Optimizing Virtual Machine Consolidation Performance on NUMA Server Architecture for Cloud Workloads

Ming Liu and Tao Li
Intelligent Design of Efficient Architectures Laboratory (IDEAL)
Department of Electrical and Computer Engineering, University of Florida
mingliu@ufl.edu, taoli@ece.ufl.edu

Abstract

Server virtualization and workload consolidation enable multiple workloads to share a single physical server, resulting in significant energy savings and utilization improvements. The shift of physical server architectures to NUMA and the increasing popularity of scale-out cloud applications undermine workload consolidation efficiency and result in overall system degradation. In this work, we characterize the consolidation of cloud workloads on NUMA virtualized systems, estimate four different sources of architecture overhead, and explore optimization opportunities beyond the default NUMA-aware hypervisor memory management.

Motivated by the observed architectural impact on cloud workload consolidation performance, we propose three optimization techniques incorporating NUMA access overhead into the hypervisor's virtual machine memory allocation and page fault handling routines. Among these, estimation of the memory zone access overhead serves as a foundation for the other two techniques: a NUMA overhead aware buddy allocator and a P2M swap FIFO. Cache hit rate, cycle loss due to cache miss, and IPC serve as indicators to estimate the access cost of each memory node. Our optimized buddy allocator dynamically selects low-overhead memory zones and "proportionally" distributes memory pages across target nodes. The P2M swap FIFO records recently unused <PFN, MFN> lists for mapping exchanges to rebalance memory access pressure within one domain. Our real system based evaluations show a 41.1% performance improvement when consolidating 16-VMs on a 4-socket server (the proposed allocator contributes 22.8% of the performance gain and the P2M swap FIFO accounts for the rest). Furthermore, our techniques can cooperate well with other methods (i.e. vCPU migration) and scale well when varying VM memory size and the number of sockets in a physical host.

1. Introduction

Due to increasingly attractive benefits such as a 50% reduction in hardware and operating costs as well as an 80% reduction in energy consumption [1], server consolidation, enabled via virtualization technologies, has been widely adopted by large-scale cloud computing platforms, such as Amazon EC2 [2] and Windows Azure [3]. A recent survey [4] shows that as computing demand increases, rather than adding additional server resources to existing data center facilities, over 60% of data center operators will consolidate their workloads (encapsulated in virtual machines). However, when scaling virtual platforms to handle the proliferation of cloud applications, challenges arise due to constraints on performance degradation (as a result of performance guarantees in the customer service level agreement), especially in those environments where VM consolidation density continues to grow. Therefore, optimizing VM consolidation performance is increasingly becoming one of the key challenges faced by the cloud provider community.

Meanwhile, server manufacturers like Dell and HP gradually replace traditional *Uniform Memory Access (UMA)* machines with *Non-Uniform Memory Access (NUMA)* ones due to their higher memory bandwidth and better system scalability. In a NUMA configuration, multiple sockets share memory and communicate with each other through a fast communication interconnect. With current QPI [5] and Hyper Transport [6] technologies, the difference between local and remote memory access latencies can be minimized within 30% [7]. Presently, consolidated workloads are shifting from conventional single-task computation-oriented applications [8] to large-scale cloud and big data workloads, such as NoSQL data storage [9], Hadoop cluster computing [10] and online transaction processing [11]. These cloud workloads generate many diverse interactions and communication patterns (e.g. request partition and collaboration, massive data caching, data scanning, heartbeat checking, etc.). Moreover, some of these applications have a large irregular memory footprint and high memory consumption, which presents a significant challenge to optimizing the efficiency of virtual machine consolidation.

To adapt these two trends, we characterize and optimize cloud workload consolidation in NUMA-based virtualized environments. Note that both CPU scheduling and memory management techniques can be applied to NUMA system optimization. Prior work in this context falls into following categories: (1) NUMA-aware scheduling [12], which pins virtual CPU (vCPU) to the nearest physical cores or those with the lowest access penalty; and (2) NUMA-aware VM placement [13] and dynamic load balancing [14]. Such hypervisors (e.g. VMware ESX [15] or Citrix XenServer [16]), initialize VM memory by minimizing the overhead due to remote memory access, and rebalance VMs deployment by moving VMs to lightly loaded nodes via page migration. Our work also aims at enhancing the efficiency of hypervisor memory management. Unlike existing strategies, which only consider remote memory access penalty, our optimizations incorporate other sources of architectural overhead that are equally important.

Toward this goal, we analyze four NUMA memory access overheads, namely, 1) last level cache contention, 2)

978-1-4799-4394-4/14/$31.00 © 2014 IEEE

memory controller congestion, 3) interconnection congestion, and 4) remote memory access latency. To estimate each overhead, we also explore three architectural metrics: last level cache hit rate, cycle loss due to last level cache miss, and IPC. We propose three optimizations: namely, memory zone access overhead analysis, NUMA overhead-aware buddy memory allocator, and P2M swap FIFO. The first technique collects virtual machine architectural metrics and estimates each memory zone's overhead in the hypervisor. The L3 cache hit rate, cycle loss due to L3 misses (local and remote), and the IPC ratio between local and remote nodes are used to estimate the impact on performance due to these overheads. The other two proposed techniques depend on the above overhead analysis; instead of reserving memory directly on the target node, the optimized buddy allocator examines memory node access overhead, selects a small list of memory zones, and distributes memory page allocation. To guarantee fast address translation, a binary tree organization is maintained. The P2M swap FIFO stores a list of recently unused <PFN, MFN> (<physical frame number, machine frame number>) mappings for each allocated memory node within each virtual machine. This enables rebalancing of node access latencies via mapping exchanges, which further improves performance.

We prototype these techniques on Xen 4.1.2 hypervisor and evaluate their efficiency on an IBM x3850 server. Our empirical results show that the proposed schemes improve both performance and architectural metrics. For example, on a 4-socket system with 16 consolidated VMs, our NUMA overhead-aware buddy allocator and P2M swap FIFO schemes improve performance by 22.8% and 18.3% respectively. Moreover, our optimizations cooperate well with dynamic vCPU scheduling and we verify their scalability to various VM memory sizes and different numbers of sockets in the physical host. Our current implementation incurs a 0.46s latency to create a 4GB VM and adds less than 7.46KB of memory usage in the hypervisor, which can be ignored.

The rest of this paper is organized as follows: Section 2 provides an overview of NUMA architectural overhead and hypervisor memory management. Section 3 characterizes cloud workload consolidation in NUMA virtualized systems. Sections 4 and 5 present our design and implementation. Section 6 evaluates our prototype and compares with existing methods. Section 7 discusses related work and Section 8 concludes this paper.

2. Background

In this section, we provide the background (i.e. memory access overhead in NUMA system, and domain guest memory management in hypervisor) relevant to our study.

2.1. Memory access overhead in NUMA architecture

Non-Uniform Memory Access (NUMA), a technology widely adopted in multiprocessor design, outperforms UMA in terms of scalability and memory bandwidth. In today's market, most server systems, such as Dell's PowerEdge [17] and HP's ProLiant [18], are equipped with at least two chip-multiprocessors communicating via an on-chip interconnect. Figure 1 illustrates a dual socket NUMA system. In this example, each multi-core chip consists of four cores that share the last level cache, memory controller, and physical memory DIMMs. Sockets communicate with each other via point-to-point interconnects (e.g. Intel QPI [5]). In a NUMA system, memory accesses can be classified as: either (a) local (requests to chip's own memory), or (b) remote (requests to memory of other sockets). Due to off-socket communication overhead, local is faster than remote access.

There are four sources of overhead [19] when considering memory access in NUMA systems. These are labeled in Figure 1:

- Last level cache contention (*A*): cores within the same chip-multiprocessors contend for the shared last level cache. Frequent cache evictions result in significant performance degradation.
- Memory controller congestion (*B*): memory requests issued to the same memory module share the same memory controller, leading to access congestion. Note that the controller also includes a memory queue unit.
- Interconnection congestion (*C*): Since NUMA systems allow memory access from other sockets; excessive cross-socket traffic can result in interconnection congestion.
- Remote memory access latency (*D*): This is the result of off-socket communication overhead. Actual latency depends on the distance between source and destination sockets.

In the past, various performance counters have been used to measure the above mentioned memory access overhead for non-virtualized NUMA systems [19, 7]. However, this becomes more challenging to do in a virtualized environment since only limited architectural performance counting metrics are supported [20]. In this study, we explore the use of IPC, L3 cache miss rate, and cycle loss due to the L3 cache to quantify the above four overheads. Details are explained in Section 4.

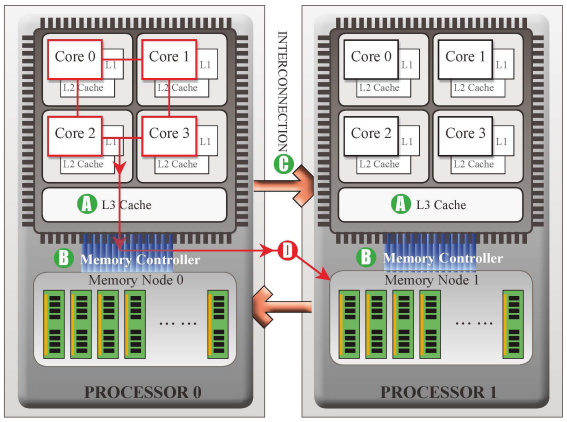


Figure 1: Schematic view of a dual-socket NUMA system

2.2. Hypervisor memory management for domain guests

Memory allocation and page table management are two mechanisms used by the hypervisor to manage the physical memory of virtual machines. In this work, we leverage these two knobs to improve performance of virtual machine consolidation for NUMA architectures. We use Xen hypervisor as an example.

During the initialization phase, Xen applies several memory allocation strategies: e820 memory allocator, boot allocator and heap allocator. E820 runs first after the system boots up. It records initial system memory information in the *e820_raw* array and reserves a range of page frames. Next, the boot allocator runs to establish the page frame bitmap and to register memory from the e820 array with a one-to-one mapping between physical addresses and linear addresses. The heap allocator (also known as the buddy allocator), which also supplies memory for domain guests, is the primary allocator during Xen execution. It performs nearly the same as the Linux memory allocator by associating memory space with <*Node*, *Zone*, *Order*> triples. *Node* refers to the memory location and *Zone* indicates the range of memory space. For instance, *Zone* n comprises all page frames between $[2^n, 2^{n+1}-1]$. *Order* indicates the number of pages to be allocated upon a request. To simplify address computation, the heap allocator sets the basic memory block size to be power-of-twos. To minimize fragmentation, each memory request is assigned the smallest possible block. The entire memory space is organized as a binary tree and a larger memory block can be split into two similar but smaller blocks if needed. Two contiguous free blocks (children) can be merged to form a parent block. Via the *alloc_heap_pages* function call, both Xen heap and Dom heap invoke the buddy allocator for page allocation. Note that the current buddy allocator within the Xen hypervisor already supports NUMA awareness, allocating local memory blocks to physical cores.

Since virtualization provides an isolated execution environment for the guest, the virtual address requires two layers of translation. The first layer translates the virtual address to a guest physical address via a process page table (maintained by the guest OS). The second translation further converts the guest physical address to a machine address using a shadow page table or nested page table (maintained by the hypervisor). Shadow paging follows a traditional page walk without any architectural support. It directly maps the guest virtual address to a system physical address upon a TLB miss through a per-process based shadow page table, which duplicates the guest OS's process page table in the hypervisor. On a page table update, the request is intercepted to maintain consistency of the corresponding shadow table.

Unlike software-based approaches, hardware assisted paging (HAP) leverages a two-dimensional page walker and establishes both the guest page table (VA→PA) and a nested page table (PA→MA) separately. There are two CR3 registers (X86) for each CPU core: one for the guest page table and the other for the nested page table. The guest OS maintains its page table without any hypervisor intervention. A TLB miss causes the walker to traverse these two tables to obtain the final mapping. When a nested page fault occurs, it first traps into the hypervisor to check for a violation of an associated nested page table entry and then performs an update. If the entry doesn't exist, the hypervisor will populate a new page frame and create an entry with the physical frame number (PFN), machine frame number (MFN), and access rights. The nested page table can be viewed as a special organization of PFN-to-MFN mappings in the hypervisor.

In this work, we target the heap allocator and nested page table management optimizations by taking NUMA memory access overhead into consideration. Sections 4 and 5 describe our prototype design and implementation.

3. Characterizing cloud workloads in NUMA virtualized systems

In this section, we setup experiments on real systems to study: 1) how NUMA architecture features affect the performance of virtualized cloud workloads, especially with high VM consolidation densities; and 2) how to estimate the four above mentioned NUMA architectural overheads with appropriate hardware performance counters.

Benchmark	Description
YCSB with MySQL	Benchmarking data read/write/update using MySQL database via YCSB interface
Memcached	Simulating the behavior of a Twitter caching server using Twitter dataset
NPB/IS NPB/UA	HPC benchmarks from NAS Parallel Benchmark Suite
TPC-C	Benchmarking the OLTP with a warehouse-centric order processing application
TunkRank	Analyzing influence of a Twitter user based on the number of that user's followers

Table 1: Cloud workload description

3.1. Experimental environment

All experiments are performed on an IBM x3850 system equipped with 8-socket based computing nodes. Each socket contains one Intel Xeon X7550 (Nehalem architecture [21]) processor and 64GB of DDR3 physical memory. Each processor further consists of 8 physical cores running at 2.0 GHz (2.4 GHz when the turbo boost feature is enabled). Each physical core has a 32KB L1 instruction cache, a 32KB L1 data cache, and a unified 256KB L2 cache. The processor is also equipped with an 18MB last level cache (L3). Socket communication is conducted via the Intel QuickPath Interconnect (QPI) [5] with a throughput of 6.4 GT/s. In our experiments, we use the performance governor, which sets each physical core to the highest frequency, as the power management policy. Additionally, we enable the turbo boost and hyper threading (16 logical cores) features of the processor. Therefore, there are a total 128 cores and 512GB of memory in each computing node. We configure the storage system with a 146GB RAID 1 system disk and a 536GB RAID 0 data disk. The system uses a Broadcom NetXtreme II 5709c 1 Gigabit Ethernet NIC for networking.

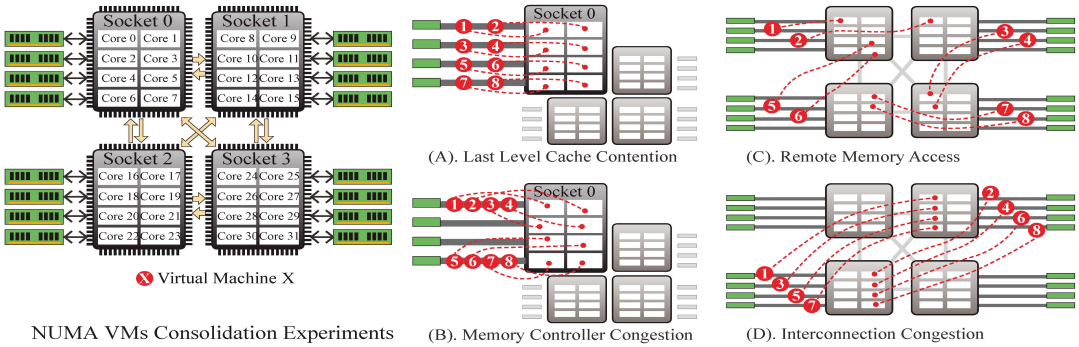
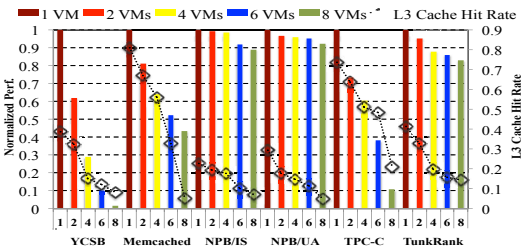
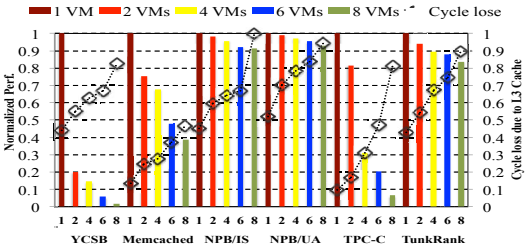


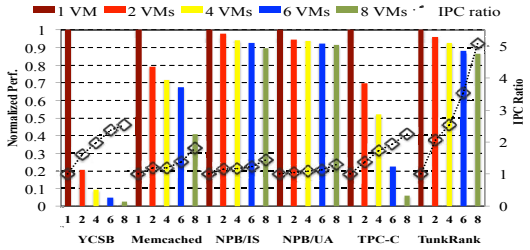
Figure 2: The four scenarios that we use to characterize NUMA architecture overheads. All VMs are created with NUMA-aware memory allocation mechanism and pinned to the specified physical cores.



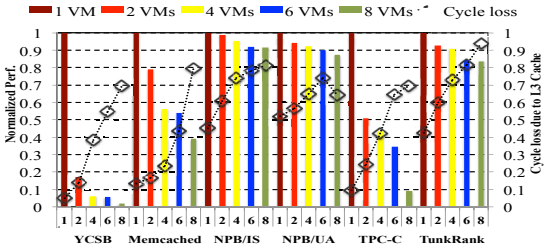
(a). Last level cache contention



(b). Memory controller congestion



(c). Remote memory access latency



(d). Interconnection congestion

Figure 3: The normalized performance along with corresponding architectural metrics to capture performance overhead. The values on X-axis are number of consolidated VM in system.

Note that for our characterization experiments, we opt to configure the IBM x3850 system with 4 sockets, which represents the current mainstream of NUMA system.

We use the Xen 4.1.2 [22] hypervisor with Intel VT enabled and Linux 2.6.32.40 as the Domain 0' kernel for evaluation. Each HVM VM guest runs Debian Squeeze configured with 2 vCPUs and 4GB memory. We select a set of high memory consumption and large irregular memory footprint benchmark [23, 24], as listed in Table 1. Among those, *YCSB* [25], developed by Yahoo! Research, aims to compare emerging cloud data serving systems (such as MySQL and Cassandra [26]). *Memcached* [27] is widely used as the distributed in-memory key-value store to improve the performance of web applications. *NPB/IS* stands for integer sort with random data read while *NPB/UA* represents unstructured adaptive mesh, dynamic and irregular memory access [28]. *TPC-C* is from OLTP-Bench [11], tailored for evaluating on-line transaction processing (OLTP) and web-oriented workloads. *TunkRank*, a package in GraphLab [29] targeting machine learning and data mining analysis for graphics, serves as an extension workload to CloudSuite [30]. Among these workloads, *NPB/IS*, *NPB/UA*, and *TunkRank* are evaluated using execution time while others are measured via throughput. All experiments are performed three times to obtain the average statistics.

3.2. Evaluation of NUMA architecture overheads

To identify and estimate NUMA architecture overheads, we carefully design four different scenarios by varying vCPUs and memory mappings among the multiple sockets, as shown in Figure 2.

In the last level cache contention case (Figure 2(A)), we deploy all VMs on socket 0 and spread them across four memory controllers. This is similar to the memory controller congestion case (Figure 2(B)), except that in Figure 2(B) VMs 1-4 are mapped to controller 0 and VMs 5-8 occupy controller 3. Note that in Figure 2(B), we did not consolidate all 8 of the VMs on one controller since the maximum memory capacity governed by a single controller is 16GB, and thus can only accommodate four 4GB VMs. To stress remote memory access, in Figure 2(C), all VMs are evenly distributed to four sockets and VMs 2-8 remotely read/write data from neighboring sockets. Figure 2(D) shows VM

placement in the interconnection congestion scenario. Similar to Figure 2(C), it also includes remote memory access. The difference is that we allocate the memory of these four VMs (i.e. VMs 2, 4, 6, 8) on socket 1 while pin their vCPUs to the physical CPUs (i.e. cores 17, 19, 21, 23) on socket 2 and this placement is mirrored for the other 4 VMs, resulting in traffic congestion. To avoid vCPU scheduling effects, we statically pin these vCPUs to different physical cores. Figure 3 shows the normalized performance along with the corresponding performance counter value on the experimental NUMA system as VM consolidation density increases. To estimate various NUMA architectural overheads, we use *IPC*, *last level cache miss rate* (***miss rate***) and the *ratio of cycle loss due to last level cache misses* (***cycle loss***), which are collected via the open source Intel Performance Counter Monitor tools [31]. Our characterization study shows that above mentioned performance counters are well-correlated with the workloads' performance under different NUMA architecture overheads. This motivates us to use these indirect indicators to recognize the NUMA overheads that cause applications' performance degradation. The following subsections discuss various NUMA overheads and most related indicators in detail.

3.2.1 Last level cache contention

Figure 3(a) shows that, on average, last level cache contention degrades NUMA system performance by 53.4% when 8 VMs are consolidated together. The L3 cache hit rate drops from 0.48 to 0.10 as VM density increases. The correlation coefficient between the L3 cache hit rate and normalized performance is 0.90, indicating that the last level cache contention overhead in NUMA machines can be accurately captured using the proposed architectural metric. Since the 8 VMs are spread across 4 sockets in this case, memory controller congestion has little impact on performance.

3.2.2 Memory controller congestion

As shown in Figure 3(b), memory controller congestion decreases performance by 22.1% on average when two VMs run together, similar to the last level cache contention experiment. However with 8 VMs consolidated, more performance degradation (i.e. 18.4%) is observed under the memory controller congestion scenario when compared to the last level cache contention case. With four VMs sharing the same memory node, access congestion extends the memory request processing latency, especially on read intensive applications, such as *YCSB* and *TPC-C*. Note that although both last level cache contention and memory controller congestion overheads co-exist in this case, we find that cycle loss has a higher correlation coefficient than the L3 cache hit rate (0.91 vs. 0.79).

3.2.3 Remote memory access latency

In NUMA architectures, the remote (i.e. off-socket) memory access latency has been a well-known target for optimizations (e.g. NUMA-aware memory allocator in OS kernels [19, 7] or hypervisors [12, 13]). As shown in Figure 3(c), the average IPC decreases from 0.62 to 0.31 as VM consolidation density increases. Even though other architectural metrics vary with performance, IPC exhibits the most relevance. Note that instead of directly employing IPC, we opt to calculate the IPC ratio between local and remote memory accesses as an overhead estimator since it manifests a higher performance correlation coefficient (0.92) than IPC (0.79).

3.2.4 Interconnection congestion

In this experiment, VM resources (core and memory) are deployed to only two sockets. We create a pair of VMs on these sockets every time and configure them to use memory within its own socket and the CPU within another socket. This deployment introduces significant data traffic across the interconnect, especially for memory intensive applications. Figure 3(d) shows that performance drops 13.3% on average when a new pair of VMs is deployed. *NPB/IS*, *NPB/UA* and *TunkRank* are the least affected benchmarks since they issue a massive number of memory requests during the initialization phase, yet minimize the number of requests during the rest of the execution period. Note that although last level cache contention exists; it is not the major contributor to performance degradation. This is because the L3 cache hit rate only drops from 0.54 to 0.32 when VM density increases. We observe that among all three hardware performance counters, cycle loss is the most accurate with a correlated coefficient value of 0.86 to identify interconnection congestion overhead.

Prior studies on VM consolidation for NUMA systems largely focus on minimizing remote memory access. In [13], the authors treat this as a bin packing problem [32] and propose a NUMA-aware VM memory allocation with *greedy*, *packed*, and *spread* policies. However, these methods fail to take the entire memory access overhead of current, state-of-the-art systems into consideration, and the remote memory access latency is sometimes overemphasized. Our characterization results show that all four NUMA access overheads are equally important and can be inexpensively captured with appropriate performance counters, which motivates our NUMA overhead-aware design.

4. Design

This section describes the key enabling techniques for incorporating NUMA overhead awareness within hypervisor memory management: namely, memory zone access overhead estimation, buddy memory allocator and hardware assisted page fault handling optimizations.

4.1. Memory zone access overhead estimator

To estimate the overhead in a virtualized NUMA system, it is imperative to analyze and summarize memory activities of all cores where vCPUs are mapped. To this end, we use three performance counters (i.e., IPC, the L3 cache hit rate, and cycle loss due to L3 miss) to quantify memory access characteristics of vCPUs within each guest domain. Each guest domain periodically collects its vCPU(s) statistics and forwards them to the managed domain, which is responsible for mapping them to the corresponding physical cores and then dispatches to the hypervisor.

```
//Classification thresholds definition
1. #define LLC_Contention_Lx              LLC_threshold_level_X
2. #define MC_Congestion_Lx               MC_threshold_level_X
3. #define Interconnect_congestion_Lx     INTER_threshold_level_X
4. #define RML_latency_Lx                 RML_threshold_level_X
// Clear overhead data of the array
5. clearup_overhead_array();
// Compute four source overheads of each memory node
6. for each memory node i
7. begin
8.      i.LLC_contention = classify_LLC_overhead(i.cache_hit);
9.      i.MC_congestion  = classify_MC_overhead(i.cycle_loss_local);
10.     for each remote memory node j
11.     begin
12.       i.j.INTER_congestion   = classify_INTER_overhead(i.j.cycle_loss_remote);
13.       i.j.RML_latency        = classify_RML_overhead(i.j.remote_ipc);
14.     end
15. end
// Compute access overhead of each memory node
16. for each memory node i
17. begin
18.     if i == local_node
19.     begin
20.       overhead[i] += i.LLC_contention + i.MC_congestion;
21.     end
23.     else
24.     begin
25.       overhead[i] += i.MC_congestion + local_node.i.LLC_contention;
26.       overhead[i] += local_node.i.INTER_congestion + local_node.i.RML_latency;
27.     end
28. end
```

Figure 4: Pseudo code for memory zone overhead computation

The hypervisor continues tracking domain-based performance characteristics for each memory node (or memory zone), under which all associated domains store their memory access characteristics in the form of a quadruple <running CPU id, ipc, l3 hit miss rate, cycle loss due to l3 miss> list. The creation of a virtual machine will add a new entry to the list of corresponding memory zones where the VM memory is allocated. VM destruction, pause, migration and checkpointing will remove the corresponding entry from the list. To allow concurrent access to the list, the update mechanism is protected via locking. To improve accuracy and robustness, a sliding window mechanism is used to obtain smoothed statistics. Note that in a remote memory access, multiple memory zones are involved and updated.

A diagnostic process uses the following architectural metrics (based on our characterization results in Section 3.2) to quantify memory access overhead:

- **Last level cache contention:** The last level cache hit rate is used for this purpose. The hypervisor traverses each memory zone list, computes the average local hit rate and chooses the highest value to represent contention.
- **Memory controller congestion:** The cache hit rate and cycle loss are jointly considered to represent this overhead. When cache hit rate fluctuation is small, the performance correlated cycle loss variation is from memory controller congestion. By inspecting all of the domains within a memory zone list, the hypervisor chooses the domain with the maximum cycle loss and uses it to estimate congestion overhead.
- **Interconnection congestion:** The cycle losses due to L3 cache misses represent interconnect congestion. Unlike the prior two metrics, interconnect congestion is dictated by remote nodes. For each memory node, the hypervisor collects corresponding remote quadruples, attributes them to different sources (i.e. issuing sockets) and CPUs within a given socket. Finally, the interconnect congestion between local node *A* and remote node *B* is obtained by calculating the maximum values from all CPUs running on B.
- **Remote memory access latency:** The IPC ratio between local and remote accesses is used as a metric. The hypervisor computes the average IPC for the local node and each remote node by traversing the corresponding lists.

Using these metrics, the access overheads of a memory zone are assessed using the algorithm shown in Figure 4. When multiple overheads exist in one memory node, we evaluate each overhead and then accumulate them together. The main control flow is: (1) using predefined threshold values to divide each overhead metric into multiple levels (Line 1 ~ Line 4); (2) classifying the estimated overhead value to a certain level (Line 6 ~ Line 15); and (3) aggregating them together for each memory zone (Line 16 ~ Line 28). The estimated overheads are utilized in the NUMA-aware buddy allocator and P2M swap FIFO.

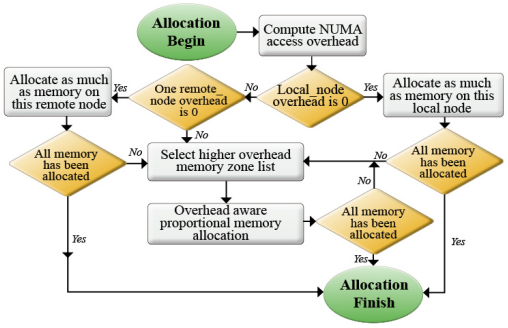
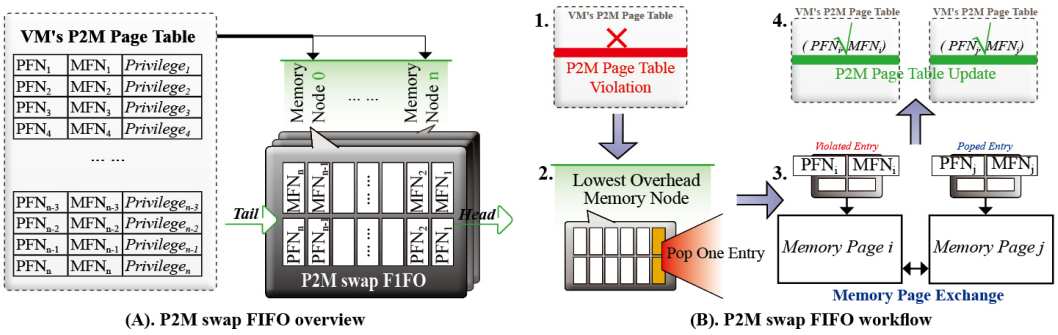


Figure 5: Flowchart of NUMA overhead aware buddy allocator

4.2. NUMA overhead-aware buddy allocator

The buddy allocator, which is widely used in the Linux kernel, splits memory blocks into power-of-two size partitions and organizes the entire memory as a recursive binary-tree. Hypervisors such as Xen use the buddy allocator for virtual machine allocation. According to the size of the memory request, the allocator searches available memory blocks and assigns the smallest block with a power-of-two size. Contemporary hypervisors support NUMA awareness during virtual machine memory initialization. Initially, it tries to allocate as much memory as possible on the local node (to pinned CPU) and later considers remote nodes until local memory is unavailable. However, NUMA-aware allocation without access overhead awareness leads to performance degradation during multiple VM consolidation.

Figure 5 illustrates the flowchart of our optimized allocator. Upon receiving a memory block allocation request, the hypervisor first computes the access overhead of all memory nodes. The local node (1st choice) or remote nodes (2nd choice) with no active VMs have the highest priority to serve the memory request. Hence, the allocator reserves the maximum possible memory chunk. Lack of such cases or unavailability of space on these nodes directs the hypervisor to inspect the memory zone lists. To select a subset of memory nodes, we use a step function that has an initial zero overhead value and increase that value by two in each step. The selection stops after four candidate nodes have been chosen. This optimization allows the buddy allocator to



(A). P2M swap FIFO overview (B). P2M swap FIFO workflow

Figure 6: Organization and workflow of NUMA overhead-aware P2M swap FIFO

determine the access overhead of memory nodes and appropriately distribute memory among low overhead nodes, which results in performance improvements.

4.3. NUMA overhead-aware P2M swap FIFO

Note that although our proposed NUMA overhead-aware buddy allocator allows virtual machine memory to be apportioned across various low overhead nodes, care should be used to balance guest memory accesses when a VM occupies several memory nodes and access penalties of these memory nodes are quite different. The proposed P2M swap FIFO of each VM could gradually resolve this imbalance and improve the overall performance of the collocated guests.

In HVM type VMs, the physical frame number to machine frame number mapping (P2M) is maintained by the hypervisor to enable nested page tables and handle HAP page faults. We propose a NUMA overhead-aware *P2M swap FIFO,* which is a software-managed buffer with a first-in-first-out policy maintained by each memory node of the virtual machine. Note that we use the FIFO feature to increase page table access locality. Each entry of the buffer contains one <PFN, MFN> mapping, as shown in Figure 6 (A). A P2M swap FIFO is created after the VM memory initialization phase on each allocated memory node. When a VM P2M page table is updated due to a new entry addition or existing entry modification, the hypervisor will perform two operations: (a) add this entry to the P2M swap FIFO until it is full; and (b) inspect the corresponding memory node access overhead.

When a P2M page table violation occurs and the estimated overhead of the associated memory node exceeds a predefined threshold (Figure 6 (B-1)), the hypervisor will identify the memory node with the lowest overhead in the system. If the estimated overhead of the identified memory node is two-levels lower (measured in the same way as Figure 4 shows), its P2M swap FIFO will pop one entry for memory page exchange (Figure 6 (B-2)). To avoid concurrency, the hypervisor initially locks the two P2M page tables and then performs page data swapping (Figure 6 (B-3)). After that, it updates the two P2M page tables with the new mapping (Figure 6 (B-4)). Note that due to the complexity of this mapping exchange, it is triggered only when a virtual machine exhibits extreme access imbalance.

5. Implementation

This section describes the implementation details of the memory nodes' access overhead assessment, enhanced buddy memory allocator and page fault handling architecture using P2M swap FIFO.

5.1. Architectural details of memory access overhead estimator

The estimator is comprised of three components: an architectural metrics collector, a hypervisor and managed domain connector, and an overhead analyzer. To begin with, the managed domain (i.e. domain 0 in Xen), which serves as a bridge between virtual machines and the hypervisor, initiates the daemon process immediately after boot up. In order to collect architectural data from guest domains, we have implemented a new hypercall named *HYPERVISOR_build_memzone_o*p, which enables communication through sockets and forwards data to the hypervisor. Furthermore, each virtual machine creates its own daemon process, registers its domain ID, obtains performance counter values with Intel open source tools [31], marshals the data, and periodically communicates with the managed domain.

When the hypervisor receives a *HYPERVISOR_build_memzone_o*p call, it unpacks the performance data packet, extracts the related domain ID and updates the maintained domain linked-list for each memory node. The hypervisor tracks all such events, protects the list from concurrent accesses for each memory node and maintains a 16-entry sliding window for each domain. On an overhead evaluation request, the hypervisor executes the overhead estimation algorithm described in Figure 4.

5.2. Enhanced buddy allocator

We extend the basic buddy allocator with NUMA overhead awareness detection, node selection, and page proportional reservation. The overhead detection stage leverages performance overhead statistics from the estimator. The selection policy is implemented by grouping the memory nodes based on the overhead level, sorting all the groups in ascending order of access overhead, and choosing memory nodes based on the requested allocation space.

Page reservation starts on the selected memory node(s). Note that page reservation follows approximation of the proportion based on binary tree organization. The requests

are split using the power-of-two rule. For example, assuming a memory request of 2^8 GB and there are three target nodes A, B and C with overheads 2, 4 and 5 respectively. The approximated overhead based on binary tree organization will be 1, 2 and 2. Therefore, nodes A, B and C will allocate memory blocks of size $2^{(8-1)}$, $2^{(8-2)}$ and $2^{(8-2)}$ that is 2^7, 2^6 and 2^6 GB, respectively.

5.3. Optimized page fault handler

We have augmented the handler with a P2M swap FIFO. It consists of a fixed size array of 256-entries for each memory node. The first-in first-out feature is implemented using a queue data structure. The FIFO structure is created during the P2M page table initialization phase and is destroyed upon guest removal. The overhead threshold is determined empirically.

During the exchange phase, each PFN needs to obtain its privilege from the original P2M page table using *get_entry* method. Machine page frame data exchange is conducted using the *memory_exchange* function. To avoid deadlock events of P2M page table spin-lock, we protect the critical section using another lock. On completion of the data exchange, old entries are removed and the updated entries are appended in the page table. The P2M swap FIFO further pops one entry, making a free entry for next entry assignment. Note that the reduction of overhead imbalance could minimize the probability of mapping exchange.

6. Evaluation

We implement our mechanisms in the Xen 4.1.2 hypervisor and compare them to the existing NUMA-aware mechanisms. Our implementation spreads across guest and managed domains, and hypervisor with 169 LOCs, 302 LOCs and 806 LOCs. In this section, we first compare *NUMA overhead-aware* to *NUMA-aware* under various scenarios. We then evaluate the effectiveness of the P2M swap FIFO and investigate whether our approaches work well with alternative methods (e.g. vCPU migration). In addition, we explore the scalability of our proposed mechanisms by varying VM memory size and the number of sockets within the physical host. The experimental setup (4 sockets of the IBM x3850 machine) and benchmarks are the same as in Section 3. As in our characterization experiments, we use execution time for performance measurement of *NPB/IS*, *NPB/UA*, and *TunkRank* and use throughput for the others. The overhead classification thresholds are empirical, which were obtained via regression analysis between performance degradation and architecture metrics during characterization. Each experiment is performed three times to obtain average statistics.

6.1. *NUMA overhead-aware* vs. *NUMA-aware*

We compare our proposed scheme (*NUMA overhead-aware*) with *NUMA-aware* using five scenarios. The first four are described in Figure 2. They emphasize last level cache contention, memory controller congestion, remote memory access, and interconnect congestion respectively. The last scenario contains a mix of all kinds of overheads.

- *Last level cache contention scenario*

On average and across all workloads, the *NUMA overhead-aware* mechanism achieves a 4.9%, 7.8%, 17.9% and 23.3% performance improvement on 2, 4, 6 and 8 VMs respectively, as shown in Figure 7(a). Similarly, the L3 cache hit rate increases by 0.042, 0.092, 0.105, and 0.151 respectively. Instead of consolidating all VMs on socket 0 as shown in Figure 2(a), the proposed approach selects less stressed memory nodes to minimize memory request latency. For instance, when VM2 joins, the hypervisor may choose socket 1 as its memory location if the last level cache contention in socket 0 becomes significant. As the number of consolidated VMs and access overhead increase, the hypervisor then spreads its allocation proportionally across all the sockets (e.g. memory allocation for VMs 7 and 8).

- *Memory controller congestion scenario*

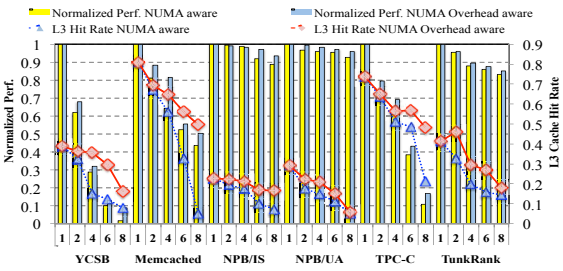
When using *NUMA-aware* VM memory allocation, as shown in Figure 2(b), VMs 1-4 initialize their memory on controller 1 of socket 0 while VMs 5-8 use memory controller 4, which causes significant data traffic congestion. With our proposed method, the cycle loss metric identifies this access overhead and triggers the *NUMA overhead-aware* memory allocation. First, it spreads memory allocation to other controllers within the same socket and then selects remote low-overhead memory nodes. As shown in Figure 7(b), performance increases by 4.7%, 20.3%, 22.8%, and 45.1% (along with 0.074, 0.089, 0.125, and 0.240 drop in cycle loss) under 2, 4, 6, and 8-VMs consolidation cases. We also enable L3 cache miss rate indicators, which improve detection of memory node overhead, resulting in greater performance improvement.

- *Remote memory access scenario*

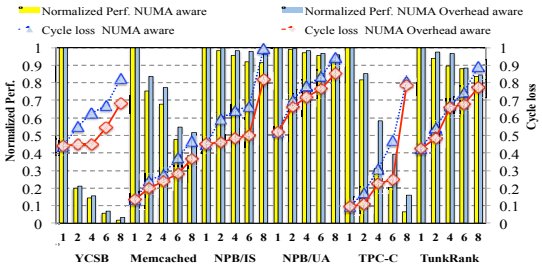
We design this experiment via statically pinning vCPUs to the physical cores out of their memory node's socket. When there are fewer consolidated VMs (such as two or four VMs), both *NUMA overhead-aware* and *NUMA-aware* methods distribute their VM's memory and vCPUs across different sockets, resulting in similar performance (e.g. 3.2% and 5.7% difference respectively). However, as more VMs are deployed, the number of overhead-free memory nodes drops (each socket has at least one VM) and the proposed method starts to select sockets with the least access overhead for memory allocation. Some remote memory requests become local and data read/write operations are balanced among all of the sockets. Therefore, performance improves by 13.7% and 16.9% for 6- and 8-VMs consolidation (along with 0.34 and 0.64 IPC ratio increase respectively), as shown in Figure 7(c).

- *Interconnect congestion scenario*

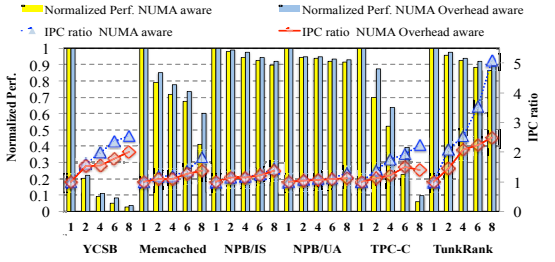
In this scenario, we distribute all VMs on nodes 1 and 2 to generate interleaved memory requests, resulting in a "data traffic jam". Our *NUMA overhead-aware* mechanism uses the cycle loss metric to detect the communication overhead for these two sockets and then allocates incoming VMs on other nodes. Consequently, the traffic jam problem is successfully solved through even distribution. As Figure 7(d)



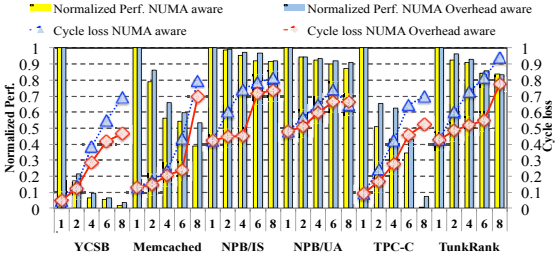
(a). Last level cache contention



(b). Memory controller congestion



(c). Remote memory access



(d). Interconnect congestion

Figure 7: Performance comparison between NUMA overhead-aware and NUMA-aware approaches. We also reported architectural metrics in the first four cases. In all scenarios, our proposed method outperforms the NUMA-aware, especially with high VM consolidation density.

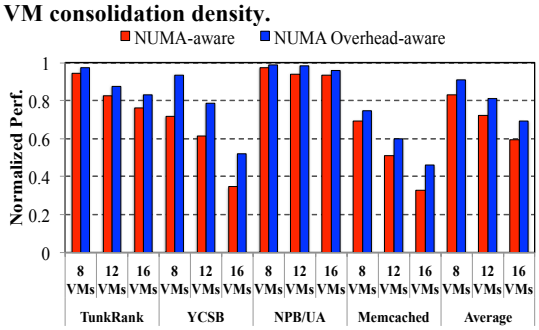
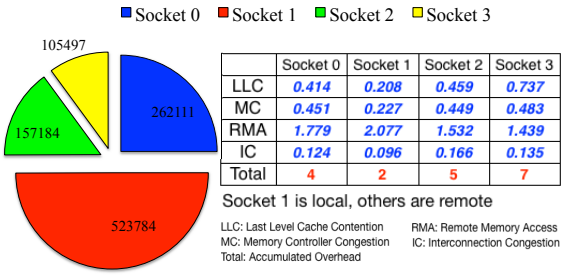


Figure 8: Performance comparison between two approaches for mixed scenario. All application's performance is normalized to single VM case.



	Socket 0	Socket 1	Socket 2	Socket 3
LLC	0.414	0.208	0.459	0.737
MC	0.451	0.227	0.449	0.483
RMA	1.779	2.077	1.532	1.439
IC	0.124	0.096	0.166	0.135
Total	4	2	5	7

Figure 9: Proportional VM memory allocation across 4 sockets for NUMA overhead-aware approach. We present architectural metrics along with accumulated overhead.

shows, on average, performance increases from 10.9% for 2-VMs scenario to nearly 143.9% for 8-VMs deployment among all benchmarks.

- *Mixed scenario*

In this scenario, all of the above four architectural overheads are present. We design our experiment with 8-VMs, 12-VMs, and 16-VMs and use 2 (8/4) VMs, 3 (12/4) VMs, and 4 (16/4) VMs to create a given type of overhead as described above. As shown in Figure 8, our optimization achieves 91.1%, 81.1%, and 69.3% of the optimal performance in these three cases, which translates to a 10.7%, 13.8% and 25.8% improvement over the *NUMA-aware*. Figure 9 provides a breakdown of the memory partitions when creating a VM on socket 1 (i.e. local socket) along with the estimated metrics. The overhead proportional value (4: 2: 5: 7) of four sockets results in nearly 25%, 50%, 15% and 10% memory page allocation from socket 0 to socket 3, instead of only initializing 1,048,576 pages on the local socket. From Figure 8, it is observed that *YCSB* and *Memcached* benefit the most (nearly 52.1% and 46.0% in 16-VMs case). Both benchmarks continuously send random irregular memory requests based on clients' read/update operations, which take advantage of these distributed memory regions with the least access overhead to minimize access latency. *TunkRank* and *NPB/UA* behave differently: they read a number of memory blocks and then perform data calculations and there are fewer memory requests during their execution.

6.2. Evaluation of P2M swap FIFO and alternative methods

This section verifies the effectiveness of the P2M swap FIFO and explores the adaptability of our techniques to other methods (such as vCPU migration). We configure the experiment on a 4-socket physical host with 8-VMs. All four architectural overheads exist in this scenario. We also implement a dynamic vCPU migration policy in the management domain. It works as follows: the daemon process leverages our architectural overhead evaluation module and dynamically pins vCPUs to physical cores on the nearest and less stressed sockets. We perform five comparison scenarios containing various combinations of *NUMA-aware*, *NUMA overhead-aware*, P2M swap FIFO, and vCPU migration approaches. Figure 10 presents our evaluation results. On aver-

age, the P2M swap FIFO further improves the *NUMA overhead-aware* approach by 18.3% across the four benchmarks. Since our *NUMA overhead-aware* allocator selects a set of memory nodes exhibiting similar overhead, it is possible that a virtual machine occupies several sockets. As the number of VMs that share one memory node increases, the probability that they access the same node concurrently increases, resulting in data traffic. The P2M swap FIFO, serving as a VM "internal" memory balancer, gradually exchanges memory to low overhead nodes to minimize access penalties. As previously explained, *YCSB* and *Memcached* benefit the most; achieving 18.5% and 31.2% improvement while performance of *NPB/UA* only increases 2.8%.

It is well known that both CPU scheduling and memory management contribute to performance degradation in NUMA machines. From Figure 9, it is observed that the *NUMA-aware* method works well with vCPU migration, leading to a 13.9% performance improvement on average. Similarly, our approaches don't conflict with vCPU migration. Optimizations combing *NUMA overhead-aware*, P2M swap FIFO and vCPU migration outperform the case without vCPU migration by 4.6% across all workloads. The small performance increase is due to the fact that our mechanisms have already increased performance by 41.1% when compared to the *NUMA-aware* optimization. Therefore, we conclude that *NUMA overhead aware* + P2M swap FIFO memory optimizations can cooperate well with vCPU scheduling technique and a full-fledged design and implementation will be our future work.

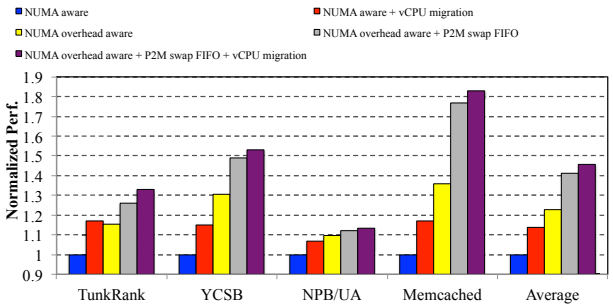


Figure 10 Performance comparison among various optimizations. Application's performance is normalized to NUMA-aware method.

6.3. Scalability of VM memory size

In this experiment, we vary the VM memory from 4GB to 32GB. The experimental setup is the same as before: 8-VMs consolidated on a four-socket host combining the four architectural overheads. In Table 2, "Default" means *NUMA-aware* approach while "Optimized" refers to *NUMA overhead-aware* memory allocation + P2M swap FIFO.

For *NUMA-aware* approach, as memory size of the VM increases, the memory of the local node can be easily exhausted. Therefore, the allocator randomly chooses one socket with available memory. This may cause unpredictable request congestion. However, our optimizations always distribute data traffic across low overhead nodes, which improve performance. It is expected that the performance gap between these two mechanisms will increase when scaling the memory size. For example, on *Memcached*, the performance difference rises from 6.1% to 17.7% when the memory size is increased. Similarly, *YCSB* manifests a 1525 rps/s throughput difference in 32GB memory compared to 1213 rps/s with 4GB memory. Note that *NPB/UA* issues dynamic and irregular memory requests. Since we use a standard C class data set with 3.35×10^4 elements, it can't consume such a large memory space, resulting in nearly stable performance when increasing memory size.

6.4. Scalability of physical host's sockets

This section explores the scalability of our optimization with an increase in the number of sockets on the physical host. We use a Dell 2U PowerEdge R710 server as the 2-socket machine and configure the IBM x3850 system to 4-socket and 8-socket hosts respectively. To fully evaluate the system, the 2-socket server is consolidated with 4-VMs while the 4-socket and 8-scoket hosts are deployed with 8- and 16- VMs. Each VM has 2 vCPUs and 4GB memory. We use the same VMs setup combining 4 architectural overheads. In Table 3, "Default" means *NUMA-aware* approach while "Optimized" stands for *NUMA overhead-aware* memory allocation + P2M swap FIFO.

When more sockets exist in the system, our *NUMA overhead-aware* memory allocator has more choices for selecting low-overhead sockets. Additionally, as a VM spreads across more sockets, the P2M swap FIFO can easily balance data traffic internally with memory exchange operations. Therefore, our optimizations outperform the default scheme. For example, in the dual-socket scenario, the proposed mechanism performs 7.9% better than the *NUMA-aware* method across four benchmarks while this performance gap rises to 14.1% and 2X for 4-socket and 8-socket cases. *YCSB* benefits the most as it achieves a 5X improvement with 8 sockets since the *NUMA-aware* method squeezes all VMs' memory to the nearest nodes, leading to high last-level cache contention and memory controller congestion while our mechanism successfully mitigates this overhead.

6.5. Overhead discussion

Sources of overhead for our optimized schemes are: data storage and operation latency. The NUMA overhead-aware buddy allocator requires 442 bytes (includes 16-entries slide window for 3 architectural metrics) for each domain. For each VM, the P2M swap FIFO allocates 1KB space for every socket. In the case where there are 8-VMs, each with 4GB memory spread across 4 sockets; the total memory space requirement is less than 7.46KB, which is negligible. Since the optimized buddy allocator incurs overhead due to NUMA overhead detection, node selection, and proportional distribution, VM management performance could experience some degradation. For example, the "xm create" operation requires additional 0.46s.

	TunkRank (s)		YCSB (rqs/s)		NPB/UA (s)		Memcached (rqs/s)	
	Default	Optimized	Default	Optimized	Default	Optimized	Default	Optimized
4G	154.5	140.3 (9.2%)	6262.0	7475.0 (19.4%)	473.2	464.9 (1.8%)	7195.0	7634.0 (6.1%)
8G	153.1	142.1 (7.2%)	6076.3	7295.2 (20.1%)	476.4	462.9 (2.8%)	6627.3	7124.4 (7.5%)
16G	156.2	142.4 (8.8%)	6121.7	7432.8 (21.4%)	475.9	464.1 (2.5%)	6590.8	7623.6 (15.7%)
32G	158.5	142.3 (10.2%)	6080.4	7605.3 (25.1%)	475.6	464.1 (2.4%)	6660.1	7841.0 (12.3%)

Table 2: Performance variation among 4 benchmarks while increasing VM memory size. The percentage numbers are performance improvement of the proposed techniques.

	TunkRank (s)		YCSB (rqs/s)		NPB/UA (s)		Memcached (rqs/s)	
	Default	Optimized	Default	Optimized	Default	Optimized	Default	Optimized
2 Sockets	122.4	118.9 (2.9%)	8274.0	9373.3 (13.3%)	440.3	432.5 (1.8%)	9555.0	10844.9 (13.5%)
4 Sockets	153.4	138.6 (9.7%)	6347.1	7578.4 (19.4%)	479.4	450.6 (6.0%)	6432.8	7719.4 (20.0%)
8 Sockets	184.7	143.3 (22.4%)	1133.0	5597.5 (4.9X)	530.0	467.5 (11.8%)	6303.0	7354.0 (16.7%)

Table 3: Performance comparison among 4 benchmarks when scaling out the number of sockets on the physical host. The percentage numbers are performance improvement of the proposed techniques.

7. Related Work

This study addresses the NUMA access overhead problem in virtualized systems via optimizing hypervisor memory management. Our research study spans several areas from cloud workloads and virtual machine consolidation to dynamic memory management and NUMA system optimizations.

Cloud workload study. Cloud computing has emerged as one dominant computing paradigm to deliver scalable online services ranging from web search to social networks. Its flexible communication workflow and explosive growth of large volume data sets bring new challenges for traditional system design and optimization. Lately, the architecture research community has been emphasizing the analysis of cloud workloads characteristics and redesigning conventional mechanisms to adapt to these new features. At the processor micro-architecture level, Ferdman [30] et al. firstly introduced a set of scale-out workloads, *CloudSuite*, to identify inefficiencies in today's multiprocessors when executing these applications; Atta [24] focused on instruction stalls resulting from large instruction footprint in online transaction processing (OLTP) workloads and proposed a hardware method to improve instruction reuse in first level cache. At the server system level, Basu [23] aimed to replace page-based virtual memory with a *direct segment* method to alleviate TLB miss overhead in big memory systems. The irregular and large memory footprint feature of multiple consolidated VMs in virtualized cloud also motivates us to explore the integration of memory access overhead estimation with hypervisor memory management for NUMA machine.

Virtual machine consolidation. Virtualization enables multiple virtual machines running different applications to share one physical system, which is widely used in cloud computing platform. Performance interference, which affects application's quality-of-service (QoS), is a critical performance issue when applying consolidation and deserves comprehensive analysis and optimization. To this end, Nathuji [33] developed a QoS-aware control framework to reserve suitable resources for workloads. Paul [34] performed a detailed characterization of co-locating different types of VMs under various core placement schemes and proposed an interference metric and regression model. Our study focuses on performance improvement via hypervisor memory management, especially for high VM consolidation scenario.

Dynamic memory management. Disco [35] and VMware ESX [36] are two typical systems providing dynamic memory management. Disco implements a dynamic page migration and a page replication system for CC-NUMA machines to maintain locality between a virtual CPU's cache misses and the associated memory pages to which the cache misses occur. VMware ESX [36] applies a statistical sampling approach to obtain aggregate VM working set directly without any guest involvement. This accurate estimation of the fraction of memory in active use results in the system responding rapidly to memory usage increases while more gradually to the decrease of memory utilization. Our performance counter driven memory management approach, which is motivated by these systems, differs from them in two aspects: (1) instead of maximizing local memory accesses, our optimization also evaluates three other NUMA architecture overheads (last level cache contention, memory controller congestion, and interconnection congestion); (2) We use various performance counters (cache miss, cycle loss, and IPC) online to identify and estimate each overhead respectively.

NUMA system optimization. How to leverage NUMA systems' architecture features and minimize memory request overhead is an interesting research topic. Prior studies can be found in both non-virtualized and virtualized environments.

In non-virtualized systems, Lachaize [37] developed the first NUMA memory profile (MemProf) based on temporal flows of interactions between threads and in-memory objects. The goal is to provide precise and valuable information for multithreaded execution in NUMA multicore machines. Zhuravlev [38] addressed shared resource contention among multicore processors via DI and DIO algorithms. They [19] further extended the contention analysis to NUMA systems and proposed DINO NUMA-aware management algorithm. Dashti [7] investigated the overhead of NUMA systems and concluded that traffic congestion (memory controller and interconnection) is the major overhead.

In virtualized environments, NUMA optimization involves coordination between vCPU scheduling and memory management. Rao [12] proposed using a "un-core" penalty

as a performance index to dynamically determine the optimal vCPU-to-core assignment for NUMA sensitive virtual machines. Recently, the Xen hypervisor incorporated NUMA-aware VM memory initialization with three heuristics (i.e., *greedy*, *packed*, and *spread*) placement policies. In addition, VMware ESX [14] supports VM dynamic rebalancing and intelligent memory migration mechanisms: it transparently moves one VM to the least-loaded node and gradually migrates memory from the original socket to a new one to eliminate remote memory access penalty. Nevertheless, all of these mechanisms only consider remote memory access overhead and use the VM consolidation ratio to trigger balancing and migration operations. Our study profiles and incorporates four NUMA architectural overhead estimations with VM memory management. Furthermore, we verify that our *NUMA overhead-aware* allocator + P2M swap FIFO techniques can work seamlessly with vCPU scheduling optimizations. Note that the studies on extending NUMA topology to the VM guest [39, 40] (such as *vNUMA-mgr*) are orthogonal to the optimization proposed in this paper.

8. Conclusions

This work explores performance optimization opportunities when consolidating cloud workloads in NUMA virtualized systems. Based on our comprehensive characterization results from multiple experimental scenarios, three techniques are proposed: memory zone access overhead estimation, a NUMA overhead-aware buddy allocator, and a P2M swap FIFO. The overhead analysis approach takes the cache hit rate, the cycle loss due to cache misses and the IPC as indicators to analyze architecture overhead. Our main idea is to leverage NUMA overhead awareness in the hypervisor's memory management. The optimized allocator reserves memory pages across selected low-overhead memory nodes. The P2M swap FIFO provides mapping exchange when memory pressure is unbalanced across guest's different nodes. Our prototyped implementation shows notable performance improvements when consolidating multiple virtual machines on a real-world server (IBM x3850) system. We perform further evaluation to examine its adaptability to other approaches, scalability among different VMs and physical hosts, and feasibility in terms of operation latency and storage overhead.

Acknowledgements

We thank the anonymous reviewers and our shepherd, Boris Grot for their help and feedback. This work is supported in part by NSF grants 1320100, 0845721(CAREER), and by Microsoft Research Safe and Scalable Multi-core Computing Awards. Ming Liu is also supported by University of Florida Graduate Fellowship.

References

[1] Server Consolidation Benefits, http://www.vmware.com/solutions/consolidation/consolidate.html
[2] Amazon EC2, http://aws.amazon.com/ec2/
[3] Windows Azure, http://www.windowsazure.com/en-us/
[4] The 2012 Uptime Institute Data Center Industry Survey, The Uptime Institute, 2012
[5] Intel QPI, "An Introduction to the Intel QuickPath Interconnect", White Paper, 2009
[6] AMD HyperTransport, http://en.wikipedia.org/wiki/HyperTransport
[7] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema and M. Roth, "Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems", In *ASPLOS*, 2013
[8] SPEC CPU2006, http://www.spec.org/cpu2006/
[9] NoSQL Database, http://nosql-database.org
[10] Apache Hadoop, http://hadoop.apache.org
[11] OLTP-Bench, http://oltpbenchmark.com/wiki/index.php?title=Main_Page
[12] J. Rao, K. Wang, X. Zhou and C. Xu, "Optimizing Virtual Machine Scheduling in NUMA Multicore System", in *HPCA*, 2013
[13] Scheduling and Placement of NUMA in Xen System, http://wiki.xen.org/wiki/Xen_Numa_Scheduling_and_Placement
[14] VMware ESX Server 2 NUMA Support, White Paper
[15] VMware ESX, http://www.vmware.com/products/vsphere-hypervisor
[16] Citrix XenServer, http://www.citrix.com/products/xenserver/overview.html
[17] Dell PowerEdge Server, www.dell.com/PowerEdge
[18] HP Proliant Server, www.hp.com/go/proliant
[19] S. Blagodurov, S. Zhuravlev, M. Dashti and A. Fedorova, "A Case for NUMA-aware Contention Management on Multicore Systems", In *USENIX ATC*, 2011
[20] R. Nikolaev and G. Back, "Perfctr-Xen: A Framework for Performance Counter Virtualization", in *VEE*, 2011
[21] Intel whitepaper, "First the Tick, Now the Tock: Next Generation Intel Microarchitecture (Nehalem)", 2008
[22] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt and A. Warfield, "Xen and the Art of Virtualization", in *SOSP*, 2003
[23] A. Basu, J. Gandhi, J. Chang, M. D. Hill and M. M. Swift, "Efficient Virtual Memory for Big Memory Servers", in *ISCA*, 2013
[24] I. Atta, P. Tozun, X. Tong, A. Ailamaki and A. Moshovos, "STREX: Boosting Instruction Cache Reuse in OLTP Workloads Through Stratified Transaction Execution", in *ISCA*, 2013
[25] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB", in *SoCC*, 2010
[26] Apache Cassandra, http://cassandra.apache.org
[27] Memcached, http://memcached.org
[28] NAS Parallel Benchmarks, http://www.nas.nasa.gov/publications/npb.html
[29] GraphLab, http://graphlab.org
[30] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki and B. Falsafi, "Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware", in *ASPLOS*, 2012
[31] Intel Performance Counter Monitor, http://software.intel.com/en-us/articles/intel-performance-counter-monitor-a-better-way-to-measure-cpu-utilization
[32] Bin packing problem, http://en.wikipedia.org/wiki/Bin_packing_problem
[33] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-Clouds: Managing Performance Interference Effects for QoS-Aware Clouds", in *Eurosys*, 2010
[34] I. Paul, S. Yalamanchili, and L. K. John, "Performance Impact of Virtual Machine Placement in a Datacenter", in *IPCCC*, 2012
[35] E. Bugnion, S. Devine, and M. Rosenblum, "Disco: Running Commodity Operating Systems on Scalable Multiprocessors", in *SOSP*, 1997
[36] C. A. Waldspurger, "Memory Resource Management in VMware ESX Server", in *OSDI*, 2002
[37] R. Lachaize, B. Lepers and V. Quema, "MemProf: a Memory Profiler for NUMA Multicore Systems", in *USENIX ATC*, 2012
[38] S. Zhuravlev, S. Blagodurov and A. Fedorova, "Addressing Shared Resource Contention in Multicore Processors via Scheduling", in *ASPLOS*, 2010
[39] D. S. Rao and K. Schwan, "vNUMA-mgr: Managing VM Memory on NUMA Platforms", in *HiPC*, 2010
[40] Q. Ali, V. Kiriansky, J. Simons and P. Zaroo, "Performance Evaluation of HPC Benchmark on VMware's ESXi Server", in *ICPP*, 2011