In Search for Contention-Descriptive Metrics in HPC Cluster Environment

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

Alexandra Fedorova
Systems Research Lab
Simon Fraser University
fedorova@cs.sfu.ca

ABSTRACT

In this paper, we argue that the modern HPC cluster environments contain several bottlenecks both within cluster multicore nodes and between them in the cluster interconnects. These bottlenecks represent resources that can be of high demand to several jobs, concurrently executing on the cluster. As such, the jobs can compete for accessing these resources and experience performance degradation due to contention. We point out, that, although the contention for shared resources like memory hierarchy of the cluster nodes, accessing the cluster interconnects or sharing the floating point unit can incur severe performance degradation to the cluster workload, the state-of-the-art cluster schedulers do not contain adequate means of addressing it. To fill this gap, we propose a new set of metrics that models shared resource contention and represents a fine-grained information about each job's resource utilization and communication patterns. The necessary information can be obtained with the performance counters within cluster nodes and cluster interconnect monitoring between them.

Categories and Subject Descriptors

D.4.1 [**Process Management**]: Scheduling

General Terms

Algorithms, Management, Measurement, Performance

Keywords

HPC Clusters, Multicore systems, Scheduling, Shared Resource Contention

1. INTRODUCTION

Assume the target environment of a High-Performance Computing (HPC) cluster. The nodes in the cluster are connected through a cluster network and are managed by a cluster scheduler *as one entity*. HPC cluster is a *batch processing system*. It executes a job at a time chosen by the cluster scheduler according to the requirements set upon job submission, defined scheduling policy and the availability of resources (unlike *an interactive system* where commands are executed when entered via the terminal or *a transactional system*, where the jobs are executed as soon as they are initiated by a transaction request from outside the cluster).

A job submitted to the HPC cluster is typically a shell script which 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.

HPC cluster scheduler 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 policy, the scheduler starts the job and executes it for the duration specified in the submission script. If the job terminates before that time, scheduler will try to use the resources freed by the job termination to run other processes. However, it might be that no jobs will be eligible to run at that time, so, in general, the cluster user will be charged for the time specified in the submission script. 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. In both cases, it is essential for HPC cluster user to *correctly predict the job execution time* so that the user will not be charged for the unnecessary resources if the job terminates early and so that her job will not be killed by the cluster scheduler due to its extended execution time.

The right prediction of job execution time relies on knowledge of *what resources are necessary for the job to complete in the required amount of time*. The existing cluster schedulers 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. All of this information, however, does not reflect how sensitive the job is to the resource contention from different jobs that will be simultaneously executing in the same cluster with the submitted one. The possible resource bottlenecks that can result in performance degradation due to job contention for them, include:

- *Shared resource contention between the applications in the memory hierarchy of each cluster node.* We assume all nodes to be multicore systems. In a multicore system (Figure 1), cores share parts of the memory hierarchy, which we term *memory domains*, *compete* for resources such as last-level caches (LLC), system request queues and memory controllers [10, 13].
- *Contention and overhead of accessing cluster interconnects*

ICPE'11, March 14–16, 2011, Karlsruhe, Germany.

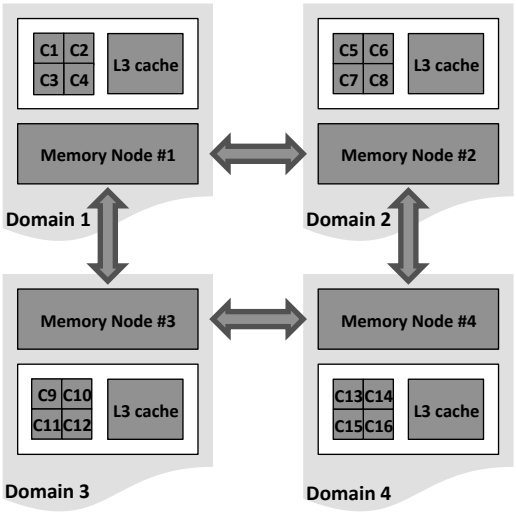


Figure 1: A schematic view of a cluster node with four memory domains and four cores per domain. There are 16 cores in total, and a shared L3 cache per domain.

(cluster network). It can occur when (a) cluster uses a file server to store the data for the cluster jobs, (b) several processes of the same job spread among cluster nodes would want to communicate their data between each other (cluster jobs are usually created using MPI, a Message Passing Interface, or other APIs that would allow their processes to exchange the data between each other, even if the processes are running on different machines).

- *Contention for a limited computational resource, most notably Floating-Point Unit (FPU).* This contention can occur on certain processor models (e.g., UltraSPARC T1), where there is only one Floating-Point Unit, shared between several computational cores.

The types of resources specified upon job submission 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, to the internode exchange of the data or to the shared use of an FPU). Because of that, the application may encounter shortage of actual computational resources allocated to it (e.g. cache space, memory controller bandwidth, internode interconnect bandwidth or the percentage of time on FPU), even though the resource requirements specified during the job submission (the number of nodes, cores per node, memory and so on) are perfectly met.

This will in turn result in *the incorrectly predicted execution time* for the contention-sensitive job. The probability of an incorrect prediction increases in HPC clusters, as they 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.

We propose to address this problem with a new set of *contention descriptive metrics* representing a fine-grained information about each job's resource utilization and communication patterns. It can be used both by the cluster scheduler to help it make scheduling decisions and by the cluster users to properly describe the jobs they submit and to estimate the slowdown due to cluster sharing.

The rest of this paper is organized as follows: Section 2 describes the experimental platform that we used, Sections 3, 4 and 5 each devoted to a specific Contention-Descriptive Metric that addresses one of the performance degrading factors outlined above. Section 6 concludes the paper with the discussion of possible ways the proposed metrics can be used, the description of our future steps and the feedback we would like to receive from the community.

2. EXPERIMENTAL PLATFORM

Table 1 shows the tools that we use for this work. We chose state-of-the-art cluster schedulers (Maui and Moab) to test our techniques with and several benchmarks, representative of a typical HPC cluster workload. Some of them are clarified in more detail below.

The High Energy Physics (HEP) SPEC benchmark is a set of test applications which stress the processor with operations and algorithms used commonly in applications from the physics community. HEP-SPEC is based on the SPEC CPU2006 benchmark suite. The reason why SPEC CPU2006 is not used as is, is because the percentage of FP tests in the SPEC CPU2006 does not match that of a typical HEP code. A subset of SPEC CPU2006 benchmarks called *all_cpp*, was proposed [1] as a representative HEP benchmark suite. All_cpp is a set of seven benchmarks, three (471.omnetpp, 473.astar, 483.xalancbmk) from the integer suite and four (444.namd, 447.dealII, 450.soplex, 453.povray) from the floating point suite, that gives a good match of the integer and FP instruction set. SPEC HEP is used as a representation of the CERN workload.

HPCC stands for High Performance Computing Challenge benchmark and is actually a suite of benchmarks that measure performance of the CPU, memory subsystem and interconnect. It consists of 7 benchmark tests packed into one application - HPL (High Performance LINPACK), DGEMM (Double-precision GEneral Matrix-Matrix multiply), STREAM, PTRANS (Parallel TRANSpose, Random Access, FFT (Fast Fourier Tranform) and communication bandwidth/latency. Before testing, HPCC should be configured to run on a particular cluster. We used Intel recommendations [6] when we changed the input parameters in the configuration file hpccinf.txt for this workload.

The NAS Parallel Benchmarks (NPB) is a set of benchmarks which was derived from computational fluid dynamics (CFD) applications targeting performance evaluation of highly parallel supercomputers. They are developed and maintained by the NASA Advanced Supercomputing (NAS). In our experiments, we use MPI implementation of NPB 3.3 with input type C.

We constructed the cluster from the following systems:

Dell-Poweredge-R805 (AMD Opteron 2350 Barcelona) has eight cores placed on two chips. Each chip has a 2MB 32-way 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 4 GB memory block, for a total of 8 GB main memory. The server was configured with a single 109 GB SCSI hard drive.

Dell-Poweredge-R905 (AMD Opteron 8435 Istanbul) has 24 cores placed on four chips. Each chip has a 5MB 48-way 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.

Dell-Poweredge-R905 (AMD Opteron 8356 Barcelona) has sixteen cores placed on four chips. Each chip has a 2MB 32-way 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 16 GB memory block, for a total of 64 GB main memory. The server was configured with a single 109 GB SCSI hard drive.

System aspect	*Tool(s) used*
Resource allocator and job scheduler	TORQUE PBS (TORQUE Portable Batch System, a job scheduler for UNIX clusters), Maui (Open Source, used in conjunction with PBS), Moab (proprietary). Maui and Moab are cluster schedulers for use on clusters and supercomputers. They are capable of supporting multiple scheduling policies, dynamic priorities, reservations, and fairshare capabilities.
Workload	SPEC MPI 2007 V2.0 SPEC High Energy Physics (HEP), representative of CERN workload SPEC CPU2006 HPC Challenge NAS Parallel Benchmarks 3.3 (MPI version) Intel MPI Benchmarks 3.2

Table 1: Tools used within this study.

Sun UltraSPARC T1 has eight cores with 4 thread contexts per core placed on one chip. The chip has a 3MB 12-way L3 cache shared by its four cores. Each core also has a private L1 instruction and data caches. It is a UMA system: the machine has a total of 32 GB main memory. The server was configured with a single 30 GB SCSI hard drive.

Since we are focused on CPU-bound workloads, which are *not* likely to run with more threads than cores [9, 11], we only evaluate the scenarios where the number of threads does not exceed the number of cores. If you schedule more processes to run than there are available cores, this is referred to as *oversubscribing*, which can result in performance degradation and hence is not recommended [7] (*mpirun* Linux tool, for example, has a special –nooversubscribe option which returns an error and does not execute the command if the number of processes requested is greater than the cores available on the cluster).

All systems were running Linux Gentoo 2.6.29 release 6.

3. MISSRATE: A METRIC OF CONTENTION FOR MEMORY HIERARCHY WITHIN CLUSTER NODE

Multiple studies investigated ways of reducing resource contention *within a mulicore machine* (a cluster node). One of the promising approaches that emerged recently is memory contention-aware scheduling [10, 13]. 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 [12] 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 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. We used it to design and implement a new scheduling algorithm called Distributed Intensity Online (DIO). DIO separates devils on different levels of memory hierarchy of the multicore system as far from each other as possible [10], 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).

We showed experimentally on two different multicore systems that DIO performs better than the default Linux scheduler, delivers much more stable execution times than the default scheduler, and performs within a few percentage points of the theoretical optimal. DIO dynamically reads miss counters and schedules applications in real time. DIO was implemented at user-level (just like a typical cluster scheduler), and although it could be easily implemented inside the kernel, the user-level implementation was sufficient for evaluation of this algorithm's key properties [10].

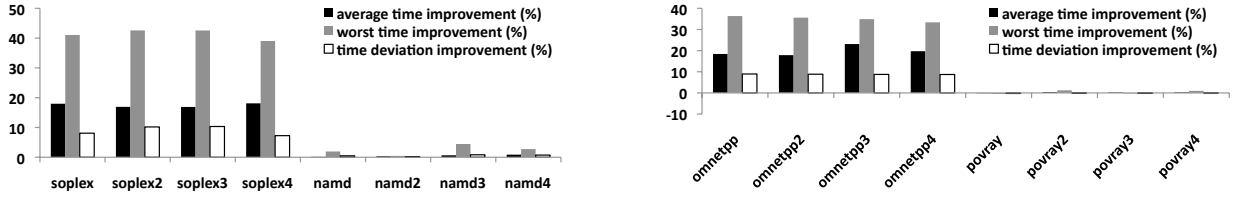
In the rest of this section we provide the experimental results of comparing performance under DIO to the default contention-unaware scheduler in Linux (referring to the latter as DEFAULT) *within the cluster environment*. The goal of these experiments is to show, that the LLC missrate metric retains its value for the MPI and HEP workloads, typical of an HPC cluster.

The prefetching hardware was fully enabled during these experiments. To account for the varied execution times of benchmark we restart an application as soon as it terminates (to ensure that the same workload is running at all times). An experiment terminates when the longest application had executed three times.

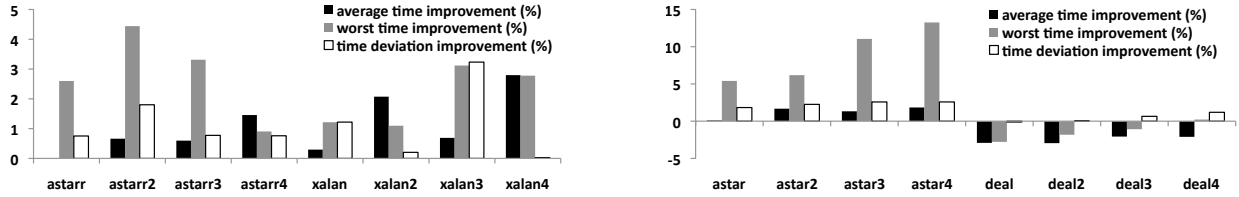
SPEC HEP. Figure 2 shows performance of DIO relative to the default Linux scheduler. The average, the worst-case performance improvement relative to the default scheduler, as well as reduction in the standard deviation of completion times are shown. The results demonstrate that DIO performs better than Default (by as much as 43% in some cases). Higher numbers are better in both cases. Worst-case performance improvement is obtained by comparing the worst-case performance (across all the runs) under the both algorithms. These metrics indicate that contention-aware algorithms provide more stable performance for the HEP workloads. While Figure 2 suggests that in all cases, DI-NUMA outperforms default, the benefit from contention-aware scheduling is the most noticeable when scheduling processes with highly distinctive memory access patterns. Workloads comprised of strong devils (high miss rate) and strong turtles (low miss rate) reap higher benefit from DIO (Figure 2(a)). The workloads comprised of semidevils (the applications whose memory intensiveness slips in between devils and turtles and so is hard to classify) (Figure 2(b)) do not leave a lot of room for improvement, so the benefits here are moderate.

It is important to keep in mind that HEP-SPEC is not a traditional clustered HPC application, and does not require communication or data coherency across cluster nodes. For this reason, clusters designed to maximize HEP-SPEC throughput may not be ideally suited for clustered HPC applications [5]. Hence, scheduling algorithms for general-purpose clusters should not be based solely on HEP-SPEC results, but should be supplemented with standard cluster benchmarks such as HPC Challenge, NAS, Intel MPI benchmarks or SPEC MPI.

HPCC and NAS. Figure 3 shows the performance benefit from using DIO with the workloads comprised of HPCC, HEP SPEC and NAS benchmarks on AMD machine with 8 cores. A process title with a pound sign '#' next to it means that the process was a part



(a) Relative performance improvement of the DIO over the DEFAULT for strong devils/turtles.



(b) Relative performance improvement of the DIO over the DEFAULT for semi devils.

Figure 2: Relative performance improvement of the DIO over the DEFAULT for HEP SPEC on AMD machine with 8 cores.

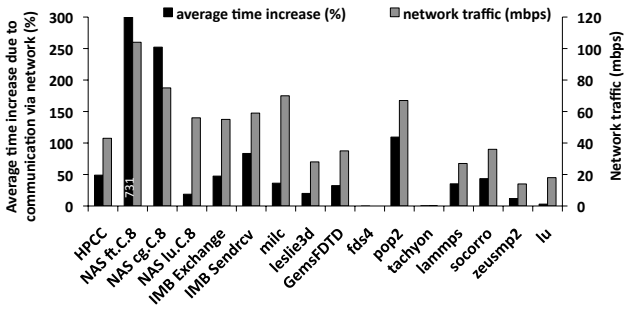


Figure 4: Average time increase and network traffic for the 8 process MPI jobs scheduled on 2 nodes (4 processes per node) relative to a schedule on one node. The bars represent the execution time decrease and the amount of traffic for the entire job. Correlation between performance degradation and traffic is 0.73.

of an MPI job. The results are generally similar to those for HEP SPEC alone showing that contention-aware scheduling algorithms can be beneficial for MPI and mixed workloads as well.

4. NETWORK TRAFFIC: A METRIC FOR DEGRADATION DUE TO ACCESSING CLUSTER INTERCONNECTS

A typical HPC cluster job can span several nodes within the cluster. In case the job processes that are running on different nodes would want to communicate between each other, they could do so by explicitly exchanging messages according to a specific protocol, defined at the program creation. The technology to allow cross-node communication in a cluster is usually MPI.

Figure 4 shows the degradation that MPI jobs from SPEC MPI2007 suite suffer when their processes are forced to communicate between each other using cluster interconnect. In these experiments, each MPI job is comprised of 8 processes. The execution time of the job was recorded for a setup where 8 processes were spread across 2 cluster nodes (4 processes on each node) and for a setup when all the processes were running on the same node. The execution time degradation (black bars) is given for the first setup relative to the second one. As can be seen, the slowdown varies greatly from job to job, but it can be as high as 778% for some MPI applications. The grey bars on the same graph show the average traffic that each of the nodes in the two-node setup send and receive in megabits per second (mbps) of a job execution. The traffic for the one-node setup was very close to zero, suggesting that the job processes were primarily exchanging data between each other, rather than with the NFS server on which job inputs were stored.

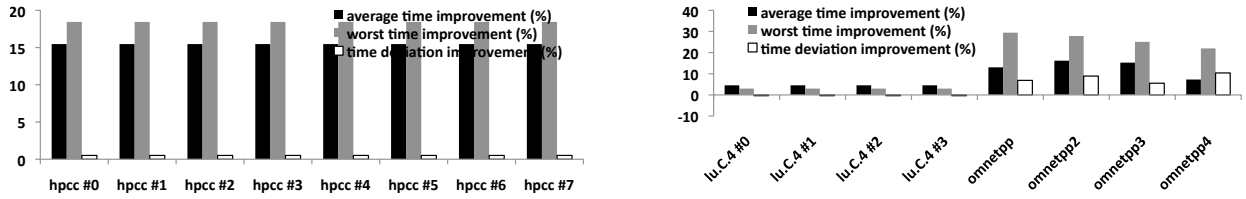
We used capstats [3] to measure traffic of each node. Capstats is a package that is shipped with Bro (a Unix-based Network intrusion detection system) [2] to measure the bandwidth used by a network connection.

The data on Figure 4 suggests a correlation between the degradation due to accessing of cluster interconnects and the amount of traffic that nodes exchange per second. Hence, the traffic measured in mbps in real time can be used as a contention descriptive metric.

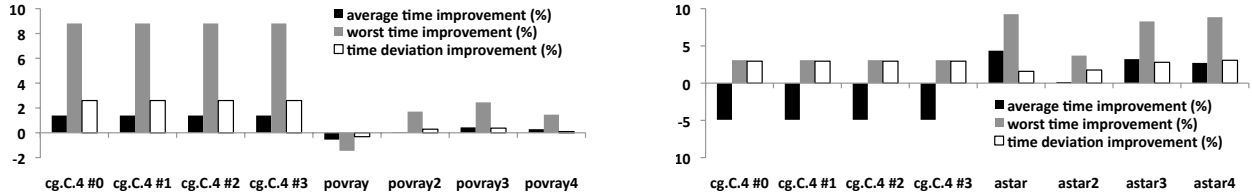
5. FP INSTRUCTION AMOUNT: A METRIC FOR DEGRADATION DUE TO FPU SHARING

The UltraSPARC T1 microprocessor with its 32 thread contexts spread among 8 cores was designed for network-facing high-demand servers, such as high-traffic web servers and mid-tier Java application servers, which often utilize a large number of separate threads. Small database applications (e.g. MySQL) which have a large thread count was also shown to perform well on it [8].

One of the limitations of the T1 design is that a single floating point unit is shared between all 8 cores, making the T1's floating point workload vulnerable for the performance degradation due to accessing of the shared FPU. Table 2 illustrates this phenomenon. Here we chose six benchmarks from SPEC CPU2006, three of those are from Integer category of the benchmark suite and the rest three are from FP category. The table presents the performance degradation of co-running an instance of application in the left vertical column of the table with the seven instances of an application shown in the upper row of the table. Thus, at any point during the experiments, eight programs were running on the system. Each instance run on its own T1 core, so that there were no sharing of core's computational resources through thread con-



(a) Relative performance improvement of the DIO over the DEFAULT for HPCC, NAS and HEP-SPEC for strong devils/turtles on AMD 8 cores.



(b) Relative performance improvement of the DIO over the DEFAULT for semi devil NAS and HEP SPEC workloads on AMD 8 cores.

Figure 3: Relative performance improvement of the DIO over the DEFAULT for the mixed workloads comprised of HPCC, HEP SPEC and NAS on AMD 8 cores.

texts. The benchmarks of different memory intensity (devils, turtles) were chosen from each category.

Table 2 suggests that the degradation of co-running an FP benchmark with a benchmark from an FP category results in much higher degradation than in the case where integer benchmarks were involved. So in order to predict, whether the program will experience contention due to FPU sharing, we could use the number of FP instructions in the program as a metric.

Sun provides a tool called *cooltst* for analysing an application's level of parallelism and use of floating point instructions to determine if it is suitable for use on a T1 platform [4]. Cooltst looks at the workload being executed by the system by all processes and bases its recommendations on two main criteria:

1. Percentage of instructions which are floating point. If the floating point content of the workload is high then the workload may not be suitable for an UltraSPARC T1 processor.

2. Parallelism. cooltst evaluates the degree of potential thread level parallelism, as measured by the spread of CPU consumption among software threads, and instruction level parallelism, as measured by the cycles per instruction (CPI). A highly parallel workload may well exploit the hardware parallelism of CMT processors to achieve high throughput. A workload with low parallelism may not.

6. CONCLUSION AND FUTURE WORK

The contention-descriptive metrics outlined above can be used in HPC cluster environment in the following ways:

- *Estimation of the job execution time by the user.* The metrics contain valuable information about the sensitivity of a particular job to the resource contention in an HPC cluster. If several metrics indicate that the job is not contention sensitive (e.g. profiling during a test run revealed that the job has a low missrate, close to zero traffic between its processes and low percentage of FP instructions), then the job will likely not experience the change in execution time when submitted on cluster, regardless of how its processes will be spread across cluster nodes or what jobs will run alongside with it.

- *Supplying cluster scheduler with the sensitivity profile of the job upon submission.* The metrics can be obtained by the user during a test run and then included in the resource list specified in the submission script together with the existing parameters of number of cores, memory, etc. This new information will help cluster scheduler to make better decisions on when and how schedule that job in the cluster. For example, if the job appears to be sensitive to accessing the cluster interconnect, as is suggested by the big network traffic value, the scheduler might take it into account and try to schedule the job on as few nodes as possible. The high percentage of FP instructions is a good reason for a scheduler to prevent the job submission on T1 nodes together with other FP jobs.

- *Online monitoring of cluster workload by the scheduler.* The mechanism of hardware performance counters, that exists on all major processor models, allows cluster scheduler to obtain the sensitivity metrics of the jobs online, as they execute. As a result, it can take into account dynamic change in application behaviour and react accordingly, delaying or migrating sensitive jobs away from each other.

Designing a cluster scheduler that will accommodate all the points outlined above is the main future course of this study. We are also working on improving the precision of our metrics. For example, LLC missrate can be potentially used with different parameters that describe the memory access patterns of the workload. The network traffic metric can be revised to distinguish between different jobs executing on the node[1]. Another direction of research would be testing the prediction accuracy of sending/receiving traffic separately and the sensitivity of the performance degradation to the phases in application execution and traffic intensity. While the amount of FP instructions can be detected offline with the tools like cooltst, it is currently not clear how to obtain this metric online, as hardware counters on T1 lack the information about floating point calculations in its workload. We are also working on identifying other sources of performance degradation in HPC clusters and on devising a descriptive metrics for them.

[1]Although our data obtained from an industry-size HPC cluster suggests that running several jobs on the same node is perhaps not as typical as we initially thought.

		Integer Benchmarks			FP Benchmarks		
		sjeng	*perlbench*	*omnetpp*	*povray*	*zeusmp*	*milc*
Int	sjeng	0.71	0.24	0	0.08	0.94	0.79
	perlbench	0	0	0	0	3.17	0
	omnetpp	0.20	0	0	8.55	3.67	13.24
FP	povray	0.36	0	0	**62.41**	**27.37**	**85.04**
	zeusmp	0	2.33	0	**46.51**	**13.95**	**55.81**
	milc	0.57	0.77	0.77	**78.35**	**34.87**	**82.76**

Table 2: Percentage of performance degradation due to FPU sharing for SPEC CPU2006 benchmarks on SPARC T1 machine. The cases when FP benchmarks were co-scheduled together resulting in increased performance degradation are highlighted in bold.

We would be very interested to hear expert opinions on our ideas and, possibly, a suggestions about the further direction in which our research should go.

7. REFERENCES

[1] Benchmark and market analysis of worker node for HEP farms. *[Online] Available: http://www.infn.it/CCR/server/.*

[2] Bro Quick Start Guide. *[Online] Available: http://www.bro-ids.org/Bro-quick-start.pdf.*

[3] Capstats: a quick hack to get some NIC statistics. *[Online] Available: http://www.icir.org/robin/capstats/.*

[4] CoolThreads Selection Tool. *[Online] Available: http://www.opensparc.net/sunsource/cooltools/www/cooltst/.*

[5] Designing research computing solutions for the CERN/ATLAS program. *[Online] Available: http://www.dell.com/downloads/global/power/ps4q09-20100175-Stemple.pdf.*

[6] Hpl application note. *[Online] Available: http://software.intel.com/en-us/articles/performance-tools-for-software-developers-hpl-application-note/.*

[7] Oversubscribing nodes. *[Online] Available: http://docs.sun.com/source/819-7480-11/ExecutingPrograms.html#50634758_48929.*

[8] UltraSPARC T1. *[Online] Available: http://en.wikipedia.org/wiki/UltraSPARC_T1.*

[9] D. an Mey, S. Sarholz, and C. Terboven et al. The RWTH Aachen SMP-Cluster User's Guide, Version 6.2. 2007.

[10] S. Blagodurov, S. Zhuravlev, and A. Fedorova. Contention-aware scheduling on multicore systems. *ACM Trans. Comput. Syst.*, 28:8:1–8:45, December 2010.

[11] R. van der Pas. The OMPlab on Sun Systems. In *Proc. of IWOMP'05*, 2005.

[12] Y. Xie and G. Loh. Dynamic Classification of Program Memory Behaviors in CMPs. In *CMP-MSI*, 2008.

[13] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing Contention on Multicore Processors via Scheduling. In *ASPLOS*, 2010.