Towards the contention aware scheduling in HPC cluster environment

Sergey Blagodurov
Simon Fraser University
E-mail: sergey_blagodurov@sfu.ca

Alexandra Fedorova
Simon Fraser University
E-mail: alexandra_fedorova@sfu.ca

Abstract.
Contention for shared resources in High-Performance Computing (HPC) clusters occurs when jobs are concurrently executing on the same multicore node (there is a contention for shared caches, memory buses, memory controllers and memory domains). The shared resource contention incurs severe degradation to workload performance and stability and hence must be addressed. The state-of-the-art HPC clusters, however, are not contention-aware. The goal of this work is the design, implementation and evaluation of a virtualized HPC cluster framework that is contention aware.

1. Introduction

High Performance Computing applications pose great demands on the clusters that host them. Highly parallel in nature, they are able to fully utilize computational cores allocated to them on the cluster multicore nodes. Although cluster schedulers typically put each process (rank) on its own CPU core, the processes collocated on the same node are not isolated from each other. On a multicore server, cores share parts of the memory hierarchy, such as caches, system request queues, memory controllers and memory domains [9, 10, 17, 7]. The job processes compete for those shared memory resources and suffer performance penalty as a result.

In this paper we present our solution to the problem of shared resource contention in HPC. Our contention-aware HPC framework enables cluster administrators to detect and mitigate contention effects on cluster compute nodes. We address the problem both on node level via scheduling on each cluster node and on cluster level via live migration of the MPI cluster workload from the contended servers. Mitigating shared resource contention results in faster runtimes for the cluster workload.

This paper provides the following contributions:

1) We summarize our experiments performed on a number of industry level multicore servers and identify those hardware that is prone to the shared resource contention from the cluster workload.

2) Not all of the cluster workloads are contention-sensitive. Our framework allows to detect those that suffer from the shared resource contention on-the-fly with a robust, non-intrusive profiling method.

3) Our proposed framework addresses shared resource contention in two ways:

a. Our contention-aware scheduling algorithm runs on every compute node. It periodically rebinds job processes to cores and migrates process memory between server NUMA domains to reduce shared resource contention on the node. This is done automatically on-the-fly. No involvement from the cluster administrators is necessary.

b. Higher performance benefits are possible if the load can be spread away from the contended node. Our framework provides cluster administrators with such opportunity. The framework reports those nodes that experience high shared resource contention and allows live migration of process ranks across the compute nodes in the cluster. This is done through a light-weight OS-level virtualization installed in the framework. We provide further details below.

The rest of this paper is organized as follows: Section 2 defines the problem scope and answers in what conditions shared resource contention is the most severe. Section 3 introduces our cluster framework and provides summary of contributions made within the project. Section 4 presents experimental results. Section 5 concludes the paper with the discussion of possible ways the developed framework can be used and the description of our future steps.

2. Problem Scope

Shared resource contention appears when the processes compete for the access to the memory resources of a multicore server, such as last-level caches (LLC), system request queues, memory controllers and memory domains. When these resources are a bottleneck, the contention results in performance degradation for the server workload. To identify the hardware that is prone to contention, we have conducted extensive experimentation on several industry level multicore servers manufactured by Dell, HP and IBM with the processors from both AMD and Intel. All servers were running Linux 2.6.32. We used applications from SPEC MPI2007 as our workload. MPI2007 is a benchmark suite for evaluating MPI-parallel, floating point, compute intensive performance across a wide range of cluster and SMP hardware. These benchmarks pose high requirements for the memory capacity of the compute nodes: typically 1 GB per rank for the medium benchmark subset and 2 GB per rank for the large subset.

We summarize our findings as follows:

1) The servers built with Intel processors based on Core microarchitecture (Xeon X5365, Xeon E5320) and all the AMD Opterons we tested (2350 Barcelona, 8356 Barcelona, 2435 Istanbul, 8435 Istanbul) are prone to the shared resource contention. In our previous work, we have identified Front Side Bus and memory controllers to be the main bottlenecks, although all levels of the memory hierarchy are contributing to the contention overhead [10, 9, 17, 7]. Due to space limitations, we only present experimental results for Opteron servers in Section 4.

Servers built with Intel Nehalem microarchitecture (Xeon X5570, Xeon X5650 were tested) do not exhibit a significant shared resource contention. Memory hierarchy of these machines tends to perform better under memory intensive CPU-bound load. There is, however, one exception. We detected high performance degradation when HyperThreading feature is enabled. Although previous studies have reported that some HPC workloads can benefit from HyperThreading [1], we do not believe that it is widely used in HPC these days. Hence, we omit HyperThreading results from this paper (the interested reader can access them at [4]).

2) Performance degradation that applications experience from the shared resource contention is not uniform. The important question is then what metric to use in order to characterize the processes as sensitive to resource contention within the memory hierarchy of a compute node?

Consider a workload of memory-intensive applications, i.e., applications that are characterized by a high rate of requests to main memory. Following the terminology adopted in an earlier study [16] we will refer to these applications as *devils*. Applications with a low rate of memory requests are referred to as *turtles*. Our methodology from the earlier work [9, 10, 17] allowed us to identify the *LLC miss rate*, which is defined to include all requests issued by LLC to main memory including pre-fetching, as one of the most accurate predictors of the degree to which applications will suffer when co-scheduled in the memory hierarchy of a multicore node. We consider the process a devil if it has more than 30

LLC misses per 10,000 retired instructions. Otherwise, the process is a turtle. We further divide devils into two subcategories: *regular devils* and *semi-devils*. Regular devils have a miss rate that exceeds 45 misses per 10,000 instructions. Semi-devils have between 30 and 45 misses per 10,000 instructions. We explain why such classification is necessary in Section 4. We obtain LLC missrate online using hardware performance counters available on all modern processor models. Profiling with performance counters is robust and does not impose slowdown to the workload. We have also determined that most of the programs from SPEC MPI 2007 are devils and prone to shared resource contention (see Section 4 for details).

3. Contention-aware HPC framework

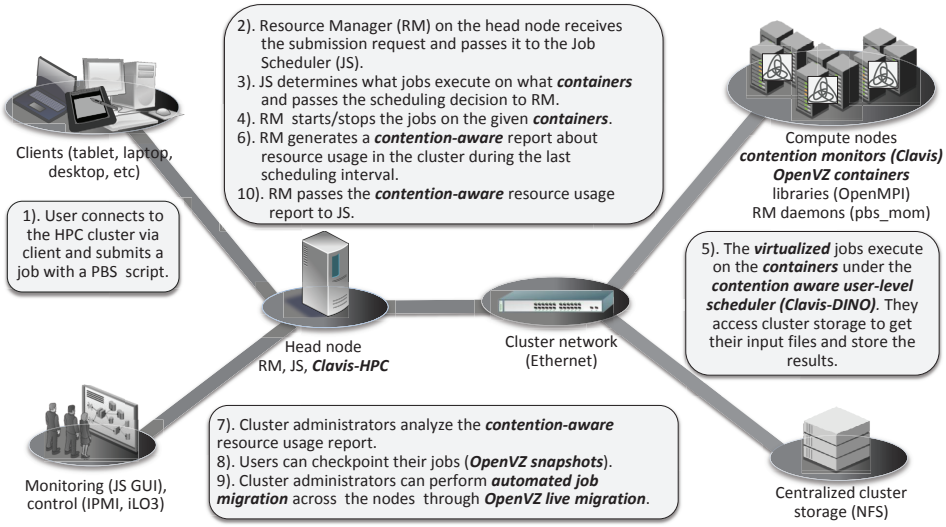

Figure 1: HPC cluster setting with contention-awareness.

The point of this work is to augment a typical HPC cluster installation with contention awareness, which it is currently lacking. Figure 1 depicts our contention-aware HPC framework and a job management cycle in it. If there are job submission requests posted in the last scheduling interval, the framework goes through steps 1–10. Otherwise, only steps 3–10 are being executed. For the most part, the management cycle is self-explanatory. We will clarify the remaining parts below. One thing we would like to highlight at that point is that there is a *two level scheduling* typically present in an HPC cluster: (a) cluster-wide scheduling on the head node (step 3) and (b) the OS-wide scheduling performed within each compute node (step 5). The first level decides what job processes to put on what nodes in a cluster, the second level then deals with scheduling of job processes assigned to a given node within that node. The first level scheduling is done by the job scheduler, the second level is usually done by the OS kernel running on the node.

The contention-related modifications are highlighted in italic. The list of changes include:

1) A virtual framework (OpenVZ) on each cluster node that allows installing Resource Manager monitoring tools (pbs_moms) and software libraries (OpenMPI, etc) inside the virtual machines (OpenVZ containers) instead of on the nodes themselves. This allows running jobs inside the containers as opposed to baremetal. The actual compute nodes are thus hidden from resource manager (RM) and cluster job scheduler (JS), they now treat virtual containers as nodes when performing cluster-wide scheduling (steps 3 and 4) or when checkpointing jobs for fault tolerance (step 8).

2) Online detection of the contention-sensitive jobs as they run on the compute nodes with Clavis user level daemon. Using a contention-aware scheduling algorithm called DINO within each node as opposed to the default OS scheduler to address shared resource contention locally (step 5).

3) Using the data obtained from Clavis, the framework periodically generates a report about shared resource contention in the cluster. It then shares the report with the job scheduler and cluster administrators upon request (steps 6, 7 and 10).

4) Using the information in the report, cluster administrators can then live migrate OpenVZ containers running memory-intensive MPI processes to the idle nodes on-the-fly, thus reducing job execution time. The framework allows to do it independently of the job scheduler, without affecting its job queues (step 9).

The instructions and scripts on deploying the framework are available at [4]. We next describe our modifications in more details.

3.1. HPC framework virtualization

Virtualization is a technique that allows running several OS instances simultaneously within a single hardware node. It is wildly adopted in the cloud, and some projects suggested to use the cloud computing resources like Amazon to host HPC clusters for educational purposes, e.g. StarHPC from MIT [13]. However, so far virtualization did not gain much popularity in HPC. Supercomputers are still largely scheduling jobs on their compute nodes baremetal. One of the main reasons is that virtualization is not free, there is an overhead of using it. The cluster workloads pose great demands on CPU, memory, network and IO subsystems of the supercomputer. And while CPU virtualization is not a problem these days due to Hardware Assisted Virtualization [2] implemented in most modern processor models, the overhead of virtualizing memory, network and IO can be substantial [14, 11, 15].

There exist many different virtualization solutions, each of which utilizes its own approach to virtualization. A good candidate for an HPC-based virtualization must provide near native memory, network and IO performance for the cluster MPI workloads. The Open Source OS-level virtualization solution called OpenVZ does offer such capabilities. Unlike other solutions (KVM, VMWare, Xen to name a few) that can run multiple operating systems at once, OpenVZ uses a single patched Linux kernel that is running both on the host and in the guests. It uses a common file system so each VM is just a directory of files that is isolated using chroot and special Linux kernel modules. However, because it does not have the overhead of a full hypervisor, it is very fast and efficient. Different studies have shown an average performance overhead of 1–3% for virtualizing OpenVZ as compared to using a standalone server [3, 14].

Memory management in OpenVZ highly favors contention-aware scheduling on NUMA multicore nodes. In other virtualization solutions memory management is usually performed by the guest and host OSes independently. It is generally very hard to say which pages of a total VM memory footprint belong to a particular MPI rank that is executing inside the guest. The memory in OpenVZ, on the other hand, is managed by the host kernel. As a result, the migration of the guest memory between different NUMA domains is as easy as migrating memory of the workload executing baremetal.

We found the following considerations to be important when using OpenVZ for the HPC-based virtualization:

1) OpenVZ does support live migration. Live migration allows moving a full process environment (processor context, memory, sockets, etc) between the hardware nodes on-the-fly. This is different from what is used in HPC nowadays, namely checkpointing a job, terminating it and restoring on different set of nodes. If done correctly, live migration does not terminate the open network connections of the job.

OpenVZ uses the approach to live migration that can be described as “suspend – move – resume”, in which the VM is being frozen and its whole state is saved to a file on disk. This file is then copied to another machine and the VM can then be unfrozen (restored) there. This approach is different from incremental memory migration used in KVM and Xen in which the source VM continues running while certain pages are moved across the network to the new machine. To ensure consistency, pages modified during this process must be re-sent. However, if the memory footprint changes too rapidly, the incremental approach may simply hang. That is why the “suspend – move – resume” approach may be more reliable, although it could take longer to complete. Although we only use Open Source tools within

this project, we note that a proprietary virtualization solution called Virtuozzo which is built on top of OpenVZ does support incremental memory migration [5].

2) For the live migration to be successful, the VM memory footprint should be locked pages free. That assumes the PBS and MPI libraries installed inside the VM as well as the cluster workload running on it should be compiled without locked pages support.

3) OpenVZ does not offer an InfiniBand support, so it is currently applicable only to the clusters with Ethernet interconnect. This is a disadvantage relative to other virtualization solutions that do support InfiniBand, KVM being one example [12]. However, developers may include IB support in future releases of OpenVZ [14].

In order to run cluster jobs inside the containers as opposed to baremetal, we install RM monitoring tools (pbs_moms) and software libraries (OpenMPI, etc) inside the OpenVZ virtual containers instead of on the nodes themselves. The actual compute nodes are now hidden from RM and JS, they now treat containers as nodes when performing cluster-wide scheduling. That allows to seamlessly, without interfering with JS or RM work, move the workload across the nodes if necessary. When deciding how many containers to create per each hardware node, we must take into account the number of physical cores, memory and the disk space available. Although most of the kernel modules are shared between containers on the node, each container creates its own Linux file structure on the disk. For our experimental machines, we found that the two containers per node is a reasonable value: it allows to mitigate the contention effects while not consuming much of a disk space or network traffic during migrations.

3.2. The node-level scheduling

Previous studies have demonstrated that the contention effects can incur severe degradation to the CPU-intensive cluster workloads [9, 6]. At the same time, the default Linux kernel scheduler that usually assigns ranks to cores within each compute node is not contention aware. In order to address this gap, we created Clavis. Clavis is a user level daemon written in C that is designed to implement various scheduling strategies under Linux running on multicore and NUMA machines. It is released under Academic Free License (AFL) and its source is available for download [8].

Clavis is started on each physical server when it is booted. It then proceeds as follows:

1) It automatically detects the OpenVZ containers present on the system and compute bound MPI ranks running in each of them. It then monitors the LLC missrate of the thus detected processes and classifies them as either devils or turtles.

3) It schedules the detected ranks using its contention-aware scheduling algorithm. For the NUMA multicore servers, we use the algorithm called Distributed Intensity NUMA Online (DINO) [9]. DINO separates devils in the memory hierarchy of the multicore system as far from each other as possible, thus reducing contention for memory hierarchy of the node. It uses LLC missrate metric to detect the memory intensiveness of the applications (whether its a devil or a turtle). When DINO moves the process away from its memory on the NUMA memory socket, it later pulls the memory close to the process to preserve memory locality.

4) Finally, it creates a concise profile of the workload in each container and saves it on the cluster file server. The report includes information about CPU utilization, allocated memory size, traffic and contention class of the MPI ranks running inside the container.

Clavis is able to improve workload performance on the node level in the following situations:

1) When all the ranks executing on the node are memory intensive (devils). The performance improvements in this case are modest, because, no matter how DINO schedules the ranks across the cores within a node, the shared resource contention still prevails. However, it is still possible to provide improvement relative to Linux Default scheduler in this case due to better memory locality achieved under DINO. We provide the experimental results in Section 4.

2) When devils are running alongside with turtles on the compute node. We studied this situation extensively before [9, 10].

3) When ranks are restored on the new compute node after live migration. Although live migration mechanism is able to fully restore process working environment on the new node, the distribution of the process' memory across NUMA domains after migration is completely random (hypervisor just restores the pages on the first NUMA domain available). Hence, proper scheduling of processes and their memory by DINO is able to rip big benefits in comparison with Default Linux scheduler that does not perform memory migration at all (Section 4).

3.3. Contention-aware resource usage reports

We now need the ability to aggregate the resource profiles collected by Clavis daemons on various compute nodes into contention-aware resource usage reports for HPC cluster as a whole. This is done by the RM. We use a popular Open Source RM called Torque. Its pbs_mom and pbs_server tools were modified within this work to periodically perform the following:

In each scheduling interval, Torque generates a list of active containers by parsing the *pbsnodes* command output. It reads the Clavis resource profiles for each container from the cluster storage. Torque determines which MPI ranks belong to which cluster jobs. The information aggregated per job in Torque is then supplied to the cluster scheduler via standard communication means.

3.4. The cluster-wide scheduling with Clavis-HPC

The task of a cluster scheduler is to maintain job queues and decide which nodes (containers in our implementation) will host the job ranks during execution. The scheduler does not consider contention effects in its scheduling decisions. It is also not able to migrate the load across the cluster on-the-fly, away from the contended nodes. To demonstrate the potential of our framework, we created Clavis-HPC, a user level cluster-wide load balancer written in Python. It runs in parallel with Maui on the head node and allows cluster administrators to seamlessly live migrate containers, along with the MPI ranks inside them, across the cluster. Clavis-HPC can be used to reduce shared resource contention as follows:

1) The cluster administrator uses Maui command-line tools to view the contention-aware resource usage report. The report includes information about which containers host contention-intensive workload (we call such containers *devilish*) and on what nodes these containers are located.

2) If there are compute nodes that host 2 devilish containers, it is worthwhile to migrate one of them away from this contended node, thus reducing the shared resource contention on it. There can be two cases: (a) the devilish container is migrated to the idle node and (b) the devilish container is swapped with a container that host turtles on some other node.

These load balancing decisions do not interrupt the active cluster workload or get in the way of the Maui queue management. Section 4 further describes the results of our scheduling.

4. Evaluation

We use the following facilities for testing.

Dell_Opteron: a small experimental 48-core cluster constructed from 3 servers. The servers have the following hardware configuration:

Two Dell-Poweredge-R805 (AMD Opteron 2435 Istanbul) servers have twelve cores placed on two chips. Each chip has a 6MB cache shared by its six cores. It is a NUMA system: each CPU has an associated 16 GB memory block (32 GB in total). Each server had a single 70 GB SCSI hard drive. We denote these machines as nodes X and Y.

One Dell-Poweredge-R905 (AMD Opteron 8435 Istanbul) server has 24 cores placed on four chips. Each chip has a 5 MB L3 cache shared by its six cores. Each core also has a private unified L2 cache and private L1 instruction and data caches. It is a NUMA system: each CPU has an associated 4 GB memory block, (16 GB in total). The server was configured with a single 76 GB SCSI hard drive. We treat this machine as two 12 core nodes with 2 NUMA domains each. We denote these nodes V and Z.

The servers were configured with Linux Gentoo 2.6.32 release 9. We use Maui as our cluster scheduler, Torque as a resource manager and SPEC MPI2007 as our workload. We run all the MPI

jobs with 24 ranks in total. 1GbE network is used as an interconnect between all the nodes. In case of the nodes V and Z, the traffic passes through the same 1GE interface to get to the other nodes. Sharing the same interface may be the bottleneck. To reduce its influence, we use nodes X and Y for our 12 rank per node runs. We only use nodes X, Y, V, Z for 6 ranks per node runs, thus minimizing the traffic via the shared interface (see below).

First of all, we have decided to perform the experiments to see if our DINO contention-aware algorithm provides any benefits to the MPI cluster workloads when scheduling on the node level. Series A on Figure 2 shows the average performance improvement for devils and turtles across different runs on Dell_Opteron cluster. As can be seen, DINO outperforms Linux Default for most of the applications tested.

Next, we wanted to demonstrate the benefits from the contention-aware cluster-wide scheduling. Series B shows that, for all the devils, there is a benefit in scheduling a job with 6 ranks per node (and taking twice as many nodes) rather than scheduling it with 12 ranks per node. The challenge here is to detect which ranks are devils and then spread only those apart. Series C–F shows the performance improvement of a solution where the job class is detected online and devils are spread across the nodes with OpenVZ live migration feature. We make the following conclusions from these results:

1) OpenVZ containers incur little overhead on its own (Series C). There can be performance degradation after migration though, if contention-unaware algorithm (like Linux Default) is used on the cluster nodes (Series E). Our contention-aware DINO algorithm (Series F), on the other hand, is able to provide solution, which is close to the best separated one of Series B (when ranks are started on different nodes from the start and no migration occurred).

2) Separating devil processes across nodes is beneficial for the overall performance of an MPI job (Series F). The influence of such separation is twofold: it relieves pressure on shared resources within a node, but it also increases pressure on the network. Series F results state that, even for a slow 1GbE connection and migration overhead included, performance benefit of reducing shared resource contention outweighs the interconnect slowdown for all devils. Those jobs that cannot benefit from separation (turtles) or those for whom migration penalty outweighs performance benefits (semi-devils) can be dynamically detected by the framework and used for load balancing (to plug the idle slots on partially loaded nodes). It is generally very hard to precisely measure what is the impact of each of the two conflicting factors on the overall performance of a job, since they interact in complex ways. Series D provides an estimate of the sensitivity of our workloads to exchanging data via the 1GbE interconnect. We obtained it by increasing the number of jobs running on the node: instead of 12 ranks of a single job, the ranks of two jobs were present on the node (6 ranks of each job). Both jobs were different instances of the same program. Such experimental configuration keeps the shared resource contention on the same level (12 ranks per node), but it increases the pressure on the network. Series D shows that most of the programs we considered do not take big hit when using the network and so the performance benefit of Series F is roughly due to the relief in shared resource contention. We do work on a metric that characterizes the interconnect sensitivity of a job, just like a LLC miss rate characterizes its sensitivity to shared resource contention. However, due to space limitations, we leave this topic outside the scope of the paper.

5. Conclusion

In this paper we demonstrated how the shared resource contention can be addressed when creating a scheduling framework for a scientific supercomputer. The proposed solution is comprised of the Open Source software that includes the original code and patches to the widely-used tools in the field. The solution (a) allows an online identification of the contention-sensitive jobs and (b) provides a way to make and enforce contention-aware scheduling decisions both on cluster level and within each multicore node.

We are currently working on improving our algorithms to simultaneously satisfy several conflicting scheduling goals (reduce contention, reduce power, reduce communication overhead) within the

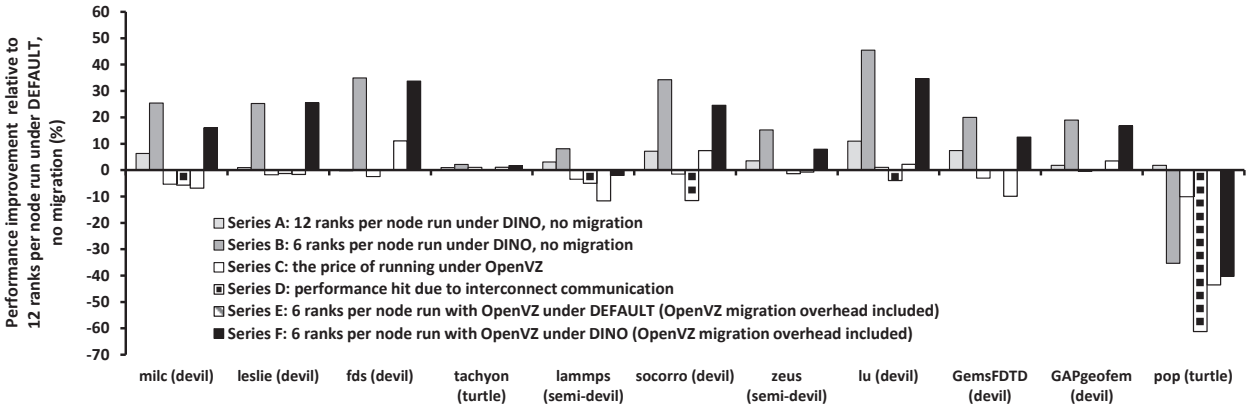

Figure 2: Results of the contention-aware experiments on Dell_Opteron.

described cluster environment simultaneously.

6. Acknowledgment

This material is based upon work supported in part by the National Science Foundation under Grant No. 0910812 to Indiana University for "FutureGrid: An Experimental, High-Performance Grid Test-bed." Partners in the FutureGrid project include U. Chicago, U. Florida, San Diego Supercomputer Center - UC San Diego, U. Southern California, U. Texas at Austin, U. Tennessee at Knoxville, U. of Virginia, Purdue U., and T-U. Dresden.

[1] A Study of Hyper-Threading in High-Performance Computing Clusters. *[Online] Available: http://www.dell.com/content/topics/global.aspx/power/en/ps4q02_leng*.

[2] Hardware Assisted Virtualization. *[Online] Available: http://en.wikipedia.org/wiki/Hardware-assisted_virtualization*.

[3] Open VirtualiZation. *[Online] Available: http://en.wikipedia.org/wiki/OpenVZ*.

[4] Optimizing Shared Resource Contention in HPC Clusters (project webpage). *[Online] Available: http://hpc-sched.cs.sfu.ca/*.

[5] Zero-downtime migration. *[Online] Available: http://download.swsoft.com/virtuozzo/virtuozzo4.0/ docs/en/lin/VzLinux-UG/14373.htm*.

[6] BLAGODUROV, S., AND FEDOROVA, A. In search for contention-descriptive metrics in HPC cluster environment. ICPE.

[7] BLAGODUROV, S., AND FEDOROVA, A. User-level scheduling on NUMA multicore systems under Linux. In *Linux Symposium* (2011).

[8] BLAGODUROV, S., AND FEDOROVA, A. User-level scheduling on NUMA multicore systems under Linux. In *Linux Symposium* (2011).

[9] BLAGODUROV, S., ZHURAVLEV, S., DASHTI, M., AND FEDOROVA, A. A Case for NUMA-Aware Contention Management on Multicore Systems. In *USENIX ATC* (2011).

[10] BLAGODUROV, S., ZHURAVLEV, S., AND FEDOROVA, A. Contention-aware scheduling on multicore systems. *ACM Trans. Comput. Syst. 28* (December 2010), 8:1–8:45.

[11] EL-KHAMRA, Y., KIM, H., JHA, S., AND PARASHAR, M. Exploring the Performance Fluctuations of HPC Workloads on Clouds. CLOUDCOM.

[12] HILLENBRAND, M., MAUCH, V., STOESS, J., MILLER, K., AND BELLOSA, F. Virtual infiniband clusters for hpc clouds. CloudCP '12.

[13] IVICA, C., RILEY, J., AND SHUBERT, C. StarHPC – Teaching parallel programming within elastic compute cloud.

[14] REGOLA, N., AND DUCOM, J.-C. Recommendations for Virtualization Technologies in High Performance Computing. CLOUDCOM.

[15] TIKOTEKAR, A., VALLÉE, G., NAUGHTON, T., ONG, H., ENGELMANN, C., AND SCOTT, S. L. Euro-par 2008 workshops - parallel processing. 2009, ch. An Analysis of HPC Benchmarks in Virtual Machine Environments.

[16] XIE, Y., AND LOH, G. Dynamic Classification of Program Memory Behaviors in CMPs. In *CMP-MSI* (2008).

[17] ZHURAVLEV, S., BLAGODUROV, S., AND FEDOROVA, A. Addressing Contention on Multicore Processors via Scheduling. In *ASPLOS* (2010).