2016 IEEE International Conference on Cluster Computing

vProbe: Scheduling Virtual Machines on NUMA Systems

Song Wu, Huahua Sun, Like Zhou, Qingtian Gan, Hai Jin
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
{wusong, hjin}@hust.edu.cn

***Abstract*—With the development of multi-core platforms and cloud computing, *Non-Uniform Memory Access* (NUMA) architecture has been dominant in cloud data centers in recent years. However, NUMA architecture is not well supported in virtualized environments. Because of the semantic gap introduced by the virtualization layer, hypervisors know little about the characteristics of applications running in *virtual machines* (VMs). More importantly, in order to guarantee hypervisors' applicability, load balance strategies of virtual CPU (VCPU) schedulers do not consider the memory access characteristics of applications running in VMs, which probably introduces significant shared resource contention and unnecessary remote memory accesses.**

In this paper, we propose a NUMA-aware VCPU scheduler based on Xen, named *vProbe*, to improve the performance of memory-intensive applications while maintaining the transparency of the virtualization layer in NUMA-based servers. It collects *performance monitoring units* (PMU) data for each VCPU and analyzes their memory access characteristics. Then, according to the memory access characteristics of each VCPU, it periodically reassigns all memory-intensive VCPUs to each NUMA node evenly while preferentially allocating them to their local nodes, which aims to alleviate shared resource contention and reduce unnecessary remote memory accesses. Moreover, when a physical CPU (PCPU) becomes idle, it preferentially steals a VCPU from the run queues of PCPUs in the local node to this PCPU, which helps to maintain balanced *last-level cache* (LLC) contention and reduce extra remote memory accesses. Our evaluation shows that *vProbe* can significantly improve the performance of memory-intensive applications (e.g., up to 45.2% performance improvement compared with the Credit scheduler) while introducing negligible overheads.

***Index Terms*—NUMA, Virtualization, VCPU Scheduling, Load Balance Strategy**

I. Introduction

NUMA architecture has been dominant in the multi-core server market in recent years due to its high memory bandwidth and good scalability. NUMA-based servers are widely deployed in existing cloud data centers and host many kinds of applications. Though previous researches have addressed various scheduling problems with NUMA-based servers, there are still significant challenges in resource management in virtualized servers based on NUMA architecture.

First, current VCPU schedulers of hypervisors usually use load balance strategies to improve the CPU resource utilizations in the multi-core servers, but they do not consider the characteristics of applications running in VMs, which probably aggravates shared resource contention and introduces unnecessary remote memory accesses (as demonstrated in Section II-B). Because of the semantic gap between hypervisors and guest operating systems (guest OSes), hypervisors know little about the characteristics of applications running in VMs. More importantly, in order to guarantee hypervisors' applicability, load balance strategies of VCPU schedulers do not consider the memory access characteristics of applications running in VMs, which probably aggravates LLC contention and brings extra remote memory accesses when VCPUs that have lots of LLC accesses are frequently migrated among different nodes to utilize available CPU resources. Previous studies [1–3] optimize thread scheduling strategies to improve the performance of memory-intensive applications, but they are ineffective in virtualized environments because NUMA architecture is invisible to guest OSes. Rao *et al.* [4] present a NUMA-aware VCPU scheduler to schedule VCPUs according to their *uncore penalties* determined by the penalty to access the "uncore" memory subsystem. However, because all performance-degrading factors are treated equally when calculating VCPUs' *uncore penalties*, it cannot give precise optimization for each factor.

Second, it is difficult to improve the performance of memory-intensive applications while maintaining the transparency of the virtualization layer in virtualized servers based on NUMA architecture. In virtualized systems, hypervisors take control of resource management, which introduces a semantic gap between the underlying NUMA architecture and guest OSes. As a result, guest OSes know little about the underlying memory distribution and hardware resource allocations, which makes it hard to improve the performance of memory-intensive applications at the guest OS level. Rao *et al.* [5] present a vNUMA-mgr approach which exposes virtual NUMA topology to VMs for performance optimization at the guest OS level. Liu *et al.* [6] leverage NUMA overhead awareness in the hypervisor's memory management. However, both of them need to modify the guest OS kernel, which breaks the transparency of the virtualization layer and restricts the applicability of the proposed methods. Thus, it is a significant challenge to design a NUMA-aware VCPU scheduler which can improve the performance of memory-intensive applications while maintaining the transparency of the virtualization layer.

The above challenges motivate us to design a NUMA-aware VCPU scheduler, named *vProbe*. According to the memory access characteristics of applications running in VMs, *vProbe* optimizes VCPU migration strategies to alleviate shared resource contention and reduce unnecessary remote memory accesses while maintaining the transparency

2168-9253/16 $31.00 © 2016 IEEE
DOI 10.1109/CLUSTER.2016.60

of the virtualization layer, which improves the performance of memory-intensive applications on virtualized servers based on NUMA architecture. Moreover, we implement a working prototype of *vProbe* and evaluate its effectiveness with various benchmarks and real-world applications.

The main contributions of this paper are as follows:

- We propose a NUMA-aware VCPU scheduler, named *vProbe*, which schedules VCPUs according to their memory access characteristics calculated by the collected PMU data, to improve the performance of memory-intensive applications while maintaining the transparency of the virtualization layer.
- We implement a prototype in the Xen [7] hypervisor based on the Credit scheduler [8] and verify its effectiveness through different kinds of test scenarios and applications. The experimental results show that our scheduler can significantly improve the performance of memory-intensive applications (e.g., up to 45.2% performance improvement compared with the Credit scheduler) while introducing negligible overheads.

II. Background and Motivation

In this section, we give a detailed description of NUMA architecture and the load balance strategy of Xen's Credit scheduler, and experimentally show the performance problems in virtualized servers based on NUMA architecture.

A. *NUMA Architecture*

Nowadays, with the increasing number of cores per chip, memory bandwidth has become a significant performance bottleneck in centralized memory architecture, i.e., *Uniform Memory Access* (UMA), which promotes the appearance of NUMA architecture. A typical NUMA system consists of multiple nodes and each node has its own memory controller and memory blocks. With distributed memory controllers, memory controller contention can be effectively alleviated, which guarantees high memory bandwidth. As NUMA architecture has become increasingly popular because of its high memory bandwidth and good scalability, NUMA-based servers are widely deployed in existing data centers and host a wide variety of applications.

However, it is difficult to guarantee the performance of memory-intensive applications on NUMA-based servers due to the performance-degrading factors: (a) With distributed memory controllers, NUMA architecture introduces some additional performance-degrading factors, i.e., remote memory access latency, memory controller contention and interconnect link contention [1, 9]; (b) LLC contention [1, 3, 10] is also an important factor that affects the performance of memory-intensive applications although it is not relevant to NUMA architecture.

B. *Load Balance Strategy of Xen's Credit Scheduler*

The VCPU scheduler is an important component of a hypervisor, which is responsible for the allocation of CPU resources among VMs. More importantly, the load balance strategy is the major component of a VCPU scheduler, which contributes to improving the CPU resource utilizations in the multi-core servers. However, the load balance strategy of Xen's Credit scheduler cannot support NUMA architecture well because it does not consider the memory access characteristics of applications running in VMs.

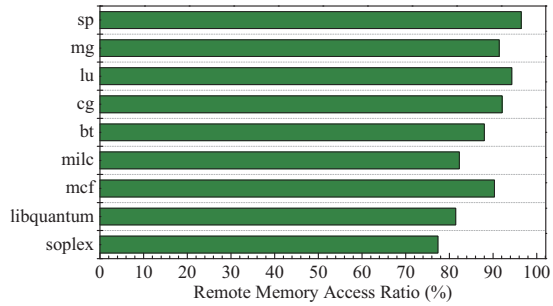


Fig. 1: The percentage of remote memory accesses number over Xen's VCPU scheduler

TABLE I: The detailed configuration of our NUMA system

	Intel Xeon E5620
Cores	4 cores (2 sockets)
Clock frequency	2.40 GHz
L1 cache	32 KB ICache, 32 KB DCache
L2 cache	256 KB unified
L3 cache	12 MB unified, shared by 4 cores
IMC	25.6 GB/s bandwidth, 2 memory nodes, each node has 12 GB memory
QPI	2 links, 5.86 GT/s

To study the influence of the load balance strategy of Xen's Credit scheduler, we conduct experiments to evaluate the remote memory access ratios of some memory-intensive applications. The remote memory access ratio reflects the percentage of accessed pages belonging to each node, which is determined by the load balance strategy. Our NUMA system comprises of two quad-core Intel Xeon CPUs, which share an *integrated memory controller* (IMC) and an Intel *QuickPath Interconnect* (QPI). More detailed configuration is described in Table I. All VMs (VM1∼VM3) have 8 VCPUs. VM1 and VM2 are configured with 8GB memory and run a 4-threaded application of NPB [11] or four identical instances of an application of SPEC CPU2006 [12] concurrently. VM3 is configured with 2GB memory and runs 8 hungry-loop applications to consume available CPU resources. As shown in Figure 1, because the load balance strategy only guarantees the fair share of CPU resources and does not consider the underlying NUMA architecture and the characteristics of applications running in VMs, the remote memory access ratios of these applications are more than 80% except for *soplex* (77.41%), which shows significant potential for performance improvement by reducing remote memory accesses. Besides, the large number of remote memory accesses can also aggravate interconnect link contention.

In summary, NUMA architecture is not well supported in virtualized systems, which motivates us to design a NUMA-aware VCPU scheduler to improve the performance of memory-intensive applications while maintaining the transparency of the virtualization layer.

III. Design

In this section, we introduce the design of our VCPU scheduler, named *vProbe*. We first describe the system

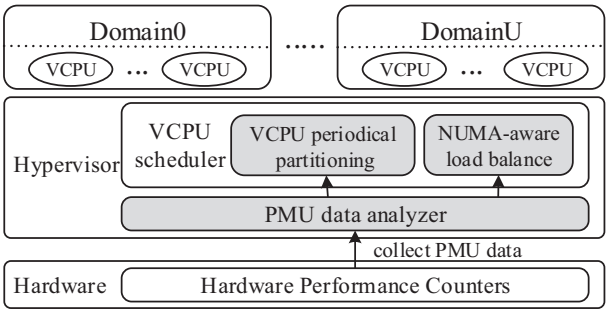


Fig. 2: Overview of *vProbe*

overview of *vProbe*. Then, we give a detailed description of each component of *vProbe*.

A. *Overview*

vProbe improves the performance of memory-intensive applications on virtualized servers based on NUMA architecture via optimized VCPU allocation and migration strategies, which considers applications' memory access characteristics to alleviate shared resource contention and reduce unnecessary remote memory accesses while maintaining the transparency of the virtualization layer.

Figure 2 gives an overview of *vProbe*, which consists of three major components: *PMU data analyzer*, *VCPU periodical partitioning*, and *NUMA-aware load balance* mechanisms. More specifically, the *PMU data analyzer* collects PMU data to analyze the memory access characteristics of each VCPU. At the end of each sampling period, in order to alleviate shared resource contention and reduce unnecessary remote memory accesses, the *VCPU periodical partitioning* mechanism reassigns all memory-intensive VCPUs to each node evenly while preferentially guaranteeing them to run on their local nodes, according to their memory access characteristics. Moreover, the *NUMA-aware load balance* mechanism enables an idle PCPU to preferentially steal a VCPU from the run queues of PCPUs in the local node, which aims to maintain balanced LLC contention while avoiding extra remote memory accesses.

B. *PMU Data Analyzer*

In order to make proper scheduling decisions in NUMA-based servers, the VCPU schedulers need to know the memory access characteristics of applications running in VMs. In this section, we present the *PMU data analyzer* to collect PMU data for each VCPU and analyze their memory access characteristics.

In the following part, we introduce *memory node affinity* and *LLC access pressure* which are considered as the typical memory access characteristics of VCPUs. The former represents the distribution of node memory accessed by a VCPU, and the latter represents a VCPU's demand for shared resources.

*1) **VCPU's Memory Node Affinity**:* As discussed in Section II-A, remote memory access latency can significantly affect the performance of memory-intensive applications on virtualized servers based on NUMA architecture. Thus, we need to know the memory location of each VCPU, i.e., the node that this VCPU's pages exist in, which enables VCPU schedulers to optimize scheduling strategies to reduce unnecessary remote memory accesses and alleviate interconnect link contention.

We introduce *memory node affinity* of a VCPU to denote which memory node this VCPU's pages exist in. More specifically, the *PMU data analyzer* sets a VCPU' *memory node affinity* to the *id* of the node that has the maximum number of pages accessed by this VCPU during the current sampling period. It is obvious that the *memory node affinity* of a VCPU indicates the node that is most suitable for this VCPU to run on during the current sampling period if we only consider the influence of remote memory access latency. What is more, because of the data locality, the *memory node affinities* of all VCPUs are crucial for the optimizations on remote memory access latency in the next sampling period.

Before discussing the calculation of the *memory node affinity*, we define some variables as follows:

1) vc: a VCPU.
2) N: the number of nodes in the NUMA system.
3) $N(vc, i)$: the number of pages in the ith node, which are accessed by vc during the current sampling period.

At the end of the current sampling period, the *PMU data analyzer* calculates the *memory node affinity* of vc as follows. It first collects PMU data for vc, including the number of pages accessed by vc in each node, i.e., $\{N(vc,1), N(vc,2), \ldots, N(vc,N)\}$. Then, it sets the *memory node affinity* of vc to the node id where id is represented as follows.

$$N(vc, id) = \max_{1 \le i \le N} \{N(vc, i)\} \tag{1}$$

*2) **VCPU's LLC Access Pressure**:* Apart from remote memory access latency, shared resource contention can significantly affect the performance of memory-intensive applications as well. As discussed in Section I, the VCPU schedulers of hypervisors probably introduce significant shared resource contention due to their disregards for the characteristics of applications running in VMs. To address this problem, we need to find an effective metric that can represent VCPUs' demands for shared resources.

In this paper, we introduce *LLC access pressure* of a VCPU to denote its demand for shared resources. This is because alleviating LLC contention can also reduce the number of memory accesses which alleviates memory controller and interconnect link contentions. Previous studies [13–15] use the LLC miss rate to represent an application's LLC demand, but the LLC miss rate is instable because it has a direct relation with interfering workloads. Therefore, we calculate a VCPU's *LLC access pressure* according to Equation (2), whose effectiveness and stability have been demonstrated in [16].

$$R_{LLC_{ref}} = \frac{LLC_ref}{InstrucRetired} \cdot \alpha \tag{2}$$

The variables used in Equation (2) are defined as follows:

1) LLC_{ref}: The number of last level cache references.
2) $InstrucRetired$: The number of retired instructions.
3) α: A constant which is used to adjust the ratio of LLC_{ref} and $InstrucRetired$ to an appropriate order of magnitude.

What's more, in order to help VCPU schedulers to make more flexible scheduling decisions, we need to classify all

VCPUs into different *types* according to their *LLC access pressures*. VCPUs that belong to the same *types* have similar *LLC access pressures*. More specifically, we empirically classify VCPUs into three *categories* as follows:

- **LLC thrashing (LLC-T)**: LLC-T VCPUs have lots of LLC misses because of their large cache demands even when there is no interfering VCPUs.
- **LLC fitting (LLC-FI)**: The capacity of a LLC can meet the requirements of LLC-FI VCPUs, but the LLC miss rate of these VCPUs can significantly increase if interfering VCPUs introduce serious LLC contention.
- **LLC friendly (LLC-FR)**: The performance of the applications running on LLC-FR VCPUs has almost no correlation with LLC capacity because of their negligible demands on LLC.

We classify all VCPUs into different *types* due to the following reason. Although *LLC access pressure* is an effective metric that can help VCPU schedulers to alleviate LLC contention, memory allocation of VMs on NUMA system makes performance optimization more complicated. As the VCPUs that belong to the same *types* have similar *LLC access pressures*, it gives greater flexibility for the VCPU scheduler to schedule VCPUs according to their *types* instead of *LLC access pressures*.

Besides, we also define two bounds, named *low* and *high*, to distinguish VCPUs from different *categories* according to their *LLC access pressures* as shown in Equation (3). (We will illustrate how to determine the values of these bounds in Section IV-A.)

$$VCPU\,type = \begin{cases} LLC\text{-}FR & if\ R_{LLC_{ref}} < low \\ LLC\text{-}FI & if\ low \leq R_{LLC_{ref}} < high \\ LLC\text{-}T & if\ R_{LLC_{ref}} \geq high \end{cases} \quad (3)$$

In summary, the *PMU data analyzer* collects PMU data to calculate all VCPUs' *memory node affinities* and *LLC access pressures*. Because PMU data are collected at the hypervisor level, guest OSes do not need to be modified, which maintains the transparency of the virtualization layer. In the following, we describe how to optimize VCPU scheduling strategy by considering all VCPUs' *memory node affinities* and *LLC access pressures*.

C. VCPU Periodical Partitioning

In this section, we propose a *VCPU periodical partitioning* mechanism to alleviate shared resource contention while reducing unnecessary remote memory accesses. It periodically reassigns all memory-intensive VCPUs whose *types* are LLC-T or LLC-FI to each node evenly while preferentially allocating them to their local nodes, according to their *memory node affinities* and *types*. Because LLC-FR VCPUs are non-memory-intensive ones, we use the default scheduling strategy to schedule them to maintain load balance. In the following, we describe this mechanism in detail.

The variables and functions used in this section are presented in Table II.

At the end of each sampling period, the *VCPU periodical partitioning* mechanism is activated. Initially, it marks all memory-intensive VCPUs as *unassigned*. Then, it continuously finds MIN-NODE and assigns a suitable *unassigned* VCPU to this node, and marks this VCPU as *reassigned*.

TABLE II: Variables and Functions used in this section

Variables and Functions	Description
$node_i$	the ith NUMA node where $1 \leq i \leq N$
reassigned VCPU	a memory-intensive VCPU which has been reassigned to a node
unassigned VCPU	a memory-intensive VCPU which has not been reassigned to any node
MIN-NODE	the node with the minimum number of *reassigned* VCPUs
groupOfVc(c,p)	a group of *unassigned* VCPUs whose *categories* are *c* and *memory node affinities* are *p*
node.reassigned_load	the number of memory-intensive VCPUs reassigned to *node*
$\max_{1 \leq i \leq N}$ {*groupOfVc(Type,i)*}	calculate the group that has the maximum number of VCPUs whose *type* are *Type*

Because LLC contention on MIN-NODE is relatively small, preferentially assigning a suitable memory-intensive VCPU to MIN-NODE can help to balance LLC contention among all nodes. More specifically, the *VCPU periodical partitioning* mechanism finds MIN-NODE and preferentially selects a VCPU from the *unassigned* LLC-T VCPUs according to their *memory node affinities* as follows.

First, MIN-NODE is set to the node that has been assigned the minimum number of memory-intensive VCPUs, i.e., the node that has the smallest value of *reassigned_load*. The value of *reassigned_load* is updated as follows: (a) Initially, all nodes' *reassigned_loads* are set to 0, which means that no memory-intensive VCPUs are assigned to any node when the *VCPU periodical partitioning* mechanism is activated; (b) The value of *reassigned_load* of a node is increased by 1 when a memory-intensive VCPU is assigned to this node.

Second, the *VCPU periodical partitioning* mechanism selects a VCPU from the *unassigned* LLC-T VCPUs while considering their *memory node affinities*. More specifically, it preferentially selects the first VCPU of the group *groupOfVc(*LLC-T, MIN-NODE.id*)* to guarantee that the selected VCPU will be assigned to its local node, which alleviates LLC contention while avoiding unnecessary remote memory accesses. However, if there is no available VCPU in that group, it selects the first VCPU of the group $\max_{1 \leq i \leq N}$ {*groupOfVc(*LLC-T, i*)*}, which has the maximum number of VCPUs. In this case, it alleviates LLC contention while minimizing the differences in the sizes of all *groupOfVc(*LLC-T, i) where $1 \leq i \leq N$, which improves the possibility that other VCPUs will be assigned to their local nodes.

Moreover, if all LLC-T VCPUs have been assigned, it uses the same principles to assign an *unassigned* LLC-FI VCPU to MIN-NODE.

The pseudo-code of the *VCPU periodical partitioning* mechanism is shown in Algorithm 1. When there are *unassigned* VCPUs, it finds MIN-NODE and assigns a suitable memory-intensive VCPU to this node as follows. First, it calls function *getMinNode()* to find MIN-NODE (line 2) and determines the *type* (denoted *Type*) of the VCPU which will be assigned to this node (line 3∼6). Second, it selects a suitable VCPU that should be assigned to MIN-NODE (line 7∼11). Finally, it marks the selected VCPU as *reassigned* (line 12) and migrates it to MIN-NODE (line 13).

Algorithm 1: VCPU Periodical Partitioning Algorithm

Input: N: number of nodes in NUMA system, statistics of *groupOfVc(c,p)* where $c \in$ {LLC-T, LLC-FI} and $1 \leq p \leq N$.
Output: scheduling decision

```
1  while there are unassigned VCPUs do
2      MIN-NODE ← getMinNode(all_nodes);
3      Type ← LLC-T ;
4      if there is no unassigned LLC-T VCPU then
5          Type ← LLC-FI;
6      end
7      if groupOfVc(Type,MIN-NODE.id) is not empty
       then
8          set vc to the first VCPU of groupOfVc(Type,
           MIN-NODE.id);
9      else
10         set vc to the first VCPU of
           max_{1≤j≤N} {groupOfVc(Type,j)};
11     end
12     mark vc as a reassigned VCPU;
13     migrate(vc,MIN-NODE);
14 end
15 return;
```

D. NUMA-aware Load Balance

In this section, we propose a *NUMA-aware load balance* mechanism to maintain balanced LLC contention while avoiding unnecessary remote memory accesses via appropriate VCPU migration when PCPUs become idle. Because of the semantic gap between hypervisors and guest OSes, the VCPU schedulers of hypervisors pay little attention to the characteristics of applications running in VMs. As a result, although the *VCPU periodical partitioning* mechanism can balance LLC contention, the default load balance strategy may break this balance by frequently migrating memory-intensive VCPUs among different nodes when PCPUs become idle, which probably causes significant performance degradation. Besides, the number of remote memory accesses significantly increases due to the migration of memory-intensive VCPUs as well. Thus, we propose the *NUMA-aware load balance* mechanism to address these problems. When a PCPU becomes idle, it preferentially selects a suitable VCPU from the run queues of PCPUs in the local node, which helps to maintain balanced LLC contention and avoid unnecessary remote memory accesses. In the following, we describe this mechanism in detail.

When a PCPU becomes idle, it activates the *NUMA-aware load balance* mechanism and preferentially steals a VCPU from the run queues of PCPUs in the local node. Moreover, in order to reduce context switches and keep the load balance among all PCPUs, it preferentially checks the run queue of the PCPU that has the heaviest workload. If there are several runnable VCPUs in the PCPU's run queue, it steals the one that has the smallest value of *LLC access pressure*. Because VCPUs with smaller *LLC access pressure* have a relatively small impact on LLC contention, migrating them to the idle PCPU can avoid breaking the balance of LLC contention. Otherwise, if no runnable VCPUs are found, it uses the same principles to check the run queues of other PCPUs in the local node.

What is more, if there are no runnable VCPUs on the local node, it steals a suitable VCPU from the remote nodes to utilize available CPU resources. The principles of stealing a VCPU from the remote nodes are the same as for the local node. Because migrating a memory-intensive VCPU from the remote nodes to the idle PCPU may break the balance of LLC contention and introduce extra remote memory accesses, it preferentially steals a runnable VCPU with smaller *LLC access pressure*, which aims to alleviate these problems.

Algorithm 2: NUMA-aware Load Balance Algorithm

Input: run-queue information of the PCPUs where the scheduler resides, the idle $pcpu$, $loadList$ of each node which maintains its PCPUs sorted in descending order according to their workloads
Output: load balance decision

```
1  node ← pcpu_to_node(pcpu);
2  while true do
3      foreach p in loadList of node do
4          if there are runnable VCPUs in p's run queue
           then
5              vcpu ← a runnable VCPU in p's run queue,
               which has the smallest value of LLC access
               pressure;
6              break;
7          end
8      end
9      node ← nextNode();
10 end
11 migrate(vcpu, pcpu);
12 return;
```

The pseudo-code of the *NUMA-aware load balance* mechanism is shown in Algorithm 2. When a PCPU becomes idle, it steals a VCPU from the run queues of peer PCPUs. It preferentially checks the local node (line 1), then the remote ones obtained by the function *nextNode()* (line 9). Besides, among all the PCPUs in a node, it preferentially steals a runnable VCPU with the smallest value of *LLC access pressure* from the run queue of the PCPU (*p* in the algorithm) that has the heaviest workload (line 3∼8). Finally, it migrates the VCPU to the idle PCPU by the function *migrate()* (line 11).

IV. System Implementation

We implement a prototype of *vProbe* based on the Credit scheduler of Xen-4.0.1. In the following, we first describe how to determine the bounds for VCPU type. Then, we describe the modification to the Credit scheduler.

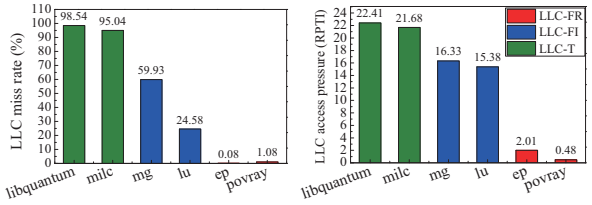
A. Determining Bounds for VCPU Type

We conduct experiments to estimate the two bounds, i.e *low* and *high*, in Equation (3) for *VCPU type*. We first measure the LLC miss rates of the selected applications and classify them into different *types*. Then, we calculate the *LLC access pressures* of different categories of applications to determine the range of the two bounds.

The detailed configuration of our NUMA system is presented in Table I. In this experiment, we run a VM (VM1)

which is configured with 4 GB memory and 1 VCPU pinned to the local node.

According to the definition of VCPU types described in Section III-B2 and test results shown in Figure 3(a), we choose two applications for each *types*: LLC-FR (povray, ep), LLC-FI (lu, mg) and LLC-T (milc, libquantum).



(a) Results of LLC miss rate (b) Results of RPTI

Fig. 3: LLC miss rate and LLC *references per thousand instructions* (RPTI) for each applications

With the collected information, we calculate *LLC access pressure* of each application according to Equation (2) (the α parameter is empirically set to 1000). As shown in Figure 3(b), *LLC access pressures* of LLC-FR VCPUs are less than 3 (i.e., povray is 0.48 and ep is 2.01) and *LLC access pressures* of LLC-T VCPUs are greater than 20 (i.e., milc is 21.68 and libquantum is 22.41). As to LLC-FI VCPUs, *LLC access pressures* of lu and mg are 15.38 and 16.33 respectively. Thus, we choose 3 and 20 as the bounds. Besides, if the value of *low* or *high* changes, the number of LLC-T (or LLC-FI) VCPUs will increase (or decrease), which can affect the *VCPU periodical partitioning* mechanism.

B. Modification to the Credit Scheduler

We implement *vProbe* based on the Credit scheduler. In the following, we describe the modification in detail.

First, we patch Xen with *Perfctr-Xen* [17] to obtain runtime information for each VCPU from low-level hardware performance counters, including LLC reference, instructions that retire execution, and the number of local and remote memory accesses. A running VCPU's runtime information is updated before VCPU context switch or every 10ms after this VCPU burns its credits.

Second, we add two fields named *low* and *high* to structure *csched_priv*, which denote the bounds used for VCPU types and are initialized to 3 and 20 respectively. We also add three fields named *node_affinity*, *LLC_pressure*, and *vcpu_type* to structure *csched_vcpu*, which denote the *memory node affinity*, *LLC access pressure*, and *type* of a VCPU. At the end of each sampling period, the *PMU data analyzer* updates each VCPU's *node_affinity* according to Equation (1), *LLC_pressure* according to Equation (2), and *vcpu_type* according to Equation (3). Moreover, the sampling period is experimentally set to 1s. (We will illustrate how the sampling period affects the performance of *vProbe* in Section V-C2.)

Finally, we add a variable *workload* to each PCPU, which records the number of VCPUs in its run queue. A PCPU's *workload* is increased by 1 if a VCPU inserts into its run queue. Besides, a PCPU's *workload* is decreased by 1 if a VCPU removes from its run queue. When a PCPU becomes idle, it preferentially selects a runnable VCPU with the smallest value of *LLC_pressure* from the run queues of PCPUs in the local node, then the remote nodes if necessary. Moreover, as to all PCPUs in a certain node, it preferentially checks the run queue of the PCPU with the largest value of *workload*.

V. Performance Evaluation

In this section, we evaluate the performance of *vProbe* through different kinds of test scenarios and workloads, and study the overheads introduced by *vProbe*. In the following, we first describe the experimental methodology. Then, we analyze the experimental results.

A. Experimental Methodology

(1) Experimental platform. The detailed configuration of our NUMA system is shown in Table I. The hypervisor is Xen-4.0.1. All VMs run the CentOS 5.5 Linux distribution with a Linux 2.6.32.39 kernel.

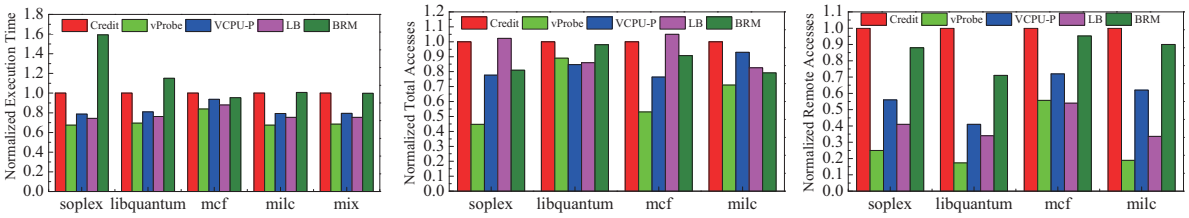
In the evaluation, we conduct experiments on three VMs (VM1∼VM3) configured with 8 VCPUs, and VM1 runs memory-intensive applications while VM2 and VM3 run interfering workloads. The memory allocations and workloads of all VMs are as follows: (a) VM1 has 15GB memory, which is split into two nodes to provide a more variable and complicated runtime environment. (b) VM2 has 5GB memory. It is an interfering VM which runs the same workloads as VM1. (c) VM3 is also an interfering VM which runs eight hungry-loop applications to consume available CPU resources. Because of the small memory demand, VM3 only has 1GB memory. We collect runtime information of applications running in VM1 to evaluate our system performance.

(2) Scheduling approaches. In this evaluation, we evaluate the performance of applications under several scheduling approaches as follows:

- Credit: the default scheduler of Xen hypervisor.
- *vProbe*: the scheduler proposed in this paper, which consists of *PMU data analyzer*, *VCPU periodical partitioning*, and *NUMA-aware load balance* mechanisms.
- *VCPU periodical partitioning (VCPU-P)*: It only implements the *VCPU periodical partitioning* mechanism, which is used to demonstrate its importance.
- *NUMA-aware Load balance (LB)*: It only implements the *NUMA-aware load balance* mechanism, which is used to demonstrate its importance.
- *Bias Random vCPU Migration (BRM)*: It is a Bias Random vCPU Migration scheduler proposed in [4], which aims to minimize the system-wide *uncore penalty*.

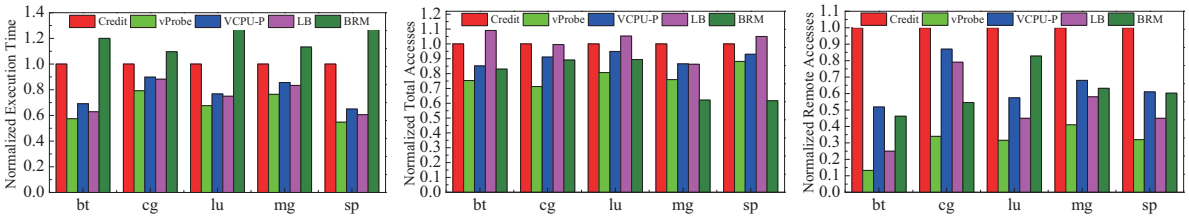
(3) Metrics. Apart from the general performance metric, i.e., average throughput for Redis and normalized execution time for the other applications, we use two other metrics to evaluate our system performance: total and remote memory accesses numbers. The total number of memory accesses reflects memory controller and LLC contention situations. The number of remote memory accesses reflects the remote memory accesses and interconnect link contention situations. These two metrics will help to give an insight on how the above scheduling approaches affect the system performance.

(4) Classification of experiments. We conduct two different categories of experiments to evaluate both the improvement and the overhead of *vProve* on NUMA system.



(a) Normalized Execution Time (b) Normalized Total Memory Accesses (c) Normalized Remote Memory Accesses

Fig. 4: The experimental results of SPEC CPU2006



(a) Normalized Execution Time (b) Normalized Total Memory Accesses (c) Normalized Remote Memory Accesses

Fig. 5: The experimental results of NPB

(5) Benchmarks and applications. The benchmarks and applications used in our experiments are as follows.

- SPEC CPU2006 [12]: a set of single-threaded programs, which is used to quantify the performance of a system's computer processor, memory subsystem and compiler.
- NPB [11]: a set of MPI programs used to evaluate the performance of clusters and supercomputers.
- Memcached [18]: An in-memory key-value store, which is used to improve the performance of web applications.
- Redis [19]: An in-memory key-value database, which is used to provide high performance for database systems.

B. Experiments with Memory-Intensive Applications

In this section, we evaluate the performance of *vProbe* with several memory-intensive applications and compare it with other related scheduling approaches.

*1) **Experiments with SPEC CPU2006**:* We select four memory-intensive applications (i.e., *soplex, libquantum, mcf,* and *milc*) from the SPEC CPU2006 suite and form five workloads: (a) We conduct four identical workloads for each application and each workload runs four identical instances to evaluate the influence of different VCPU schedulers on these applications. Specifically, because VM2's memory can only support two instances of the *mcf* application, we run six instances of the *mcf* in VM1 and two instances in VM2 to guarantee that all four workloads have the same total number of instances. (b) The fifth workload *mix* consists of all four applications and each application runs one instance to evaluate the influence of different VCPU schedulers on a mixed workload. We run the five workloads above in VM1 and VM2, and calculate the average runtime of applications in VM1. As to the workload *mix*, we calculate the normalized execution time of each application, and use the average of their normalized execution times as the final result. The test results are shown in Figure 4.

The normalized execution time, total and remote memory accesses numbers of these applications are shown in Figure 4(a)∼(c) respectively. The results show that *vProbe* has the best performance in all workloads. As shown in Figure 4(a), *vProbe* obtains the best performance improvement in workload *soplex*. Compared with the Credit scheduler, *VCPU-P* and *LB*, *vProbe* achieves 32.5%, 16.6%, and 10.2% improvements respectively. *VCPU-P* improves the performance, but performs worse than *vProbe* because of ignoring the load balance strategy. Besides, *LB* can reduce remote memory accesses and improve the performance of these applications. But the total number of memory accesses of some applications, such as *soplex* and *mcf*, increases slightly. This is because *LB* ignores LLC contention problems, which can reduce the LLC hit rate, and increase the total memory accesses. As a result, *LB* attains worse performance than *vProbe*. As to *BRM*, it attains similar or worse performance than the Credit scheduler, although it can reduce the total and remote memory accesses. This is because *BRM* schedules VCPUs according to each VCPU's *uncore penalty*, and it needs to acquire a system-wide lock before updating a VCPU's *uncore penalty*. Thus, when the number of VCPUs is large, i.e., greater than 8, the lock contention problem introduces significant overheads.

*2) **Experiments with NPB**:* We select five memory-intensive applications (i.e., *bt, cg, lu, mg,* and *sp*) from the NPB suite while each application is configured with four threads. For each test, we run the same application in VM1 and VM2 under different VCPU schedulers, and analyze the execution time and memory accesses of the application running in VM1. The experimental results are shown in Figure 5.

As shown in Figure 5(a)∼(c), *vProbe* still outperforms other related VCPU schedulers because it can alleviate LLC contention and reduce unnecessary remote memory accesses. Figure 5(a) shows that the best case of performance improvement happens when we run the workload *sp*. For workload *sp*, compared with the Credit scheduler, *VCPU-P*, and *LB*, *vProbe* achieves 45.2%, 15.7%, and 9.6% improvements re-

spectively. As shown in Figure 5(b), compared with the Credit scheduler, *LB* increases the total number of memory accesses of some applications (e.g., *bt*, *lu*, and *sp*) due to its ignorance of LLC contention. Moreover, although *LB* has a larger total number of memory accesses, it outperforms *VCPU-P* because *LB* preferentially maintains VCPUs to run on the current node when VCPU migration occurs and alleviates remote memory accesses. Unfortunately, *BRM* still presents the worst performance due to its lock contention problem, which is similar to the SPEC CPU2006 results.

*3) **Experiments with Memcached**:* Memcached is a widely used distributed memory caching system. In this test, each memcached server is configured with eight working ports for the load process and deployed in VM1 and VM2 respectively. We use the *memslap* [20] tool to evaluate the performance of the memcached server. For each test, we start up *memslap* with a variable number of concurrent calls ranging from 16 to 112, and record the total time of continuously executing the given test 50,000 times. The experimental results are shown in Figure 6.

As shown in Figure 6(a)∼(c), *vProbe* has the best performance for memcached servers. Figure 6(a) shows that the greatest performance improvement happens when the number of concurrent calls is 80. With this number of concurrent calls, compared with the Credit scheduler, *VCPU-P* and *LB*, *vProbe* achieves 31.3%, 13.2%, and 17.3% improvements respectively. *LB* outperforms *VCPU-P* when the number of concurrent calls is small, e.g., 16 and 32, because the LLC contention problem is not serious and remote memory access latency is the major performance-degrading factors. However, with the increasing number of concurrent calls, the LLC contention becomes the major degradation factor and *VCPU-P* attains better performance than *LB*. Besides, due to the lock contention problem, *BRM* attains better performance than the Credit scheduler and worse performance than the other VCPU schedulers.

*4) **Experiments with Redis**:* Redis is an open-source, networked, in-memory and key-value database, which has a large number of memory accesses. In this test, we use the *redis-benchmark* tool to generate a load *get* for performance evaluation of redis servers. For each test, we run four redis servers and four *redis-benchmark* tools in VM1 and VM2 respectively. For each *redis-benchmark* tool, the number of parallel connections ranges from 2,000 to 10,000, and the total number of requests is 100 million. We conduct experiments under different VCPU schedulers and present the experimental results in Figure 7.

The throughput, normalized total and remote memory accesses numbers are shown in Figure 7(a)∼(c). We can see that *vProbe* outperforms other related VCPU schedulers. Figure 7(a) shows that the greatest performance improvement happens when the number of parallel connections is 2000. With this number of parallel connections, compared with the Credit scheduler, *VCPU-P* and *LB*, *vProbe* achieves 26.0%, 13.3%, and 16.8% improvements respectively. As shown in Figure 7(a), *VCPU-P* outperforms *LB*. This is because *VCPU-P* can significantly alleviate LLC contention, which is the major performance-degrading factor for redis servers. Besides, although *BRM* can reduce the number of remote memory accesses, it attains similar performance to the Credit scheduler due to the lock contention problem.

*5) **Experimental Results Analysis**:* In this subsection, we analyze the results of the above experiments and draw the following conclusions.

- The Credit scheduler does not consider the memory access characteristics of applications running in VMs, which probably introduces significant LLC contention and remote memory accesses. Thus, it cannot guarantee the performance of memory-intensive applications.
- *vProbe* has the best performance among all the related VCPU schedulers. This is because it optimizes VCPU allocation and migration strategies, which reduces LLC contention and unnecessary remote memory accesses.
- Both *VCPU-P* and *LB* outperform the Credit scheduler, but they have lower performance than *vProbe*. This is because *VCPU-P* ignores the importance of the load balance strategy, which probably brings unbalanced LLC contention and remote memory accesses by frequently migrating memory-intensive VCPUs among all nodes. *LB* ignores the importance of balanced LLC contention, which also causes performance degradation.
- *BRM* attains lower or similar performance to the Credit scheduler. This is because it needs to acquire a system-wide lock before updating a VCPU's *uncore penalty*. Thus, although it can reduce the total and remote memory accesses, the lock contention problem can introduce significant performance degradation.

C. Overheads

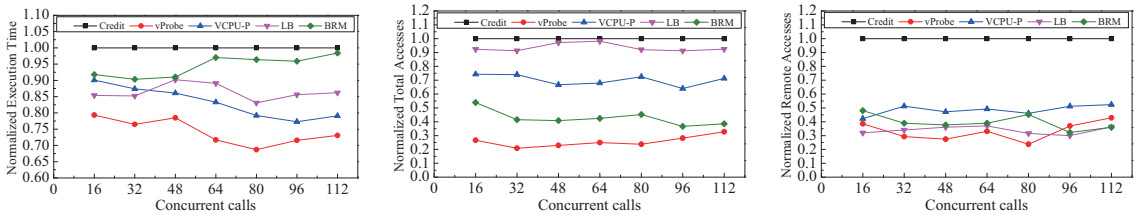
In this section, we first evaluate the overheads introduced by *vProbe*. Then, we conduct experiments to analyze how the sampling period affects the performance of *vProbe*.

*1) **Overheads of vProbe**:* The overheads of *vProbe* come from several sources: (a) the time required to collect PMU data; (b) the time required for the *VCPU periodical partitioning* mechanism to reassign all memory-intensive VCPUs to suitable nodes. For simplicity, we call the above extra cost "overhead time". In this section, we conduct experiments to evaluate the percentage of "overhead time" in the total execution time. We create one to four VMs configured with 4GB memory and 2 VCPUs. Each VM runs two instances of the *soplex* application. When there are 4 VMs, the number of VCPUs is equal to that of PCPUs, which means all PCPUs are busy and the amount of PMU data collection in parallel reaches its maximum limit.

TABLE III: The experimental results of "overhead time"

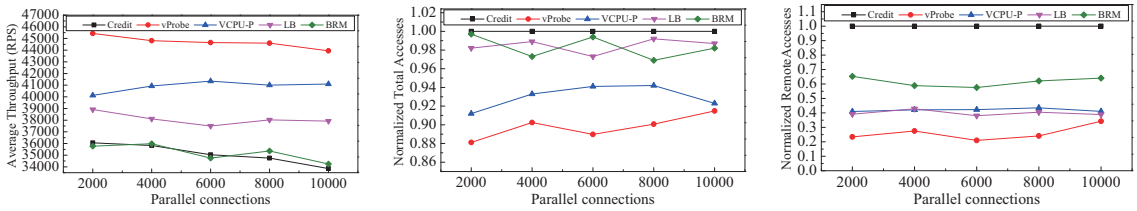
Number of VMs	The percentage of "overhead time"
1	0.00847
2	0.01206
3	0.01619
4	0.01062

As shown in Table III, with the increasing number of VMs, the percentage of "overhead time" increases. However, as concerns the case of 4 VMs, the number of total VCPUs of DomUs is 8, which is equal to the number of PCPUs. As a result, the amount of VCPU migration reduces, which decreases the frequency of PMU data collection and extra overheads. Thus, the "overhead time" is even smaller when running 4 VMs. Table III shows that the percentage of



(a) Normalized Execution Time (b) Normalized Total Memory Accesses (c) Normalized Remote Memory Accesses

Fig. 6: The experimental results of Memcached



(a) Average Throughput (b) Normalized Total Memory Accesses (c) Normalized Remote Memory Accesses

Fig. 7: The experimental results of Redis

"overhead time" is much lower than 0.1%. That is to say, *vProbe* introduces negligible overheads.

*2) **The Sampling Period**:* To study the influence of the sampling period, we set the sampling period vary from 0.1s to 10s and run workload *mix* described in Section V-B1. The experimental results are presented in Figure 8.

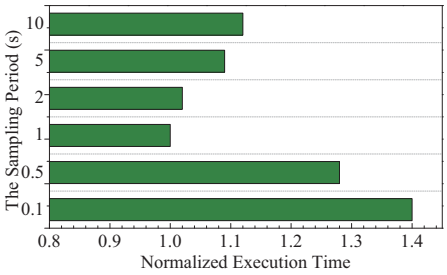


Fig. 8: The experimental results of workload *mix* under variable sampling period

As show in Figure 8, the performance of *vProbe* increases when the sampling period increases from 0.1s to 1s, because the "overhead time" becomes serious when the sampling period is short. However, *vProbe* presents worse performance when the sampling period is bigger than 1s, because the memory access characteristics of VCPUs cannot be updated timely. As a result, the VCPU scheduler cannot make suitable scheduling decisions timely. Thus, we choose 1s as the sampling period in this paper.

VI. Discussion and Future Work

Dynamic bounds for *VCPU type*. In this paper, the bounds for *VCPU type* used by *vProbe* are set manually. Although *vProbe* can significantly improve the performance of memory-intensive applications in VMs based on NUMA system, dynamically adjusting these bounds according to the workload in VMs will make *vProbe* more adaptable to different real-word applications. Thus, our future work is to study how to adjust these bounds dynamically according to the workloads in the system.

Page migration. In this paper, currently we only focus on optimizing VCPU scheduling strategy. In fact, page migration may be an alternative way to alleviate shared resource contention and reduce remote memory accesses. On the one hand, page migration can effectively alleviate memory controller contention and avoid remote memory access. On the other hand, the cost of page migration is expensive while VCPU scheduling strategy is relatively cheap. Thus, in the future, we will study how to combine VCPU scheduling and page migration strategies to improve the performance of our system.

VII. Related Work

There are many studies that optimize NUMA system performance in traditional environments.

On one hand, many studies optimize NUMA system performance via proper thread scheduling. Su *et al.* [2] present a NUMA-aware thread placement algorithm which considers the critical path to improve the performance of OpenMP programs. Blagodurov *et al.* [1] present a method to schedule threads according to their LLC misses. Besides, it migrates the partial pages along with the thread. Majo *et al.* [3] present a novel thread placement algorithm which attempts to benefit from data locality while avoiding significant LLC contention.

On the other hand, page management strategies are also effective in improving the performance of memory-intensive applications. Lachaize *et al.* [9] improve the performance of memory-intensive applications by reducing the number of remote memory accesses via page replication, interleaving and allocation strategies. Dashti *et al.* [10] identify that remote memory access latency is not the major performance-degrading factor, and alleviate shared resource contention via page interleaving, page co-location, page replication, and thread clustering. Goglin *et al.* [21] implement a system call to enable high-performance memory migration in Linux.

Besides, many studies [22–25] improve NUMA system performance at the application level. However, the above studies are not fit for virtualized environments because NUMA architecture is invisible to guest OSes.

In virtualized environments, Rao *et al.* [4] present a NUMA-aware VCPU scheduler to schedule VCPUs according to their penalties to access the "uncore" memory subsystem. However, it needs to acquire a system-wide lock

before updating a VCPU's *uncore penalty*, which restricts its scalability. Rao *et al.* [5] present a vNUMA-mgr approach that exposes the virtual NUMA topology to VMs, which enables developers to optimize system performance at the guest OS or application level. Liu *et al.* [6] leverage NUMA overhead awareness in the hypervisor's memory management, but it ignores the importance of VCPU scheduling. Ibrahim *et al.* [26] use partitioning and direct inter-VM shared memory support to improve the performance of high performance computing applications. However, this only improves the access latency of shared memory blocks. What is more, the above studies need to modify the guest OS kernel, which breaks the transparency of the virtualization layer and restricts the applicability of the proposed methods.

VIII. Conclusion

In this paper, we design and implement a NUMA-aware VCPU scheduler, named *vProbe*. By considering the memory access characteristics of each VCPU, *vProbe* optimizes VCPU allocation and migration strategies to alleviate shared resource contention and reduce unnecessary remote memory accesses while maintaining the transparency of the virtualization layer, which improves the performance of memory-intensive applications on virtualized servers based on NUMA architecture. More specifically, we introduce *PMU data analyzer* to collect PMU data for each VCPU and calculate their *memory node affinities* and *LLC access pressures*. In order to alleviate shared resource contention and reduce remote memory accesses, we present *VCPU periodical partitioning* mechanism to periodically reassign all memory-intensive VCPUs to suitable nodes according to their *memory node affinities* and *categories*. We also propose a *NUMA-aware load balance* mechanism to preferentially steal a suitable VCPU from run queues of PCPUs in local node to an idle PCPU, which aims to maintain balanced LLC contention and avoid extra remote memory accesses. Finally, we conduct experiments to validate the effectiveness and overheads of *vProbe*. The experimental results show that *vProbe* can significantly improve the performance of memory-intensive applications (e.g., up to 45.2% performance improvement compared with the Credit scheduler) while introducing negligible overheads.

IX. Acknowledgement

This research is supported by National Science Foundation of China under grants No. 61472151 and No. 61232008, National Key Research and Development Program under grant 2016YFB1000500, National 863 Hi-Tech Research and Development Program under grant No. 2015AA01A203, and the Fundamental Research Funds for the Central Universities under grant HUST:2015TS067.

References

[1] S. Blagodurov, S. Zhuravlev, A. Fedorova, and A. Kamali, "A case for NUMA-aware contention management on multicore systems," in *Proc. of USENIX ATC*, 2010, pp. 557–558.

[2] C. Su, D. Li, D. S. Nikolopoulos, M. Grove, K. Cameron, and B. R. De Supinski, "Critical path-based thread placement for NUMA systems," *ACM SIGMETRICS Performance Evaluation Review*, pp. 106–112, 2012.

[3] Z. Majo and T. R. Gross, "Memory management in NUMA multicore systems: Trapped between cache contention and interconnect overhead," in *Proc. of ISMM*, 2011, pp. 11–20.

[4] J. Rao, K. Wang, X. Zhou, and C.-Z. Xu, "Optimizing virtual machine scheduling in NUMA multicore systems," in *Proc. of HPCA*, 2013, pp. 306–317.

[5] D. S. Rao and K. Schwan, "vNUMA-mgr: Managing VM memory on NUMA platforms," in *Proc. of HiPC*, 2010, pp. 1–10.

[6] M. Liu and T. Li, "Optimizing virtual machine consolidation performance on NUMA server architecture for cloud workloads," in *Proc. of ISCA*, 2014, pp. 325–336.

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, 2003.

[8] L. Cherkasova, D. Gupta, and A. Vahdat, "Comparison of the three CPU schedulers in Xen," *SIGMETRICS Performance Evaluation Review*, vol. 35, no. 2, pp. 42–51, 2007.

[9] R. Lachaize, B. Lepers, and V. Quéma, "Memprof: A memory profiler for NUMA multicore systems," in *Proc. of USENIX ATC*, 2012, pp. 53–64.

[10] 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 *Proc. of ASPLOS*, 2013, pp. 381–394.

[11] D. Baliley, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, "The NAS parallel benchmarks-summary and preliminary results," in *Proc. of SC*, 1991.

[12] Standard Performance Evaluation Corporation CPU 2006 (SPECCPU2006). http://www.spec.org/cpu2006/.

[13] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *Proc. of ASPLOS*, 2010, pp. 129–142.

[14] A. Jaleel, H. H. Najaf-Abadi, S. Subramaniam, S. C. Steely, and J. Emer, "Cruise: cache replacement and utility-aware scheduling," in *Proc. of ASPLOS*, 2012, pp. 249–260.

[15] D. Tam, R. Azimi, and M. Stumm, "Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors," in *Proc. of EuroSys*, 2007, pp. 47–58.

[16] S.-g. Kim, H. Eom, and H. Y. Yeom, "Virtual machine scheduling for multicores considering effects of shared on-chip last level cache interference," in *Proc. of IGCC*, 2012, pp. 1–6.

[17] R. Nikolaev and G. Back, "Perfctr-xen: a framework for performance counter virtualization," in *Proc. of VEE*, 2011, pp. 15–26.

[18] A distributed memory object caching system (memcached). http://www.memcached.org.

[19] Redis. http://www.redis.io/.

[20] M. Zhuang and B. Aker, "memslap: Load testing and benchmarking tool for memcached." http://docs.tangent.org/libmemcached/memslap.html.

[21] B. Goglin and N. Furmento, "Enabling high-performance memory migration for multithreaded applications on Linux," in *Proc. of IPDPS*, 2009, pp. 1–9.

[22] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot, "Scale-out NUMA," in *Proc. of ASPLOS*, 2014, pp. 3–18.

[23] S. Li, T. Hoefler, and M. Snir, "NUMA-aware shared-memory collective communication for MPI," in *Proc. of HPDC*, 2013, pp. 85–96.

[24] L. M. Maas, T. Kissinger, D. Habich, and W. Lehner, "Buzzard: a NUMA-aware in-memory indexing system," in *Proc. of ACM SIGMOD ICMD*, 2013, pp. 1285–1286.

[25] T. Kissinger, T. Kiefer, B. Schlegel, D. Habich, D. Molka, and W. Lehner, "Eris: A NUMA-aware in-memory storage engine for analytical workloads," *The VLDB Endowment*, vol. 7, no. 14, 2014.

[26] K. Z. Ibrahim, S. Hofmeyr, and C. Iancu, "The case for partitioning virtual machines on multicore architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 10, pp. 2683–2696, 2014.