2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems

vScope: A Fine-Grained Approach to Schedule vCPUs in NUMA Systems

Qingtian Gan, Song Wu, Hai Jin, Kun Wang
Service Computing Technology and System Lab
Cluster and Grid Computing Lab
School of Computer Science and Technology
Huazhong University of Science and Technology, Wuhan, 430074, China
{ganqt, wusong, hjin, wkhust}@hust.edu.cn

***Abstract*—*Non-Uniform Memory Access* (NUMA) architecture has become the dominant architecture and is widely used in virtualization platforms. In NUMA-based cloud computing platform, arbitrary topology of vCPUs and memory may cause significant performance degradation for VMs, which introduces great challenges for *virtual machine monitors* (VMMs) to efficiently manage the vCPUs and memory. Previous studies mainly sample the characteristics of the vCPUs to indicate the optimizing strategies to reduce the NUMA overheads in virtualization platforms. But the typical periodical sampling methods have some deviations with the real vCPU characteristics. This leads to the inaccurate sampling and scheduling decisions for the optimizing strategies.**

Motivated by the inaccuracy in sampling methods and scheduling decisions, we propose a fine-grained scheduler, named *vScope*, which makes accurate scheduling decisions according to the guest OS processes in the vCPUs, to improve the performance of memory-intensive workloads in cloud platforms. In *vScope*, the VMM identifies the guest OS processes in the vCPUs and calculates the NUMA affinity of each process from the PMU data. At the end of vCPU's scheduling cycle, the scheduler appropriately schedules the vCPUs to their local NUMA node to alleviate the unnecessary NUMA overhead. We implement *vScope* in Xen-4.5.1 VMM and evaluate its effectiveness with some memory-intensive benchmarks. The experimental results shows that *vScope* can achieve up to 11.5% performance improvement for these workloads when compared with the Credit scheduler in Xen. Moreover, *vScope* only introduces limited overhead into the system.

***Index Terms*—Virtualization, NUMA, vCPU Scheduler, Semantic-Gap**

I. Introduction

Non-Uniform Memory Access (NUMA) architecture has become the dominant memory architecture in high-performance servers for its high memory bandwidth and good scalability. Compared with previous *Uniform Memory Access* (UMA) architecture, NUMA architecture can provide higher memory bandwidth and avoid the contention for memory controller when the number of processors increases. Currently NUMA-based servers have been widely deployed in cloud computing platforms. However, arbitrary vCPU-memory topology in NUMA platforms may cause unnecessary remote memory access latency and shared resource contention [1], which may significantly decrease the performance of memory-intensive workloads [2–4]. *Virtual machine monitors* (VMMs) should carefully manage the vCPU and memory to avoid these potential performance bottlenecks.

Previous virtual machine monitors only focus on maximizing the resource utilization, but do not take underlying NUMA topology into account. Recent years, several mechanisms have been proposed to relieve the remote access latency and shared resource contention in NUMA-based virtualization systems. In Xen VMM, a famous open-source virtualization software, Automatic VM Placement strategy and NUMA-aware Scheduling strategy [5] are applied to balance the remote access latency among all nodes. Rao *et al.* [3] find that uncore penalty is an effective metric to predict program performance in NUMA system, and use this metric to indicate the scheduling of vCPUs. Ming *et al.* [2] find that the shift of physical server architectures to NUMA may have bad impact on the cloud workload consolidation performance, and propose a NUMA-aware VM memory management strategy in Xen hypervisor. Sun *et al.* [4] classify the vCPUs into different types according to the LLC reference pressure, and migrate the vCPUs to their local NUMA nodes as well as ensuring the balance of each NUMA node's LLC reference at the same time.

But limited by the semantic gap [6] introduced by virtualization layer, VMMs cannot be aware of the information of applications running in the virtual machines. Rao *et al.*, Ming *et al.* and Sun *et al.* all characterize the vCPUs status to indicate vCPU scheduling and memory management. But in fact, the memory accessing operation is essentially done by the processes in the virtual machines. The characteristics of vCPUs is the average representation of a set of guest processes. There exist some deviations between the periodical sampling result and the real vCPU status, especially when the processes running in each vCPU switches. The average characteristics of vCPUs cannot accurately stand for the processes in virtual machines.

These enlighten us to design a fine-grained vCPU scheduler, called *vScope*, to improve the performance of memory-intensive applications running in virtual machines. The *vScope* scheduler observes the guest OS's process switching in each vCPUs and sample the memory access characteristics of each process. In this way the VMM can get the accurate characteristics of vCPUs in each scheduling cycle, and can manage the scheduling of vCPUs more effectively. We implement *vScope* based on Xen-4.5.1 VMM and evaluate its effectiveness with different memory-intensive benchmarks.

978-1-5386-2588-0/17 $31.00 © 2017 IEEE
DOI 10.1109/HPCC-SmartCity-DSS.2017.69

The main contributions of this paper are as follows:

- We implement the method to observe the switching of guest OS processes in each vCPUs, and propose a fine-grained scheduler, named *vScope*, to schedule the vCPUs according to the characteristics of process running in each vCPU. The characteristics of guest OS processes are sampled and calculated from PMU data.
- We implement the prototype of *vScope* based on the Credit scheduler of Xen-4.5.1 VMM and evaluate the effectiveness of *vScope* with a set of memory-intensive workloads in SPEC CPU2006 and NPB benchmarks. According to the experimental results, *vScope* achieves up to 11.5% performance improvement compared with Credit scheduler in Xen VMM.

The rest of this paper is organized as follows. In Section II, we briefly introduce the background of NUMA overhead in virtualization platforms, and the motivation to do this work. We discuss the design of *vScope* in Section III and describe the implementation in Section IV. In Section V we evaluate the effectiveness of *vScope* as well as the overhead. Finally in Section VII, we make a conclusion about this paper and discuss the future works.

II. Background and Motivation

In this section, we introduce the NUMA architecture, its advantages and limitations in more detail. Then we discuss about the challenges of virtualization technology on NUMA system.

A. NUMA Architecture

Multiprocessor system has become the mainstream choice to improve the performance of computing platform. Traditional symmetric multiprocessor architectures, such as *Uniform Memory Access* (UMA) architecture, have to face the challenge of memory controller and memory bus contention when the number of cores increases. This limits the development of multiprocessor and promotes the population of NUMA architecture. In NUMA architecture, processors are divided into different NUMA nodes, and each NUMA node has each own memory controller and memory channels [7]. Different NUMA nodes are connected with interconnection links, such as Intel QuickPath [8]. All cores can access both memory on the same NUMA node, which is also called local memory, and memory on other NUMA node, which is also called remote memory. This kind of design relieves the contention for memory controllers and increases the overall memory bandwidth.

However in NUMA architecture, accessing memory in remote NUMA node takes more cost than accessing memory in local node [9]. The arbitrary topology of CPUs and accessed memory may introduce unnecessary remote memory access latency for the system. What's more, the improper locality of processes and memory may cause significant shared resource contention: contention for Last-Level Cache, contention for memory controller, and contention for Interconnection Links [10–12]. These four performance degradation factors interplay together and bring great challenges for system and application developers.

B. Motivation

In virtualization environment, the two-level mapping from virtual machines' processes and memory to physical CPUs and memory makes the problem even more complex. This demands the VMMs to manage the vCPUs and memory carefully because the efficiency of managing these resource will dramatically affect the performance of memory-intensive workloads running in VMs. Jia [3], Ming [2] and Sun [4] *et al.* study the NUMA overheads in virtualization systems and propose their solution for VMMs to reduce the remote memory accessing latency and shared resource contention. The proposed methods almost all rely on sampling the vCPUs characteristics from physical PMU data. However, in our research we find that these periodical sampling and scheduling methods have a lot of deviation during the runtime and this may affect the accuracy of the NUMA-oriented optimization strategies. In order to verify the existence of the deviation, we conduct experiments to compare the sampling results seperately with method in the previous solutions and the accurate fine-grained sampling method.

TABLE I: Physical Configuration of the NUMA Server

Processor Model Name	Intel Xeon E5620
Number of NUMA Nodes	2
Cores on each Node	4 cores
CPU frequency	2.40 GHz
Max Memory Bandwidth	25.6 GB/s memory bandwidth
QPI Bandwidth	5.86 GT/s QPI
Cache Size	L1 Cache: 32 KB ICache, and 32 KB DCache, L2 Cache: 256 KB, L3 Cache: 12MB

Our experiments are based on a two-socket NUMA system. Each NUMA node has 4 cores and 12GB memory. The detail configurations are listed in Table I. This is also the physical experimental platform of the following experiments. We install Xen-4.5.1 as the VMM and create a virtual machine in the system. The virtual machine is configured with 8 vCPUs and 8GB memory. We run 8 instances of the same memory-intensive workload (*soplex*) in the virtual machine and use two different sampling methods to observe the ratio of accessing Node0's memory from a specific vCPU. The periodical sampling method is the general method used by previous optimizing strategies and the fine-grained sampling method collects the vCPUs memory access information at each vCPU cycle, which is the most accurate sampling method. During the experiment, we choose vCPU0 of this virtual machine as the target vCPU to be observed and calculate the ratio of accessing memory of Node0. We randomly select a continuous set of the observation result and remove the invalid data with too little memory access times. The experimental result is illustrated in Figure 1.

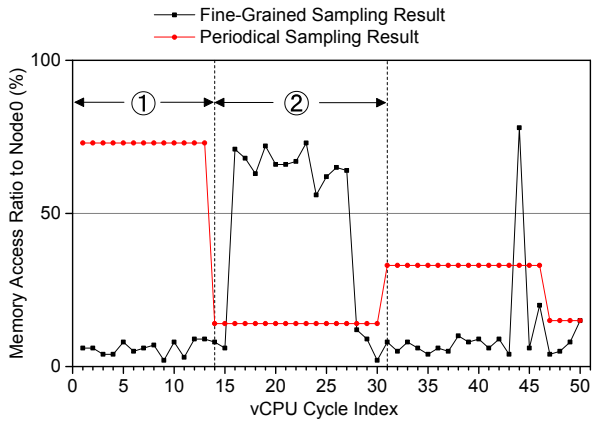


Fig. 1: The Percentage of Memory Access Ratio to Node0 with Two Sampling Methods

From Figure 1 we can see that, there are a lot of mismatching between the sampling results of these two sampling method. In order to analyze the result in more detail, we define two regions in the figure: vCPU cycle 1∼14 as region 1, and 14∼30 as region 2. At the 14th vCPU cycle, periodical sampling policy calculates the memory access ratio according to the memory access data before the 14th vCPU cycle, which is indicated by the black dots in region 1. The calculating result is 15% and is indicated by the red dots in region 2. This means that during region 0, 15% of the memory vCPU0 accessed is on Node0. The value 15% is then treated as the vCPU's status to indicate the scheduling of vCPUs during the vCPU cycles in region 2. But in fact, the real status of the vCPU in each cycle, indicated by the black dots in region 2, does not strictly match with the periodical sampling result. That is because the guest OS process in the vCPU changes, and the characteristics of the vCPU changes as well. To conclude, there exist two deviations here: (1) the periodical sampling result is the average status during the vCPU cycles from the previous sampling point to current sampling point; (2) the real status of the vCPU may be much different from the average sampling status because of the process switching in the vCPU. Using this average periodical sampling result as the status of the vCPU to indicate the managing of vCPUs cannot provide accurate decisions.

In this experiment we use the memory access ratio to specific NUMA node as an example to present the problems. However the deviations between the average sampling result and the real status of the vCPU also exist when sampling other vCPU characteristics. These deviations may lead to the inaccurate of the vCPUs scheduling and memory managing policy in the previous researches. This motivates us to propose a fine-grained vCPU scheduler, which samples the vCPU's characteristics and schedules the vCPUs according to the guest OS processes running in the vCPUs.

III. Design

In this section, we will introduce the design of our fine-grained scheduler *vScope*: the PMU Data Collector, Guest OS Process Cache, and the vCPU Scheduling Policy. The overview of each part's design is illustrated in Figure 2.

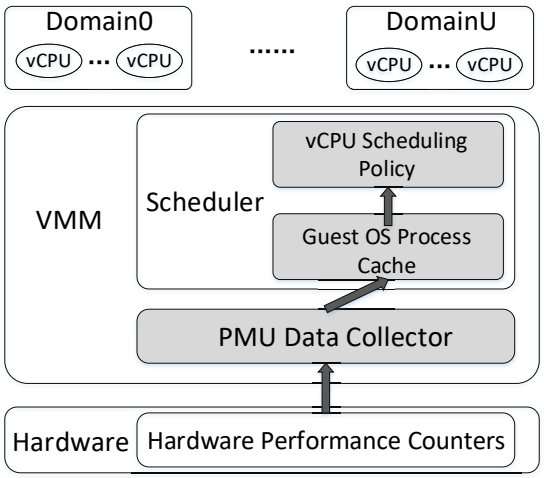


Fig. 2: Overview of vScope

A. *PMU Data Collector*

In order to efficiently manage the vCPUs during scheduling, *vScope* should firstly get aware of the memory-accessing characteristics of processes running in the vCPUs. This characteristic is calculated based on the PMU data collected from physical registers. Traditional profiling tools, such as Perf [13] and OProfile [14], can sample the PMU data of each process in the operating system. But it is difficult for the VMM to get the accurate PMU data of processes in virtual machines. In this section, we firstly describe how to identify the guest OS processes, and then we discuss how to sample and analyze PMU data.

Stephen *et al.* [15] find a method to track the switching guest OS processes in the vCPUs using the value of CR3 registers without any modification to the VM operating systems. In the virtual machine's operating system, the page directory base address stored in the vCPU's CR3 register specifies each process's memory space. Each process has its own memory space. The value of page directory base address is unique for each process and is flushed into the vCPU's CR3 register when the guest OS process is scheduled into the vCPU. This means the value of vCPU's CR3 register can be used to identify the guest OS process running in this vCPU. Moreover, the value of vCPU's CR3 register can be easily accessed by the VMM.

Based on this idea, we monitor the value of the CR3 register to identify the guest OS processes in each vCPU, and collect the PMU data of these guest OS processes during each vCPU scheduling cycle.

In more detail, we collect the times of accessing memory in each NUMA node by the guest OS processes, and calculate the affinity node of processes. In order to describe the calculating method, we introduce a set of variables in the following list:

* N : The number of NUMA nodes.
* $Node_i$: The ith NUMA node.
* $C(gp, i)$: The number of times the guest OS process gp accessing memory from $Node_i$, $0 <= i < N$.

* $NodeAffinity(gp)$: The node affinity of the guest OS process gp.

For guest OS process gp in vCPU vc, we collect the memory access time of gp to each node $C(gp, i)$, and then find the node j with maximum value, which is described in Formula1. Then process gp's node affinity $NodeAffinity(gp)$ is set to be j.

$$C(gp, j) = \underset{0<=i<N}{max}\{C(gp, i)\} \quad (1)$$

B. GuestOS Process Cache

In this section we describe the design of Guest OS Process Cache. The design of vCPU Scheduling Policy, which is described in the following section, needs to refer to the historical node affinity information of the guest OS processes. This demands that the VMM should cache the processes' node affinity information in the past scheduling duration. But in a virtual machine, there may have thousands of processes being generated and terminated. It is a great challenge to cache these processes' historical information with tolerable system overhead in the VMM.

*1) **Design of Guest Process Cache**:* Generally, in an operating system, only a small set of processes is heavily scheduled and take up most of the CPU cycles. We design an experiment to observe the executing time of guest OS processes. We install Xen-4.5.1 VMM in the physical server as configured in Table I and start a VM with 8 vCPUs and 8GB memory. In the VM, we run 8 instances of *soplex* application as the workload, and observe the vCPU cycle each guest OS process takes up in the VM. The result is presented as a histogram plot in Figure 3.

As shown in the figure, most processes only take up less than 20 vCPU cycles in the sampling period, and only 13 hot processes take up more than 500 vCPU cycles, which is extremely more than other processes. In order to dive more deep into these data, we calculate the percentage of process number and percentage of total vCPU cycles in different vCPU cycle regions. The statistic results are shown in Table II. From the table we can observe in more detail that, in the virtual machine, about only 2.2% processes take up about 87.7% vCPU scheduling cycle. These 2.2% of the processes are the hot processes in the virtual machine. Caching these hot processes' historical node affinity information can cover most of the demand.

TABLE II: Total vCPU Cycles of Processes in Different Region

vCPU Cycle Region	Percentage of Process Number	Percentage of Total vCPU Cycle
0 ∼ 499	97.8%	12.3%
500 ∼	2.2%	87.7%

Inspired by this phenomenon in the OS, we design a *Least-Recently-Used* (LRU) queue with limited length as the guest OS process cache to store the historical information of processes. A LRU queue is often applied in cache system to store the hot information [16], which just fits the characteristic of the hot processes in the operating systems.

*2) **Update GuestOS Process's Node Affinity**:* In *vScope*, after calculating the $NodeAffinity(gp)$ of the guest OS process gp, the VMM then searches the historical record in the LRU queue of GuestOS Process Cache to compare and update the node affinity information. We use the pseudo-code in Algorithm 1 to describe the strategy of updating node affinity information. If the node affinity value of gp is different from the historical value in the LRU queue, then the historical record will be updated with current $NodeAffinity(gp)$. If gp's information is not cached in the LRU queue, a new record will be created and added to the queue. Each time gp's record is accessed, it will be moved to the head of the LRU queue.

The length of the LRU queue is strictly limited. Once the number of record in the LRU queue is excess, the record in the tail of LRU queue will be removed. In this way, the information of the hot guest OS processes will be cached at the head of the queue and can be accessed quickly. The information of infrequently scheduled guest OS processes will be gradually eliminated from the queue to save the space overhead.

Algorithm 1: Guest OS Process Cache Updating Algorithm

Input: gp: current guest OS process
$NodeAffinity(gp)$: the node affinity of guest OS process gp;
$lrulist$: the LRU queue that chaches the processes' affinity data;
Output: the updating result

```
foreach lruitem in lrulist do
    if lruitem is the cache of gp then
        if luritem.node_affinity not equal to NodeAffinity(gp) then
            lruitem.pincount -= 1;
        else
            lruitem.pincount ← MAX-PINCOUNT;
        end
        if luritem.pincount < 0 then
            lruitem.node_affinity ← NodeAffintiy(gp);
        end
        move luritem to lrulist head;
    end
end
if no luritem matched with gp then
    lruitem ← CreateNewLRUItem(gp, current_node);
    move lruitem to lrulist head;
end
if length(lrulist) > MAX-LRULEN then
    remove the last record in lrulist;
end
return;
```

C. vCPU Scheduling Policy

The main guideline of the vCPU Scheduling Policy is to keep the vCPUs running on its local NUMA node. The

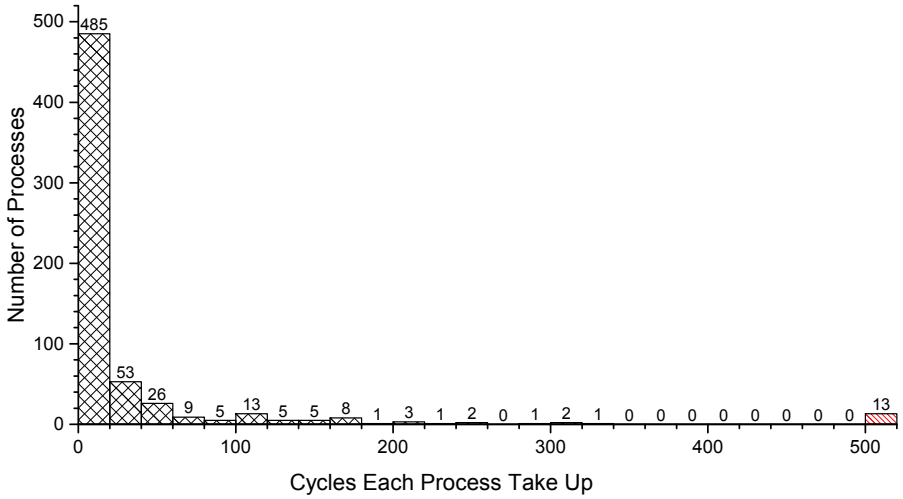


Fig. 3: Statistics on the Cycles Each Guest OS Process Takes Up

pseudo-code of this vCPU Scheduling policy is shown in Algorithm 2. At the end of each vCPU's scheduling cycle, *vScope* updates the node affinity of the vCPU to be the node affinity of the process running in this vCPU (line 2). If the current node affinity of this vCPU is different from the previous one, then the scheduling strategy finds a physical CPU with least overhead on this NUMA node and pushes this vCPU into the co-locate queue of this pCPU (line 3∼4). The overhead of pCPU means the number of vCPUs co-located to this pCPU.

When choosing the candidate vCPU for next scheduling cycle, *vScope* compares the priority of the first vCPU in the co-locate queue and the first vCPU in the run queue, and selects the one with higher priority to be the candidate vCPU (line 6∼13). In this way, the vCPUs in the co-locate queue get chance to be stolen to their local node.

Algorithm 2: NUMA-Aware vCPU Scheduling Algorithm

Input: vc: current vCPU;
gp: the guest OS process running in the vc;
pc: current pCPU;
Output: vCPU migration decision

1 **if** *vc.node_affinity != gp.node_affinity* **then**
2 $vc.node\_affinity \leftarrow gp.node\_affinity$;
3 $vc.cpu\_affinity \leftarrow PickLeastOverheadPCPU(vc.node\_affinity)$;
4 put vc into colocate queue of $vc.cpu\_affinity$;
5 **end**
6 $vcnext \leftarrow FirstVC(pc.runq)$;
7 $vcco \leftarrow FirstVC(pc.colocate\_queue)$
8 **if** $vcnext.pri < vcco.pri$ **then**
9 steal $vcco$ from its runq;
10 remove $vcco$ from colocate queue;
11 $vcnext \leftarrow vcco$;
12 **end**
13 set $vcnext$ to be the next scheduled vCPU;
14 **return**;

IV. Implementation

In this section, we will describe the modifications we do to the default VMM to implement each part of system *vScope*.

A. *Implementation of PMU Data Collector*

The PMU Data Collector is the basis of *vScope*. In the system, we use *Perfctr-Xen* [17] as the tool to obtain the node accessing counts from hardware performance monitor units. We patch the code of *Perfctr-Xen* to Xen-4.5.1 VMM and modify it to count the times of accessing each node's memory by the processes. What's more, the operation to get the PMU data follows the description in Intel software developer manual [18]. The node accessing count of each process is updated at the end of each vCPU scheduling cycle.

B. *Implementation of GuestOS Process Cache*

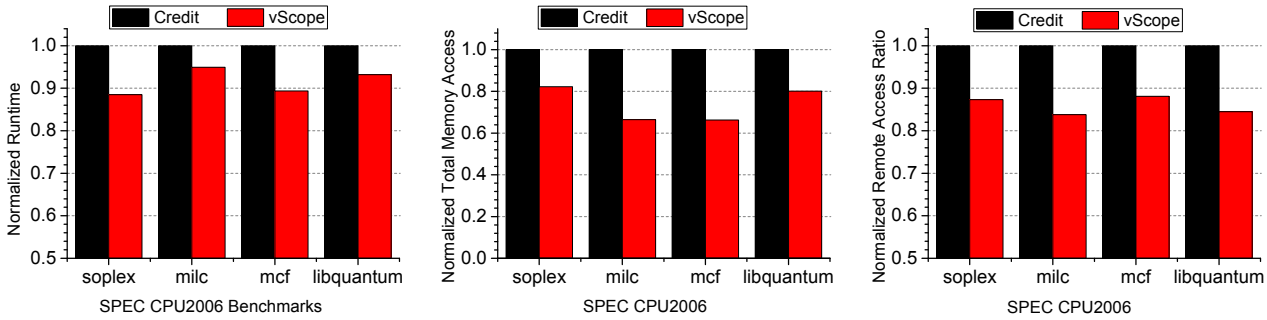
The LRU queue of GuestOS Process Cache is the central part of the system. We add a doubl-link list to each pCPU's data structure to indicate the LRU queue. The list items contain the information of each process's node affinity information. We create this LRU queue for each pCPU so that the queue only needs to cache the hottest processes running on this pCPU. This can shorten the time to search the record in the queue and improve the accuracy of the cache. The size of the LRU queue is tunable and can be modified to adjust the overhead of the whole system.

C. *Implementation of vCPU Scheduling Policy*

Our modification is done based on the default Credit scheduler in Xen-4.5.1 VMM. We add variables in the vCPU's data structure to record their node affinity and cpu affinity. The node affinity is updated from the guest OS process running the vCPU when the vCPU's current scheduling cycle ends.

V. Evaluation

In this section, we conduct a set of experiments to verify the effectiveness of our scheduler *vScope* using different kind of workloads, and study the overhead of the *vScope*. In the following, we first introduce the design of evaluation experiments, and then we discuss about the experimental results with different kinds of workloads.



(a) Normalized Executing Time (b) Normalized Total Memory Access (c) Normalized Remote Access Ratio

Fig. 4: Experimental Results of SPEC CPU2006

A. *Experiment Method*

Our experiments are all based on the server shown in Table I. We use Xen-4.5.1 as the VMM and apply Credit scheduler and *vScope* separately to compare the system performance of these two schedulers. In the experiments, we create three virtual machines (VM1∼VM3). VM1 and VM2 take up 8GB of memory and VM3 takes up 1GB of memory while all VMs are configured with 8 vCPUs. Each virtual machine runs CentOS 5.5 OS and Linux-3.2.30 kernel. When starting the experiments, we run 4 instances of the same workloads in both VM1 and VM2, and run 8 instances of computing-intensive workloads in VM3 as the background overheads.

We use a set of memory-intensive benchmarks as the workload in the experiments. These benchmarks are as following:

- **SPEC CPU2006** [19]: A set of single-thread, industry-standardized benchmark. We choose several memory-intensive applications from SPEC CPU2006: *soplex*, *libquantum*, *mcf*, and *milc*.
- **NPB** [20]: A set of programs used to evaluate the performance of supercomputers. In our experiments, we use the NPB-MPI version and choose the memory-intensive applications in the set: *bt*, *cg*, *lu*, *mg*, *sp*.

In order to compare the performance of default Credit scheduler and *vScope*, we observe the executing time, the total memory access, and the remote access ratio during the runtime to compare the effectiveness of *vScope*. We set the experimental result of each application in Credit scheduler as the baseline, and normalize the experimental result in *vScope*.

B. *Experiments with SPEC CPU2006*

We separately use the four SPEC CPU2006 benchmarks (*soplex, milc, mcf, libquantum*) as the workloads and conduct the experiments as described in Section V-A. In the experiments, we observe the experimental metrics in two different schedulers. We execute each application for five times and calculate the average of the results, and present the normalized experimental results in Figure 4.

As shown in Figure 4(a∼c), the *vScope* shortens the executing time of all the four benchmarks by 11.5%, 5%, 11%, and 7%, respectively. In more detail, the *vScope* reduces the total memory access by 17.8%, 33.5%, 33.7% and 20%, respectively. This is because *vScope* migrates the vCPUs to the local NUMA node and the affinity cpu of the guest process in the vCPU. On one hand, scheduling the vCPU in the affinity pCPU may increase the LLC cache hit ratio and reduce the times of memory access; on the other hand, scheduling on the local NUMA node can relieve the situation of accessing memory in remote NUMA nodes and decrease the remote memory access ratio. These two factors work together and improve the whole system performance.

C. *Experiments with NPB Benchmarks*

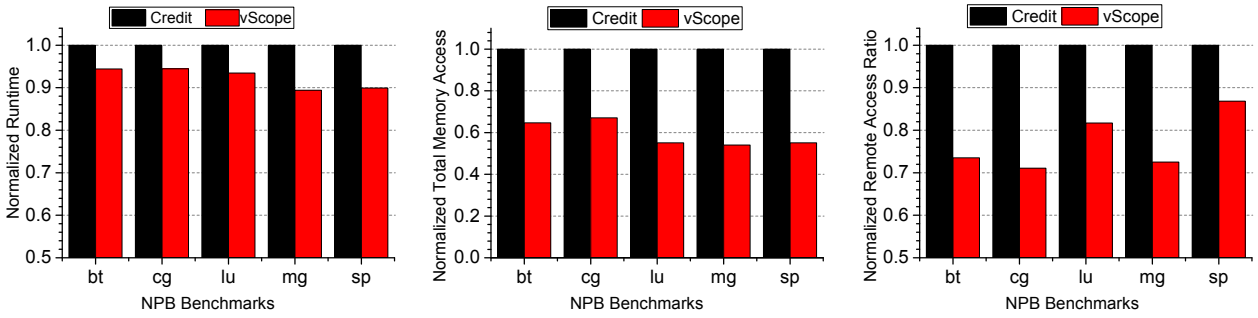
We run the selected NPB benchmarks (*bt, cg, lu, mg, sp*) in VM1 and VM2. We use the implementation of NPB-MPI and in each virtual machine we run four single-thread instances. The same as in the experiments with SPEC CPU2006, we run each application for five times and calculate the average of the results in both the Credit scheduler and *vScope*. The normalized results are shown in Figure 5.

The results of NPB is almost the same as that of SPEC CPU2006. The five NPB application achieves separately 5.6%, 5.5%, 6.5%, 10.6%, and 10.1% performance improvement in *vScope* scheduler when compared with Credit scheduler. What's more, *vScope* reduces the number of remote memory access as well as total memory access. The reduction in total memory access is contributed by the improvement in LLC hit ratio.

D. *Overheads*

The overheads of *vScope* are mainly contributed by the delay in collecting PMU data and the cost of searching the GuestOS Process Cache queue. In this section we conduct experiments to evaluate these overheads. These overheads are related to the number of vCPUs and the number of hot processes in the VMs. We start 1, 2, 4, and 8 VMs separately in the VMM and execute two instances of NPB (*mg*) application in each VM. Each VM is configured with 2 vCPUs and 2GB memory. During the experiments, we observe the overhead time caused by the code of *vScope*. The observed overhead time is shown in Table III.

As shown in Table III, with the number of VMs and the number of hot processes increase, the overhead time of *vScope* also increases. When starting 8 VMs in the system,



(a) Normalized Executing Time (b) Normalized Total Memory Access (c) Normalized Remote Access Ratio

Fig. 5: Experimental Results of NPB

TABLE III: Overhead Time of vScope

Number of VMs	Overhead Time Percentage
1	0.02%
2	0.05%
4	0.14%
8	0.35%

the overhead time takes up about 0.35% of the runtime, which is almost negligible. That's to say, *vScope* improves the performance of NUMA-based virtualization systems while introducing only a little overhead into the system.

VI. Related Works

This study works on optimizing the resource management in VMMs to improve the performance with a Semantic-Gap Bridging method in NUMA-based virtualization system. The NUMA overhead problems have also been widely discussed by other researchers.

The overhead of NUMA system is caused by the four performance degradation factors: remote access latency [9], LLC contention, memory controller contention, and interconnection contention [10, 11, 21]. Rao *et al.* [3] design a metric to measure the overall system overhead and indicate the scheduling of the vCPUs to minimize this overhead. Liu *et al.* [2] optimize the memory management policy in the VMM and reduce the overhead of the memory accessing to improve the performance of workloads in virtual machine consolidation scenario. Kim *et al.* [22] evaluate the average data access latency of each VM and migrate the vCPUs or memory to alleviate the latency in KVM. The KVM also introduces the Automatic NUMA Balancing [23] strategy to schedule the vCPUs and migrates memory along with vCPUs in some cases.

Another kind of solutions for NUMA overheads focus on weakening the semantic gap to mitigate the obstacle in resource management efficiency. Rao *et al.* propose the method of exporting the underlying NUMA topology to the virtual machines and make use of the NUMA optimization method in the guest OS. Hypervisors such as Xen [5] and VMware [24] also apply the vNUMA strategy in the VMMs.

VII. Conclusion and Feature Work

In summary, we propose a fine-grained approach to efficiently manage the location of vCPUs of virtual machines in this paper. Our proposed scheduler, *vScope*, tracks the guest OS processes in the vCPUs and collects the PMU information of guest OS processes to calculate their affinity nodes. In the scheduling decision part, *vScope* tries to migrate the suitable vCPUs to their local NUMA nodes. The experiments verify that *vScope* reduces the remote memory access ratio of vCPUs in the system and improves the performance of memory-intensive workloads by up to 11.5% with only a little overhead.

In this work, we only aim at alleviating the remote access latency when scheduling the vCPUs. But the LLC contention may also dramatically decrease the system performance. In the future, we will study on combining the considering about LLC contention to make better scheduling decisions.

What's more, the NUMA architecture may also affect the effectiveness of IO devices when device interface and vCPUs are not in the same NUMA nodes [25, 26]. An extension plan is to take advantage of the semantic-gap method in this work to alleviate the overhead in IONUMA scenario.

VIII. Acknowledge

This research is supported by National Key Research and Development Program under grant 2016YFB1000501, 863 Hi-Tech Research and Development Program under grant No. 2015AA01A203, and National Science Foundation of China under grants No. 61232008.

References

[1] F. Gaud, B. Lepers, J. Funston, M. Dashti, and A. Fedorova, "Challenges of memory management on modern NUMA systems," *Communications of the ACM 2015*, vol. 58, no. 12, pp. 59–66, 2015.

[2] L. Ming and L. Tao, "Optimizing virtual machine consolidation performance on NUMA server architecture for cloud workloads," in *Proceedings of the 41st International Symposium on Computer Architecture (ISCA'14)*, 2014, pp. 325–336.

[3] J. Rao, K. Wang, X. Zhou, and C.-Z. Xu, "Optimizing virtual machine scheduling in numa multicore systems," in *Proceedings of the 19th International Symposium on*

High Performance Computer Architecture (HPCA'13), 2013, pp. 306–317.

[4] H. Sun, S. Wu, Q. Gan, L. Zhou, and H. Jin, "vProbe: Scheduling virtual machines on NUMA systems," in *Proceedings of the 2016 IEEE Cluster Computing (CLUSTER'16)*. IEEE, 2016, pp. 70–79.

[5] Citrix, "Xen NUMA roadmap," 2015. [Online]. Available: http://t.cn/RoiaLQP

[6] P. M. Chen and B. D. Noble, "When virtual is better than real [operating system relocation to virtual machines]," in *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HotOS'01)*. IEEE, 2001, pp. 133–138.

[7] C. Lameter, "NUMA (non-uniforme memory access): An overview," *ACM Queue*, vol. 11, no. 7, p. 40, 2013.

[8] B. Mutnury, F. Paglia, and J. Mobley, "QuickPath interconnect (QPI) design and analysis in high speed servers," in *Proceedings of 19th Electrical Performance of Electronic Packaging and Systems (EPEPS'10)*, 2010, pp. 265–268.

[9] Z. Majo and T. R. Gross, "Memory management in NUMA multicore systems: Trapped between cache contention and interconnect overhead," in *Proceedings of the 2011 International Symposium on Memory Management (ISMM'11)*, 2011, pp. 11–20.

[10] S. Blagodurov, S. Zhuravlev, A. Fedorova, and A. Kamali, "A case for NUMA-aware contention management on multicore systems," in *Proceedings of the 2011 USENIX Annual Technical Conference (USENIX ATC'11)*, 2010, pp. 557–558.

[11] 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 *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*, 2013, pp. 381–394.

[12] B. Sergey, Z. Sergey, and F. Alexandra, "Contention-aware scheduling on multicore systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 28, no. 4, p. 8, 2010.

[13] Linux, "Perf: Linux profiling with performance counters," 2015. [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page

[14] J. Levon, W. Cohen, and P. Elie, "Oprofile: A system profiler for linux," 2015. [Online]. Available: http://oprofile.sourceforge.net/news/

[15] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Antfarm: Tracking processes in a virtual machine environment," in *Proceedings of the 2006 USENIX Annual Technical Conference (ATC'06)*, 2006, pp. 1–14.

[16] E. J. O'neil, P. E. O'neil, and W. Gerhard, "The LRU-K page replacement algorithm for database disk buffering," *ACM SIGMOD Record*, vol. 22, no. 2, pp. 297–306, 1993.

[17] R. Nikolaev and G. Back, "Perfctr-Xen: a framework for performance counter virtualization," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'11)*, 2011, pp. 15–26.

[18] C. Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2*, 2016.

[19] SPECCPU, "SPEC CPU2006," 1988. [Online]. Available: http://www.spec.org/cpu2006/

[20] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber *et al.*, "The NAS parallel benchmarks-summary and preliminary results," in *Proceeding of Supercomputing (SC'91)*, 1991, pp. 158–165.

[21] R. Lachaize, B. Lepers, and V. Quéma, "MemProf: A memory profiler for NUMA multicore systems," in *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC'12)*, 2012, pp. 53–64.

[22] C. Kim and K. H. Park, "Credit-based runtime placement of virtual machines on a single NUMA system for QoS of data access performance," *IEEE Transactions on Computers*, vol. 64, no. 6, pp. 1633–1646, 2015.

[23] Redhat, "Automatic NUMA balancing," 2014. [Online]. Available: http://t.cn/RofX8kY

[24] J. Simons, "vNUMA: What it is and why it matters," 2011. [Online]. Available: https://octo.vmware.com/vnuma-what-it-is-and-why-it-matters/

[25] L. Shelton, "High performance I/O with NUMA systems in linux," 2013. [Online]. Available: http://t.cn/Ro8Pbx0

[26] R. Mehta, Z. Shen, and A. Banerjee, "NUMA aware I/O in virtualized systems," in *Proceedings of the 2015 IEEE High-Performance Interconnects (HOTI'15)*. IEEE, 2015, pp. 10–17.