A User-Level NUMA-Aware Scheduler for Optimizing Virtual Machine Performance

Yuxia Cheng, Wenzhi Chen, Xiao Chen, Bin Xu, and Shaoyu Zhang

College of Computer Science and Technology, Zhejiang University, Hangzhou, China
{rainytech,chenwz,chunxiao,xubin,zsy056}@zju.edu.cn

Abstract. Commodity servers deployed in the data centers are now typically using the Non-Uniform Memory Access (NUMA) architecture. The NUMA multicore servers provide scalable system performance and cost-effective property. However, virtual machines (VMs) running on NUMA systems will access remote memory and contend for shared on-chip resources, which will decrease the overall performance of VMs and reduce the efficiency, fairness, and QoS that a virtualized system is capable to provide. In this paper, we propose a "Best NUMA Node" based virtual machine scheduling algorithm and implement it in a user-level scheduler that can periodically adjust the placement of VMs running on NUMA systems. Experimental results show that our NUMA-aware virtual machine scheduling algorithm is able to improve VM performance by up to 23.4% compared with the default CFS (Completely Fair Scheduler) scheduler used in KVM. Moreover, the algorithm achieves more stable virtual machine performance.

Keywords: NUMA, virtual machine, scheduling, memory locality.

1 Introduction

Multicore processors are commonly seen in today's computer architectures. However, a high frequency (typically 2-4 GHz) core often needs an enormous amount of memory bandwidth to effectively utilize its processing power. Even a single core running a memory-intensive application will find itself constrained by memory bandwidth. As the number of cores becomes larger, this problem becomes more severe on Symmetric Multi-Processing (SMP) systems, where many cores must compete for memory controller and bandwidth in a Uniform Memory Access (UMA) manner. The Non-Uniform Memory Access (NUMA) architecture is then proposed to alleviate the constrained memory bandwidth problem as well as to increase the overall system throughput.

Commodity servers deployed in today's data centers are now typically using the Non-Uniform Memory Access (NUMA) architecture. The NUMA system links several small and cost-effective nodes (known as NUMA nodes) via the high-speed interconnect, where each NUMA node contains processors, memory controllers, and memory banks. The memory controller on a NUMA node is responsible for the local NUMA node memory access. An application accessing

C. Wu and A. Cohen (Eds.): APPT 2013, LNCS 8299, pp. 32–46, 2013.

remote NUMA node memory requires the remote memory controller to fetch the data from remote memory banks and send back the data through the high-speed interconnect, thus the latency of accessing remote node memory is larger than accessing local node memory.

The difference of memory access latency between local NUMA node and remote NUMA node will severely impact an application's performance, if the application is running on one NUMA node while its memory is located in another NUMA node. For example, the Linux default task scheduler CFS takes little consideration of the underlying NUMA topologies and will scheduling tasks to different cores depending on the CPU load balance, which eventually will result in applications running on different cores and their memory being distributed on different NUMA nodes. Especially for memory sensitive applications, the remote memory access latency will greatly impact the overall application performance.

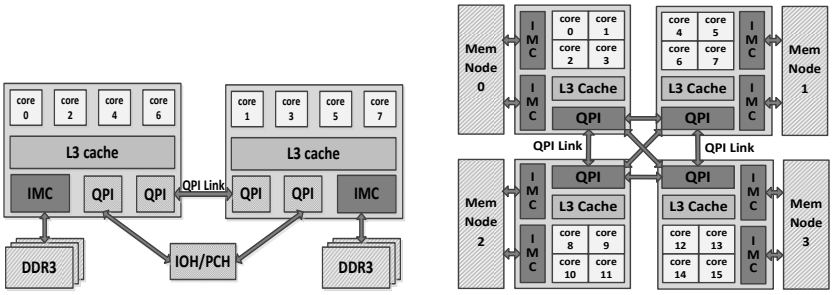
Virtualization poses additional challenges on performance optimizations of the NUMA multicore systems. Existing virtual machine monitors (VMMs), such as Xen [6] and KVM [12], are unaware of the NUMA multicore topology when scheduling VMs. The guest operating system (OS) running in a virtual machine (VM) also have little knowledge about the underlying NUMA multicore topology , which makes application and OS level NUMA optimizations working ineffectively in virtualized environment. As a result, the VMs running both in Xen and KVM are frequently accessing remote memory on the NUMA multicore systems, and this lead to sub-optimal and unpredictable virtual machine performance on NUMA servers.

In this paper, we propose a "Best NUMA Node" based virtual machine scheduling algorithm and implement it in a user-level scheduler that can periodically adjust the placement of VMs running on NUMA systems and make NUMA-aware scheduling decisions. Our solution not only improves VM's memory access locality but also maintains system load balance. And each VM achieves more stable performance on NUMA multicore systems.

The rest of this paper is organized as follows: the NUMA performance impact is analyzed in section 2. We present the proposed NUMA-aware scheduling algorithm and describe the implementation of the user-level scheduler in section 3. In section 4, the performance evaluation of the proposed algorithm is presented. Finally, we discuss the related work in section 5 and draw our conclusion in section 6.

2 NUMA Performance Impact

The NUMA architecture introduces more complex topology than UMA (Uniform Memory Access) systems. Applications (especially for long-running applications such as VMs) may have a high probability of accessing memory remotely on NUMA systems. The CPU, memory bandwidth, and memory capacity load balance among NUMA nodes put much burden on OS and VMM schedulers to properly take advantage of the NUMA architecture. The main focus of these schedulers is to load balance CPU processing resource and seldom consider the



(a) 2-NUMA-node Westmere-EP System

(b) 4-NUMA-node Westmere-EX System

Fig. 1. Dual Socket NUMA Multicore System Performance Impact

NUMA memory effect. In this section, we conduct some experiments to show that the existing VM scheduler CFS (Completely Fair Scheduler) used in KVM [12] will schedule VMs onto different NUMA nodes which results in VMs' remote memory access, and we also evaluate the performance degradation caused by remote memory access on NUMA systems.

Table 1. Hardware Configuration of multicore NUMA servers

Server models	Dell R710	Dell R910
Processor type	Intel Xeon E5620	Intel Xeon E7520
Number of cores	4 cores (2 sockets)	4 cores (4 sockets)
Clock frequency	2.4 GHz	1.87 GHz
L3 cache	12MB shared, inclusive	18MB shared, inclusive
Memory	2 memory nodes, each with 16GB	4 memory nodes, each with 16GB

We use two experimental systems for evaluation. One is a two-NUMA-node Dell R710 server, the other is a four-NUMA-node Dell R910 server. The detailed hardware configuration is shown in table 1. Both servers are commonly seen in today's data centers. The R710 server has two 2.40 GHz Intel (R) Xeon (R) CPU E5620 processors based on the Westmere-EP architecture (shown in Fig.1 (a)). Each E5620 processor has four cores sharing a 12MB L3 cache. The R710 server has a total of 8 physical cores and 16GB memory, with each NUMA node having 4 physical cores and 8 GB memory. The R910 server has four 1.87 GHz Intel (R) Xeon (R) CPU E7520 processors based on the Nehalem-EX architecture (shown in Fig.1 (b)). Each E7520 processor has four cores sharing a 18MB L3 cache. The R910 server has a total of 16 physical cores and 64 GB memory, with each NUMA node having 4 physical cores and 16 GB memory.

We briefly describe the NUMA architecture of our evaluation platforms. The 2-NUMA-node Intel Xeon Westmere-EP topology is shown in Fig.1 (a). In the

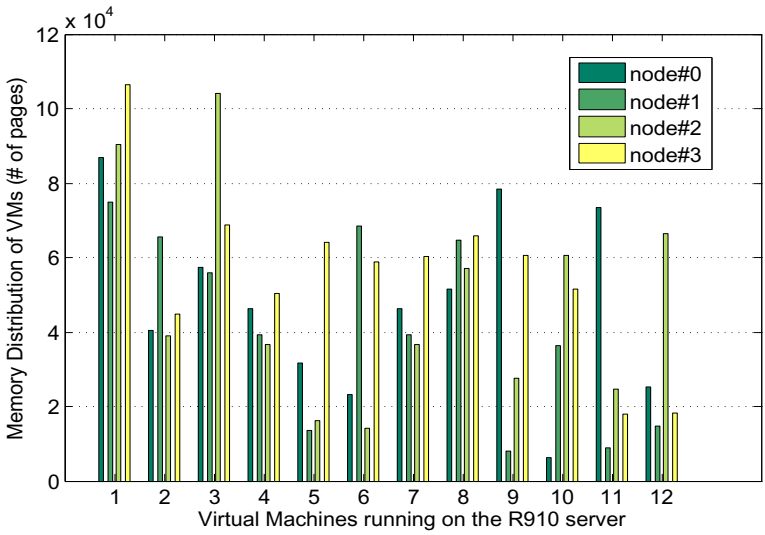


Fig. 2. Memory Distribution of VMs running on NUMA Systems

Westmere-EP architecture, there are usually four or six cores sharing the Last Level Cache (LLC, or L3 cache) in a socket, while each core has its own private L1 and L2 cache. Each socket has the Integrated Memory Controller (IMC) connected to the local three channels of DDR3 memory. Accessing to the physical memory connected to a remote IMC is called the remote memory access. The Intel QuickPath Interconnect (QPI) interfaces are responsible for transferring data between two sockets. And the two sockets communicate with I/O devices in the system through IOH/PCH (IO Hub / Platform Controller Hub) chips. Fig.1 (b) shows a 4-NUMA-node Intel Xeon Westmere-EX topology, there are four NUMA nodes interconnected by the QPI links in the system, and each node has four cores sharing one LLC with two IMCs integrated in the socket. Although other NUMA multicore processors (e.g., AMD Opteron) may differ in the configuration of on-chip caches and the cross-chip interconnect techniques, they have similar architectural designs.

2.1 Memory Distribution on NUMA Nodes

We use KVM as our experimental virtualization environment. The default Linux memory allocation strategy is allocating memory on local node as long as the task is running on that node and the node has enough free memory. Therefore, after a long period of running, a VM will migrate from one NUMA node to another NUMA node due to the CPU load balance of the CFS scheduler. Eventually, the VM's memory will be scattered on all NUMA nodes. Fig.2 shows the memory distribution of virtual machines running on the NUMA system.

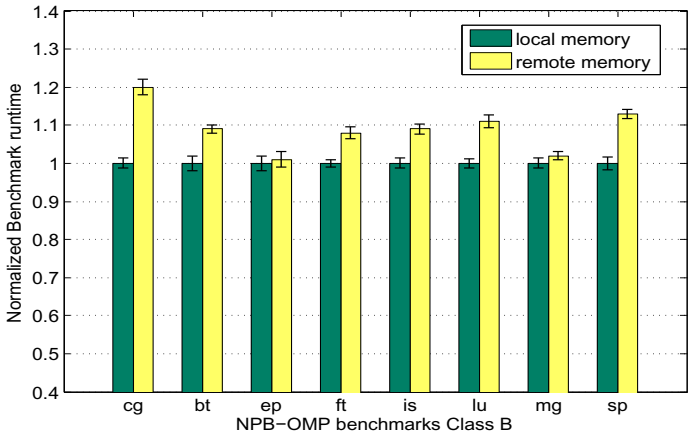


Fig. 3. Remote Memory Access Penalty on Virtual Machine Performance

The data is collected from the R910 server (described in Section 2.1) with KVM virtualization environment, twelve VMs are running on the server, and each VM is configured with 4 VCPUs and 4 GB memory.

The memory of VMs scattered on all NUMA nodes will lead to high memory access latency due to a large proportion of memory access from remote NUMA node. In section 2.2, we will study the VM's remote memory access penalty on NUMA systems.

2.2 Remote Memory Access Penalty

We run a single virtual machine on the R710 server to distinguish the remote memory access performance impact on NUMA systems. In the experimental evaluation, we first run the local memory access case that the VM's VCPUs and memory are all located in the same NUMA node. Then, we run the remote memory access case that the VM's VCPUs are pinned to one node and its memory is bound to another node (using the virsh VM configuration file). In the two cases, we record the average runtimes of NPB benchmarks (a total of five runs for each benchmark) running inside the VM.

Fig.3 shows the benchmarks' local performance compared with remote performance. The average runtime of benchmarks in the remote memory access case is normalized to the local memory access case. As the result shows, some benchmarks (such as *cg*, *lu*, *sp*) have significant performance degradation due to remote memory access latency. But there has little performance impact on other benchmarks (such as *ep* and *mg*) due to the NUMA memory effect.

From the experimental result, we find that even in a two-NUMA-node system, the remote memory access penalty is obvious, especially for NUMA sensitive workloads. Therefore, it is beneficial to improve virtual machine memory access

locality on NUMA systems via properly using NUMA-aware scheduling methods. In section 3, we present our NUMA-aware VM scheduling policy.

3 The NUMA-Aware VM Scheduler

3.1 Main Idea

Virtual machines running on multicore NUMA systems will benefit from local node execution that a VM's VCPUs are running on one NUMA node and its memory is also located on the same NUMA node. VMs running on their local nodes will reduce remote memory access latency. What's more, all VCPUs of a VM running on one NUMA node will reduce the last level cache (LLC) coherency overhead among LLCs of different NUMA nodes. If the VCPUs of a VM is running on different NUMA nodes that have separate LLCs, when they access shared data it will cause LLC coherency overhead. Finally, VMs local node execution will also reduce the interconnect contention (e.g. contention for QPI links in Intel Xeon processors). Although scheduling one VM's VCPUs on the same NUMA node will increase shared on-chip resources contention, we try to schedule different VMs onto separate NUMA nodes with best effort to mitigate shared resources contention and to maximize system throughput with a balanced memory bandwidth usage.

However, it is a big challenge to make all the VMs execute on their local NUMA node and at the same time fully utilize the scalable NUMA architecture. One simple solution is to manually bind the VMs onto NUMA nodes, so all VMs will have local node execution. But the manually bind solution lacks flexibility and may lead to system load imbalance. Some heavily loaded NUMA nodes may become the performance bottlenecks. System load imbalance will greatly impact the VMs overall performance and can not effectively and efficiently take advantage of the multicore NUMA architecture.

To improve virtual machine memory access locality and at the same time to achieve system load balance, we propose a user-level NUMA-aware VM scheduler that periodically scheduling the VCPUs onto certain NUMA nodes according to the CPU and memory usage of all VMs in the virtualized system. The NUMA-aware scheduling algorithm properly selects a "best NUMA node" for a VM that is worth scheduling onto this "best NUMA node" to improve memory access locality as well as to balance system load. Our "Best NUMA Node" based virtual machine scheduling algorithm (short for BNN algorithm) dynamically adjust the placement of VMs running on NUMA nodes as the workload behaviors of the VMs change during execution. In section 3.2, we discuss the design motivation and show a detailed description of the BNN algorithm. We implemented the BNN algorithm in our user-level VM scheduler, and the implementation of the user-level VM scheduler is presented in section 3.3.

3.2 The BNN Scheduling Algorithm

The "Best NUMA Node" based virtual machine scheduling algorithm (short for BNN algorithm) is mainly composed of three parts: (i) selecting the VMs that

are worth scheduling to improve their memory access locality; (ii) finding the "Best NUMA Nodes" for the VMs selected by the previous step; (iii) scheduling the VCPUs of the selected VMs to their "Best NUMA Node".

(i) Selecting Proper VMs

We first select proper VMs that are worth scheduling to improve the virtual machine memory access locality and the overall NUMA system load balance. To select the most actively running VMs, the CPU load of each VM is calculated online (the CPU load calculation of each VM is presented in section 3.3). VMs are then sorted by their CPU load in descending order. Then, we select the topmost k VMs as the proper VMs that are worth scheduling, such that the value k satisfies the following equation:

$$\sum_{i=1}^{k} \sum_{j=1}^{N(VM_i)} Load(V_j) > \frac{4}{5} \sum_{i=1}^{m} \sum_{j=1}^{N(VM_i)} Load(V_j) \tag{1}$$

where $N(VM_i)$ represents the number of VCPUs of VM_i, $Load(V_j)$ represents the CPU load of $VCPU_j$, and m represents the total number of VMs in the system. As denoted in the equation (1), we select the topmost k active VMs as our target scheduling VMs (the total CPU load of these k VMs occupies 80% ($\frac{4}{5}$) CPU usage of all VMs in the system) and let the default CFS scheduler take over the rest of the VMs in the system to do fine-grained load balancing job. We observe that active VMs suffer from NUMA effect more than less active VMs, therefore we select the topmost k active VMs as our target NUMA-aware scheduling VMs and the value of 80% CPU usage of all VMs is tuned by experimental results. By scheduling the most active VMs into proper NUMA nodes through our user-level scheduler and scheduling the remaining less active VMs through the system default CFS scheduler, we can effectively address the challenges of virtual machine memory access locality and system load balance on NUMA multicore systems.

(ii) Finding the "Best NUMA Node"

After selecting the proper VMs for NUMA-aware scheduling, we try to find the "Best NUMA Node" for every selected VMs. First, we examine the memory distribution of each selected VMs. The memory footprint of VMs in each NUMA node is gathered online (the calculation of memory footprint of each VM is presented in section 3.3). According to the memory footprint of the VM, we select the "Best NUMA Node" candidates (short for BNN candidates) for the VM. BNN candidates for the VM satisfy the following equation:

$$M_i > \frac{1}{n} \sum_{j=1}^{n} M_j, \quad (1 \leq i \leq n) \tag{2}$$

where M_i represents the memory footprint of the VM in NUMA node i, n represents the total number of NUMA nodes in the system. Equation (2) means the NUMA node i is selected as the BNN candidates as long as the memory footprint in NUMA node i is larger than the average memory footprint of the VM in all NUMA nodes. We select these NUMA nodes that have relatively large memory

Finding ``best NUMA node`` for each selected VMs
Input: List of selected VMs L_{vm}. The list is sorted in descending order of the VMs' CPU load. Each VM's BNN candidates set. **Output:** A mapping M_{BNN} of VMs to ``best NUMA nodes``.
Variables: the number of NUMA nodes n; the number of total VMs m; VCPU Resource VR; VM_t's BNN candidates set $BNNC_{VM_t}$;
1: Initialize VCPU Resource VR_j of each NUMA node j. 2: $VR_j = \frac{1}{n}\sum_{i=1}^{m} N(VM_i), \quad (j = 1, \ldots, n)$ 3: $M_{BNN} \leftarrow \emptyset$, $VM_t \leftarrow$ pop_front (L_{vm}); 4: **while** $VM_t \neq$ NULL **do** 5: max = node i in $BNNC_{VM_t}$ candidates set that has the largest VR value 6: **if** ($VR_{max} - N(VM_t) \geq 0$) 7: $BNN = max$; 8: **else** 9: BNN = node j that has the largest VR value among all nodes; 10: **end if** 11: $VR_{BNN} = VR_{BNN} - N(VM_t)$, push_back (M_{BNN}, (VM_t, BNN)) 12: $VM_t \leftarrow$ pop_front (L_{vm}); 13: **end while**

Fig. 4. The algorithm of finding the BNN node for each VM

footprint of the VM as its BNN candidates. Because the VM scheduling into BNN candidate nodes will have a higher probability of accessing local memory.

Then, we find the "Best NUMA node" from the BNN candidates set for each VM. Figure 4 shows the algorithm of finding the BNN node for each VM. First, the VCPU resource of each NUMA node is initialized (Line 1-2) as follows:

$$VR_j = \frac{1}{n}\sum_{i=1}^{m} N(VM_i), \quad (j = 1, ..., n) \tag{3}$$

where n represents the total number of NUMA nodes in the system, m represents the total number of VMs in the system, $N(VM_i)$ represents the number of VCPUs of VM_i. VR_j means the VCPU resource of NUMA node j, that is the number of VCPUs allocated in NUMA node j. We suppose that each NUMA node should have equal number of VCPUs to achieve system load balance and maximize system throughput, so we equalize VR_j as equation (3) shows.

After initializing VR_j, we design an approximate bin packing algorithm to find the "Best NUMA Node" for each selected VM. In the beginning, each node j has the VCPU resource capacity of VR_j. Every time, we pick up a VM_t from the sorted VM list (L_{vm}). We select a node that has the largest VR (VCPU resource) value from the BNN candidates set of the VM_t, and record the node

id as max (Line 5). If the max node has sufficient VCPU resource capacity to hold the VM_t (Line 6-7), then we select max as the VM_t's BNN node (good memory locality for VM_t and predictable load balance). Other wise, we try to find a node that has large VCPU resource to hold VM_t to maintain system load balance, so we select the node that has the largest VR value among all nodes in the system as the VM_t's BNN node. By heuristically selecting relatively large VR value node each time, we can achieve good system balance when assigning VMs to their BNN nodes. After selecting the BNN node for VM_t, we decrease the VR capacity of the BNN node and save the VM_t's BNN node mapping strategy in the mapping list M_{BNN}. Then, we find the BNN node for the next VM from the VM list L_{vm} until all selected VMs are mapped to their BNN nodes.

In the BNN algorithm, we assume that the number of a VM's VCPUs is smaller than the number of physical cores in one NUMA node and a VM's memory size is no larger than the physical memory size of one NUMA node. Therefore, we can assign each VM a BNN node to hold VMs. If the VCPU number and memory size of a VM are larger than a physical NUMA node (called huge VMs), we can configure these huge VMs with several small virtual NUMA nodes using the qemu-kvm's VNUMA functionality and make sure each virtual NUMA node of the huge VM is smaller than a physical NUMA node. Then, we can use the BNN algorithm to schedule these virtual NUMA nodes just like scheduling small VMs.

(iii) Scheduling VCPUs to BNN Nodes

After finding the "Best NUMA node" for each selected VMs, the scheduler migrates the VMs' VCPUs to their "Best NUMA nodes" according to the BNN mapping list M_{BNN}. We use the sched_ setaffinity() system call to schedule VCPUs to the proper NUMA nodes. After the VCPUs' affinities are set to their BNN nodes, the job of scheduling VCPUs within nodes is automatically done by the CFS scheduler. The unselected VMs (the less active VMs) will also be scheduled by the CFS scheduler to achieve more fine-grained system load balance.

As the VMs' workload behavior will change over time, our NUMA-aware VM scheduler will periodically execute the above three steps to dynamically adjust the BNN nodes for the selected VMs. The adjustment period is now heuristically set to 60s.

3.3 Implementation of User-Level Scheduler

The NUMA-aware VM scheduler is a user-level process that is designed to test the effectiveness of scheduling algorithms on real NUMA multicore systems. It is able to monitor the virtual machine execution online, gather VM's runtime information for making scheduling decisions, pass it to the scheduling algorithm and enforce the algorithm's decisions. The NUMA-aware VM scheduler has three major phases of execution: (i) Gathering system information online; (ii) Executing scheduling algorithm; (iii) Migrating VM's memory.

Table 2. Gathering system information for scheduling

Gathered information	Description of gathering methods
Information about the system's NUMA topology	The NUMA topology can be obtained via sysfs by parsing the contents of /sys/devices/system/node. The CPU topology can be obtained by parsing the contents of /sys/devices/system/cpu.
Information about the CPU load of each VM	VMs created by the KVM are regarded as general processes in the Linux system. VCPUs of a VM are regarded as threads of the VM process. The CPU load of the threads can be obtained via /proc/<PID>/task/<TID>/stat file (columns 13th and 14th represent the number of jiffies[1] during which the given thread are running in user and kernel mode respectively).
Information about the memory footprint of each VM in each NUMA node	The memory footprint of a VM in each NUMA node is the amount of memory stored on each NUMA node for the given VM process. The file /proc/<PID>/numa_maps contains the node distribution information for each virtual memory area assigned to the process in number of pages.

(i) Gathering system information

The detailed description of gathering system information through user-level scheduler online is shown in table 2. There are three kinds of information obtained online via parsing pseudo file systems (*proc* and *sysfs*).

(1) Information about the system's NUMA topology is obtained once the scheduler starts running.

(2) Information about the CPU load of each VM is calculated periodically. We calculate the average CPU load of each VM during one scheduling epoch, and we sort the VMs using their average CPU load in descending order.

(3) Information about the memory footprint of VMs in NUMA nodes can be obtained from the numa_map files as shown in table 2. The memory footprint of each VM is calculated when the scheduling algorithm selects BNN candidate nodes for the VM.

(ii) Executing scheduling algorithm

The user-level scheduler monitors the VMs workload behavior (the CPU load and memory distribution information), passes the gathered information to the NUMA-aware VM scheduling algorithm and enforces algorithm's decision on migrating VCPU threads onto their proper NUMA nodes. The NUMA-aware scheduling algorithm is periodically executed and the scheduling decisions are enforced using sched_ setaffinity() system call.

[1] One jiffy is the duration of one tick of the system timer interrupt.

(iii) Migrating VM's memory

The user-level scheduler also provides the function of migrating memory to a specified NUMA node using the move_pages() system call. Our NUMA-aware VM scheduler adopts two memory migration strategies to migrate a VM's proper memory pages to its BNN node. The two memory migration strategies are as follows:

(1) If a VM's BNN node is changed to another NUMA node and the VM's VCPUs are scheduled onto its new BNN node. We then use the Intel PEBS (Precise Event-Based Sampling) functionality [2] of sampling memory instructions to get the memory address of the VM. If the sampled memory address is located in the remote NUMA node, we uses the move_pages() system call to migrate the pages around the sampled address to the BNN node. The sampled addresses are considered as frequently accessed memory addresses which have a higher probability to be sampled by PEBS than those less frequently accessed addresses. In this way, we migrate the frequently accessed memory pages from remote node to the BNN node.

(2) When the system load is below a certain threshold (for example 1/p CPU usage of the total system, where p is the total number of physical cores), the scheduler will begin a memory migration phase. In each memory migration phase, the scheduler randomly selects one VM and migrates the VM's memory pages that reside in other nodes to its BNN node. Once the system load is below the previously defined threshold, the memory migration phase will restart memory migration phase.

4 Performance Evaluation

In this section, we evaluate the proposed BNN algorithm using the real-world parallel workloads. We compare the performance of BNN with KVM's default CFS (Completely Fair Scheduler) scheduler and a manually VM binding strategy in Section 4.1. Then we show the improvement of performance stability of the BNN scheduler in Section 4.2. Finally, we analyze the BNN's runtime overhead in Section 4.3.

We run the experiments on the R910 server described in table 1. The server is configured with 32 logical processors with hyperthreading enabled. In order to isolate the NUMA effect from other factors that affect VMs performance, we disable the Intel Turbo Boost in BIOS and set the processors to the maximum frequency. We ran VMs in qemu-kvm (version 0.15.1). Both the host and guest operating systems used in the experiments are SUSE 11 SP2 (the Linux kernel version 3.0.13). The proposed NUMA-aware VM scheduler runs in the host OS. We use the NAS Parallel Benchmark (NPB 3.3) [1] to measure virtual machine performance. The NPB benchmark suite is a set of benchmarks developed for the performance evaluation of parallel applications.

We simultaneously run 8 VMs on R910 server. Each VM is configured with 4 VCPUs and 8 GB memory. Inside each VM, we run one 4-threaded NPB-OMP benchmark. For example, a 4-threaded *bt* benchmark runs in VM1, a 4-threaded *cg* benchmark runs in VM2, and a 4-threaded *sp* benchmark runs in VM8.

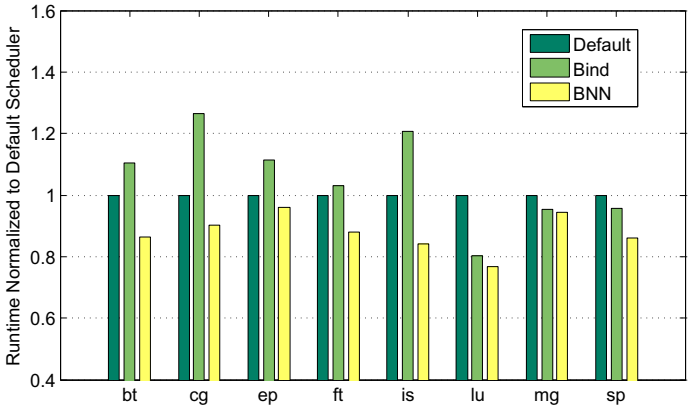


Fig. 5. Virtual machines performance compared with default scheduler

4.1 Improvement on VM Performance

Figure 5 shows the runtime of benchmarks under three different strategies: the default CFS scheduler (Default), the manually bind VMs strategy (Bind), and the BNN scheduler (BNN). We simultaneously run 8 VMs described above on R910 server. Each runtime is the average of five runs under the same strategy and is normalized to the runtime under default KVM CFS scheduler. In the Bind strategy, we manually bind two VMs into one NUMA node, and the eight VMs are evenly distributed across four NUMA nodes on R910 server with two VMs bound to every NUMA node. In the Bind strategy, VMs running *bt* and *cg* are bound to node 0, VMs running *ep* and *ft* are bound to node 1, VMs running *is* and *lu* are bound to node 2, and VMs running *mg* and *sp* are bound to node 3.

From the experimental results, we observe that BNN outperformed *Default* by at least 4.1% (ep) and by as much as 23.4% (lu). Since the *Bind* strategy is unable to adjust to the workload changes, the performance of some benchmarks degrade significantly (the performance degradation of *cg* and *is* is up to 26.4% and 20.7% respectively) compared with *Default*. From the figure, we also find that BNN is more effective for benchmarks that are more sensitive to the NUMA remote memory access latency. For example, BNN significantly improves the performance of *lu*, *bt*, and *sp* by 23.4%, 14.5%, and 14.9% respectively, while only improves the performance of *ep* and *mg* by 4.1% and 5.7%. From previous experiment in Fig.3., we can find that *lu*, *bt*, and *sp* are NUMA sensitive benchmarks, while *mg* and *ep* are insensitive to NUMA effect. The BNN scheduler considers both NUMA effect and system load balance, so BNN achieves better performance than both *Default* and *Bind*.

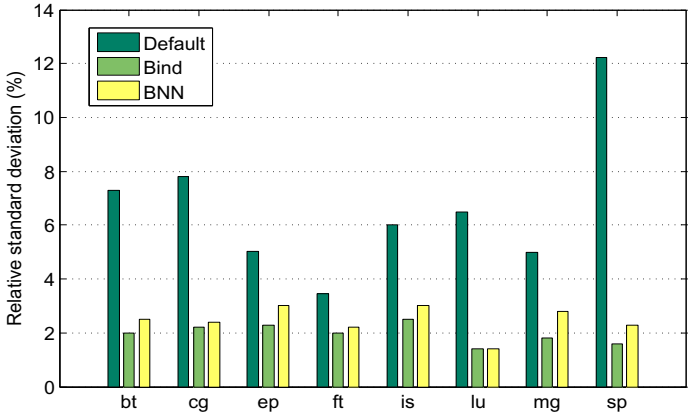


Fig. 6. Comparison of runtime variations among Default, Bind and BNN strategies

4.2 Improvement on Performance Stability

Figure 6 shows the performance stability comparison of *Default*, *Bind*, and *BNN* strategies in terms of benchmarks runtime variations. We calculated the Relative Standard Deviations (RSD) for a set of five runs of each benchmarks under different strategies. RSD measures the extent of stability across program executions. The smaller the RSD value, the more stable and consistent program performance. As expected, the manually bind strategy achieved small RSD values in all workloads with no more than 3% variations. The default CFS scheduler (that only considers CPU load when scheduling VCPUs to cores) caused much more variations than the *Bind* strategy. For the NUMA sensitive *sp* benchmark, the variations can be as high as 12.4%. In comparison, BNN achieves performance stability close to the *Bind* strategy and has significant improvement on performance stability than the *Default* strategy.

4.3 Overhead Analysis

The time complexity of BNN algorithm is O(nlgn). Sorting VMs according to their CPU load has O(nlgn) time complexity, and finding the "best NUMA node" for each VM has O(n) time complexity. As our scheduler executes the BNN algorithm every 60s, so the total overhead of BNN scheduling algorithm is very low. Our experimental results show that the proposed NUMA-aware scheduler incurs less than 0.5% CPU overhead in the system.

5 Related Work

There has been great research interest in performance optimizations of NUMA-related multicore systems. Many research efforts aim at improving application

throughput, fairness, and predictability on NUMA multicore systems. Existing work has tried to address these issues via thread scheduling and memory migration.

In UMA (Uniform Memory Access) multicore systems, thread scheduling methods have been studied to avoid the destructive use of shared on-chip resources [7,16,13] or to use the shared resources constructively [4,8]. The NUMA (Non-Uniform Memory Access) architecture introduces another performance impact factor, the memory locality factor, to be considered when scheduling threads[15]. Researchers proposed the profile-based [5] or dynamic memory migration techniques [9] to improve memory locality on NUMA systems. [7] and [13] considered both shared on-chip resources and memory locality factors to optimize applications performance on NUMA multicore systems. [10] proposed a user-level scheduler on NUMA systems to help design NUMA-aware scheduling algorithms.

Virtualization poses additional challenges on performance optimizations of NUMA multicore systems. [3] proposed a technique that allows a guest OS to be aware of its virtual NUMA topology by reading the emulated ACPI (Advanced Configuration and Power Interface) SRAT (Static Resource Affinity Table). [14] presented a method that allows the guest OS to query the VMM via para-virtualized hypercalls about the NUMA topology. [11] proposed another approach that does not assume any program or system-level optimizations and directly works in the VMM layer by using Performance Monitoring Unit (PMU) to dynamically adjust VCPU-to-core mappings on NUMA multicore systems.

In contrast, our NUMA-aware virtual machine scheduler uses the novel BNN algorithm to dynamically find the "Best NUMA Node" for each active VM and allows these VMs running on their BNN nodes and their memory also allocated in their BNN nodes. Our approach does not need modify the VMM or guest OS, and has a low overhead that only uses system runtime information available from the Linux pseudo file systems to make scheduling decisions.

6 Conclusion

In this paper, we proposed a "Best NUMA Node" based virtual machine scheduling algorithm and implemented it in a user-level scheduler in the KVM virtualized systems. The experimental results show that the BNN algorithm improves virtual machine performance. Optimizing virtual machine performance on NUMA multicore systems faces a lot of challenges, our solution tries to improve memory access locality and at the same time maintain system load balance. In the future work, we try to (1) find metrics for predicting data sharing among VMs and using these metrics to aid VM scheduling on NUMA systems; and (2) design a more adaptive memory migration strategy to further improve memory access locality on NUMA systems.

References

1. The NAS Parallel Benchmarks, `http://www.nas.nasa.gov/publications/npb.html`
2. Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 3: System Programming Guide
3. Ali, Q., Kiriansky, V., Simons, J., Zaroo, P.: Performance Evaluation of HPC Benchmarks on VMware's ESXi Server. In: Alexander, M., et al. (eds.) Euro-Par 2011, Part I. LNCS, vol. 7155, pp. 213–222. Springer, Heidelberg (2012)
4. Bae, C.S., Xia, L., Dinda, P., Lange, J.: Dynamic Adaptive Virtual Core Mapping to Improve Power, Energy, and Performance in Multi-socket Multicores. In: HPDC (2012)
5. Marathe, J., Mueller, F.: Hardware Profile-Guided Automatic Page Placement for ccNUMA Systems. In: PPoPP (2006)
6. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the Art of Virtualization. In: ACM SIGOPS Operating Systems Review (2003)
7. Blagodurov, S., Zhuravlev, S., Dashti, M., Fedorova, A.: A Case for NUMA-Aware Contention Management on Multicore Systems. In: USENIX ATC (2011)
8. Ghosh, M., Nathuji, R., Lee, M., Schwan, K., Lee, H.S.: Symbiotic Scheduling for Shared Caches in Multi-core Systems using Memory Footprint Signature. In: ICPP (2011)
9. Ogasawara, T.: NUMA-Aware Memory Manager with Dominant-Thread-Based Copying GC. In: OOPSLA (2009)
10. Blagodurov, S., Fedorova, A.: User-Level Scheduling on NUMA Multicore Systems under Linux. In: Proceedings of Linux Symposium (2011)
11. Rao, J., Wang, K., Zhou, X., Xu, C.: Optimizing Virtual Machine Scheduling in NUMA Multicore Systems. In: HPCA (2013)
12. Kivity, A., Kamay, Y., Laor, D., Lublin, U., Liguori, A.: KVM: the Linux Virtual Machine Monitor. In: Proceedings of the Linux Symposium (2007)
13. Majo, Z., Gross, T.R.: Memory Management in NUMA Multicore Systems: Trapped between Cache Contention and Interconnect Overhead. ACM SIGPLAN Notices (2011)
14. Rao, D.S., Schwan, K.: vNUMA-mgr: Managing VM Memory on NUMA Platforms. In: HiPC (2010)
15. Tang, L., Mars, J., Vachharajani, N., Hundt, R., Soffa, M.: The Impact of Memory Subsystem Resource Sharing on Datacenter Applications. In: ISCA (2011)
16. Zhuravlev, S., Blagodurov, S., Fedorova, A.: Addressing Shared Resource Contention in Multicore Processors via Scheduling. In: ASPLOS (2010)