vDARM: Dynamic Adaptive Resource Management for Virtualized Multiprocessor Systems

Jianmin Qian, Jian Li, Ruhui Ma and Haibing Guan
Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University, China
Email: {hacker_qian, li-jian, ruhuima, hbguan}@sjtu.edu.cn

***Abstract*—Modern data center servers have been enhancing their computing capacity by increasing processor counts. Meanwhile, these servers are highly virtualized to achieve efficient resource utilization and energy savings. However, due to the shifting of server architecture to non-uniform memory access (NUMA), current hypervisor-level or OS-level resource management methods continue to be challenged in their ability to meet the performance requirement of various user applications. In this work, we first build a performance slowdown model to accurate identify the current system overheads. Based on the model, we finally design a dynamic adaptive virtual resource management method (vDARM) to eliminate the runtime NUMA overheads by re-configuring virtual-to-physical resource mappings. Experiment results show that, compared with state-of-art approaches, vDARM can bring up an average performance improvement of 42.3% on an 8-node NUMA machines. Meanwhile, vDARM only incurs extra CPU utilization no more than 4%.**

I. Introduction

Multiprocessor servers have been widely adopted in high performance computing (HPC). Additionally, these servers are virtualized so that multiple independent users can run their own applications on a same physical server. Since such servers may have high number of end-users and the hosted applications may have different resource demands. For example, the in-memory graph analytics applications have large irregular memory footprints [1] and the scientific computation applications may generate massive data caching [2]. Consequently, it is difficult to design a resource management policy that satisfies every application's performance requirements [3][4].

Meanwhile, the NUMA architecture has been the dominant multiprocessor architecture due to its scalable bandwidth performance [5]. However, the NUMA architecture also introduces additional overheads, such as remote access latency and memory traffic congestion [6]. Prior studies have shown that these overheads can significantly affect the application performance. For instance, it was reported that bandwidth performance degrades at least 57% when the memory access is remote [7], and the memory traffic congestion may increase memory access latency up to 5 times [8]. Moreover, due to the semantic gap introduced by the virtualization layer, applications running inside a virtual machine (VM) know little about the underlying NUMA characteristics [9]. As a result, the application's performance cannot be guaranteed without NUMA-aware management of the VM resource.

Although previous approaches have been proposed to address the NUMA overheads in virtualized systems, there are still two major limitations. **First**, existing hypervisor-level approaches attempt to optimize the NUMA overheads through initialized resource placement. A common approach, which has been adopted by Xen [4] and VMware ESXi [10], is to allocate the VM's virtual CPUs (vCPUs) and memory on the same node to maximize data locality or interleave the VM's memory to each NUMA nodes to avoid traffic congestion. These approaches are efficient for the applications with regular access behaviors. However, current user applications may generate many irregular data interactions [3]. Therefore, only the initialized resource placement cannot always be efficient. **Second**, current user-level resource management policies also relies on dynamic resource scheduling to address the NUMA overheads. However, these approaches lack an adaptive performance slowdown model to make an accurate scheduling decision. For example, the *Carrefour* policy uses hardware performance events (HPEs) to model application's memory access performance [8]. However, not all the processor platforms (e.g.,Intel and AMD) have the HPEs that present the same function [5]. Thus, such approaches have strong platform dependency and poor portability.

In this paper, we first build a performance slowdown model to accurately identify the NUMA overheads by using a platform-independent method. Based on the model, we then propose a dynamic adaptive virtual resource management policy (vDARM) to improve the efficiency of user applications on NUMA systems. vDARM eliminate the current NUMA overheads by employing two corresponding scheduling algorithms: the locality-aware vCPU remapping to improve data locality; and the congestion-aware memory migration to balance memory traffic. Experimental results show that vDARM has an average performance improvement of 42.3% on 8-node NUMA platform. Moreover, vDARM only increases CPU utilization no more than 4%. We summarize the major contributions as follows:

1) We study the limitations of current resource management policies on NUMA multiprocessor system and propose a performance slowdown model that accurately identifies NUMA overheads on different platforms.
2) Based on our performance slowdown model, we propose vDARM, a dynamic adaptive virtual resource management policy to improve the user applications efficiency by runtime vCPU scheduling and memory migration.
3) We implement and evaluate vDARM prototype on a NUMA multiprocessor platform, demonstrating significant performance improvements.

978-3-9819263-2-3/DATE19/©2019 EDAA

II. Design

Motivated by the limitations of current approaches, we find the necessity to address the NUMA overheads for the applications with various resource access behaviors. In this work, we first build a performance slowdown model to accurately identify the current NUMA overheads. Based on the model, we than design two dynamic resource reconfiguration mechanisms that in cooperation with the default system resource management policies to eliminate the corresponding overheads.

A. *Bandwidth-aware Performance Slowdown Model*

To efficiently identify the current NUMA overheads, our performance slowdown model should address two key problems: whether the current VM is sensitive to NUMA overheads and which NUMA overhead the VM suffers from. Meanwhile, the model could be easily applied to the mainstream multiprocessor platforms. To this end, we first calculate the runtime bandwidth demands of the VM that runs different applications. Based on the VM bandwidth demands, we then design a platform-independent overheads identification method.

*1) **Calculating runtime bandwidth demand**:* There are two methods to calculate memory bandwidth demands in runtime systems. The conventional method uses last level cache (LLC) misses to calculate memory bandwidth [11]. Because LLC misses always access the main memory. Thus, as long as we know the number of LLC misses during a periodic time and the size of cache lines of the system, the memory bandwidth can be determined ($\frac{LLC\_misses*Size}{time}$). However, this method only measures the read bandwidth of the VM due to the limited performance events. The second method calculates the memory bandwidth demand by recording the number of read and write requests to the memory controller over a period time. As shown in Equation (1), $NumReads$ and $NumWrites$ are the number of memory read and write requests, respectively. The block size ($BlockSize$) of each read and write operation in the current system is 64 Bytes [5]. In this work, we use the second method. When compared with the first method, the second method calculates both memory read and write bandwidths. Thus, it is more accurate for the identification of overheads. Moreover, the OS provide the software events to acquire the number of memory reads and writes [5]. Thus, our model can be deployed on both Intel and AMD processors.

$$BW_{demand} = \frac{(NumReads + NumWrites) * BlockSize}{ProfilingInterval} \quad (1)$$

*2) **Identifying VM sensitivity to NUMA overheads**:* To determine whether the VM performance is sensitive to NUMA overhead and which NUMA overhead the VM is sensitive to, we divide the VM types into three classes: VMs that are not sensitive to NUMA overhead (**class 0**), VMs that are sensitive to remote access latency (**class 1**) and VMs that are sensitive to traffic congestion (**class 2**). To make the accurate classification, we first define a sensitive factor (SF) that is based on the VM bandwidth demands. As shown in the Equation (2), the $BW_{available}$ is the maximum available bandwidth of the memory controller in the system. Theoretically, when the SF value approximates 1, this means the VM bandwidth demand is very small and its performance may not be affected by the NUMA overheads. If the SF is less than 0, this means the VM bandwidth demand is greater than the current maximum bandwidth and its performance may suffer from memory traffic congestion. However, the SF value alone cannot well distinguish the NUMA insensitive VMs and NUMA sensitive VMs. Fortunately, the CPI value of NUMA insensitive applications are always lower than 1. Thus, we use CPI and SF to jointly determine the sensitivity. To verify the effectiveness of our identification method, we run each tested application in a VM. Figure 1 shows the results.

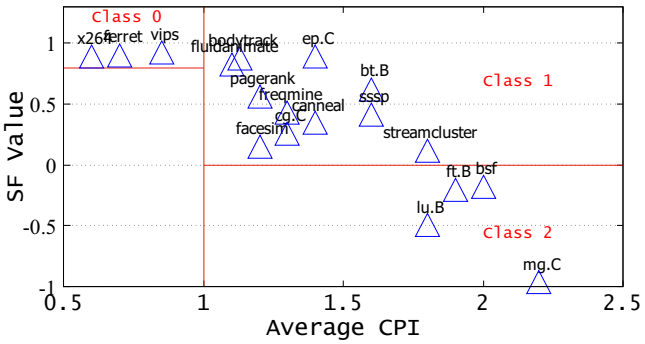

Fig. 1: SF values of the tested applications.

$$SF = \frac{BW_{available} - BW_{demand}}{BW_{available}} \quad (2)$$

We observe that classification results are in accordance with the applications performance in Figure **??**. Firstly, the *X264*, *ferret* and *vips* have CPI values less than 1 and SF values approximate to 1 (range form 0.85 to 1). These three applications are not affected by the NUMA overheads, thus they belong to class 0. For the applications whose CPI value is greater than 1 and SF value is greater than 0, such as *steamcluster* and *facesim*. Their performance are affected by the remote access latency, thus they belong to class 1. For the applications with a CPI value greater than 1 and SF value lower than 0, such as *lu.B* and *mg.C*, their performance are affected by memory traffic congestion and belong to class 2.

B. *Dynamic Adaptive Virtual Resource Management*

After the VM identification with the NUMA overhead sensitivity, the final step is to manage the VMs' resources to address the overheads. We rely on two techniques to dynamically reconfigure the VM resources: (1) vCPU migration: changing the node on which the VM's vCPU is running. (2) memory page migration: migrating memory pages of a VM from one node to another. In the following, we introduce how our management algorithms are aligned with these two techniques.

*1) **Locality-aware vCPU and page migration**:* Whenever the VM is sensitive to the remote access latency, the locality-aware vCPU and page migration algorithm will be enabled to eliminate remote memory access. Algorithm 1 presents the

Algorithm 1: Locality-aware vCPU and page migration

```
1  Input: N ←num_numa_configured_nodes();
2  M ← num_vm_vcpus();
3  D_N ← num_vm_pages_per_nodes();
4  Output: Scheduling Decisions;
   /* First Phase: gather VM pages to one node     */
5  if memory allocation policy ≠ preferred then
6      p = max_vm_pages_node();
7      for i := 0 to N − 1 do
8          if node_free_pages_p ≥ D_i && i ≠ p then
9              numa_migrate_pages(i,p,D_i);
10             node_free_pages_p -= D_i;
11         end
12     end
13 end
   /* Second Phase: migrate VM vCPUs to same node */
14 for j := 0 to M − 1 do
15     if curr_vcpu_node ≠ p then
16         migrate_vm_vcpu(curr_vcpu_node, p);
17     end
18 end
```

Algorithm 2: Congestion-aware page migration

```
1  Input: N ← num_numa_configured_nodes();
2  P ← num_vm_total_pages();
3  D_N ← num_vm_pages_per_nodes();
4  Onput: Scheduling Decisions;
5  if memory allocation policy = preferred then
6      p_node = get_vm_memory_node();
7      for i := 0 to N − 1 do
8          if node_free_pages_i ≥ vm_pages_per_node &&
             i ≠ p_node then
9              numa_migrate_pages(p_node,i,P/N);
10         end
11     end
12 end
13 else if memory allocation policy = first_touch then
14     for j := 0 to N − 1 do
15         d_j = D_j − P/N;
16     end
       // iterate until all values in d_N become 0
17     for j := 0 to N − 1 do
18         while d_j > 0 do
19             for k := 0 to N − 1 do
20                 if d_k < 0 && |d_j| >= |d_k| then
21                     numa_migrate_pages(j,k,d_k);
22                     d_j− = |d_k|; d_k = 0;
23                 end
24                 else if d_k < 0 && |d_j| < |d_k| then
25                     numa_migrate_pages(j,k,d_j);
26                     d_k+ = d_j; d_j = 0; break;
27                 end
28             end
29         end
30     end
31 end
```

detailed pseudo-code with two phases. Initially, the scheduler obtains the number of system configured NUMA nodes (N). For each VM, the scheduler obtains the number of its active vCPUs (M) and the number of its memory pages that distributed on each node (D_N). In the first phase, if the memory allocation policy is not *perferred*, the scheduler gathers all its memory pages to the node p that have the maximal distributed pages. Using this, the minimum number of memory pages will be migrated to reduce the migration cost. In the second phase, all the vCPUs of the VM are migrated to the same node that the VM's memory pages reside in. In this way, the memory access can be guaranteed locally.

*2) **Congestion-aware page migration**:* For the VMs that suffer from the traffic congestion, the scheduler employs the traffic congestion-aware page migration to rebalance the memory access traffic. As shown in the algorithm 2, the scheduler first obtains the number of system configured NUMA nodes N and the total number of memory pages P of the VM. Then, it calculates the average number of pages P/N that should be distributed on each node for the purpose of balanced memory access. After that, the scheduler checks the current memory allocation policy of the VM. If the memory allocation policy is *preferred*, the scheduler will evenly migrate the VM pages to each NUMA nodes. Specifically, the scheduler first locates the node where the VM's memory pages reside in ($page\_node$). Then, for each node, if the free memory space of this node is larger than the required memory space, the scheduler will migrate P/N pages from the $page\_node$ to this node.

If the memory allocation policy is *first_touch*, the VM memory pages can be unevenly distributed on all NUMA nodes, thus the migration solution is more complicated. To evenly distribute VM memory pages on all NUMA nodes as well as minimize the amount of migrated pages, we design a novel migration method. First, the scheduler calculates the difference values (d_N) between the VM pages that are actually distributed on each node D_N and theoretical average values (P/N) (lines 14 to 16). Then, if the value in the d_N is larger than 0, this means the node has extra pages that should be migrate to the node where its d_N value is lower than 0. The process iterates until all values in the d_N vector become 0. Finally, the VM pages can be evenly distributed on all nodes.

III. Evaluation

In this section, we first evaluate the performance of vDARM with various applications on the NUMA processor platform, and then we study the runtime overheads of our vDARM.

A. Experiment Platforms and Methodology

All experiments are conducted on a machine with 8 NUMA nodes, each node equipped with AMD Opteron 6376 processors. Each processor is with 16 cores run at 2.3GHz with a 32 GB DDR3 memory. We compare our vDARM with two NUMA-aware resource management policies: the ***vProbe*** [9] is a NUMA-aware vCPU scheduler to improve the performance of memory-intensive applications. The ***carrfour*** [4] approache dynamically migrates pages in order to increase the memory access locality, while avoiding contention on memory controllers. Note that *carrfour* only can be implemented on the AMD platform.

B. Improvement on Application Performance

Figure 2 shows the performance improvement relative to DFT policy on the AMD platforms. We can observe that vDARM achieves best performance for most applications. While for some applications, our vDARM performs slightly

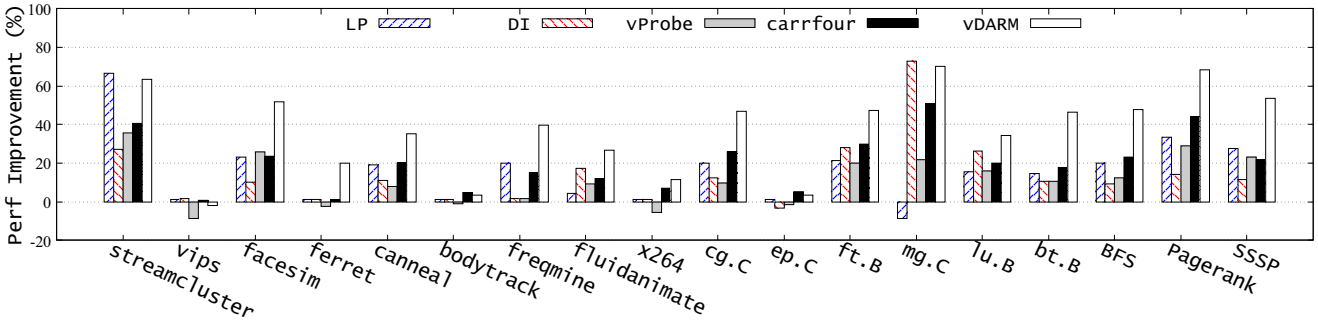

Fig. 2: Performance improvement (relative to DFT policy) on the AMD platform.

lower than the system default policies. For example, our vDARM performs closed to LP policy with streamcluster and closed to DI policy with mg.C. This because these applications have regular memory access and their performance is affected by a single NUMA overhead. Thus, our vDARM performs closely to system policies. For the applications have irregular memory accesses (e.g., canneal,freqmine and cg.C), their performance may simultaneously affected by the two NUMA overheads. Our vDARM always achieves best performance because it can optimize all the NUMA overheads at runtime. On average, vDARM improves 42.3% performance of all the tested applications. For the *vProbe* approache, since it optimizes the application performance through vCPU scheduling, it cannnot address the memory congestion when the initialized VMs memory is allocated on one NUMA node. Thus it is inefficient for the applications that suffer from memory traffic congestion. We observe that *carrfour* also performs slightly lower than our vDRAM for most applications. This is because that *carrfour* only use page migration and page replication to optimize memory traffic congestion, while do not optimize remote memory access at the same time.

C. Overheads

The overhead of vDARM is caused by collecting and storing the VM memory access behaviors and migrating memory and vCPUs scheduling. We tested the average increased CPU usage with all applications when running vDARM. Due to the limited space, we cannot put the detailed figure here. Compared with the solution that without vDARM, the maximum increased CPU usage is about 8.7% when running mg.C and the minimum value is 0.3% when running freqmine. Note that vARM optimizes memory intensive workloads through vCPU scheduling, while optimizing resource locality. The average increased CPU usage of all the applications are 3.2%.

ACKNOWLEDGMENT

This work is supported by the National Key Research & Development Program of China 2016YFB1000502

IV. CONCLUSION

This work studied the efficiency of resource management on virtualized NUMA multiprocessor systems. We first built a platform-independent performance slowdown model to accurately identified current NUMA overheads. Based on the model, we then presented vDARM, a dynamic adaptive resource management policy to optimized the NUMA overheads. Experiments with different applications from the AMD NUMA processors demonstrated an average performance improvement by 42.3%. The future work will focus on the vDARM runtime overheads optimization.

REFERENCES

[1] J. Malicevic, B. Lepers, and W. Zwaenepoel, "Everything you always wanted to know about multicore graph processing but were afraid to ask," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, Santa Clara, CA, 2017, pp. 631–643.

[2] I. Psaroudakis, T. Scheuer, N. May, A. Sellami, and A. Ailamaki, "Adaptive numa-aware data placement and task scheduling for analytical workloads in main-memory column-stores," *Proc. VLDB Endow.*, vol. 10, no. 2, pp. 37–48, Oct. 2016.

[3] B. Teabe, A. Tchana, and D. Hagimont, "Application-specific quantum for multi-core platform scheduler," in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16. New York, NY, USA: ACM, 2016, pp. 3:1–3:14.

[4] G. Voron, G. Thomas, V. Quéma, and P. Sens, "An interface to implement numa policies in the xen hypervisor," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17, 2017, pp. 453–467.

[5] X. Liu and J. Mellor-Crummey, "A tool to analyze the performance of multithreaded programs on numa architectures," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '14. NY, USA: ACM, 2014, pp. 259–272.

[6] J. Funston, M. Lorrillere, A. Fedorova, B. Lepers, D. Vengerov, and V. Quema, "Placement of virtual containers on NUMA systems: A practical and comprehensive model," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, Boston, MA, 2018, pp. 281–294.

[7] K. Zhang, R. Chen, and H. Chen, "Numa-aware graph-structured analytics," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015, 2015, pp. 183–193.

[8] 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," *SIGARCH Comput. Archit. News*, vol. 41, no. 1, pp. 381–394, Mar. 2013.

[9] S. Wu, H. Sun, L. Zhou, Q. Gan, and H. Jin, "vprobe: Scheduling virtual machines on numa systems," in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2016, pp. 70–79.

[10] A. Banerjee, R. Mehta, and Z. Shen, "Numa aware i/o in virtualized systems," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, Aug 2015, pp. 10–17.

[11] P. Tembey, A. Gavrilovska, and K. Schwan, "Merlin: Application- and platform-aware resource allocation in consolidated server systems," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC '14. New York, NY, USA: ACM, 2014, pp. 14:1–14:14.