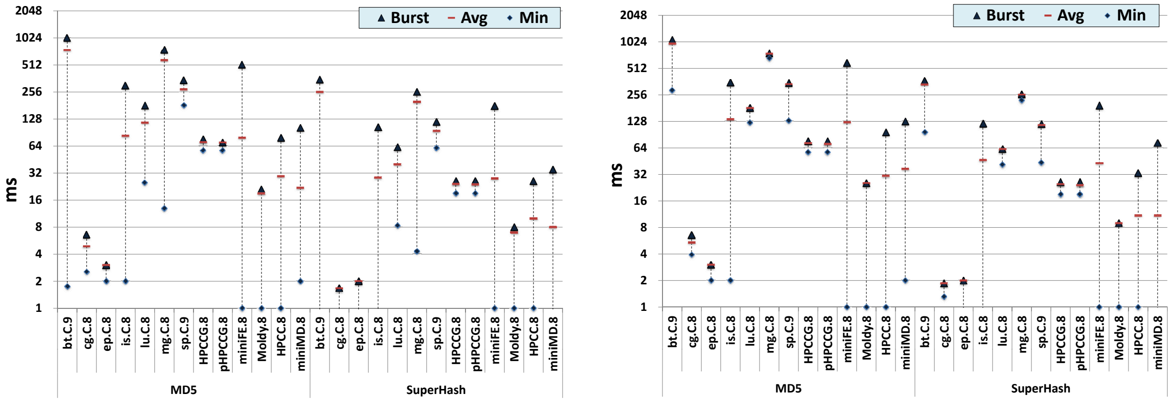
The detection system is still an in-progress project. We have investigated the CPU and network overheads needed for the memory tracer. We report the CPU time needed for the memory tracer to scan the page table and rehash all updated local memory pages, and the size of data that it sends to convey these hash updates to the DHT over the network.

In Figure 8, we show the CPU time that is spent by the memory tracer for each node to scan and rehash all locally updated memory pages during each scan round. We present the burst, average and minimal CPU times needed for each round during the entire execution the workload.

For most of the workloads, using the MD5 hash function, it takes on average less than 128ms to scan and rehash memory updates when the scan interval is set to 2s. This is less than 6.4% overhead. With a scan interval of 5s, the CPU time required is even less, resulting in a 2.6% overhead. However, several applications (such as BT and MG) have much higher memory update rates, which need on average 512ms CPU time (25% overhead) during each scan round when the interval is 2s. For these applications, when the system extends its scan interval to 5s, the CPU times needed for each scan round do not increase too much, which makes the CPU overhead decrease to less than 10%. Even for less frequently occurring burst updates during application execution, the memory tracer consumes less than 512ms, which is 25% overhead for most of the workloads. To reduce this overhead, the system can set the scan interval to 5s, which only increases the CPU time a little bit, resulting in an overhead of less than 10% even during the burst updates.

Compared with MD5, when the SuperHash hash function is used, the CPU overheads are about 1/3 as high. That means for most of the workloads, the overheads of the memory tracer are less than 2.2% with a 2s interval and less than 1% when interval is 5s, and less than 9% in 2s and 4% in 5s interval during burst updates.

As we discussed in Section 5.4, the memory tracer can dynamically adjust the scan interval for memory hash updates. If it detects that the current memory update period would cause CPU overhead to surpass a preset threshold, the memory tracer will increase the scan interval. If it dips below a higher threshold, it will decrease the interval.



(a) 2s Interval

(b) 5s Interval

Figure 8: CPU Time that is spent by the memory tracer on each node to scan and rehash all local updated memory pages during each scan round. We show results from using 2s and 5s scan intervals with MD5 and SuperHash functions. For most of the workloads when using MD5 hash, it needs less than 128ms to rehash memory updates in 2s interval, which is less than 6.4% overhead. While in 5s interval, it need less than 128ms, which is less than 2.6% overhead. When switching to SuperHash, the overheads are around 1/3 of the ones using MD5.

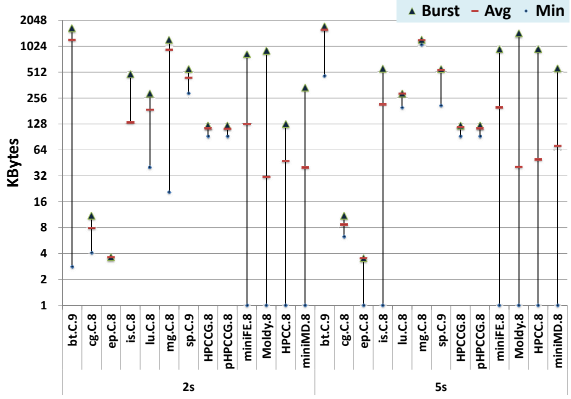


Figure 9: Size of data that is sent over the network by the memory tracer on each node during each scan round to update its memory hashes to DHT. We show results from using 2s and 5s scan intervals with MD5. For most of the workloads, the memory tracer sends less than 512Kbytes data during each round averagely and less than 1024Kbytes during burst updates.

Besides the CPU overhead, we also evaluated the data traffic that is needed for each node to send its memory hash updates to the DHT during each scan round. Figure 9 shows the amount of data that is sent over the network by each node during a 2s and 5s scan interval. Since MD5 and our extended version of SuperHash generate hash values of the same length, the data traffic needed are the same for both hash functions. From the figure, we can see that for most of the workloads, the memory tracer sends less than 512Kbytes data during each round on average and less than 1024Kbytes during burst updates. This suggests that the network overhead of memory tracer is generally less than 1% on 1Gbps network, which is very acceptable in our target cluster and HPC systems.

6. RELATED WORK

Many studies have shown the benefits of using memory sharing between VMs collocated on the same host. Content-based page sharing was introduced in VMware ESX [28] and Xen [17]. These implementations use background hashing and page comparison in the hypervisor to transparently identify same pages belong to VMs on the same host. Potemkin [27] uses flash cloning and delta virtualization to enable a large number of mostly-identical VMs on the same host. Satori [19] implements memory sharing in the Xen environment by detecting opportunities for page sharing while reading data from a block device. Difference Engine [15] has demonstrated that even higher degrees of page sharing can be obtained by sharing portions of similar, but not identical pages.

Kernel Samepage Merging (KSM) [10] lets the Linux kernel share identical memory pages amongst different processes. It can also be used in conjunction with QEMU/KVM to share identical regions of memory between multiple co-located VMs. SBLLmalloc [11] is a user library that can identify identical memory blocks on the same machine and merge them to reduce the memory usage for large scale applications in HPC systems.

The above works have the goal of reducing memory pressure on individual nodes by deduplicating intra-node memory content sharing. In contrast, our proposed system detects and tracks both intra-node *and inter-node* sharing of memory content across all the nodes. Additionally, we have argued that such detection and tracking should be factored into a separate service, on top of which other services, including dedupliction, can be built.

The work closest to ours is Memory buddies [29], which uses memory fingerprinting to discover VMs with high sharing potential and then co-locates them on the same host. It uses a centraized controller to gather fingerprints from each node, which is likely to limit scalability. In contrast, our proposal uses a scalable distributed hashing approach.

Live gang migration [13] optimizes the live migration of a group of co-located VMs on the same host by deduplicating the identical memory pages in these co-located VMs before migrating them. VMFlock [9] and Shrinker [25] present a migration service optimized for cross-datacenter transfer.

Finally, our proposed system is not a distributed memory sharing/caching system such as Memcached [3] or RAMCloud [24]. Such general-purpose distributed memory caching systems are used to speed up dynamic database-driven websites by caching data and objects in RAM to reduce the number of times an external data source must be read.

7. CONCLUSION

We have argued for a facility for dynamically detecting and tracking intra- and inter-node memory content sharing in a virtualized large scale parallel system. The main contributions of this paper that support this argument are:

- We carried out a detailed study of the memory content sharing for a range of applications and application benchmarks to support our argue that both intra- and inter-node memory content sharing is common in parallel applications.
- We described how we can simplify and enhance many services in large scale parallel systems by leveraging these memory content sharing within individual nodes and across nodes.
- We proposed a detection system which can efficiently and scalably track and identify the memory content sharing in large parallel systems with low overhead. We discussed the challenge to build such a system, and our approach and some preliminary results in building the system.

This is a work in-progress. We are currently completing the design and implementation of the memory tracer frontend and the DHT backend.

8. REFERENCES

[1] HPCC. http://icl.cs.utk.edu/hpcc/.

[2] LAMMPS. http://lammps.sandia.gov/.

[3] Memcached. http://memcached.org/.

[4] Miniapplications. https://software.sandia.gov/mantevo/.

[5] MOLDY. http://www.ccp5.ac.uk/moldy/moldy.html.

[6] NPB. http://www.nas.nasa.gov/publications/npb.html.

[7] SuperHash. redshift.sourceforge.net/superhash/.

[8] Agarwal, S., Garg, R., and Gupta, M. S. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the 18th annual international conference on Supercomputing (ICS'04)* (2004).

[9] Al-Kiswany, S., Subhraveti, D., Sarkar, P., and Ripeanu, M. VMFlock: virtual machine co-migration for the cloud. In *Proceedings of the 20th international symposium on High performance distributed computing (HPDC'11)* (2011).

[10] Arcangeli, A., Eidus, I., and Wright, C. Increasing memory density by using KSM. In *Proceedings of the Linux Symposium (OLS'09)* (2009).

[11] Biswas, S., de Supinski, B. R., Schulz, M., Franklin, D., Sherwood, T., and Chong, F. T. Exploiting data similarity to reduce memory footprints. In *Processing of the 25th IEEE International Symposium on Parallel and Distributed (IPDPS'11)* (2011).

[12] Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation (NSDI'05)* (2005).

[13] Deshpande, U., Wang, X., and Gopalan, K. Live gang migration of virtual machines. In *Proceedings of the 20th international symposium on High performance distributed computing (HPDC'11)* (2011).

[14] Ferreira, K., Stearley, J., Laros, III, J. H., Oldfield, R., Pedretti, K., Brightwell, R., Riesen, R., Bridges, P. G., and Arnold, D. Evaluating the viability of process replication reliability for exascale systems. In *Proceedings of the International Conference for High Performance Computing, Networking,Storage and Analysis (SC'11)* (2011).

[15] Gupta, D., Lee, S., Vrable, M., Savage, S., Snoeren, A. C., Varghese, G., Voelker, G. M., and Vahdat, A. Difference engine: Harnessing memory redundancy in virtual machines. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)* (2008), pp. 309–322.

[16] Huang, W., Liu, J., Abali, B., and Panda, D. K. A case for high performance computing with virtual machines. In *20th Annual International Conference on Supercomputing (ICS'06)* (2006).

[17] Kloster, J., Kristensen, J., and Mejlholm, A. On the feasibility of memory sharing: Content-based page sharing in the xen virtual machine monitor. Tech. rep., Master Thesis, Department of Computer Science, Aalborg University, 2006.

[18] Lange, J. R., Pedretti, K. T., Hudson, T., Dinda, P. A., Cui, Z., Xia, L., Bridges, P. G., Gocke, A., Jaconette, S., Levenhagen, M., and Brightwell, R. Palacios and Kitten: New high performance operating systems for scalable virtualized and native supercomputing. In *24th IEEE International Symposium on Parallel and Distributed Processing (IPDPS'10)* (2010).

[19] Miłós, G., Murray, D. G., Hand, S., and Fetterman, M. A. Satori: Enlightened page sharing. In *Proceedings of the 2009 conference on USENIX Annual technical conference (USENIX'09)* (2009).

[20] Moody, A., Bronevetsky, G., Mohror, K., and de Supinski, B. R. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)* (2010).

[21] Nagarajan, A. B., Mueller, F., Engelmann, C., and Scott, S. L. Proactive fault tolerance for HPC with Xen virtualization. In *21st Annual International Conference on Supercomputing (ICS'07)* (2007).

[22] Nath, S., Yu, H., Gibbons, P. B., and Seshan, S. Subtleties in tolerating correlated failures in wide-area storage systems. In *Proceedings of the 3rd conference on Networked Systems Design & Implementation (NSDI'06)* (2006).

[23] Nishimura, H., Maruyama, N., and Matsuoka, S. Virtual clusters on the fly - fast, scalable, and flexible installation. In *Proceedings of the 7th IEEE International Symposium on Cluster Computing and the Grid (CCGRID'07)* (2007).

[24] Ousterhout, J. K., Agrawal, P., Erickson, D., Kozyrakis, C., Leverich, J., Mazières, D., Mitra, S., Narayanan, A., Parulkar, G. M., Rosenblum, M., Rumble, S. M., Stratmann, E., and Stutsman, R. The case for ramclouds: scalable high-performance storage entirely in dram. *Operating Systems Review 43*, 4 (2009), 92–105.

[25] Riteau, P., Morin, C., and Priol, T. Shrinker: Improving Live Migration of Virtual Clusters over WANs with Distributed Data Deduplication and Content-Based Addressing. In *Proceedings of the 17th International European Conference on Parallel and Distributed Computing (EuroPar'11)* (2011).

[26] Sapuntzakis, C. P., Chandra, R., Pfaff, B., Chow, J., Lam, M. S., and Rosenblum, M. Optimizing the migration of virtual computers. In *Proceedings of the 5th symposium on Operating systems design and implementation (OSDI'02)* (2002).

[27] Vrable, M., Ma, J., Chen, J., Moore, D., Vandekieft, E., Snoeren, A. C., Voelker, G. M., and Savage, S. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)* (2005).

[28] Waldspurger, C. A. Memory resource management in vmware esx server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)* (2002).

[29] Wood, T., Tarasuk-Levin, G., Shenoy, P. J., Desnoyers, P., Cecchet, E., and Corner, M. D. Memory buddies: exploiting page sharing for smart colocation in virtualized data centers. In *Proceedings of the 5th International Conference on Virtual Execution Environments (VEE'09)* (2009).