Towards the contention aware scheduling in HPC cluster environment

Sergey Blagodurov
Systems Research Lab
Simon Fraser University
sergey_blagodurov@sfu.ca

Alexandra Fedorova
Systems Research Lab
Simon Fraser University
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 allocated CPU time, shared caches, memory bus, memory controllers, etc.) and when jobs are concurrently accessing cluster interconnects as their processes communicate data between each other. The cluster network also has to be used by the cluster scheduler in a virtualized environment to migrate job virtual machines across the nodes. The contention for cluster shared resources incurs severe degradation to workload performance and stability and hence must be addressed. The state-of-the-art HPC cluster schedulers, however, are not contention-aware. The goal of this work is the design, implementation and evaluation of an HPC scheduling framework that is contention aware.

1. INTRODUCTION

An *HPC cluster* is a group of linked computers, working together closely *thus in many respects forming a single computer* for the purpose of solving advanced computation problems. The computers (nodes) in the HPC cluster are connected through a cluster network and are treated by *a cluster resource management system* as a whole. HPC cluster is a *batch processing system*: it executes jobs at a time chosen by the cluster scheduler according to the requirements set upon job submission, defined scheduling policy and the availability of resources.

A job is submitted to the HPC cluster with a script that contains a program invocation and a set of attributes allowing cluster user to manage the job after submission and to request the resources necessary for the job execution. The attributes specify the duration of the job (*walltime*), offer control over when a job is eligible to be run, what happens to the output when it is completed and how the user is notified when it completes.

The system puts the job in a queue upon submission. The queue contains the jobs waiting for the execution on the cluster. Once the resources specified in the job submission script are available, and if the job is eligible to run according to the cluster scheduling policy, the system starts the job and executes it for the duration specified in the submission script.

In the process of its execution, the job uses the cluster resources assigned to it. In doing so, it can compete for those resources that are shared between several concurrently executing jobs. The parallel streams of execution within the same job can also compete among each other. In this work, we consider the following bottlenecks which result in performance degradation when highly contended:

- *Shared resource contention between the job processes in the memory hierarchy of each cluster node.* We assume all nodes to be multicore servers. In a multicore server (Figure 1), cores share parts of the memory hierarchy, which we term *memory domains*. When the job processes (ranks) execute on these cores, they *compete* for resources such as last-level caches (LLC), system request queues and memory controllers [14, 22, 13, 11].

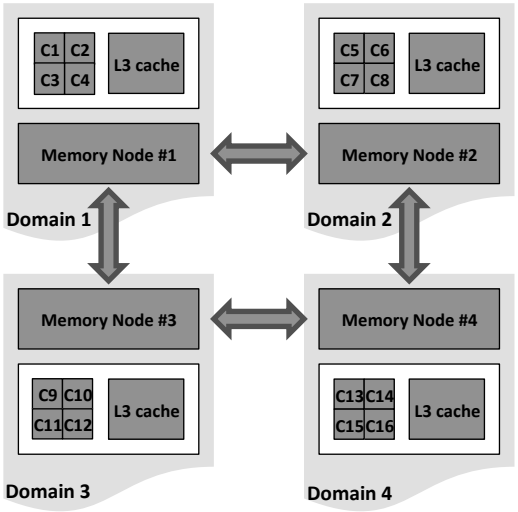

Figure 1: A schematic view of a cluster node with four memory domains and four cores per domain. There are 16 cores in total, and a shared L3 cache per domain.

- *Contention and overhead of accessing cluster interconnects (cluster network).* It can occur when (a) several processes of the same job spread among cluster nodes would want to communicate their data between each other[1]; (b) the cluster

[1]Cluster jobs are usually created using MPI, a Message Passing Interface, or other APIs that would allow their processes to *explicitly, without reliance on cache coherence* exchange the data between each other, even if the processes are running on different machines.

interconnect is used to migrate job virtual machines (VMs) across the nodes in a virtualized cluster environment.

The contention for shared resources results in *the increased execution time* for the contention-sensitive job. The probability of such increase is high, as HPC clusters are often used by many users and each of them in general does not know which jobs will be executed concurrently on the cluster at a given time. Due to these reasons, wallclock estimations have been historically poor, deviating approximately 20 to 40% from the factual value across a wide spectrum of systems [18].

If the job needs more time to execute than is specified in the script, the scheduler might try to allocate additional resources to the job. It might not be able to do so, as different jobs might be already scheduled for execution immediately after. If that happens, scheduler can terminate the job before its natural completion. *That is why it is essential to avoid the increase in execution time due to shared resource contention within the HPC cluster when possible.*[2]

The existing systems allow users to post certain coarse-grained resource demands in the submission script: the job can request a number of cluster nodes, processors, the amount of physical memory, the swap or the disk space. They, however, do not allow to provide fine grained description of resource requirements of the job (i.e. how sensitive the application is to the memory resource contention or to the internode exchange of the data). Because of that, the application may encounter shortage of actual computational resources allocated to it (e.g. cache space, memory controller bandwidth or internode interconnect bandwidth), even though the resource requirements specified during the job submission (the number of nodes, cores per node, memory and so on) are perfectly met.

The goal of this work is the design, implementation and evaluation of an HPC scheduling framework that is contention aware.

The rest of this paper is organized as follows: Section 2 describes the cluster framework assumed in this study and our experimental setup. Section 3 provides the summary of contributions made within the project so far. 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. SYSTEM OVERVIEW

Figure 2 describes an HPC cluster assumed within this study 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* present in a typical HPC cluster: (a) 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 .

In this section we will describe the state-of-the-art tools that are used in such HPC system. We will then suggest the modifications that need to be done for the HPC framework to be contention-aware in Section 3.

2.1 Experimental Platform

We use the following facilities in our work.

1. **Dell_Opteron**: a small testing 48-core cluster. The nodes have the following hardware configuration:

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, for a total of 32 GB main memory. Each server had a single 70 GB SCSI hard drive.

Dell-Poweredge-R905 (AMD Opteron 8435 Istanbul) 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, for a total of 16 GB main memory. The server was configured with a single 76 GB SCSI hard drive.

The nodes are connected through 100MbE and 1GbE networks. The nodes were configured with Linux Gentoo 2.6.32 release 9.

2. **HP_Nehalem**: a testing 96-core (192-HT-context) cluster. The nodes have the following hardware configuration:

HP ProLiant SL390 (Intel Xeon X5650 Nehalem) servers have 12 cores placed on two chips (24 thread contexts if HyperThreading is enabled). Each chip has a 12 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 24 GB memory block, for a total of 48 GB main memory. The servers were configured with two 2 TB SCSI hard drives.

We enabled HyperThreading on these machines since some HPC workloads can benefit from it [1].

The nodes are connected through 1GbE and 10GbE networks. The nodes were configured with Scientific Linux 2.6.32 release 9.

3. **IBM_Nehalem (india)**: a 1024-core cluster hosted by FutureGrid. The nodes have the following hardware configuration:

IBM iDataPlex (Intel Xeon X5570 Nehalem) servers have 8 cores placed on two chips. Each chip has an 8 MB L3 cache shared by its four 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 12 GB memory block, for a total of 24 GB main memory. The servers were configured with a single SCSI hard drive.

The nodes are connected through 1GbE and InfiniBand networks. The nodes were configured with Scientific Linux 2.6.32 release 9.

We use OpenIPMI to measure power consumption of the compute nodes and *Integrated Lights-Out (iLO 3)* feature of the Proliant servers in HP_Nehalem to turn nodes on/off. iLO is an embedded server management technology that makes it possible to monitor and perform activities on an HP server from a remote location. The iLO Ethernet card has a separate network connection (and its own IP address) to which the framework periodically connects via HTTPS and issues the power up/ power down requests.

We use SPEC MPI2007 as our workload. MPI2007 is SPEC's

[2]To prevent a premature job termination, the user can increase the walltime in the submission script. That, however, can result in early completion, in case the contention does not appear in that run. The scheduler will try to use the freed resources to run other jobs, but none might be eligible to run at that time, so, in general, the cluster user will be charged for the time specified in the submission script. The issue of charging rates (the amount users pay for compute resources) might feel irrelevant to some academic facilities, but is important to the industry level HPC clusters. To support it, Maui and Moab schedulers have a built-in ability to charge users with rates based on QoS, resources, and time of day [17]. While using HPC clusters for academic research purposes is typically free of charge, it can be limited to certain yearly allocation quotas, measured in core years [2].

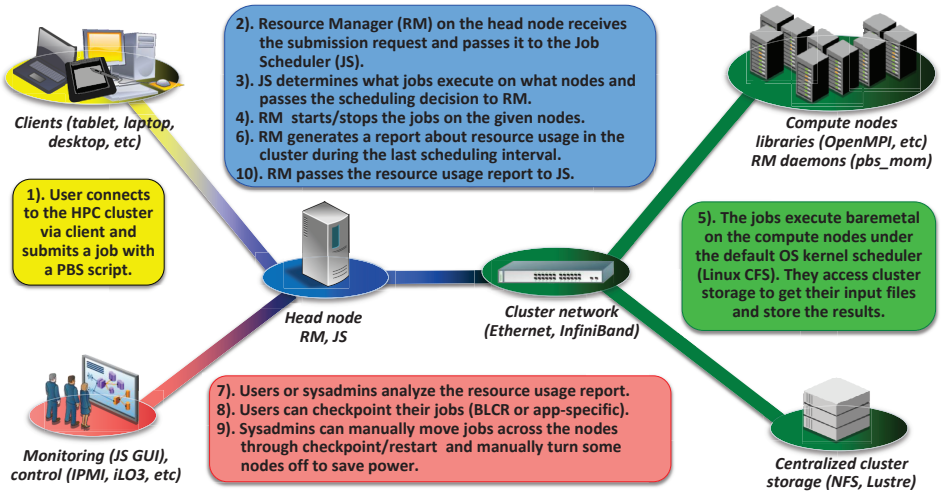

Figure 2: HPC cluster setting assumed in this study.

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 (an MPI process of the job) for the medium benchmark subset and 2 GB per rank for the large subset.

2.2 Experimental Setup

Setting up the experimental testbed for HPC research is challenging. The main purpose of the big HPC clusters is to facilitate research across the disciplines (physics, chemistry, simulations, graphics, etc.). Cluster users need to submit and execute their scientific codes. They do not typically require an access to the whole cluster framework. HPC research, on the other hand, requires full access to the cluster infrastructure. For example, augmenting the aforementioned HPC cluster framework with contention awareness and testing the proposed solution, assumes: (a) root access to the cluster RM and scheduler and (b) baremetal root access to the compute nodes. This may be in conflict with the cluster policies. Another challenge is cluster reliability. HPC cluster professionals want their expensive supercomputers to provide value. Faults are highly undesired. Doing research, on the other hand, requires to break things (inducing several faults per day and investigating their causes is a good metric for research effectiveness).

FutureGrid offers a good trade-off between these potentially conflicting requirements: net-booting of compute nodes with user supplied images. With this feature, we were able to recreate a research-friendly HPC framework within a big cluster like so:

1) Create an image that contains a Linux distribution and a set of cluster software tools (Maui, Torque, OpenMPI, etc.) that will comprise the framework.

2) Upload the resulting image into a repository of the image management tool called Rain.

3) Book compute nodes from a FutureGrid cluster that supports HPC (e.g. india) in exclusive mode. During the submission, the user specifies an additional "os" parameter to qsub command with the image name in the repository.

4) The custom image is then booted bare-metal onto the booked nodes. They are being temporarily excluded from the cluster-wide management for the duration of the job.

5) After the nodes are up and running, we need to configure our own testing framework within the booked nodes (bring up the RM, scheduler and PBS clients on each booked node, detect the amount of compute resources available, etc.). This is done automatically with a set of shell scripts.

6) Perform the experiments.

7) Upon the job completion, the nodes are rebooted into a default image by Rain.

This setup allows to quickly perform, revise and scale HPC-related experiments on a commodity scientific cluster. The instructions on configuring the framework using FutureGrid infrastructure along with the shell scripts for deploying the experimental framework are available at [7].

2.3 The Choice of the Cluster Scheduler and Resource Manager

In this section, we look closely at the cluster resource management system (the "blue" section in Figure 2) which comprises (1) a resource (workload) manager (RM) and (2) a job (cluster) scheduler [16, 17]. We describe why we chose Maui as our cluster scheduler and Torque as a Resource Manager. A brief description of their work is also provided.

The resource manager:

- Sets up a queuing system for users to submit jobs. The reason for introducing queues is to prevent jobs from competing with each other for limited compute resources of the HPC cluster.
- Maintains a list of available compute resources.
- Receives job submission requests from cluster users.
- Periodically reports to the job scheduler information, necessary for it to make a scheduling decision (updates on the status of job queues, loads on compute nodes, resource availability, etc.).
- Enforces the scheduling decision received from the job scheduler (launches the job processes on the specified nodes).
- Reports the status of previously submitted jobs to the user upon request.

The job scheduler receives periodic input from the resource manager regarding job queues and available resources, and makes a

schedule that determines the order in which jobs will be executed. In other words, job scheduler tells the resource manager what jobs to run, when to run them, and where. Most resource managers have an internal, built-in job scheduler, but system administrators usually substitute an external scheduler for the internal one to enhance functionality (the features provided by external cluster schedulers usually include advanced reservation, backfilling, preemption and many more [18]).

We were choosing among three popular cluster schedulers: Moab, Maui and Condor (Table 1). The choice was made in favor of Maui. Moab, arguably the most popular cluster scheduler in the commercial setups, does have more capabilities than Maui (especially in the amount of monitoring data that is provided by the scheduler). It is, at the same time, proprietary and quite expensive to install. Maui, on the other hand, is open-source and free. Maui and Moab are developed by the same company, with Moab being essentially an extension for Maui. Several university cluster installations has switched from Condor to Maui recently ([9] for instance). Moreover, Maui is being used by the Compute Canada clusters (in particular, by the BC-based WestGrid) and by the local HPC cluster in the Faculty of Applied Science at SFU (The Colony HPC cluster). All of this is important to us when choosing a scheduler to work with.

The choice of Maui as the job scheduler largely determined choosing Torque as the project's RM since is is closely integrated with (and as popular as) Maui itself (Table 2).

2.4 Maui Cluster Scheduler

Maui has a two-phase scheduling algorithm. During the first phase, the high-priority jobs are scheduled using *advance reservation*. In the second phase, a *backfill* algorithm is used to schedule low-priority jobs between previously scheduled jobs. Advance reservation uses execution time predictions (walltime) provided by the users to reserve resources (such as CPUs and memory) and to generate a schedule. Hence, it serves as the mechanism by which the scheduler guarantees the availability of a set of resources at a particular time. Given a schedule with advance-reserved, high-priority jobs and a list of low-priority jobs, a backfill algorithm tries to fit the small jobs into scheduling gaps. This allocation does not alter the sequence of jobs previously scheduled, but improves system utilization by running low priority jobs in between high-priority jobs. To use backfill, the scheduler also requires a runtime estimate, which is supplied by the user when jobs are submitted. The following is the sequence of work flow of Maui [16, 18, 17]:

0. Maui starts a new iteration when the following event(s) occur:

a) Job or resource change event occurs (i.e. job termination, node failure)

b) Reservation boundary event occurs

c) A configurable timer expires

d) Via an external command

1. On every iteration, Maui obtains updated data from RM regarding node, job state, configuration.

2. Historical statistics and usage information for running and completed jobs are updated.

3. Refresh Reservations. This step adjusts existing reservation incorporating updated node availability. Changes in node availability may also cause reservations to slide forward or backward in time. Reservations may be created or removed. If the job that possess a reservation is idle (waiting in the queue) it is provided an immediate access to the reserved resources.

4. A list is generated that contains all jobs which can be feasibly scheduled. Availability of resources, job credentials, etc. are taken into account in generating this list.

5. Prioritize feasible jobs according to the historical usage information (to enforce resource fair sharing), various job attributes, scheduling policies and resources required by each jobs.

6. Schedule jobs in priority order, starting the jobs it can and creating advanced reservations for those it cannot until it has made reservations for the top N jobs where N is a cluster configurable parameter.

7. Backfill. Maui determines current available backfill windows in between reservations and attempts to fill them with the remaining jobs which are eligible to run during these holes.

Maui supports job preemption. High-priority jobs can preempt lower-priority or backfill jobs if resources to run the high-priority jobs are not available. In some cases, resources reserved for high-priority jobs can be used to run low-priority jobs when no high-priority jobs are in the queue. However, when high-priority jobs are submitted, these low-priority jobs can be preempted to reclaim resources for high-priority jobs.

3. CONTENTION-AWARE MODIFICATIONS

The Torque/Maui installation outlined on Figure 2 is fully functional and can schedule jobs as is. The point of this work is to augment this typical HPC cluster setup with contention awareness, which it is currently lacking. Figure 3 provides the description of the necessary additions given in red. The list of changes include:

- An opportunity to supply contention-descriptive metrics in the PBS scripts upon job submission (step 1).
- A virtual framework (OpenVZ) on each cluster node that allows installing RM 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 RM and JS, they now treat containers as nodes when performing the cluster-wide scheduling (steps 3 and 4) or when checkpointing jobs for fault tolerance (step 8).
- Online detection of the contention- and communication-sensitive jobs as they run on the compute nodes with Clavis user level daemon. Using a contention-aware scheduling algorithm within each node as opposed to the default OS scheduler to address shared resource contention (step 5).
- Using the data obtained from Clavis, the framework periodically generates a contention-aware report about resource usage in the cluster. It then shares the report with the job scheduler and the cluster users upon request (steps 6, 7 and 10).
- Performing simple automatic contention- and power- aware cluster-wide scheduling decisions, independently of the job scheduler (step 9).

We will describe these changes below.

The contention-descriptive metrics.

The first question that we had was what metrics to use in order to characterize the cluster job as sensitive to (a) resource contention within the memory hierarchy of a cluster compute node and (b) accessing cluster interconnects?

Multiple studies investigated ways of reducing resource contention within a mulicore server. One of the promising approaches that emerged recently is contention-aware scheduling. 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 [21] we will refer to

Cluster scheduler	*Description*
RM internal schedulers	Open Source. Usually are very simple and are hence not used in a realistic cluster environment.
Maui	Open Source. Maui extends the capabilities of base RM schedulers by adding the following features: - Job priority policies and configurations - Job advance reservation support - QOS support including resource access control - Extensive fairness policies - Non-intrusive 'Test' modes - and many more [4].
Moab	Like Maui, is capable of supporting multiple scheduling policies, dynamic priorities, reservations, and fairshare capabilities. In addition, has integrated billing mechanisms and an advanced graphical cluster administration with integrated documentation and wizards. Unfortunately, Moab is proprietary software which makes it difficult to analyze within the academia.
Condor	Supports many features implemented in Maui (backfill, reservations, etc.). Maui, however, became a popular scheduler for academic HPC environments (for example, our local WestGrid cluster facilities use Maui for scheduling) so we opted for it.

Table 1: Comparison of different job schedulers.

Resource manager	*Description*
OpenPBS	An initial Open Source RM capable of a basic job maintenance and control functionality. It was a starting point for many advanced RMs, including Torque.
Torque	This is an advanced Open Source RM based on the original OpenPBS project. Torque incorporates many advances in the areas of scalability, reliability, and functionality and is currently in use at tens of thousands of leading government, academic, and commercial sites throughout the world.
SLURM	Simple Linux Utility for Resource Management is an open-source RM designed for Linux clusters. It has many good features including portability and highly tolerance of system failures (including failure of the node executing its control functions). Torque, however, is developed by the same company as Maui and is meant to be used with it by default. It is also believed that SLURM is hard to configure properly to work with Maui ([8] for instance).

Table 2: Comparison of different resource managers.

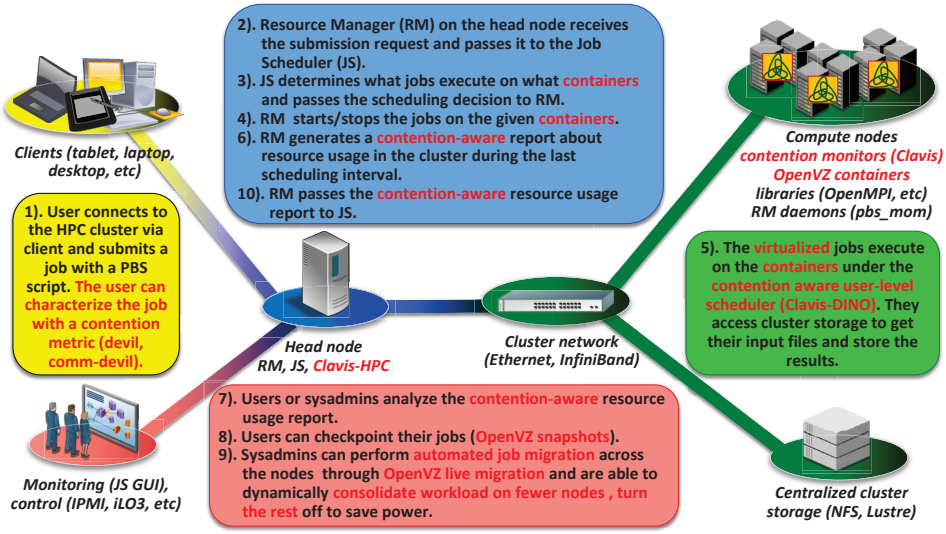

Figure 3: HPC cluster setting with contention-awareness.

these applications as *devils*. Applications with a low rate of memory requests are referred to as *turtles*. Our methodology from the earlier work [13, 14, 22] allowed us to identify the *last-level cache (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.

When choosing a metric that describes the communication intensiveness of a job (how sensitive the job is to a slowdown due to accessing cluster interconnects), we wanted to validate, if the network traffic of the job correlates with its performance level. Our initial experiments [10] have in fact shown a correlation of 0.73 between the amount of traffic transferred by a job across the network and the job slowdown due to that. For the workloads we considered (SPEC MPI 2007) the job slowdown only happens on slow networks (up to 100MbE) or when there is a network impairment

happening.

The two metrics (LLC missrate and job traffic) are parts of the sensitivity profile of the cluster job. The user can obtain them from the contention-aware resource usage report (step 6) during a profiling run. They can then be included in the resource list specified in the submission PBS script, together with the already existing parameters of number of cores, memory, etc. This new information will help the framework to perform contention-aware scheduling decisions in step 9. For example, if the job appears to be sensitive to accessing the cluster interconnect, as is suggested by the big network traffic value from the previous runs, the scheduler might take it into account and try to schedule the job on as few nodes as possible. The high missrate is a good reason for a scheduler to prevent the job submission on the nodes together with other memory intensive jobs.

HPC framework virtualization.

Virtualization is a popular technique that allows running several OS instances simultaneously within a single hardware node. It is the basis of the cloud technology: users of such services as Amazon EC2, Microsoft Azure and HP CloudSystem run their workloads on virtual machines instead of the hardware nodes themselves. Virtualization allows the datacenter administrators to effectively hide the underlying hardware infrastructure from the clients, thus making the cloud datacenter maintenance much more flexible. The virtualization offers the following main advantages when being applied to a datacenter:

The ability to live migrate workload across multiple hardware nodes. Live migration allows moving a full job environment (processes, 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[3]. If done correctly, live migration does not terminate the open network connections of the job, nor does it significantly affect the job performance. Live migration is often being used to oversubscribe the hardware nodes in the cloud. Instead of running on their own dedicated machines, several underutilized virtual instances are consolidated on fewer nodes, thus sharing their resources, which leads to a more effective usage of the datacenter infrastructure.

The ability to make a snapshot of a running VM. Unlike live migration, the preserved state of the running job is now moved not to a different node, but is stored locally on the disk, or on the centralized file storage, ready to be restored in case there are software or hardware faults in the datacenter. This ability of a virtualized datacenter framework to save the running VM state leads to a better fault tolerance in the datacenter. Users are no longer concerned about saving the intermediate data of their applications manually for a possible later restore, they can now issue a simple VM checkpoint request, independent of the software running inside the VM.

Perhaps the main outcome of creating a virtualized datacenter is that virtual workload scheduling onto the hardware nodes becomes completely hidden from the cluster users. For instance, in Amazon EC2 the users only request the virtual images to run their software on. They do not know what hardware nodes host their workload. The Amazon cluster administrators can consolidate the load at any time with live migration and turn the freed nodes down to save power. The users are usually unaware of this underlying cluster scheduling.

The virtualization can be a very useful technique, however, so far it 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 [3] implemented in most modern processor models, the overhead of virtualizing memory, network and IO can be substantial [19, 15, 20].

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 operating system-level virtualization technology called OpenVZ does offer such capabilities. Unlike other virtualization technologies (KVM, VMWare, Xen to name a few) that are capable of virtualizing the entire machine and 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 kernel modules. However because it doesn't have the overhead of a full hypervisor, it is very fast and efficient: different studies have shown a 1–3% performance penalty for OpenVZ as compared to using a standalone server [6, 19].

We would also like to point out that the memory management in OpenVZ highly favors the 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 a single kernel. As a result, the migration of the guest memory between different NUMA nodes is as easy as migrating memory of the workload executing baremetal.

The contention-aware scheduling framework introduced in this paper has OpenVZ support. We found the following considerations to be important when using OpenVZ for the HPC-based virtualization:

Like many other virtualization solutions, OpenVZ does support live migration (moving a VM from one physical server to another without shutting down the VM or terminating its open connections). Because OpenVZ is essentially a chroot environment on the host machine, the copying of the disk contents to the new server during the live migration is being done incrementally with rsync: the majority of files are copied beforehand so that, when the actual migration happens, it finished faster. Unfortunately, the same does not hold for the memory migration between the physical nodes. *OpenVZ does not currently support the incremental memory migration*[4]. Instead, it uses the approach that can be described as "suspend – checkpoint – move – restore – 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 more reliable than incremental memory migration (if the memory footprint changes too rapidly, the incremental approach may simply hang), but it can take longer to complete. This is especially true for the MPI cluster jobs that usually have big memory footprints. We provide the data

[3]The checkpointing mechanism can be implemented either manually or by using a special libraries, BLCR being one example. The difficulty with the first method is that it is program dependent: only the application developers know what data to save on the disk for the later restore. While BLCR offers a more universal approach, the restore process may not go as planned: some programs do not work with BLCR and those that do work may experience PID conflicts when being restored on different nodes. In both cases, the open connections of the moving jobs have to be terminated.

[4]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.

on live migration overhead with OpenVZ in Section 4.

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.

The fault tolerance with OpenVZ is possible, but may be limited to the lifetime of the networking sockets preserved in the snapshot. In other words, if the time period between checkpointing and restoring is longer than the tcp socket lifetime, some connections can be terminated upon restore. This can be addressed by bumping tcp limits via Linux sysfs.

OpenVZ does not currently offer an InfiniBand support. However, developers may include one in future releases of OpenVZ [19].

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. Just like with the cloud installations, the actual compute nodes are now hidden from RM and JS, they now treat containers as nodes when performing the cluster-wide scheduling. That allows to seamlessly, without terminating the jobs, 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 one creates its on Linux file structure on the disk. For our clusters, 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.

The node-level scheduling.

Previous studies have demonstrated that the contention effects can incur severe degradation to the CPU-intensive cluster workloads [13, 10]. 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 [12].

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

- It automatically detects the OpenVZ containers present on the system and compute bound MPI ranks running in each of them. The rest of the processes are no interest to us and hence are scheduled according to Default.
- It monitors the LLC missrate of the thus detected processes and classifies them as either devils or turtles.
- It also measures traffic that the workload inside the local containers have exchanged with the rest of the cluster. It creates a communication matrix that reflects what containers on the local machine are communication-bound.
- 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) [13]. 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 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.
- Finally, it creates a concise profile of the workload in each container. It includes the information about CPU utilization, allocated memory size, traffic and contention class of the container MPI ranks as follows.

Every line in the report corresponds to a particular compute bound PID and contains the information about the process usage characteristics:

<PID of the process> (<timestamp>):

<PID of the spawning shell (needed to assign the process to the particular cluster job in Torque)>

<process binary name>

<OpenVZ container ID (CTID)>

<Contention class (devil/turtle)>

<Resources used>

For example: 19415 (Sat Jun 2 00:49:06 2012): 7145 GemsFDTD_base.a 132 devil CPU 100 MISS RATE 121.34 MEM 4.000000 TRFC SNT 120.12 RCVD 5.56 IO WR 10.01 RD 25.66

This data is then saved under the name "<node hostname>.report" on the cluster storage for the later use in cluster-wide scheduling (see below).

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 Torque RM whose pbs_mom and pbs_server 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 then reads the Clavis resource usage report for each container from the cluster storage.
- After that, Torque determines which MPI ranks belong to which cluster jobs. This is done by accessing pjob->ji_qs.ji_jobid member of its internal job data structure. The value is the PID of the shell that spawned the MPI job (for the given MPI rank it is always the Parent PID of its Parent PID). This is matched with the value supplied in the usage report.

The information aggregated per job in Torque is then supplied to the cluster scheduler via standard communication means. It is saved as additional parameters in pjob->ji_wattr array and can be viewed upon request.

The cluster-wide scheduling with Clavis-HPC.

The Maui job scheduler receives periodic input from the resource manager regarding job queues and available resources. It then makes a queue that determines the order in which jobs will be executed. Maui does not consider contention effects in its decisions. It is also not able to migrate the load across the cluster on-the-fly when the conditions in which workload is running change.

To demonstrate the potential of our virtualized contention-aware cluster framework, we created Clavis-HPC, a user level cluster-wide scheduler written in Python. It works in parallel with Maui and schedules the containers across the physical nodes, while Maui manages the job queue of the workload running inside the containers. Clavis-HPC works as follows:

1) It reads the information about the contention and communication intensiveness of each container in the cluster from the resource usage report.

2) Simultaneously, it measures the power consumption of each node provided by OpenIPMI or iLO3.

3) The information obtained in the first two steps is then used by Clavis-HPC to make a decision about what to do next. The scheduler can (a) leave a given container unchanged; (b) decide to migrate the container to a different node and (c) turn some node off. A scheduling decision is thus a set of instructions about which physical servers remain active and which physical server will host each container. Each scheduling decision is valid for the next scheduling interval, upon the end of which it is reevaluated. Once the decision is made, the framework enforces it.

For now, we are performing simple scheduling decisions that try to optimize one of the three cluster-wide goals: (1) improve the performance by mitigating the shared resource contention in the cluster; (2) save power by consolidating the load on fewer servers and turning the idle nodes off; (3) improve performance by reducing the communication overhead via slower network in case of impairment. All of the 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. EXPERIMENTAL RESULTS

In this Section we demonstrate what benefits the contention-aware cluster framework described above can provide on our testing clusters. Before running the cluster-wide experiments, we decided to take a quick look at the MPI apps we are going to run (Table 3). As can be seen, more than half of the apps are memory intensive with high missrate (we call them *devils*). The rest are turtles (low missrate) or semi-devils (the missrate is just above the threshold). We have used two types of inputs in our experiments: the medium and the large input sets from SPEC MPI 2007. When a program is running with the large set, its name on the graphs starts with "l"[5].

Name	*LLC Missrate * 10.000*	*Traffic (mbps)*	*Class*
milc	15.95	70	devil
leslie3d	83.37	28	devil
fds4	78.13	0	devil
pop2	20.64	67	turtle
tachyon	0.48	0,08	turtle
lammps	35.22	27	semidevil
socorro	87.82	36	devil
zeusmp2	33.81	14	semidevil
lu	65.55	18	devil
GemsFDTD	129.12	35	devil

Table 3: SPEC MPI 2007 apps characterization.

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. Figure 4 shows the average performance improvement for devils and non-devils across different runs on Dell_Opteron cluster. As can be seen, DINO outperforms Default (denoted as "no DINO") for all devils and semi-devils and does not make any difference for turtles. This is expected, since turtles do not exhibit high memory intensiveness and hence are prone to the memory contention effects. The benefits from using contention-aware scheduling can be even bigger if the HyperThreading is enabled in the cluster as is the case for HP_Nehalem on Figure 5. Here more ranks are able to execute per each compute node and the contention effects are thus exacerbated. Figure 7 shows the modest improvements on IBM_Nehalem. This is due to the fact that the Nehalem processors installed in it have rather big (2MB per core) last-level caches which softens the contention effects.

Next, we wanted to demonstrate the benefits from the contention-aware cluster-wide scheduling. Figure 4 shows that, for all the devils, there is a benefit in scheduling a 12 process job across two nodes (6 processes per node) rather than scheduling it on one node. The slowdown for semidevils is only present when the given semidevil is co-scheduled with a devil on two nodes. Figure 4 also shows the performance improvement of a solution where the job class is detected online and devils are spread across two nodes with OpenVZ live migration feature relative to solution where the same job runs on one node till completion. We make the following conclusions from these results:

- Separating devil processes across nodes is beneficial for the overall performance of an MPI job.
- 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).
- If the job is very communication intensive (i.e., *pop*) and not memory intensive its processes should be placed on fewer nodes.
- OpenVZ containers incur little overhead on its own. There can be performance degradation after migration though, if contention-unaware algorithm (like Linux Default) is used on the cluster nodes. Our contention-aware DINO algorithm, on the other hand, is able to provide solution, which is close to the best separated one (when ranks are started on different nodes from the start and no migration occurred).

The overhead of OpenVZ migration is bigger for the large set jobs on Figure 5, as the time it takes to migrate container from one node to another is proportional to the total memory footprint of the MPI ranks running inside it.

Figure 6 demonstrates the amount of power savings that can be obtained by consolidating the load on fewer servers with live migration and turning the freed nodes off. In these experiments, the compute nodes of HP_Nehalem were half loaded in the beginning. This is a typical situation in an underloaded clusters, when the goal of the job scheduler is to load balance jobs across the nodes. Now let's assume that the power supply of the supercomputer has decreased. This can be the case if it is operating on renewable sources of energy, whose output is not stable and changes throughout the day. The workload is now migrated onto half the nodes and the rest are switched off (except for server2 which serves as the head node). This is done live without interrupting the load or affecting the Maui job queue. When the supply is risen again, the load can be again spread across the cluster.

Finally, Figure 7 shows that the performance of the cluster workload can be improved on-the-fly in case the cluster network starts to falter[6]. In these experiments, the workload has been once again consolidated on fewer nodes, but this time, the containers running processes of the same job were placed together after migration. We see that the framework is able to improve performance by reducing

[5]Due to memory limitations and the lack of enough nodes to scale the load at the time, we were not able to test some of the large set workloads on Dell_Opteron and IBM_Nehalem.

[6]We use netem kernel module on the compute nodes to induce a modest network impairment as described in [5].

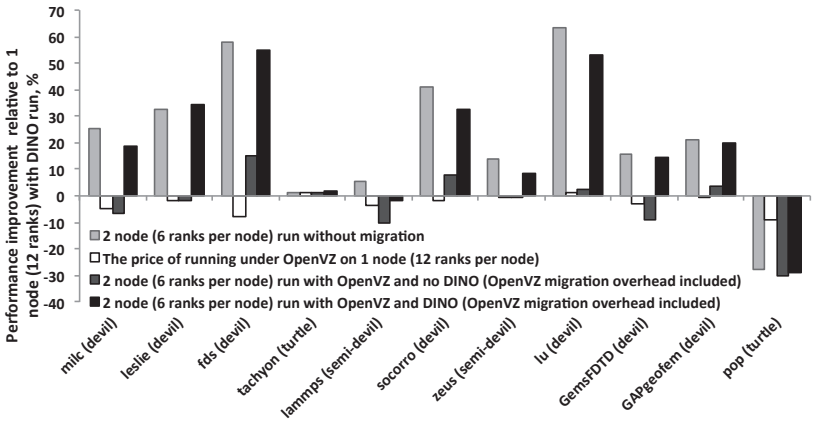

Figure 4: Performance improvement given by contention aware framework with different migration options on Dell_Opteron.

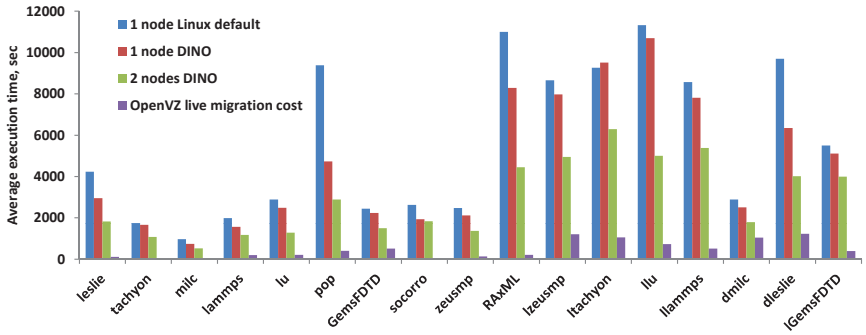

Figure 5: Performance improvement and OpenVZ migration cost given by contention aware framework on HP_Nehalem.

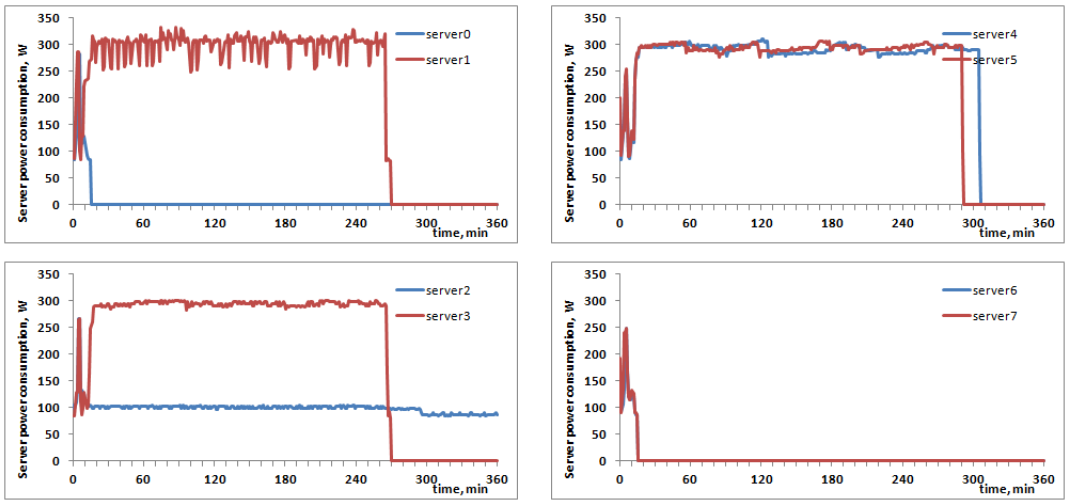

Figure 6: Power savings from the workload consolidation given by contention aware framework on HP_Nehalem (lGemsFDTD – llu workload).

the communication overhead via slower network in case of impairment.

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-intensive jobs and (b) provides a way to make and enforce a simple 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 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

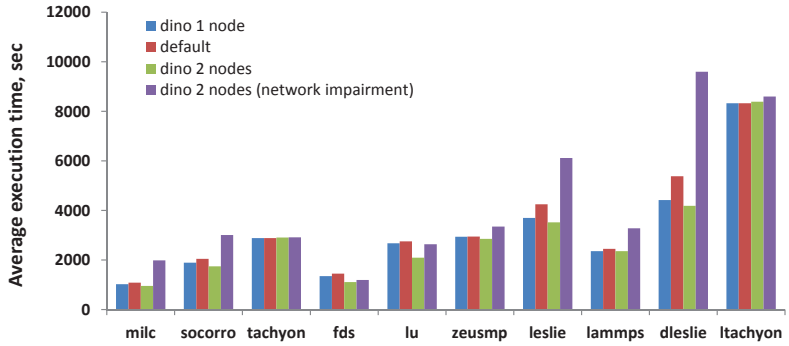

Figure 7: Performance improvement from the workload consolidation given by contention aware framework on IBM_Nehalem.

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.

7. REFERENCES

[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] Compute canada/calcul canada resources. *[Online] Available: https://ccdb.computecanada.org/browse/resources_in.*

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

[4] Maui cluster scheduler fetures. *[Online] Available: http://www.clusterbuilder.org/encyclopedia/alphabetized/m/maui-cluster-scheduler.php.*

[5] Network emulation with netem kernel module. *[Online] Available: http://www.linuxfoundation.org/collaborate/workgroups/networking/netem.*

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

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

[8] Slurm/maui configuration notes. *[Online] Available: http://www.supercluster.org/pipermail/mauiusers/2005-January/001442.html.*

[9] Transition to torque/maui. *[Online] Available: http://www.urc.uncc.edu/urc/old-announcements/transition-to-torquemaui/.*

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

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

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

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

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

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

[16] GUPTA, A., AND BARUA, G. Cluster schedulers. *[Online] Available: http://www.stanford.edu/ abhig/Docs/Cluster-SchedulerReport.pdf.*

[17] IQBAL, S., GUPTA, R., AND FANG, Y.-C. Job scheduling in hpc clusters. *[Online] Available: http://www.dell.com/downloads/global/power/ps1q05-20040135-fang.pdf.*

[18] QUINN, D. J., JACKSON, D., SNELL, Q., AND CLEMENT, M. Core algorithms of the maui scheduler. pp. 87–102.

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

[20] 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.

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

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