Dynamic Virtual Machine Scheduling in Clouds for Architectural Shared Resources

Jeongseob Ahn, Changdae Kim, Jaeung Han, Young-ri Choi†, and Jaehyuk Huh
KAIST †*UNIST*
{jeongseob, cdkim, juhan, and jhuh}@calab.kaist.ac.kr ychoi@unist.ac.kr

Abstract

Although virtual machine (VM) migration has been used to avoid conflicts on traditional system resources like CPU and memory, micro-architectural resources such as shared caches, memory controllers, and non-uniform memory access (NUMA) affinity, have only relied on intra-system scheduling to reduce contentions on them. This study shows that live VM migration can be used to mitigate the contentions on micro-architectural resources. Such cloud-level VM scheduling can widen the scope of VM selections for architectural shared resources beyond a single system, and thus improve the opportunity to further reduce possible conflicts. This paper proposes and evaluates two cluster-level virtual machine scheduling techniques for cache sharing and NUMA affinity, which do not require any prior knowledge on the behaviors of VMs.

1 Introduction

In cloud systems based on virtualization, virtual machines (VM) share physical resources. Although resource sharing can improve the overall utilization of limited resources, contentions on the resources often lead to significant performance degradation. To mitigate the effect of such contentions, cloud systems use dynamic rescheduling of VMs with live migration technique [3], changing the placement of running VMs. However, such VM migration has been used to resolve conflicts or balance load on traditional allocatable system resources such as CPUs, memory, and I/O sub-systems. VM migration can be triggered by monitoring the usages of these resources for VMs in a cloud system [4, 8].

In the meantime, the advent of multi-cores has enabled the sharing of micro-architectural resources such as shared caches and memory controllers. Contention on such micro-architectural resources has emerged as a major reason for performance variance, as an application can be affected by co-running applications even though it receives the same share of CPU, memory, and I/O. For a single system, there have been several prior studies to mitigate the impact of contention on shared caches and memory controllers by carefully scheduling threads [2, 9]. The prior studies rely on the heterogeneity of memory behaviors of applications within a system boundary. The techniques group applications to share a cache to minimize the overall cache misses for a system. However, if a single system runs applications with similar cache behaviors, such intra-system scheduling cannot mitigate contentions.

However, cloud systems with virtualization open a new opportunity to widen the scope of contention-aware scheduling, as virtual machines can cross legacy system boundaries with live migration. In this paper, we use live VM migration to dynamically schedule VMs for minimizing the contention on shared caches and memory controllers. Furthermore, this study considers the effects of non-uniform memory accesses (NUMA) in multi-socket systems commonly used in cloud servers.

We propose contention-aware cloud scheduling techniques for cache sharing and NUMA affinity. The techniques identify the cache behaviors of VMs on-line, and dynamically migrate VMs, if the current placements of VMs are causing excessive shared cache conflicts or wrong NUMA affinity. Since the techniques identify the VM behaviors dynamically and resolve conflicts with live migration, they do not require any prior knowledge on the behaviors of VMs. The first technique, *cache-aware cloud scheduling* minimizes the overall last-level cache (LLC) misses in a cloud system. The second technique, *NUMA-aware cloud scheduling* extends the first technique by considering NUMA affinity.

We evaluate our proposed schedulers using selected SPECcpu 2006 applications in various combinations. The experimental results show that the cache-aware scheduler can significantly improve the performance compared with the worst case. With our preliminary NUMA optimization, the performance is slightly im-

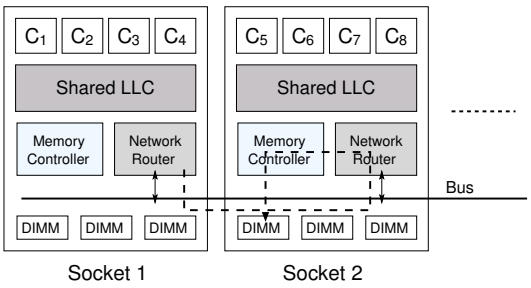


Figure 1: Shared caches and NUMA

proved for our benchmark applications, compared with that of the cache-aware scheduler.

2 Motivation

2.1 Cache Sharing and NUMA Affinity

Although shared caches can potentially improve the efficiency of caches with dynamic capacity sharing among cores, they also incur contention problems when one of cores generates excessive cache misses and evicts the cached data from the other cores. Figure 1 shows a common multi-core system with multiple sockets. In each socket, there is a shared last-level cache (LLC), and memory accesses across different sockets have longer latencies than those to the local socket.

There have been several studies to mitigate such negative interferences in shared caches with partitioning caches [6, 7] or carefully scheduling threads [5, 9]. In the scheduling-based solutions [9], threads are grouped and mapped to different sockets, aiming to minimize the sum of cache misses from all the shared LLCs. In the scheduling policy for a system with two sockets, threads are sorted by their LLC misses, and grouped into two sets with equal or similar sums of LLC misses. Minimizing the differences of LLC misses among the last-level caches reduces the overall LLC misses in the system.

However, NUMA affinity complicates such cache-aware scheduling. If an application is running on a socket different from the socket in which its memory pages reside, the cost of LLC misses will increase due to the NUMA effect. Therefore, scheduling to minimize the overall cache misses must also consider possible NUMA effects. Blagodurov et al. investigated the impact of NUMA on cache-aware scheduling to reduce negative interferences among threads [2]. In virtualized systems, some commercial hypervisor provides dynamic page migration to reduce memory access latencies for VMs, but it does not consider cache sharing effect [1].

The prior studies use thread scheduling in a single system to reduce the shared cache contention and negative NUMA effects. Such intra-system scheduling limits the

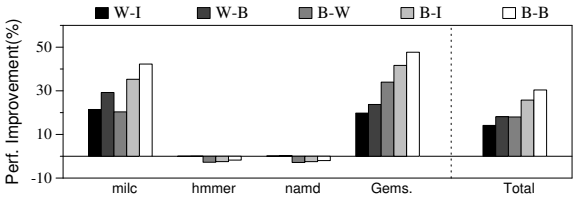


Figure 2: Performance improvements over W−W

opportunity to search the best groups of threads sharing an LLC within a system, which has only several or tens of cores at most. However, in a virtualized cloud system composed of a large number of nodes, VMs can migrate across physical system (or node) boundaries, potentially increasing the chance to find a better grouping of VMs for shared LLCs, and to support NUMA affinity.

2.2 Performance Implication in Clouds

In this section, we quantitatively show the performance implication of cache sharing and NUMA affinity in a small scale cloud system. We use a 4-node cluster with 8 cores in two sockets for each node. The details of the experiments are shown in Section 4.1. Figure 2 presents the performance of a mixed workload from 4 application types on the cluster, with 8 VM instances for each application type.

In this figure, we present six different VM mapping policies for cache sharing and NUMA affinity. For the cache sharing aspect, the best case (B) is to map VMs to cores such that the sum of LLC misses from all the sockets in the cloud system is minimized. The worst case (W) is the mapping with the highest difference between the largest and smallest per-socket LLC misses in the cloud system. For the NUMA affinity, we present three mapping policies. The worst case is that the memory pages of all VMs are allocated in their remote sockets, while the best case is that all the VM memory pages are allocated in their local sockets. Also, we show an interleaved allocation (I), which assigns the memory pages of a VM to be always in both sockets in an interleaved way.

The figure shows the combinations of cache and NUMA-aware VM scheduling policies. In each combination, the first letter denotes the cache policy and the second letter denotes the NUMA policy. For example, `B-I` represents a VM scheduling policy which is the best case for cache sharing, and the interleaved case for NUMA. The figure presents performance improvements of the policies over the `W-W` case.

As expected, there is significant potential for performance improvement by placing VMs considering architectural shared resources. The `B-B` case significantly improves the performance compared with `W-W`. Especially, `milc` and `GemsFDTD` could increase their performance by sharing caches with other less memory-

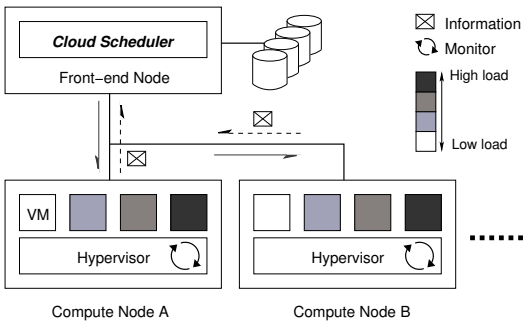


Figure 3: Memory-aware Cloud Scheduler

intensive workloads. On the other hand, `hmmer` and `namd` have no improvements, since these do not require high capacity for LLCs. The NUMA affinity also affects the overall performance significantly. Even for the best case for shared caches, a good NUMA policy can improve the performance, especially for memory-intensive workloads (`milc` and `GemsFDTD`). If optimizing the NUMA affinity is not possible, the interleaved memory allocation could be an alternative solution to avoid the worst case for NUMA.

As shown in the results, the performance variance due to VM scheduling can be large even in a small scale cluster. In public clouds, supporting consistent performance regardless of co-running VMs is critical. In a large scale cloud system, the heterogeneity of VM cache behaviors across different nodes, is expected to grow, as various users will share the cloud system. Exploiting such heterogeneity of cache behaviors, memory-aware cloud-level scheduling can potentially improve the efficiency of shared cache and NUMA affinity, avoiding the worst case scheduling.

3 Memory-Aware Cloud Scheduling

In this section, we present our memory-aware cloud scheduling techniques. For memory-aware scheduling, the cloud scheduler collects the cache behavior of each VM from computing nodes, and migrates VMs if such migration can potentially reduce the overall cache misses and the average memory access latencies by NUMA affinity in the cloud system. Figure 3 describes the overall architecture of the memory-aware cloud scheduler. In each computing node, a monitor checks LLC misses with hardware performance monitoring counters, and periodically sends the per-VM LLC miss and NUMA affinity information to the cloud scheduler. Based on the VM status information from all the nodes, the cloud scheduler makes global scheduling decisions.

We explore two scheduling policies for the memory-aware cloud scheduler. Firstly, the *cache-aware scheduler* only considers the contentions on shared caches, ignoring the NUMA effect. The policy will group VMs to minimize the overall LLC misses in the entire cloud system, even if the grouping can violate NUMA affinity. Secondly, the *NUMA-aware scheduler* extends the cache-aware scheduler for supporting NUMA affinity.

One of the advantages of the proposed memory-aware schedulers is that they use only the information of VMs measured on-line, without previous knowledge on the VMs. The memory-aware cloud schedulers initially place VMs on computing nodes, only considering CPU and memory availability for each node. However, they dynamically identify the cache behaviors of the VMs, and re-locate them to improve the memory behavior.

Algorithm 1 Cache-aware scheduler (pseudo code)

$P_{List} =< pm_1, ..., pm_n >$ // LLC misses of all compute nodes
$V_{List} =< vm_1, ..., vm_k >$ // LLC misses of VMs in a node

/* Step1: local phase */
for each node i in 1 ... n **do**
  // gather LLC misses for all VMs in node i
  $pm_i \Leftarrow$ **gather** (i)
  $V_{List} \Leftarrow$ **sort** (pm_i)
  // distribute the VMs across sockets with even LLC misses
  distribute (V_{List})
end for

/* Step2: global phase */
// find nodes with the largest and smallest LLC misses
$maxNode \Leftarrow$ **findMaxNode** (P_{List})
$minNode \Leftarrow$ **findMinNode** (P_{List})
// find VMs with largest and smallest misses from two nodes
$maxVM \Leftarrow$ **findMaxVM** ($maxNode$)
$minVM \Leftarrow$ **findMinVM** ($minNode$)

if $maxNode_{LLC} - minNode_{LLC} > threshold$ **then**
  swap ($maxVM, minVM$)
end if

Cache-Aware Scheduler: The cache-aware scheduler migrates VMs to minimize the overall LLC misses in the cloud system. It consists of local and global scheduling phases. In the local phase, VMs in each computing node are grouped and scheduled to shared cache domains (commonly sockets) in the node. Since VM migrations across physical nodes consume network bandwidth and computational capability, we attempt to minimize such VM migration by optimizing VM scheduling within a node first. In the global phase, the cloud scheduler attempts to re-distribute VMs to have even LLC misses in all the nodes in the cloud system.

Algorithm 1 presents the cache-aware scheduling with the two phases. In the local phase, VMs in each node are sorted by LLC misses, and then grouped to make each LLC have even misses. We use the same simple algorithm used by Zhuravlev et al [9]. For example, for a node with two shared cache domains, the VM with the largest number of LLC misses is assigned to the first

group, and the second VM is assigned to the second group. Among the remaining VMs, the VM with the smallest number of LLC misses is assigned to the first group, and the VM with the second smallest number of LLC misses is assigned to the second group. This continues until all VMs are assigned to one of the two groups.

In the global phase, the scheduler finds two nodes, in the cloud system, with the largest and smallest numbers of LLC misses. From the two nodes, it finds two VMs with the largest and smallest numbers of LLC misses, respectively. If their LLC miss difference is larger than a threshold, the two VMs are swapped by live migration. The scheduler periodically executes the two-phase scheduling to gradually reduce the overall LLC misses in the cloud system.

Algorithm 2 NUMA-aware scheduler (pseudo code)

$S_{List} =< sock_{1,1}, ..., sock_{n,m} >$ // LLC misses of all sockets

for each node i in 1 ... n **do**
  for each socket j in 1 ... m in node i **do**
    // gather LLC misses for all VMs in $sock_{i,j}$
    $sock_{i,j} \Leftarrow$ **gather** (i, j)
  end for
end for

// find sockets with the largest and smallest LLC misses
$maxSocket \Leftarrow$ **findMaxSocket** (S_{List})
$minSocket \Leftarrow$ **findMinSocket** (S_{List})
// find VMs with largest and smallest misses from 2 sockets
$maxVM \Leftarrow$ **findMaxVM** ($maxSocket$)
$minVM \Leftarrow$ **findMinVM** ($minSocket$)

if $maxSocket_{LLC} - minSocket_{LLC} > threshold$ **then**
  swap ($maxVM, minVM$)
end if

NUMA-Aware Scheduler: The NUMA-aware scheduler considers the NUMA affinity in addition to the reduction of LLC misses in the cloud system. Unlike the cache-aware scheduler, the NUMA-aware scheduler only provides global scheduling as the local scheduling can potentially break the NUMA affinity of VMs. In the cache-aware scheduling, the local scheduling can assign a VM to a socket different from the one, in which it was initially created and thus its memory pages reside, for minimizing LLC misses within a computing node.

Initially, the memory pages of a VM are allocated only one of the socket in a node. Such NUMA-aware memory allocation is supported by the Xen hypervisor. From the initial placement, the NUMA-aware scheduler migrates VMs to different sockets to reduce the overall LLC misses. Algorithm 2 presents the NUMA-aware scheduler with only a global phase. For each scheduling iteration, the scheduler selects two sockets, in the cloud system, with the largest and smallest numbers of LLC misses. Among the two sockets, two VMs with the largest and smallest numbers of LLC misses are selected respectively, and the two VMs may be swapped. As this socket exchange moves the VMs including their memory pages, their NUMA-affinity is still maintained.

	Memory bound		CPU-bound	
1	GemsFDTD	milc	hmmer	namd
2	omnetpp	lbm	gobmk	sjeng
	Memory bound			CPU-bound
3	cactusADM	gcc	soplex	namd
	Memory bound	CPU-bound		
4	libquantum	tonto	povray	sjeng
	Memory bound			
5	cactusADM	milc	omnetpp	soplex
	CPU bound			
6	gobmk	sjeng	namd	povray

Table 1: Selected workloads from SPECcpu 2006

For further improvements, within a physical node, only the hot pages, which are frequently accessed by a VM can be migrated to different sockets, instead of migrating entire pages belong to the VM. The hot page supports can allow less expensive local scheduling even for the NUMA-aware scheduler. The NUMA improvements based on hot page migrations will be our future work.

4 Evaluation

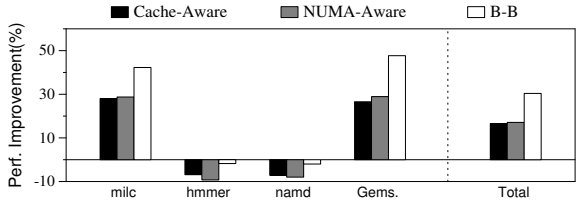
4.1 Methodology

We have implemented the proposed schedulers running in a separate cloud manager node. Each computing node is virtualized with the open source Xen hypervisor. Each node runs a monitoring tool, which records LLC misses for VMs and periodically sends the miss and NUMA affinity information to the cloud scheduler.

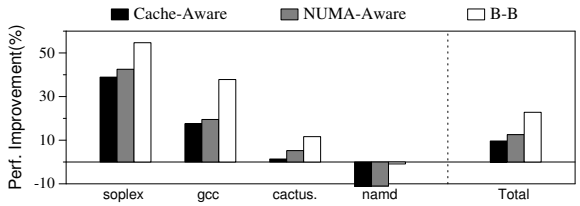
On top of the Xen hypervisor, each node runs 8 guest VMs, which use a Ubuntu distribution based on Linux kernel 2.6.18. In our small scale testbed, there are 4 physical nodes with total 32 VMs. Each physical node has 8 cores placed on two chips (sockets) and each socket on a node has a 12MB L3 cache shared by 4 cores. In the dual-socket system, memory access latencies to the local socket and remote sockets are different. Each VM employs a single core and 1GB guest physical memory size. Table 1 presents our benchmark applications. We create 6 workloads by mixing applications with various memory characteristics. Each workload has 4 different applications, and 8 instances of each application run on the 32 VMs in our testbed.

4.2 Performance Improvements

Figure 4 shows the performance improvements for individual applications on WL1 and WL3, respectively. The



(a) WL1: 2 Memory bound + 2 CPU bound



(b) WL3: 3 Memory bound + 1 CPU bound

Figure 4: Performance improvements over W−W

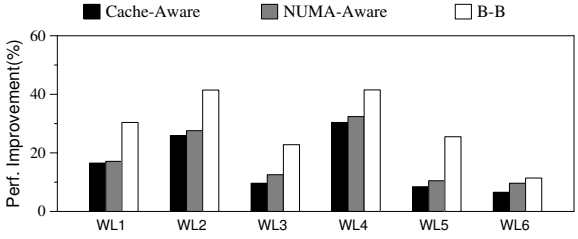


Figure 5: Performance improvements over W−W

bars show the performance improvements over the worst case (W−W), as described in Section 2.2. The cache-aware scheduler can significantly improve the performance compared with the worst case. On WL1, which consists of 2 memory and 2 CPU bound applications, the overall performance is improved by 17%. Note that the improved performance includes the overhead of VM migrations. The cache-aware scheduling significantly improves the performance of memory-bound applications, `milc` and `GemsFDTD`, which incur a large number of LLC misses, while it degrades the other two applications slightly. Although `hmmer` and `namd` are not very sensitive to the cache capacity, they have to share LLCs with the memory-intensive applications.

Our preliminary NUMA-aware scheduler improves the overall performance slightly compared with the cache-aware scheduler. As the initial design removes the local scheduling and relies only on global scheduling to maintain NUMA affinity, the cloud system slowly adjusts VM placements. Furthermore, the global migration consumes CPU resources to copy VM memory to another physical node. A local NUMA-aware scheduling, which migrates only hot pages of a VM to a different socket within a single node, can reduce unnecessary global VM migration to maintain NUMA affinity.

Figure 5 summarizes the performance improvements for our six workloads, showing similar trends except for WL6. WL6 consists of all CPU-bound applications, and thus does not benefit from memory-aware scheduling.

5 Conclusions and Future Work

In this paper, we proposed and evaluated memory-aware cloud scheduling techniques, which do not require any prior knowledge on the behaviors of VMs. This paper shows that VM live migration can also be used to mitigate micro-architectural resource contentions, and the cloud-level VM scheduler must consider such hidden contentions. We plan to extend our preliminary design of NUMA-aware scheduling for more efficient NUMA affinity supports with hot page migrations. Also, we will investigate a systematic approach based on a cost-benefit analysis for VM migrations and contention reductions.

Acknowledgments

This research was supported by the SW Computing R&D Program of KEIT(2011-10041313, UX-oriented Mobile SW Platform) funded by the Ministry of Knowledge Economy.

References

[1] VMware ESX Server 2 NUMA Support. White paper. .

[2] BLAGODUROV, S., ZHURAVLEV, S., MOHAMMAD, D., AND FEDOROVA, A. A case for numa-aware contention management on multicore processors. In *Proceedings of the USENIX Annual Technical Conference* (2011).

[3] 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* (2005).

[4] GULATI, A., SHANMUGANATHAN, G., HOLLER, A., AND AHMAD, I. Cloud-scale resource management: challenges and techniques. In *Proceedings of the 3rd USENIX conference on Hot topics in cloud computing* (2011).

[5] MERKEL, A., STOESS, J., AND BELLOSA, F. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th European conference on Computer systems* (2010).

[6] QURESHI, M. K., AND PATT, Y. N. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (2006).

[7] SUH, G. E., DEVADAS, S., AND RUDOLPH, L. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture* (2002).

[8] WOOD, T., SHENOY, P., VENKATARAMANI, A., AND YOUSIF, M. Black-box and gray-box strategies for virtual machine migration. In *Proceedings of the 4th USENIX conference on Networked systems design and implementation* (2007).

[9] ZHURAVLEV, S., BLAGODUROV, S., AND FEDOROVA, A. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the 15th International Conference on Architectural support for programming languages and operating systems* (2010).