HASO: A Hot-page Aware Scheduling Optimization Method in Virtualized NUMA Systems

Butian Huang, Jianhai Chen, Qinming He, Bei Wang, Zhenguang Liu, Yuxia Cheng
College of Computer Science, Zhejiang University
Hangzhou 310027, China
{*butine, chenjh919, hqm, wangbei, zhenguangliu, rainytech*}*@zju.edu.cn*

***Abstract*—In the situation of CPU and memory overcommit, it is inevitable that some NUMA nodes will be overloaded or hotspotted and become hot nodes, leading to the VM application performance degradation in virtualized NUMA (vNUMA) systems. However, the virtual machine monitor (VMM) can not be aware of the NUMA feature and the distribution array of hot memory pages amongst NUMA nodes effectively. Aiming at eliminating the hot nodes and load imbalance in a vNUMA system, this paper proposes a hot-page aware scheduling optimization method (HASO) and implements a HASO scheduling system. At first we monitor the NUMA node load state, find the hot nodes in which we choose the hot VMs that cause the node hotspots. Then, we predict the distribution of future hot memory pages of the hot VM, and evaluate the cost of migrating the hot pages between NUMA nodes. At last, the hot pages of hot VM with minimized migration cost are migrated to idling nodes, so as to eliminate the hot node and improve the VM application performance. In contrast to the default scheduling mechanism of VMM, our HASO scheduling method can not only improve the memory intensive benchmark cg by up to 27.06% and the benchmark stream by up to 15.63%, but also balance the load of NUMA nodes.**

***Keywords*-Virtualization; Hot Page; Memory Migration; Virtual NUMA;**

I. Introduction

With the evolution of multicore and virtualization technologies, the Non-Uniform Memory Access (NUMA) architecture computer system has taken a widespread exploitation in scientific computing clusters, modern data centers, and cloud computing infrastructures [1]. The NUMA architecture consists of multiple NUMA nodes with multiple CPUs and independent large memory blocks. Each node combines one CPU with one or more cores and a specific size of large memory block. The CPU of one node can both do *local memory access* that accessing the memory in this node, and do *remote memory access* that it accesses the memory in another node. Unfortunately, the *remote memory access* will have a worse application performance and quality of service (QoS) than local memory access. Virtualization poses additional challenges to performance optimization in virtualized NUMA (virtual NUMA or vNUMA) systems [2]. Many existing researches have proposed multiple performance optimization methods in vNUMA systems, such as vnuma-mgr [3], NUMA scheduling [4], memory migration [5], [6] methods.

However, virtualization still lacks efficient methods to solve the performance degradation caused by the overloaded NUMA nodes and load imbalance derived from the running of memory intensive applications between VMs in the situation of CPU and memory overcommit [7]. Due to the defects of the VMM resource scheduling, the vCPU and memory of VM probably are allocated onto distinct nodes, leading to many *remote memory access* operations and VM performance degradation. To avoid the *remote memory access*, the VMM needs to remain a virtual NUMA topology in which the CPU accessing memory is in the same node. We call the frequently-accessed memory pages as *"hot" pages* [8], the VM with hot pages as a "hot" VM and an overloaded NUMA node with a lot of hot pages or memory load hotspots as a "hot" node. In order to remove the hot nodes for load balancing and remain virtual NUMA topology, some VMs in the overloaded node or their hot pages are required to be migrated onto other idling nodes. But migrating the VM with very large memory will have a huge cost of migration.

In this paper, aiming at solving the NUMA hot node elimination problem and minimizing the cost of VM memory migration, we propose a hot-page aware scheduling optimization method, termed as HASO. We design and implement a HASO scheduling system. At first, we monitor and check the node load status and discover the overloaded or hot nodes. Then, we perceive the hot pages of VM in the hot node using a 5-bitmap method to identify hot page data, and further use future hot pages with a K-means clustering algorithm. Furthermore, with consideration of memory migration cost, the HASO scheduler adopts novel scheduling algorithms to provide optimal policies to migrate hot pages to low load nodes, reducing the number of hot pages in hot nodes, so as to eliminate the hot nodes and improve the VM application performance. The results of performance evaluation experiments demonstrate the effectiveness of our HASO method.

The key contributions of this paper can be summarized as follows. (1) We have proposed the HASO method to solve the elimination hot NUMA node problem in vNUMA systems. (2) We have provided a HASO scheduling system

to perceive the hot NUMA nodes and VM hot pages identified with a state weight using a 5-bitmap method. (3) We designed some novel algorithms for predicting the future hot pages of hot VMs and eliminating hot node scheduling with minimized VM migration cost. (4) We have presented overall performance evaluation experiments to demonstrate the effectiveness of the HASO method.

The rest of the paper is organized as follows. Section II presents the background and motivation. Section III describes the system design and implementation. Section IV presents the performance evaluation experimental results and Section V provides the conclusions and our future work.

II. Background and Motivation

In this section, we introduce the background of NUMA architecture and discuss its performance bottleneck in vNUMA systems, including the VM memory access and the cost of VM migration between NUMA nodes.

A. The NUMA system

Generally, before NUMA appears, the traditional computer architecture is the Uniform Memory Access (UMA). The UMA system has an off-chip shared memory controller and the memory bandwidth is often a bottleneck as the number of cores per chip increases. Besides, the UMA-based symmetric multi-processor (SMP) architecture is limited in scalability because all memory accesses pass through the same shared memory bus. The NUMA architecture is a new computer architecture and designed to effectively remove the central shared memory controller and significantly increase every core memory bandwidth.

we present a NUMA multicore machine with four NUMA nodes (CPU sockets) in Fig.1. Each CPU socket has eight cores, which share a last-level cache (L3 cache), an integrated memory controller (IMC), and an Intel QuickPath Interconnect (QPI). The cores together with their private L1 and L2 caches are called the ”core” memory subsystem. The shared L3 cache, the IMC, and the QPI together form the ”uncore” memory subsystem. The cache line requests from the cores are queued in the Global Queue (GQ) and serviced by the uncore. The physical memory is divided into four memory nodes. Each node is directly connected to a processor.

B. The Virtual NUMA System

The vNUMA system is a NUMA machine which is deployed with a virtualization environment and hosts multiple VMs to run applications. We focus on introducing the VM memory access and VM migration issues in vNUMA systems.

1) The VM memory access: The VM memory access is a general system operation during the runtime of VM in vNUMA systems. When a VM starts to run, the VM resources including the vCPU and memory are initially

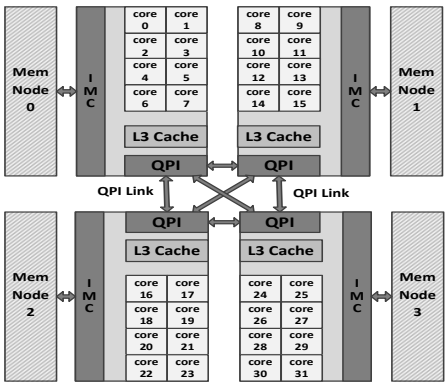


Figure 1. A four-node NUMA machine

allocated onto the NUMA nodes via the VMM. Precise Event-Based Sampling (PEBS) is an advanced sampling feature of the Intel Core-based processors in which the processor is directly recording samples into a designated memory region. perf [9] is a performance analyzing tool in Linux, available from Linux kernel version 2.6.31 and can statistically profile the entire system. It supports hardware performance counters, trace-points, software performance counters, and dynamic probes.

2) The VM memory migration cost: Like VM migration between PMs, the VM memory migration between NUMA nodes will also introduce cost. The cost is generally associated with three factors. (1) *Memory distribution*. In vNUMA systems, the VM memory pages can be distributed in many nodes. Migrating VM memory pages in many nodes will have much more cost than one node. Moreover, the VM memory pages can partly be hot or cold, and partly be shared by two or more VMs. These factors can not be ignored in conclusion of the cost of memory migration. (2) *Node weight*. To conclude the cost of migration, we should choose the other optimal node and migrate the VM memory in current node to the optimal node. Obviously the distance between nodes will be a significant factor which is related to the migration cost. (3) *The load of node*. The load of node is commonly signified by the resource usage, such as CPU usage and memory usage. Besides, the ratio of CPU overcommit is also a significant fact with relation to the node load.

C. Motivation

We consider a vNUMA system with many VMs running memory-intensive applications and a memory overcommit. Some NUMA nodes will be overloaded or in a hot state, namely, the load, for example, memory usage, of nodes is over a threshold constant value such as 90% with a total 100%. The load of all NUMA nodes can be obtained. Generally, a *load balancing degree* is used to measure the overall load state of the NUMA machine. If the load

balancing degree is larger than a certain threshold, we deem that some NUMA nodes are bound to be in a hot state, and then we must make decision of migrating VMs or pages in hot nodes to idling nodes with a low load or cold state.

The node being overloaded or hot will provide a serious impact on application performance in vNUMA system. This motivates us to solve the problem. We call it as a hotspot elimination problem. The core task of our work is to optimize the migration of VM or hot pages between nodes. But whole VM migration with a large amount of memory will be very unrealistic. We attempt to find the VMs with a large number of hot pages and migrate the hot pages to other node with low load so as to eliminate the hot node. According to the time or space locality principle, we can predict the future hot pages used in VM memory migration. In the next section we will specifically propose an overall solution to solve the hot node elimination problem.

III. Design and Implementation

In this section, we present a hot-page aware scheduling optimization (HASO) method to solve the NUMA node hotspot elimination (NHE) problem in vNUMA systems. We focus on eliminating the hotspot scenario in which there are some nodes with overloaded memory usage.

A. HASO scheduling system

We design and implement a HASO scheduling system (HASO-SS) for managing the hotspot elimination scheduling problem. As shown in Figure 2, it includes a schedule controller (SC) module, a HASO monitor (HM) module, a load state analyzer (LSA) module, a hot-page analyzer (HPA) module, a schedule decider (SD) module, a schedule algorithm (SA) module and a schedule executer (SE) module.

The HASO monitor is responsible for monitoring the load and memory page access information. The load information includes the CPU usage, memory usage of NUMA nodes and VMs. It periodically captures the load and memory page access information and stores into a shared system data base or storage. The load state analyzer can read the load data and check the load status including the nodes and VMs, for example, the node load is in overloaded or a hot state and unbalance, or VM is overloaded. The hot-page analyzer (HPA) is responsible for the analysis of memory page access state information and recording the state into a dynamically built memory page bitmap storage. The scheduler controller module is the core of HASO-SS in charge of controlling and managing the system schedule operations. It regularly invokes the LSA module to check the NUMA node system states. If a node is in hot status, it immediately triggers a hot node elimination scheduling decision request and sends the request to the SD module. The SD module calls a special schedule algorithm to make decision to schedule and gives an optimal schedule scheme result. The SD module returns

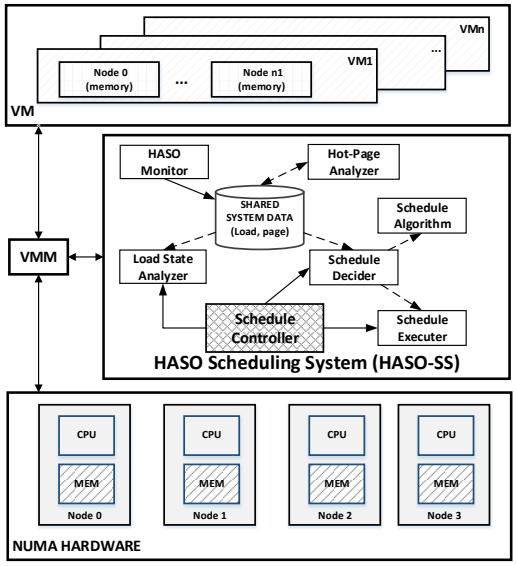


Figure 2. Overview of HASO scheduling system

the specific schedule scheme back to the SC module. In the end, the SC module calls the schedule executer to execute the specific scheduling operations.

The key techniques for solving the hot node elimination problem are detailed in the following sections.

B. Discover hot node and select hot VMs for migration

The HASO scheduling system firstly discovers the hot node and eliminates it by selecting hot VMs in the hot node migrated to other low loaded node. Generally, the load includes two types, namely, CPU usage and memory usage. If we do not consider CPU then it is just only memory.

1) Hot node and hot VM: We use a formula to conclude the load of a node or a VM. Given node or VM i, let L_i be the load of node i, CL_i be the CPU load usage, and ML_i be the *memory load usage*. Then, the formula used to conclude the load of node or VM i is $L_i = 1/(1-CL_i) * 1/(1-ML_i)$. We give a threshold denoted as a hot boundary constant of load usage. The hot node or VM is defined as follows.

Definition 1: ***Hot node or VM****. Let HOT be a threshold. A hot node is defined as a NUMA node with a load usage higher than HOT, such as 80%. A hot VM is also defined as a VM with a load usage higher than the given HOT.*

2) Hot node discovery: Besides, in situation of CPU and memory overcommit, we think that if a vNUMA system are load unbalanced between all nodes, then there must be some hot nodes. We introduce a load balancing degree (LBD) to denote a system overall load balance state and think that if the LBD is too large or larger than a given threshold, then some node load must be overloaded or hot. Specifically, let n be the number of NUMA nodes and $NL = \{NL_i\}$ where

$0 < i < n$, be a set of node load with number n. The LBD is concluded by $LBD = \sum_{i=1}^{n}(L_i - \sum_{j=1}^{n} L_j/n)^2/n$.

3) Select hot VMs for memory migration: When a hot node is found, we attempt to eliminate it by migrating the memory of some VMs in the hot node to other low memory loaded node. At first, we choose some VMs in which the memory locates to migrate. We simply use greedy, heuristic methods to choose the VMs to migrate. We think that some hot VMs in the node cause the node being hot. So we firstly check if there are some hot VMs in the hot node and select all the hot VMs. Otherwise, we calculate the load usage of all VMs in the hot node and sort the VMs by the load usage decreasingly. And then according to the decreasing order we choose the VM one by one to migrate its memory to other node until the hot node is eliminated.

C. Select hot memory blocks

After hot VM selection, due to the huge cost of migration of hot VM memory to other nodes, we then attempt to migrate the memory of the hot VM partly, namely, the hot memory page blocks to other NUMA node.

1) Record the access state of memory pages: Actually, a VM of KVM is a program process which can be monitored by Linux OS process monitoring tool. We can use perf to monitor the process of hot VM and check the memory pages being used or not. At a time point, we call a memory page being used as a hot page, otherwise cold or not hot. In our HASO scheduling system, the HASO monitor is a demon program and collects the sample hot page data periodically, with a fixed time interval such as 30 seconds.

In one time, if the memory access state of a page is being used, then we set a state value as 1 for this page, otherwise 0. For the memory page, we check and get one state value per time interval. At each time interval, we can have a set of hot memory pages identified with a state value. We define a hot page candidate set P to record the obtained pages and set it as $\emptyset$ initially. Let i be an integer, P_i be a set of hot memory pages we check and get at the ith time interval. Then, we union P_i to P and get a new P, namely, $P = P \cup P_i$. Obviously, with the past of several time intervals, the P will become increasingly bigger.

2) Store the access state of hot pages using Five-bitmap method: In order to precisely analyze the overall state of memory access in a fixed hot VM in the hot node, we need to store as many as possible the access state values for the obtained hot pages in the candidate page set. We use bitmap to store the state values. Bitmap method is widely used to solve the store issue of the access state of memory pages [10].

For simplicity, we firstly define a set of memory pages with a continuous memory address as a *memory page block* or memory block. Let N be the number of pages in a memory page block, B_ADD be the base address of the memory page block. A memory page block can be represented by $MB(B\_ADD, N)$. The address of a page $i(i < N)$, can be concluded by $B\_ADD + i$. The hot memory pages of a hot VM in a hot node will have many distributed memory blocks.

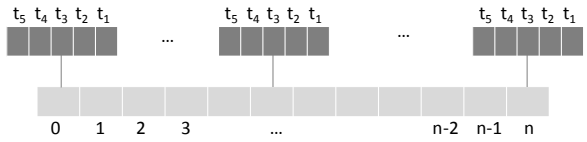


Figure 3. Bitmap with five bits

Specially, at one time interval, a given memory block with n pages requires n bits to store the memory page access state. The state value includes 1 and 0. Each page needs to save one state value. We use a bitmap with n bits to save the page access state mapping to a hot memory page block with n pages. Actually, one bitmap only can save one state value of a set of pages at one time interval. If we need to save multiple state values corresponding to multiple time intervals, it will require more bitmaps and more memory storage space.

In our work, the bitmap is dynamically built for each memory block in the monitoring. To save the storage space, we use a *5-bitmap* method, in which we choose the nearest five state values from the last five time intervals. As shown in Figure 3, we use five bitmaps to save the memory access state information corresponding to 5 time intervals for each hot page. For each page, the five state values can be viewed together as a unit. We combine the five state bits as a state string like "*bbbbb*", where $b \in \{0, 1\}$. For example, one memory page has a state string "01001" at a time interval. It means the nearest continuous five time intervals of the memory access states are $0, 1, 0, 0$, and 1, respectively. From the right to the left, we identify the sequence number as $1, 2, 3, 4$, and 5, respectively. The ith bit value denotes the ith state value.

In addition, considering the time history factor, we deem that the bigger bit index or serial number in a combined state five bits string indicates the nearer state we get at a nearer time interval. We qualitatively set a weight for each bit of the bitmap with five bits. Let w_i be the ith bit weight, we empirically set the weight value of the ith bit to i, namely, $w_i = i(i = 1, 2, \ldots, 5)$. The bigger i denotes the nearer time interval obtained the access state of a page. We set a total weight to the page to denote the hot degree of memory page access. The total weight is computed by the formula, $W = \sum_{i=1}^{5} b_i * w_i$, where $b_i \in \{0, 1\}$ is the ith state bit value. For example, if a page has a five state bits string "01110", then the weight of the page is 9, which is computed by the expression, $0 * 5 + 1 * 4 + 1 * 3 + 1 * 2 + 0 * 1$.

3) Building future hot memory blocks as migration candidates: We have obtained a candidate set of hot pages blocks. In a hot page block, each page has both a page address and a hot state weight. To eliminate the hot memory node, we determine which memory pages are required to be firstly migrated in the hot VM of a hot node. Considering the memory space locality principle, if one page is hot, then a set of pages close to this page are hot or will be hot in the near future. We will migrate both the hot pages and a fixed number of their adjacent pages. The memory addresses of these hot pages and their adjacent pages are generally continuous and form a memory block.

Each page x has two attributes, namely the page index e_x and the five bits state weight w_x. Let X be a hot page set, $\forall x \in X$, x has the form $< e_x, w_x >$. We perform K-means clustering techniques [11] on X to divide the page set into K clusters, where K is an integer parameter. This is to put the pages that are near in location and have similar weights into one cluster, and put the pages with no similarity in weights or page index into different clusters. We sort the K clusters in a decreasing order according to the average state weight and migrate the cluster hot pages one by one. The higher average weight a cluster has, the more pages in the cluster are firstly migrated. Considering the space locality principle, we give a distance offset $\mathcal{O}$, a hot page with address $ADDR$, and we extend the page address to a range between the address range $(ADDR-\mathcal{O}, ADDR+\mathcal{O})$, which constructs a page block. All hot pages are extended to a migration set of hot page blocks. If some hot page blocks have overlay addresses, then we union them to one hot page block.

D. Hot node elimination scheduling algorithm

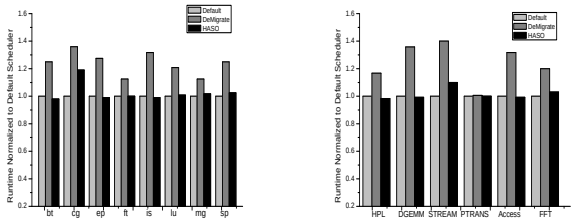
We give a scheduling algorithm to execute the VM migration and eliminate the hot node. The algorithm selects a destination node and calculate the cost of migration VM hot page blocks to the node. The node with the minimum migration cost is selected as an optimal destination node.

Let F_{ij} be the memory pages of VM j on the node i, k be the migration destination node of the VM j, and W_{ik} be the weight of the node k, then the migration cost of the VM j can be calculated as $C_{jk} = \sum_{i=1}^{n} F_{ij} * W_{ik} * NL_i(i \neq k)$.

Through this algorithm, the strategies migrate the VMs to the relative idle NUMA nodes to eliminate the hot node and balance the NUMA node load. After obtaining future hot memory pages, we need to find the NUMA node to migrate the VM's memory pages, and guarantee that the cost of migration is minimum.

IV. Performance Evaluation

In this section, we present an experimental evaluation of the proposed HASO method using the real-world parallel workloads.



(a) NPB benchmarks (b) HPCC benchmarks

Figure 4. VM benchmarks compared with default scheduler

A. Experimental environment

We use an R910 server with four 1.87 GHz Intel (R) Xeon (R) CPU E7520 processors based on the Nehalem-EX architecture. Each E7520 processor has four cores sharing a 18MB L3 cache. The R910 server has a total of 16 physical cores and 64 GB memory, with each NUMA node having 4 physical cores and 16 GB memory. The server is installed with KVM, the Qemu's version is 1.2.2, and the linux kernel's version is 3.10.7. We compare the performance of HASO with KVM's default CFS (Completely Fair Scheduler) scheduler. We select the NAS Parallel Benchmark (NPB 3.3) [12] and the HPC Challenge benchmark [13] to measure VM performance.

B. VM migration performance

In this experiment, we evaluate the performance of VM migration between NUMA nodes. We place a VM in the NUMA node 0, which is configured with 8 vCPUs and 16 GB memory and run the NPB benchmark in this VM. In the process of running the benchmark in VM, at first, we use the default VM migrate strategy, which migrates VM as a whole. And then, we use the HASO method strategy which migrates the VM hot pages to other NUMA nodes. Our experimental results show that the sample interval cannot be too small and the suitable value is set to $30s$.

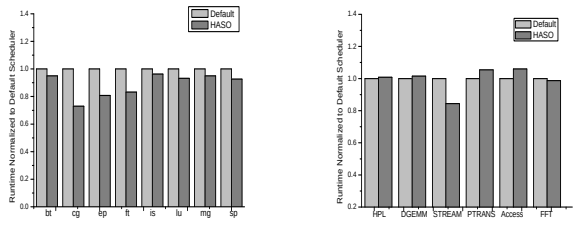
Figure 4 (a) shows the runtime of NPB benchmarks under three different strategies: the default CFS scheduler (Default), the default VM migrate strategy (DeMigrate), and the HASO strategy. Figure 4 (b) shows the runtime of HPCC benchmarks under three different strategies: the default CFS scheduler (Default), the default VM migrate strategy (DeMigrate), and the HASO strategy. We found that the HASO strategy cannot affect the migrated VM's performance, and can improve the overall performance of NUMA machine.

C. VM application performance

We create 8 VMs on R910 server, every VM is configured with 4 vCPUs and 4GB memory and bound to a single NUMA node, as shown in Table 2. Inside each VM, we run one 8-threaded NPB-OMP benchmark.

Table I
VM CONFIGURATION

VM ID	Memory	vCPU	Binding node
0	4GB	4	0
1	4GB	4	0
2	4GB	4	1
3	4GB	4	1
4	4GB	4	2
5	4GB	4	2
6	4GB	4	3
7	4GB	4	3



(a) NPB benchmarks (b) HPCC benchmarks

Figure 5. VM performance improvement compared with default scheduler

For instance, an 8-threaded bt benchmark runs in VM1, an 8-threaded cg benchmark runs in VM2, and an 8-threaded sp benchmark runs in VM8. Meanwhile, as shown in Fig.5, we have designed a memory-intensive benchmark, which is allocated 4G memory and tries to assign the random value to the applied memory all the time. Then this benchmark will take up most of the VM's memory space, and make the node's memory usage growth significantly.

Figure 5 (a) shows the runtime of NPB benchmarks under the default CFS scheduler (Default) and the HASO scheduler (HASO). We can see that HASO strategy is more effective for memory-intensive benchmarks. The VM hot pages is the main reason of cg benchmark's performance degradation. HASO strategy improves the performance of cg significantly by 27.06%, ep by 19.2%, ft by 16.8%, while performance improvement of other benchmarks is not obvious.

Figure 5 (b) shows the runtime of HPCC benchmarks under the default CFS scheduler (Default) and the HASO scheduler (HASO). We can also see that HASO strategy is more effective for memory-intensive benchmarks. Stream benchmark is memory-intensive. HASO strategy improves the performance of stream by 15.63%, and cannot improve the performance of other benchmarks.

V. CONCLUSION AND FUTURE WORK

In this paper, we have proposed the HASO method to eliminate the hot node to improve VM application performance in vNUMA systems. We have provided 5-bitmap method to record the VM memory access states, a K-means technique to cluster hot pages and predict future hot page blocks used to migrate. The HASO scheduler can not only improve the migrated VM performance but also improve the overall system performance.

We focus on migrating the hot pages instead migrating VMs to eliminate the hot node and improve the VM performance in the condition of CPU and memory overcommit, our methods improve the VM performance greatly in this paper. Since our strategies are not intended for CPU-intensive benchmarks at present. In future work, our HASO method can be continued to be optimized by tuning some parameters. Another direction is to improve performance of CPU-intensive applications based on hot pages in virtualization. These issues have gained little attention at present and deserve our further study.

ACKNOWLEDGMENT

We would like to thank the InCAS lab for providing us with excellent experimental environment.

REFERENCES

[1] L. Tang, J. Mars, X. Zhang, R. Hagmann, R. Hundt, and E. Tune, "Optimizing googles warehouse scale computers: The numa experience," in *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA2013)*. IEEE, 2013, pp. 188–197.

[2] M. Chapman and G. Heiser, "vnuma: A virtual shared-memory multiprocessor." in *Proceedings of 2009 USENIX Annual Technical Conference (ATC 2009)*, 2009.

[3] D. S. Rao and K. Schwan., "vnuma-mgr: Managing vm memory on numa platforms," in *Proceedings of the 2010 International Conference on High Performance Computing (HiPC 2010)*. IEEE, 2010, pp. 1–10.

[4] Y. Cheng, W. Chen, X. Chen, B. Xu, and S. Zhang, "A user-level numa-aware scheduler for optimizing virtual machine performance," *Advanced Parallel Processing Technologies*, pp. 32–46, 2013.

[5] V. Mishra and D. Mehta, "Performance enhancement of numa multiprocessor systems with on-demand memory migration," in *Proceedings of IEEE 3rd International Advance Computing Conference (IACC 2013)*. IEEE, 2013, pp. 40–43.

[6] Q. Ali, V. Kiriansky, J. Simons, and P. Zaroo, "Performance evaluation of hpc benchmarks on vmwares esxi server," *Euro-Par 2011: Parallel Processing Workshops*, pp. 213–222, 2012.

[7] W. Jian, D. Wei, J. Cong-Feng, and X. Xiang-Hua, "Over-committing memory by initiative share from kvm guests," *International Journal of Grid and Distributed Computing*, vol. 7, no. 4, pp. 65–80, 2014.

[8] J. Ahn, C. Kim, J. Han, Y.-R. Choi, and J. Huh, "Dynamic virtual machine scheduling in clouds for architectural shared resources," in *Proceedings of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 2012)*, 2012.

[9] "perf." [Online]. Available: https://perf.wiki.kernel.org

[10] Y. Luo, B. Zhang, X. Wang, Z. Wang, Y. Sun, and H. Chen, "Live and incremental whole-system migration of virtual machines using block-bitmap," in *Proceedings of 2008 IEEE International Conference on Cluster Computing (Cluster 2008)*. IEEE, 2008, pp. 99–106.

[11] J. A. Hartigan and M. A. Wong, "A k-means clustering algorithm," *Applied statistics*, pp. 100–108, 1979.

[12] 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., H. D. Simon., V. Venkatakrishnan., and S. K. Weeratunga., "The nas parallel benchmarkslsummary and preliminary results," in *Proceedings of the 1991 ACM/IEEE conference on Supercomputing (ICS 1991)*. ACM, 1991, pp. 158–165.

[13] "Hpcc." [Online]. Available: http://icl.cs.utk.edu/hpcc/index.html