Proceedings of the Joint Workshops on

Co-Scheduling of HPC Applications (COSH 2017)

and

Virtualization Solutions for High-Performance Computing (VisorHPC 2017)

Stockholm, Sweden, January 24, 2017
Co-located with HiPEAC 2017

Workshop Co-Chairs

Carsten Trinitis and Josef Weidendorfer (COSH 2017)
Carsten Clauss and Stefan Lankes (VisorHPC 2017)

ISBN: 978-3-00-055564-0
DOI: 10.14459/2017md1344297

Published in the TUM library (mediaTUM), January 2017

(This page intentionally left blank)

Contents

(This page intentionally left blank)

Foreword to the Proceedings of the Joint Workshops on

Co-Scheduling of HPC Applications (COSH 2017) and Virtualization Solutions for High-Performance Computing (VisorHPC 2017)

Co-located with HiPEAC 2017
in Stockholm, Sweden, January 24, 2017

These proceedings consist of the papers from the following joint workshops: the 1st Workshop on Virtualization Solutions for High-Performance Computing (VisorHPC) and the 2nd Workshop on Co-Scheduling of HPC Applications (COSH 2017), co-located with the HiPEAC 2017 conference in the wonderful city of Stockholm, Sweden. By merging the two workshops for this year and entering the stage as a single event at HiPEAC, we could present five paper presentations, two keynote talks, and a vivid live demo all around co-scheduling and virtualization in HPC.

Upon the separate calls for papers, the two workshops received seven submissions from five countries in total and each of these papers has been reviewed by at least three program committee members. In the end, we could select five high quality papers (three for COSH plus two for VisorHPC) for presentation and publication.

It was a great honor that Prof. Dhabaleswar K. (DK) Panda of the Ohio State University volunteered to deliver an invited keynote talk about opportunities and challenges in designing high-performance MPI and Big Data libraries on virtualized InfiniBand clusters. In addition, we very much appreciated a second invited talk by Dr. Mikko Byckling of Intel Corporation about recent hardware support and latest developments in microprocessor technology within the context of co-scheduling, virtualization and energy challenges for future exascale systems.

We would like to say a big thank you to the members of our program committees who did a great job in reviewing all submissions thoroughly and conscientiously. Furthermore, we also want to take the opportunity and give thanks the whole HiPEAC Conference Committee and especially to the Workshops & Tutorials

Chairs as well as to the on-site organization team for making this event possible and successful.

Stockholm was a wonderful location for the conference and its workshops and we hope that the discussions around co-scheduling and virtualization at the joint COSH and VisorHPC event were fruitful for the attending audience and resulted in new insights.

January 2017

Carsten Trinitis, Josef Weidendorfer
(COSH Co-Chairs)

Carsten Clauss, Stefan Lankes
(VisorHPC Co-Chairs)

COSH: Workshop Background and Topics

The task of a high performance computing system is to carry out its calculations (mainly scientific applications) with maximum performance and energy efficiency. Up until now, this goal could only be achieved by exclusively assigning an appropriate number of cores/nodes to parallel applications. As a consequence, applications had to be highly optimised in order to achieve even only a fraction of a supercomputer's peak performance which required huge efforts on the programmer side.
This problem is expected to become more serious on future exascale systems with millions of compute cores. Many of today's highly scalable applications will not be able to utilise an exascale system's extreme parallelism due to node specific limitations like e.g. I/O bandwidth. Therefore, to be able to efficiently use future supercomputers, it will be necessary to simultaneously run more than one application on a node. To be able to efficiently perform co-scheduling, applications must not slow down each other, i.e. candidates for co-scheduling could e.g. be a memory-bound and a compute bound application.
Within this context, it might also be necessary to dynamically migrate applications between nodes if e.g. a new application is scheduled to the system. In order to be able to monitor performance and energy efficiency during operation, additional sensors are required. These need to be correlated to running applications to deliver values for key performance indicators.

COSH Workshop Topics:

- co-scheduling concepts and challenges,
- modeling of performance and energy efficiency,
- application classification regarding resource demands,
- task migration for workload balancing,
- case studies in co-scheduling.

VisorHPC: Workshop Background and Topics

Although virtualization solutions are quite common in server farms and cloud computing environments, their usage is in contrast by far not prevalent in the domain of high-performance computing (HPC). This is because, at least up to now, virtualization solutions like the employment of virtual machines (VM) have proven to be too heavyweight to be acceptable in scalable HPC systems. However, future exascale systems, equipped with a much higher degree of computing power but also with a much larger amount of computing cores than today's HPC systems, will demand for resiliency and malleability features that may only be provided by increasing likewise the degree of virtualization within the systems. Moreover, recent advancements, e.g. concerning container-based virtualization or regarding hardware abstraction of high-performance interconnects, currently propel the idea of introducing more virtualization solutions even in the domain of HPC. Prominent examples for such added values stemming from virtualization and fostering resiliency, usability and malleability also in HPC systems are the possibility of live-migration and transparent checkpoint/restart, as well as the option to provide each user with an individual environment in terms of customized VM images.

VisorHPC Workshop Topics: All subjects concerning virtualization solutions in HPC, especially (but not limited to):

- Employment and management of virtual machines and/or software containers in HPC,
- Checkpointing and/or migration solutions for resiliency and malleability in HPC,
- Solutions for network and I/O virtualization regarding HPC interconnects,
- Parallel I/O and HPC storage solutions taking advantage from virtualization,
- Hypervisors and other virtualization solutions tailored for HPC systems,
- Virtualization solutions for dealing with heterogeneity in HPC environments,
- Extensions to resource managers and job schedulers with respect to virtualization,
- Adaptation of cloud-computing technologies to HPC environments,
- Operating system support for virtualization in HPC systems,
- Performance and power analysis of virtualized HPC systems,
- Debugging and/or profiling in virtual environments.

Technical Program Committees

COSH

Jens Breitbart, Bosch, DE
Carsten Clauss, ParTec Cluster Competence Center GmbH, DE
Georgios Goumas, National Technical University of Athens, GR
Stefan Lankes, RWTH Aachen University, DE
Thomas Lundqvist, University West, SE
Konstantinos Nikas, National Technical University of Athens, GR
Carsten Trinitis, Technische Universität München, DE
Josef Weidendorfer, Technische Universität München, DE
Andreas de Blanche, University West, SE

VisorHPC

Frank Bellosa, Karlsruhe Institute of Technology (KIT), DE
Carsten Clauss, ParTec Cluster Competence Center GmbH, DE
Stefan Lankes, RWTH Aachen University, DE
Julián Morillo Pozo, Barcelona Supercomputing Center (BSC), ES
Lena Oden, Argonne National Laboratory (ANL), US
Pablo Reble, Intel Corporation, US
Josh Simons, VMware, Inc., US
Josef Weidendorfer, Technische Universität München, DE

(This page intentionally left blank)

Abstracts of Invited Talks

HPC Meets Cloud: Opportunities and Challenges in Designing High-Performance MPI and Big Data Libraries on Virtualized InfiniBand Clusters

Dhabaleswar K. (DK) Panda, Ohio State University, US

Significant growth has been witnessed during the last few years in HPC clusters with multi-/many-core processors, accelerators, and high-performance interconnects (such as InfiniBand, Omni-Path, iWARP, and RoCE). To alleviate the cost burden, sharing HPC cluster resources to end users through virtualization for both scientific computing and Big Data processing is becoming more and more attractive. The recently introduced Single Root I/O Virtualization (SR-IOV) technique for InfiniBand and High Speed Ethernet provides native I/O virtualization capabilities and is changing the landscape of HPC virtualization. However, SR-IOV lacks locality-aware communication support, which leads to performance overheads for inter-VM communication even within the same host. In this talk, we will first present our recent studies done on MVAPICH2-Virt MPI library over virtualized SR-IOV-enabled InfiniBand clusters, which can fully take advantage of SR-IOV and IVShmem to deliver near-native performance for HPC applications under Standalone, OpenStack, and Containers environments. In the second part, we will present a framework for extending SLURM with virtualization-oriented capabilities, such as dynamic virtual machine creation with SR-IOV and IVShmem resources, to effectively run MPI jobs over virtualized InfiniBand clusters. Next, we will demonstrate how high-performance solutions can be designed to run Big Data applications (like Hadoop) in HPC cloud environments. Finally, we will share our experiences of running these designs on the Chameleon Cloud testbed.

DK Panda is a Professor and University Distinguished Scholar of Computer Science and Engineering at the Ohio State University. He has published over 400 papers in the area of high-end computing and networking. The MVAPICH2 (High Performance MPI and PGAS over InfiniBand, Omni-Path, iWARP and RoCE) libraries, designed and developed by his research group, are currently being used by more than 2,700 organizations worldwide (in 83 countries). More than 405,000 downloads of this software have taken place from the project's site. As of Nov'16, this software is empowering several InfiniBand clusters (including the 1st, 13th, 17th, and 40th ranked ones) in the TOP500 list. The RDMA packages for Apache Spark, Apache Hadoop, Apache HBase, and Memcached together with OSU HiBD benchmarks from his group are also publicly available. These libraries are currently being used by more than 200 organizations in 29 countries. More than 19,000 downloads of these libraries have taken place. He is an IEEE Fellow.

An Update on Intel Technologies for High Performance Computing
Mikko Byckling, Intel Corporation, Finland

We will give an update on technologies for high-performance computing (HPC) by Intel® and relate them to the context of co-scheduling and virtualization. Our main focus will be on the 4th generation Intel® Xeon® Processor® (co-denamed Broadwell) and the 2nd generation Intel® Xeon Phi™ processor (co-denamed Knights Landing). We will give some examples on how using such different processor architectures affects the time and energy to solution of real HPC applications.

D.Sc. Mikko Byckling is a senior application engineer at Intel. He is currently focusing on code modernization and optimization on the 2nd generation Intel® Xeon Phi™ processor. His professional interests include parallel algorithms, parallel programming, scientific software development and exascale computing.

Disallowing Same-program Co-schedules to Improve Efficiency in Quad-core Servers

Andreas de Blanche
Department of Engineering Science
University West, Sweden
andreas.de-blanche@hv.se

Thomas Lundqvist
Department of Engineering Science
University West, Sweden
thomas.lundqvist@hv.se

ABSTRACT

Programs running on different cores in a multicore server are often forced to share resources like off-chip memory, caches, I/O devices, etc. This resource sharing often leads to degraded performance, a slowdown, for the programs that share the resources. A job scheduler can improve performance by co-scheduling programs that use different resources on the same server. The most common approach to solve this co-scheduling problem has been to make job-schedulers resource aware, finding ways to characterize and quantify a program's resource usage. We have earlier suggested a simple, program and resource agnostic, scheme as a stepping stone to solving this problem: Avoid Terrible Twins, i.e., avoid co-schedules that contain several instances from the same program. This scheme showed promising results when applied to dual-core servers. In this paper, we extend the analysis and evaluation to also cover quad-core servers. We present a probabilistic model and empirical data that show that execution slowdowns get worse as the number of instances of the same program increases. Our scheduling simulations show that if all co-schedules containing multiple instances of the same program are removed, the average slowdown is decreased from 54% to 46% and that the worst case slowdown is decreased from 173% to 108%.

Keywords

Co-scheduling; Same Process; Scheduling; Allocation; Multicore; Slowdown; Cluster; Cloud

1. INTRODUCTION

For several years now multi-core processors have been the standard processor architecture used in everything from mobile phones to supercomputers. While multi-core processors harness an extraordinary computational capacity, they suffer from the fact that a large number of cores share all other resources, like an off-chip memory system, caches, or I/O devices. Many studies have identified the limited off-chip bandwidth in conjunction with the shared off-chip memory system, often referred to as the *memory wall* [1], as the potentially largest bottleneck resource.

In order to make use of the capacity of a multi-core processor several processes have to be co-scheduled on the same computer, i.e. several processes must simultaneously execute on different cores in the same multi-core chip. However, it is common knowledge that co-scheduling programs that have a high degree of resource usage, will have a negative impact on the performance of said programs. Still, we will, in most cases, increase the overall performance by co-scheduling programs since the overall execution time for all processes will typically be lower than if the processes were executed sequentially, one after the other. This is not always true, though, in [2] two co-scheduled programs experienced a super-linear slowdown due to memory traffic contention, i.e. the programs' execution times were more than doubled. Hence, co-scheduling processes, with no knowledge of how the co-scheduling will affect the programs, can lead to severe performance degradation.

The risk of severe performance degradation caused by improper co-scheduling has led to a limited use of co-scheduling. According to Breitbart, Weidendorfer and Trinitis [3] many large HPC centers mostly use co-scheduling for single core jobs and [4, 5] reports that the utilization rate of Mozilla's, VMWare's, Google's and Microsoft's datacenters are all below 50%. Hence, if programs can be co-scheduled using all cores, with an average slowdown of less than 50% we could capitalize on the poor utilization and either double the throughput or halve the number of servers, which would save both money and energy. Several studies have shown that this is possible [6, 7].

In [8] we proposed a simple scheme that leads to improved co-scheduling without the need for any prior knowledge of a program's resource usage. This approach is thus program agnostic and does not require any characterization or measurement activities to be performed since it is simply based on avoiding the co-scheduling of twins, i.e. two instances of the same program. This simple, Terrible Twins, scheme is based on two observations. The first one is that the performance can be improved not only by selecting the best ways, but also by avoiding the worst ways in which programs can be co-scheduled. The second observation is that co-schedules containing multiple instances of the same program are over represented among co-schedules with very low and very high resource usage. Later, in [9], the initial study was extended with a more in-depth analysis covering varying start times and empirical data from a second processor architecture. Both studies covered dual-core co-scheduling.

COSH-VisorHPC 2017 Jan 24, 2017, Stockholm, Sweden

ISBN 978-3-00-055564-0
DOI: 10.14459/2017md1344414

In this paper we extend the earlier work to quad-core co-scheduling. Using a probabilistic analysis and empirical data we show that the conclusions made for dual-core co-scheduling are valid also for quad-core co-scheduling: We should still avoid co-scheduling multiple instances of the same program, i.e., avoid twins, triplets, quads, etc.

To evaluate the performance degradation of quad-core co-scheduling we did two evaluations. First, we performed a full-factor experiment with the serial versions of the NAS parallel benchmark suite. That is, the software was executed in all possible co-scheduling combinations on a quad-core processor and the corresponding slowdown was recorded. Then, the full-factor results were used in a second evaluation that examined the cluster-wide scheduling impact.

Based on our experimental evaluation, we draw the following conclusions:

- Avoiding same-program co-schedules (twins, triplets and quads) is a viable scheme for improving the performance of four-way co-scheduling.
- Our results show that the average and worst-case slowdowns are much lower when twins, triplets, and quads are not co-scheduling.
- Schedules containing no same-program co-schedules are better than those containing twins, which are better than the triplets, which in turn are better than the quads. We find that as the number of same programs instances increases in a co-schedule, the average slowdown also increases and performance suffers.
- Only a small part, 0.5% of our simulated schedules contains no twins, triplets, or quads, indicating that this program agnostic scheme can be used to reduce the number of schedules for further optimization.

The rest of this paper is organized as follows. In Section 2 we first give a brief background on resource aware co-scheduling. Then, in Section 3, we revisit earlier probabilistic arguments and extend these arguments into the quad-core realm. This is followed, in Section 4, by a description of the hardware and software used in the experiment. Sections 5 and 6 present the experimental methodology and evaluation results before concluding the article with discussions and conclusions, in Sections 7 and 8, respectively.

2. RESOURCE AWARE CO-SCHEDULING

As stated earlier, when two or more programs are executing on the same server at the same time they share many of the server's resources, such as memory or disks. If several processes are executing on the same processor chip, they also, most likely, will share one or more cache memories. Resource sharing almost always degrades the performance of the co-scheduled processes. In order to improve performance, we must limit this degradation.

A large body of work has been done on increasing the performance of co-scheduled programs in the context of operating system scheduling. For example, early research by Stone et al. [10] proposed a cache partitioning scheme and Kim et al. [11] found that for a set of co-scheduled benchmarks the throughput increased by 15% when the access to the shared resource was made more fair. Furthermore, Snavely and Tullsen [12] suggested symbiotic co-scheduling as an approach to determine how processes should be co-scheduled on a processor with support for simultaneous multithreading (SMT). The idea behind symbiotic co-scheduling is to determine which processes have the most "compatible" resource demands and then co-schedule these processes. Eyeman and Eeckhout [13] showed that their probabilistic symbiosis approach achieved a 19% reduction in job turnaround time for a four thread SMT processor, without having to evaluate the processes before execution. These results mean that there is a great potential for methods to reduce the performance degradation due to resource contention.

This paper focuses on the placement strategies of a cluster, grid or cloud scheduler. The main issue for this kind of high level job-scheduler is to, before execution, determine on which computer a specific task should execute. Thus, it must determine which programs to co-schedule before the execution starts. Relocation of running processes is not an option in an HPC environment and even if possible, it would be a costly task. Making a good co-scheduling decision before the execution starts is preferred over having to relocate running processes due to excessive resource contention. Current research in the co-scheduling area is focused on developing techniques aimed at online or offline characterization of a program's resource requirements as well as creating taxonomies that can be used to classify programs based on their resource needs [14, 15]. After characterization, a cluster or cloud scheduler can then use this information to decide which programs to co-schedule in order to minimize the performance degradation that occur due to resource sharing. Some of the methods [16, 17] characterize both how aggressively a program uses the resource as well as how sensitive it is to resource competition. This is done by measuring how much pressure they put on, and how much they are slowed down by specifically tailored micro-benchmarks. Hence, the pressure they put on other processes is modeled separately from how much they themselves are affected by the resource sharing. The memory wall [1] has been identified as the largest cause of contention in modern multi-core systems and much research is focused on this area [18, 19, 20, 21].

2.1 The Terrible Twins Approach

Methods relying on characterization and categorization fall short when no knowledge of a program's resource use behavior can be derived. In our previous studies, [8] and [9], a simple scheme to improve co-scheduling is suggested which does not rely on a-priori knowledge of a program's behavior – it is program agnostic and requires no data collection, no instrumentation, and no logging or learning. The main idea of the scheme is to "*Avoid the Terrible Twins*", i.e. avoid co-scheduling several instances of the same program on the same computer. Two important observations are made: (1), when scheduling jobs in a large cluster or cloud environment, one type of bad co-schedules is when all co-scheduled programs utilize the same resource to a high degree. And (2), co-scheduling programs that does not use any shared resources should be considered equally bad, given that there are other programs that could have benefited from being co-scheduled with these programs. A program with no, or low, resource usage will never degrade the performance of other co-scheduled programs. For this reason, we should avoid terrible twins.

Terrible twins, i.e., two-core co-schedules consisting of two instances of the same program, are more likely to have a very high or very low degree of similar resource use compared to

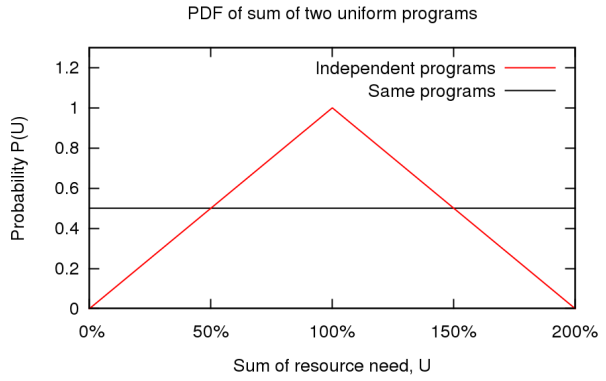


Figure 1: The probability density function (PDF) of the sum of the resource need of two co-scheduled programs.

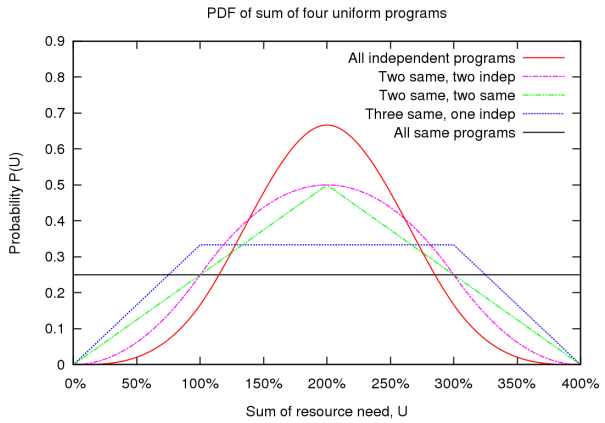


Figure 2: The probability density function (PDF) of the sum of the resource need of four co-scheduled programs where each program has a uniform resource need probability between 0% and 100%. Combining independent programs leads to a more centered PDF meaning that there is less risk for the total resource need to be very low or very high.

a co-schedule consisting of two independent programs. The simple reason behind this is that two instances of the same process use, or do not use, the same set of resources. In our previous study, we used a probabilistic argument to show this. When co-scheduling two or more jobs, the combined resource need is the sum of the individual programs' resource needs. For two programs P_1 and P_2, assuming a uniformly distributed random resource need u_1 and u_2, we get the following combined resource need U:

$$U = \begin{cases} u_i + u_j & \text{if } P_1 \text{ and } P_2 \text{ are independent} \\ 2u_i & \text{if } P_1 \text{ and } P_2 \text{ are the same} \end{cases}$$

This means that combining two independent programs results in a **uniform sum distribution** while combining two instances of the same program results in preserving the **uniform distribution**. This is illustrated in Figure 1 where we see that the sum of independent programs has the uniform **sum** distribution (red, pyramid-shaped line) and that the sum of two instances of the same programs has a uniform distribution (black, flat line). Thus, we can expect a higher concentration of same-process co-schedules in both the low and high ends of the spectrum.

For a more in-depth explanation we refer the reader to [10] and [11]. The next section expands upon this framework to show that, also for quad-core co-scheduling, the co-scheduling of several instances of the same program, would still result in a lower or higher aggregate resource need than when combining independent processes from different programs.

3. PROBABILISTIC ANALYSIS

We now do a probabilistic analysis of four-way co-scheduling. We assume that the resource need of a randomly picked program can be modeled as a random variable between 0% and 100%. For simplicity, we assume a uniform distribution. This means that when we have a program to co-schedule, it will have an equal probability of having a resource need somewhere between 0% and 100%. This could, for example, represent the use of a resource like the memory bus.

Now, we would like to explore what happens when we co-schedule programs. To do this, we simply look at the sum of the resource needs for the programs. We need to distinguish between two cases: (1) When combining instances of the same program, the resource need of each running process will be similar, and can be modeled as dependent random variables. In the other case, (2), when combining instances from different programs, the resource need will be independent and not correlated to each other. For four randomly picked programs, P_1, P_2, P_3, and P_4, with resource needs u_1, u_2, u_3, and u_4, we get the combined resource need U as:

$$U = \begin{cases} u_1 + u_2 + u_3 + u_4 & \text{if all } P \text{ is independent} \\ u_1 + u_2 + 2u_3 & \text{if } P_3 \text{ and } P_4 \text{ is same (twin)} \\ 2u_1 + 2u_3 & \text{if } P_1 \text{ and } P_2, \text{ is same and} \\ & P_3 \text{ and } P_4 \text{ is same (2 twins)} \\ u_1 + 3u_2 & \text{if } P_2, P_3, \text{ and } P_4 \text{ is same} \\ & \text{(triplet)} \\ 4u_1 & \text{if all } P \text{ is same (quad)} \end{cases}$$

This means that combining independent programs will result in a **uniform sum distribution** while combining four instances of the same program (the quads) will preserve the original **uniform distribution**. This is illustrated in Figure 2 where we see that the sum of four independent jobs has the uniform sum distribution (red, top-most line) and that four instances of the same programs (quads) has a uniform distribution (black, bottom-most line). The middle cases where two or three programs are the same (twins, double twins, and triplets) are seen in-between the lowest and highest curves. The more independent programs that are combined, the more centered around the middle the resource need becomes. Twins, triplets, and quads are more spread out and have a higher relative probability of ending up with a lower or higher aggregate resource need.

The conclusion one can make from this is that twins, triplets, and quads all behave similarly to what we earlier concluded when looking only at twins. All same-program co-schedules will have a higher possibility of a lower or higher resource use, meaning that we should avoid these co-schedules since, as we argued earlier, a high resource use is very of-

ten bad and as [8] showed, a low resource use might also be bad because of a potentially missed opportunity to combine these programs with higher resource users. Thus, avoiding same-program co-schedules should improve performance from a probabilistic point of view. In the next sections, we verify this probabilistic reasoning using experimental data in our full-factor evaluation.

4. EXPERIMENTAL SETUP

In this section, we give a brief overview of the hardware and software used in the experiments. The evaluation was carried out on a computer equipped with the Intel Ivy Bridge i5-3470 processor. The i5-3470 has four cores and a three-tier on-chip cache architecture with private level 1 and level 2 caches. The last level cache (level 3) is shared by all cores. Each computer was equipped with 8 GB of RAM with a bus speed of 5 GT/s.

The computer was running CentOs Linux version 7 and the workload used in all experiments was the ten benchmarks of the Numerical Aerospace Simulation (NAS) parallel benchmark suite (NPB) reference implementation [22] designed at NASA (input size C). The NPB benchmark suite is a collection of five kernels, three pseudo programs, and two computational fluid dynamics programs.

5. FULL-FACTOR EVALUATION

The purpose of the full-factor evaluation was to measure the actual slowdowns experienced when running the benchmark program on different cores in the same processor. First, to determine a baseline, the solo execution time of all ten NAS Parallel benchmarks (NPB) were measured by executing them alone on the Intel i5-3470 quad-core computer. Then, a full factor measurement, covering all possible co-scheduling combinations of the ten benchmarks were performed and the slowdown for all combinations were recorded. The measurements were performed using overlapping executions where four programs were started at the same time on different cores. The first program to finish was replaced with another instance of the same program so that the other programs are exposed to constant co-scheduling pressure, see Figure 3. This was repeated, until all programs performed at least one full execution.

The impact of not varying the start times of the NPB programs while using overlapping executions was evaluated in [9]. The evaluation showed that using synchronized start times gave, on average, 1.01% higher slowdown readings, on an i5 processor, then when using varying start times, indicating that the exact start times are of minor importance for the accuracy of the results.

5.1 Full-Factor Measurement Results

Figure 4 shows the slowdown distribution of all possible four-way co-schedules for the NPB programs. The slowdown of a four-way co-schedule is defined as the average of the four programs' individual slowdowns. As seen in the figure, the slowdowns of all co-schedules range from around 0% up to 260%, and the average slowdown of all co-schedules is 53%. Many co-scheduling combinations show low slowdowns and there are few really high ones. To examine the slowdown impact of the co-schedules containing multiple instances of the same program: twins, triplets, and quads, we divided the measurement results into different subsets.

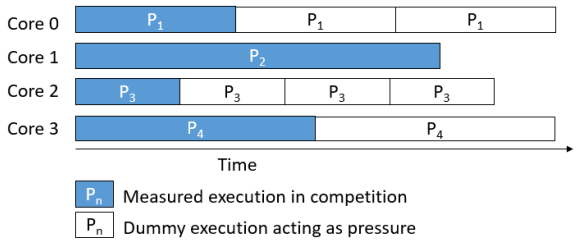


Figure 3: The methodology used when measuring the co-scheduling slowdown of programs P_1 to P_4. When the first instance of a program finishes, it is replaced with another instance of the same program. This procedure is repeated until the first instance of all programs have finished executing.

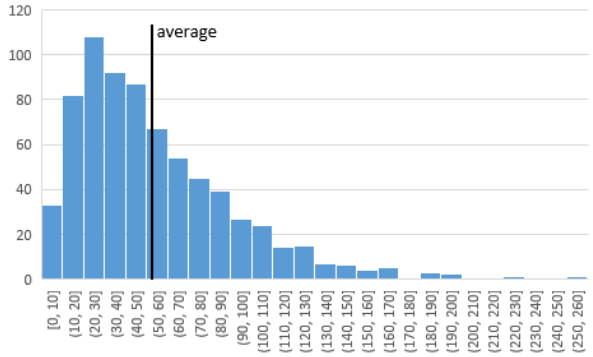


Figure 4: Histogram showing the measured slowdown distribution of all possible four-program co-schedules. The black line marks the average slowdown.

In Figure 5 and Table 1 the co-schedules have been divided into four subsets. The No-same subset consists only of co-schedules that include no more than only one instance of each program. The Twins subset consists of all co-schedules that contain two instances of the same program, and a few of these co-schedules are double twins, i.e., two same-program pairs. In the Triplets subset, we find all co-schedules where three instances of the same program are combined with one other program and finally in the Quads subset, all programs in a co-schedule are the same.

Table 1 shows the best, average, and worst slowdown for each subset. The same subsets have also been plotted in Figure 5 as a box-and-whiskers plot, which gives a high-level overview of the slowdown distribution of the different co-schedules. The ends of the whiskers (lines) represent the best and worst slowdowns, the boxes span percentile 10 to 90 (i.e. the grey box contains 80% of all co-schedules) and the line with a dot marks the median.

Turning to the values in Table 1 the average slowdown of the no-same subset is between 8.39 percentage points lower than the average of the set containing all possible schedules. Furthermore, it is between 8.89 and 32.70 percentage points lower than the subsets containing same-program co-schedules, e.g. twins, triplets and quads. The average slowdown is important because it is the slowdown that we, over time would converge towards, when executing a very large

	All [%]	No-Same [%]	Twins [%]	Triplets [%]	Quads [%]
Best	**3.71**	*6.94*	5.60	**3.71**	6.56
Average	53.64	**45.25**	54.14	68.29	*77.95*
Worst	*253.66*	**120.88**	194.62	253.66	224.68

Table 1: Co-schedules divided in subsets. The best values are in bold and the worst in italics.

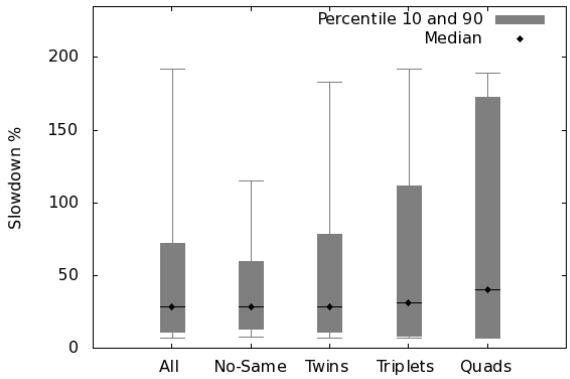


Figure 5: The filled box represents the co-schedules for each subset with slowdowns between the 10th and the 90th percentile. The dot and line in each box is the median slowdown and the end of the whiskers show the min and max slowdowns.

number of jobs from that category. This shows that avoiding same-program co-schedules clearly improves the average co-scheduling slowdowns.

An even more drastic improvement can be seen when looking at the highest slowdown values. The highest slowdown of the no-same schedule is 121%, which is 74 percentage points better than the subset containing twin co-schedules and 134 percentage points better than the subset containing triplets. The co-schedule with the overall lowest slowdown has a performance degradation of 3.71 percent. The subset with the highest minimum is the No-same subset which has a slowdown of 6.94%. The difference between the lowest slowdowns of the subsets is quite small compared to the huge difference in the maximum (worst) slowdowns. Thus, avoiding same-program co-schedules not only improves the average but drastically improves the worst co-scheduling slowdowns.

Although, the difference between the best and 10th percentile values in Figure 5 is quite small, it becomes obvious that the median slowdown increases as the number of same-program instances in the co-schedule increases. Worth noticing is that the worst value for the No-same subset is better than the 90th percentile of the subsets containing triplets and quads.

In general, the average, the median, the 90th percentile, and the worst values all increase as the number of same-program instances in the co-schedules increases. The one exception being the maximum value for the Triplets subset which is higher than the maximum of the Quads subset. However, the measurement data shows that the worst value in the Quads subset has the second highest slowdown overall.

To conclude, our measurements show that as the degree of same-program co-scheduling increases the potential slowdown also increases. Having no same-process co-schedules is better than twins, which are better than triplets which are better than quads. Hence, avoiding same-program co-schedules (twins, triplets and quads) is a viable scheme to identify co-schedules with a lower average and worst-case slowdown.

	All	No-Same	Twin	Triplet	Quad
# billion schedules	351 100%	1.6 0.5%	200 57.0%	140 39.8%	10 2.8%
Min	**10.1**	10.9	**10.1**	10.7	*12.1*
Average	53.5	**45.7**	51.6	56.0	*58.9*
Max	*172.7*	**107.9**	157.2	164.9	*172.7*

Table 2: Comparison between the set of all schedules and the subsets that contain no-same, twins, triplets, or quads co-schedules. The best values are in bold and the worst in italics. The No-same subset clearly outperforms the other sets.

6. IMPACT ON SCHEDULING

Recall that the goal of co-scheduling, in a cluster or cloud environment, is to schedule jobs on servers in such a way that the overall slowdown is kept as low as possible, i.e. increasing the efficiency of the system. As explained in Section 2, the approach of avoiding same-program co-schedules is program agnostic and does not require any knowledge of the program before allocating it to a server. To evaluate its impact on the overall cluster or cloud performance we constructed and simulated a job-scheduling scenario. The simulated system consisted of five computers equipped with one quad-core Intel i5-3470 each, for a total of 20 cores. As input to the simulation we used the full-factor measurement data presented in Section 5.

We simulated all possible ways in which 40 program instances could be scheduled, and co-scheduled on the nodes. Since there were 40 program instances and 20 cores, only half of the instances were used in each schedule. This amounted to a total of 351 billion different schedules, i.e. ways to schedule a program to each core using between zero and four instances of each benchmark program. The simulation results are summarized in Table 2 and Figure 6.

In Table 2 we can see that only 0.5% of the schedules contained no same-process co-schedules. The No-same process co-schedule set is hardly even visible at the bottom left part of Figure 6. While the No-same set is very small the Quads set makes up 2.8% of all schedules and has the, by far, worst performance of all sets shown in Table 2. Not surprisingly, the No-same-process set has the lowest average slowdown of all sets, 45.7%. Hence, we can conclude that removing all same-process co-schedules (all twins, triplets and quads) will decrease the average slowdown by 7.8 percentage points.

Turning to the average slowdown of the sets containing twins, triplets, and quads we can see that the average slowdown increases as the degree of the same-program co-schedules increases. This is not surprising since we saw the same pattern in Section 5 when looking at the slowdown of the individual co-schedules. However, the average slowdown for the Twins set is 5.9 percentage points higher than the No-same set and it is also 1.9 percentage points lower than the set of all schedules. Hence, removing only the schedules

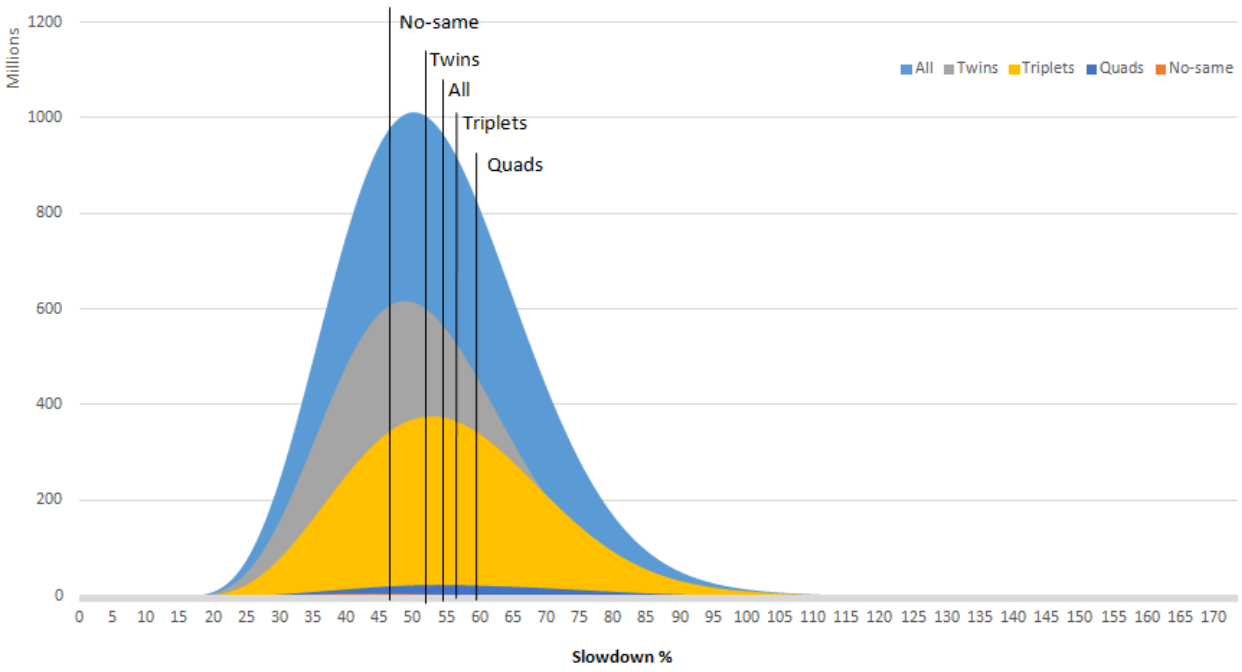


Figure 6: A histogram of all 351 billion evaluated schedules as well as the same subsets as in Table 2. The black lines mark the average slowdown of each subset. The bin size is 0.1.

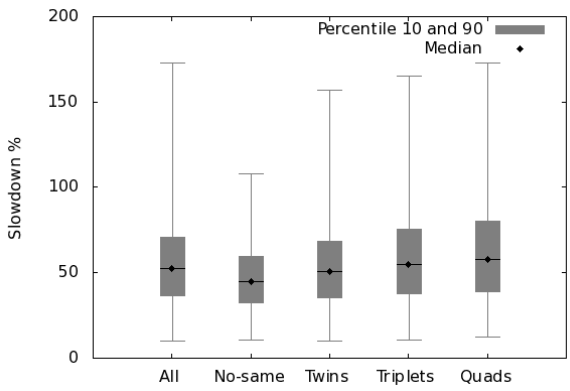


Figure 7: A box and whiskers plot of the same subsets as in Table 2. The filled box covers the 10th to the 90th percentile. The dot and line in each box is the median value and the end of the whiskers the min and max slowdowns.

containing triplets and quads (keeping No-same and Twins) will improve the performance by two percentage points while lowering the worst-case slowdown from 172.7 to 157.2.

Figure 7 shows the scheduling set data in a box and whiskers plot. The ends of the whiskers (lines) represent the best and worst slowdowns, the boxes span the 10th to the 90th percentile and the dot marks the median value for the set. In this plot, it becomes quite obvious that a greater number of same-process co-schedules leads to higher performance degradation. All indicators, except the minimum slowdowns, get worse as the number of same-program instances in a co-schedule increases. For example, when looking at the 10th percentile the No-same program schedule set is 2.9 percentage points better than the twins set, 5 percentage points better than the Triplets set, and 6.5 percentage points better than the Quads set.

Nevertheless, the most notable differences are found when looking at the maximum slowdown. The No-same set has a maximum slowdown of 107.9% and it increases to 157.2%, 164.9%, and 172.7% for the other three sets. Hence, removing all same-process co-schedules (Twins, Triplets and Quads) will decrease the worst case slowdown by at least 50 percentage points.

7. CONCLUSION

In this paper we extend the dual-core based Terrible Twins scheme [8] with a probabilistic analysis and experimental measurements that cover quad-core co-scheduling. The results from the analysis and evaluation reestablish the dual-core findings also for quad-core co-scheduling. Thus, it is possible to decrease the performance degradation caused by resource contention without any knowledge of a program's resource usage profiles and without performing any measurements or instrumentation whatsoever.

A full factor experiment with the serial versions of the NAS parallel benchmark suite was performed where all possible quad-core co-schedules were executed and the slowdowns recorded. As predicted by the probabilistic model, the co-scheduling sets that contain several instances of the same program were overrepresented among the co-schedules with the lowest and highest slowdowns, i.e., their 10th and 90th percentiles were lower and higher than those of the no-same process co-schedule set. And as determined in [8] both extremely low-, and high-slowdown co-schedules have a negative impact on the overall job-scheduling slowdowns of a cluster or cloud system.

To evaluate if same-process co-schedules (twins, triplets and quads) have a negative impact on job-scheduling performance we simulated a job-scheduling scenario based on the slowdown measurements obtained during the full factor experiments. Our scheduling simulations show that the average slowdown is decreased from 54% to 46% and that the worst case slowdown is decreased from 173% to 108% if all co-schedules containing several instances of the same program are removed. Furthermore, we found that only a small part, 0.5% of our simulated schedules contains no twins, triplets, or quads, indicating that this program agnostic scheme can be used to reduce the number of schedules to consider as basis for further optimization.

In conclusion, we find that the program and resource agnostic approach of avoiding same-program co-schedules (twins, triplets and quads) is a viable scheme for improving the performance of quad-core co-scheduling. Further studies are motivated to examine if the scheme can be extended and generalized to cover any number of cores. Also, if it can be applied to programs with parallel threads and to evaluate other workloads and scheduling scenarios as well.

8. REFERENCES

[1] W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious In *SIGARCH Computer Architecture News - Volume 23*, pages 20–24, New york, NY, USA, 1995.

[2] E. Koukis and N. Koziris. Memory and network bandwidth aware scheduling of multiprogrammed workloads on clusters of smps. In *International Conference on Parallel and Distributed Systems - Volume 1*, pages 345–354, Washington, DC, USA, 2006. IEEE Computer Society.

[3] J. Breitbart, J. Weidendorfer, and C.R. Trinitis. Automatic Co-scheduling based on Main Memory Bandwidth Usage. In *Proceedings of the 20th Workshop on Job Scheduling Strategies for Parallel Processing*, Chicago, US, 2016.

[4] R. McMillian. Data center servers suck - but nobody knows how much. In *Wired magazine, www.wired.com/2012/10/data-center-servers*, October, 2012.

[5] H. Yang, A. Breslow, J. Mars, and L. Tang. Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In *SIGARCH Computer Architecture News*, June, ACM, New York, NY, USA, 2013.

[6] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *ISCA '11: Proceeding of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 283–294, New York, NY, USA, 2011. ACM.

[7] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ASPLOS on Architectural support for programming languages and operating systems*, pages 129–142, New York, NY, USA, 2010. ACM.

[8] A. de Blanche and T. Lundqvist. Terrible Twins: A Simple Scheme to Avoid Bad Co-Schedules. *1st COSH Workshop on Co-Scheduling of HPC Applications, HiPEAC*, Prague, January, 2016.

[9] A. de Blanche and T. Lundqvist. Initial Formulation of why Disallowing Same Program Co-schedules Improves Performance. *In Co-Scheduling of HPC Applications, book chapter, ed. Trinitis, C. and Weidendorfer, J., Advances in Parallel Computing - Volume 28*, IOS Press, 2017.

[10] H. S. Stone, J. Turek, and J. L. Wolf. Optimal partitioning of cache memory. In *In IEEE Transactions on Computers. 14, 9*, Sep, 1992.

[11] S. Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, PACT '04*, 2004.

[12] A. Snavely and D.M. Tullsen Symbiotic jobscheduling for a simultaneous multithreaded processor. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems (ASPLOS IX)*. ACM, New York, NY, USA, 234-244, December 2000.

[13] S. Eyerman and L. Eeckhout. Probabilistic Modeling for Job Symbiosis Scheduling on SMT Processors. In *ACM Transactions on Architecture and Code Optimizations (TACO)*, Vol 9, No 2, June 2012

[14] Y. Xie and G. Loh. Dynamic classification of program memory behaviors in CMPs. In *2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2008.

[15] A-H. Haritatos, K. Nikas, G. Goumas, and N. Koziris. A resource-centric Application Classification Approach. *1st COSH Workshop on Co-Scheduling of HPC Applications, HiPEAC*, Prague, Jan, 2016.

[16] A. de Blanche and T. Lundqvist. Addressing characterization methods for memory contention aware co-scheduling. *The Journal of Supercomputing*, 71(4):1451–1483, 2015.

[17] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *MICRO '11: Proceedings of The 44th Annual IEEE/ACM International Symposium on Microarchitecture*, New York, NY, USA, 2011. ACM.

[18] J. Weidendorfer and J. Breitbart. Detailed Characterization of HPC Applications for Co-Scheduling. *1st COSH Workshop on Co-Scheduling of HPC Applications, HiPEAC*, Prague, Jan, 2016.

[19] A. Fedorova, S. Blagodurov, and S. Zhuravlev. Managing contention for shared resources on multicore processors. *Commun. ACM*, 53(2):49–57, Feb. 2010.

[20] A. de Blanche and S. Mankefors-Christiernin. Method for experimental measurement of an applications memory bus usage. In *International Conference on Parallel and Distributed Processing Techniques and Applications*. CRSEA, July 2010.

[21] A. de Blanche and T. Lundqvist. A methodology for estimating co-scheduling slowdowns due to memory bus contention on multicore nodes. In *International Conference on Parallel and Distributed Computing and Networks*, February 2014.

[22] NASA. NAS parallel benchmarks, 2013. NASA Advanced Supercomputing Division Publications.

(This page intentionally left blank)

Improving QoS and Utilisation in modern multi-core servers with Dynamic Cache Partitioning

Ioannis Papadakis
School of ECE, NTUA
ypap@cslab.ece.ntua.gr

Konstantinos Nikas
School of ECE, NTUA
knikas@cslab.ece.ntua.gr

Vasileios Karakostas
School of ECE, NTUA
vkarakos@cslab.ece.ntua.gr

Georgios Goumas
School of ECE, NTUA
goumas@cslab.ece.ntua.gr

Nectarios Koziris
School of ECE, NTUA
nkoziris@cslab.ece.ntua.gr

ABSTRACT

Co-execution of multiple workloads in modern multi-core servers may create severe performance degradation and unpredictable execution behavior, impacting significantly their Quality of Service (QoS) levels. To safeguard the QoS levels of high priority workloads, current resource allocation policies are quite conservative, disallowing their co-execution with low priority ones, creating a wasteful tradeoff between QoS and aggregated system throughput. In this paper we utilise the cache monitoring and allocation facilities provided by modern processors and implement a dynamic cache partitioning scheme, where high-priority workloads are monitored and allocated the amount of shared cache that they actually need. This way, we are able to simultaneously maintain their QoS very close to the levels of full cache allocation and boost the system's throughput by allocating the surplus cache space to co-executing, low-priority applications.

Keywords

Co-scheduling; Cache partitioning; Cache management; QoS

1. INTRODUCTION

Multi-core systems have become the norm for high performance servers that are widely used in both HPC and cloud datacenters. These systems encapsulate several cores that share critical resources, such as cache space and memory bandwidth. When applications are executed simultaneously on such a system, contention for shared resources impacts the quality of service (QoS) and can lead to performance degradation.

To address this problem, researchers have mainly developed two orthogonal approaches. The first one extends the schedulers operating at different levels, from inside a single server [3, 5, 11, 12, 18–20, 22, 27, 30] up to a supercomputer [6], datacenter or cloud environment [7–9, 21, 29], to deal with contention. These contention-aware schedulers typically classify applications based on their utilisation of one or more of the shared resources. The classification is then used to identify co-schedules that mitigate contention and maximise the overall throughput or maintain performance fairness. The second approach investigates ways to effectively partition the shared resources among the concurrently running applications [10, 14, 16, 17, 23–26, 28]. Most of these works focus on the shared cache space and attempt to manage its allocation to the applications in an attempt to isolate or optimise their performance.

Nevertheless, until today, supercomputers and cloud datacenters do not employ any of the above solutions. Instead, in order to guarantee QoS, they typically do not allow co-execution of applications on a multi-core server, leaving a subset of cores idle to avoid any interference [6, 8, 29].

Intel® has recently developed and released as part of its latest Xeon® processors that power server platforms, the Intel® Resource Director Technology (RDT) [2], a hardware framework to monitor and manage shared resources. The goal of our work is to leverage this technology to provide a mechanism that alleviates the effects of contention, making the co-execution of applications a viable choice even when QoS needs to be guaranteed.

To this end, we implement a dynamic cache partitioning scheme that monitors the throughput of a high-priority application and allocates to it the amount of shared cache that it actually needs, safeguarding its QoS as the performance penalty compared to full cache allocation is minimised to 5% on average. At the same time, the cache surplus is allocated to the co-executing low-priority applications, boosting their performance by a factor of 5× compared to when the cache is fully assigned to the high-priority workload, thus increasing the aggregated system throughput.

2. BACKGROUND & MOTIVATION

2.1 Architectural Support

Intel® RDT provides the necessary hardware support to monitor and manage the last level cache (LLC) and the memory bandwidth [15]. This is achieved through four technologies:

- Cache Monitoring Technology (CMT), which monitors the usage of the shared LLC.
- Cache Allocation Technology (CAT), which enables the management of the distribution of the shared LLC

COSH-VisorHPC 2017 Jan 24, 2017, Stockholm, Sweden

ISBN 978-3-00-055564-0
DOI: 10.14459/2017md1344298

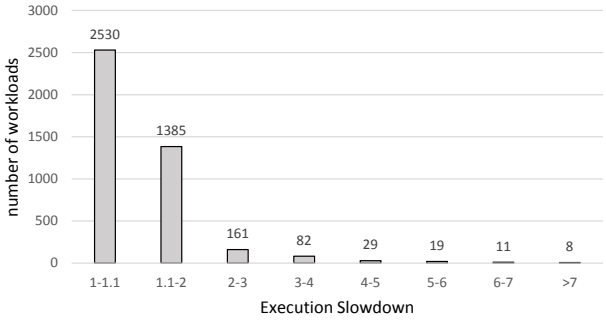


Figure 1: Distribution of HP application's slowdown when running together with 21 LP applications.

among the concurrently running applications. The allocated cache areas can overlap or be isolated.

- Memory Bandwidth Monitoring (MBM), which monitors the usage of the memory links.
- Code and Data Prioritization (CDP), which enables separate control over code and data placement in the LLC.

All these technologies can be programmed and utilised via Model Specific Registers (MSR) on a hardware thread basis. Therefore, they can be used in all modern Operating Systems, which provide APIs to read and write the MSRs.

2.2 Motivation

To guarantee QoS, supercomputers and datacenters typically avoid placing multiple applications on the same multicore server [6, 8, 29]. To showcase the problems that arise when attempting to utilise all the processing cores of a server, we execute multiprogrammed workloads on an Intel Xeon E5-2699 v4 server. The system is configured with a 2.2 GHz processor with 22 cores, SMT disabled, 55MB L3 cache (LLC), and 64GB of memory.

We employ 11 benchmarks from the Parsec 3.0 benchmark suite [4] (serial versions) and 28 benchmarks from the SPEC CPU 2006 suite [13], 9 of which can be used with multiple inputs, bringing the total number of applications to 65. Each multiprogrammed workload is created by nominating one of the 65 benchmarks as a "High-Priority" application (HP) and placing it on one of the cores of the server. Then another benchmark is selected as the "Low-Priority" application (LP) and 21 instances of it are placed on the remaining cores of the system. Every time an LP instance finishes, we restart it to make sure that the system is always full until HP finishes its execution.

We execute all the 4,225 workloads and compare the execution time of the HP application to when it is executed alone on the system. As shown in Figure 1, in around 60% of the workloads, HP suffers at most 10% slowdown, while in the remaining 40% there is a significant impact on the QoS. In around 33% of the cases, HP's execution time can double compared to when running alone, while in 310 workloads co-execution thrashes HP, causing its execution time to increase even more than 8 times in some cases.

It is evident that in order to increase the utilisation of the platforms while guaranteeing QoS, we must diminish the impact of co-execution due to resource contention.

3. CACHE-BASED QoS

The CAT technology has been shown to be able to restore the performance of an HP application in multiprogrammed workloads by reserving for that application a percentage of the shared LLC while restraining all other applications in the remaining cache space [15]. However, to leverage this technology on a multi-core platform of a supercomputer or a datacenter, one would need to know the exact amount of cache space that the HP application requires to be isolated from the others.

This amount depends on the behaviour of the actual HP application as well as on the behaviour of its co-runners. Therefore, the system would need to know at any given moment which applications are running, how they behave and how much space HP requires in the currently executed workload.

Alternatively, a system could conservatively limit all the LP applications in the minimum possible cache space, i.e., a single way of the set-associative LLC, reserving the rest of the cache space for the HP application. We refer to this conservative, static allocation as "Cache-Takeover QoS" (CT-QoS). CT-QoS is expected to provide the best possible performance for HP, at the expense however of the LP applications. At the same time, there is a high chance that HP does not utilise all the allocated cache space and it could have achieved similar performance with less cache. Furthermore, the cache requirements of an application typically vary during its execution, making the static assignment of cache space for the whole execution of the application a poor choice.

In this paper we propose a mechanism that dynamically adapts the HP application's LLC allocation, trying to match its cache requirements at any moment of its execution. We refer to this approach as "Dynamic Cache-Partitioning QoS" (DCP-QoS). DCP-QoS attempts to guarantee similar performance for the HP application to that achieved by CT-QoS, while at the same time allowing more cache space to be used by the LP applications. This extra space allocated to LPs, will enable them to achieve higher performance compared to CT-QoS, thus increasing the aggregated throughput of the system.

3.1 DCP-QoS

The flow chart of our scheme is presented in Figure 2. DCP-QoS makes initially the same conservative choice as CT-QoS, i.e., all the LP applications are restrained within the minimum possible cache space. Assuming a system with an N-way associative LLC, DCP-QoS allocates $N-1$ ways to HP and only 1 way to LPs. It then waits for HP to reach a steady state before attempting any changes.

Once our scheme detects that the performance of HP is stable, it starts gradually reducing its allocation by one way, which gets assigned to the LP applications. When the cache space has been reduced to a point where HP's performance is no longer stable, our mechanism increases HP's allocation by a single way. If this does not make the performance stable again, our mechanism resets, i.e., allows HP to takeover the LLC and limits LPs to a single way. If it does, then we assume that we have reached a "Balanced state", where the HP application has enough cache space allocated to enable it to achieve similar performance to when running with $N-1$

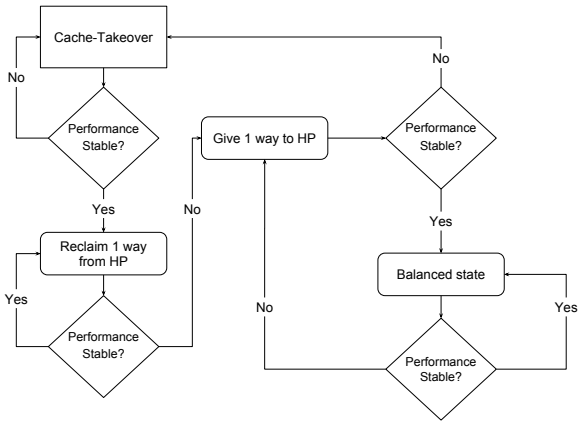


Figure 2: Flowchart of the proposed DCP-QoS mechanism.

ways.

While in the Balanced state, our mechanism continues to monitor HP's performance. If it detects a change, then it increases HP's allocation by a single way and checks whether the extra space was enough to make the performance stable again. If not, our mechanism resets and HP receives again $N-1$ ways.

3.1.1 Determining performance stability

To determine whether the performance of HP is stable or not, we define a monitoring period with length T and measure the IPC HP obtained during that time. As IPC is expected to fluctuate between two monitoring periods, we define that IPC_{t_1} and IPC_{t_2} are considered equal if :

$$(1-a) * IPC_{t_1} \leq IPC_{t_2} \leq (1+a) * IPC_{t_1} \qquad (1)$$

At the end of each time interval i, we compare HP's IPC, IPC_i, with its IPC during the previous two periods, IPC_{i-1} and IPC_{i-2}. If:

- $IPC_{i-1} = IPC_{i-2}$ and $IPC_i = IPC_{i-1}$, then the performance is considered stable.
- $IPC_{i-1} = IPC_{i-2}$ and $IPC_i \neq IPC_{i-1}$, then the performance is considered not stable.
- $IPC_{i-1} \neq IPC_{i-2}$ and $IPC_i \neq IPC_{i-2}$, then the performance was found to be not stable at the end of the previous monitoring interval. This caused our mechanism to allocate one more cache way to the HP application, without however succeeding in restoring the performance. Therefore, the performance is determined to be unstable again, leading our mechanism to reset.
- $IPC_{i-1} \neq IPC_{i-2}$ and $IPC_i = IPC_{i-2}$, then the performance that was found to be unstable before has now been restored again to its previous levels. Therefore, we consider the performance to be stable and our mechanism has reached the Balanced state.

As is evident by Equation 1, our mechanism can deduce that performance is not stable and reset, even when HP's IPC has increased. This design choice is justified by the fact that a significant change in the IPC, regardless of whether it is increased or not, could imply a phase change of the application. Therefore, our mechanism resets and starts looking for the new Balanced state from the beginning. Note that different metrics could also be used for driving the mechanism, such as LLC misses; we leave the exploration of such options for future work.

3.1.2 Cache Utilization Monitoring

We further augment DCP-QoS to take into consideration also the cache metrics offered by the Intel technologies. During each monitoring period T, using CMT and MBM, we acquire n samples of how much space HP occupies in the LLC and how much memory bandwidth it uses, recording only the maximum cache occupancy and the maximum bandwidth utilisation.

At the end of T, once the stability of the performance has been determined and a decision regarding HP's allocation has been made, our mechanism compares the recorded utilisation of the memory channel by HP to a predefined threshold. If the utilisation is higher than the threshold, we can speculate that HP tends to fully utilise its allocated space and proceed with modifying its allocation based on the performance stability, as depicted in Figure 2.

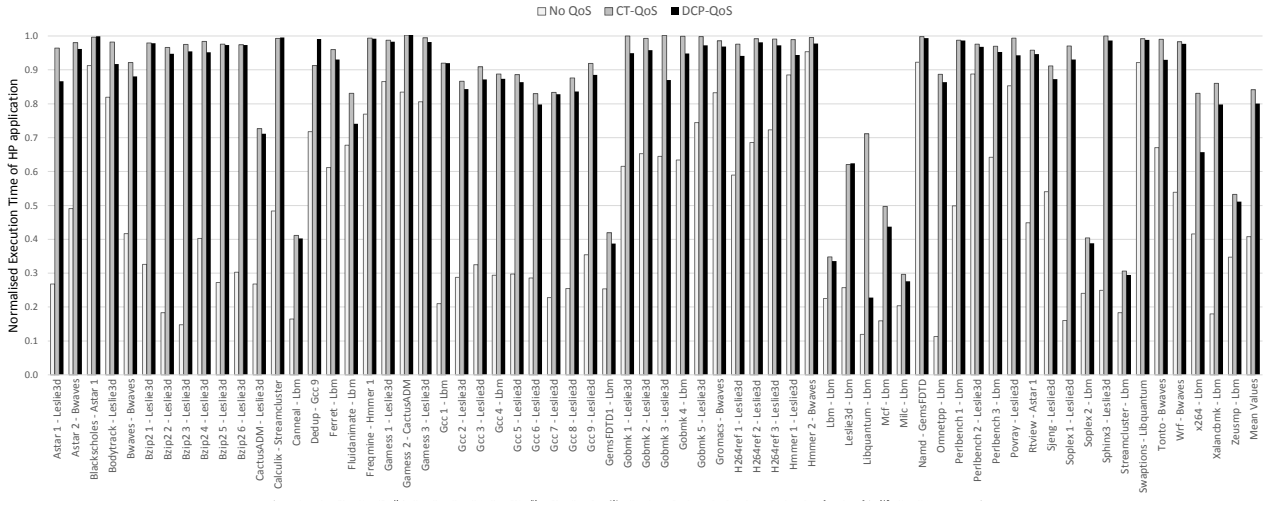
If, however, the utilisation is below the predefined threshold, we can deduce that HP is content with its LLC allocation, as it does not suffer many LLC misses. At this point, we compare the recorded maximum LLC occupancy to the cache space allocated to HP by our mechanism. If HP is found to occupy less space than allocated, then the redundant cache ways are removed from HP and assigned to LPs. That way, we are able to reduce HP's allocation more aggressively than by removing just one cache way at the end of every monitoring period.

Finally, we use the memory bandwidth utilisation to avoid unnecessary resets while in the Balanced state. Specifically, if our system has reached the Balanced state and the performance is found unstable, before resetting, we look at the recorded memory bandwidth measurement of the last monitoring period. If it is found to be less than the predefined threshold, then we speculate that HP will not utilise the extra space provided by a reset and choose to remain in the Balanced state.

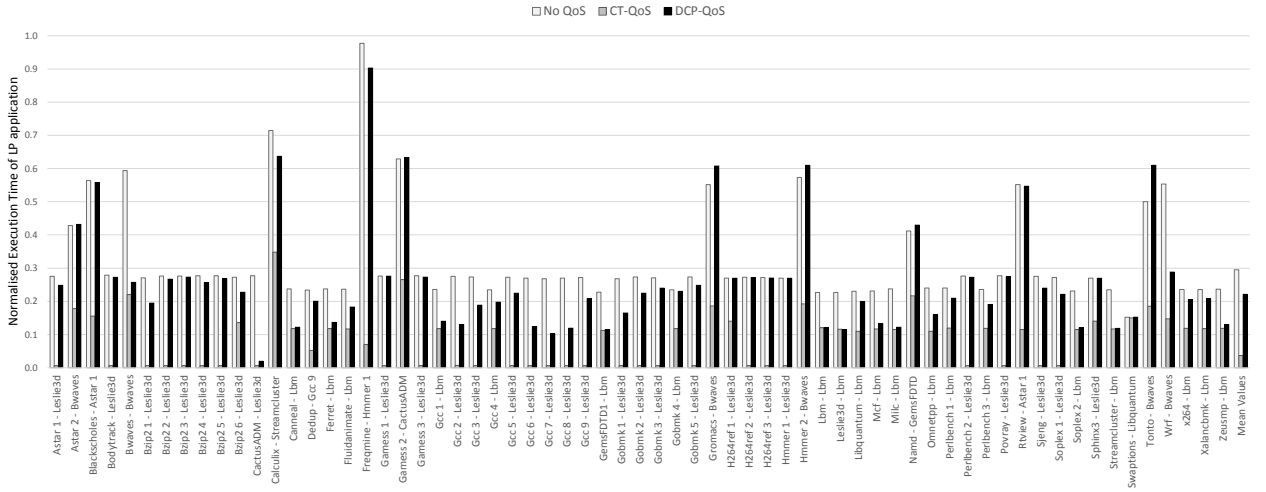
4. EVALUATION

Processor	Intel Xeon E5-2699 v4 (Broadwell)
	22 cores, 2.2GHz, SMT disabled
Memory	64GB DDR4
Memory Bandwidth	153.6 Gbps per channel
LLC	55MB, 20-way set associative
DCP-QoS	$a = 0.05$
	$T = 100ms$
	$n = 10$
	$bandwidth\_threshold = 27.5Mbps$

Table 1: System Configuration



(a) HP



(b) LP

Figure 3: Execution time of the (a) HP application and (b) LP applications, normalized to when each application runs alone on the system (higher is better). DCP-QoS maintains the QoS of the HP application very close to the levels of CT-QoS and at the same time improves the performance for the LP applications by allocating the surplus cache space to them.

4.1 Methodology

We implement DCP-QoS by extending the Intel RDT Software Package v0.1.5 [1], an open source stand-alone library that provides support for CMT, MBM, CAT, and CDP. As the library controls these technologies via the MSRs, it runs in privileged mode. In addition, the library calculates various metrics such as IPC. The evaluation is performed on an Intel Xeon E5-2699 v4 processor. The details of the processor, together with the predefined parameters of DCP-QoS are presented in Table 1.

For the evaluation, we employ 11 benchmarks from the Parsec 3.0 benchmark suite [4] (serial versions) and 28 benchmarks from the SPEC CPU 2006 suite [13], 9 of which can be used with multiple inputs, bringing the total number of applications to 65. We create multiprogrammed workloads by selecting one benchmark as the HP application and 21 instances of another as the LP applications. We first execute all the 4,225 possible co-executions on our system without any QoS mechanisms, allowing full contention for the shared resources. Based on the results, we select for each of the 65 benchmarks its worst co-execution, i.e., the LP co-runners that caused its performance to degrade the most.

We then use the selected 65 co-executions to compare the proposed DCP-QoS mechanism with two other configurations. First the "No QoS" configuration, where no QoS mechanism is used, hence the HP and the LP applications experience full contention on the LLC and memory channel. Second, the CT-QoS configuration, where 19 ways are statically allocated to the HP application and only 1 way is

allocated to the LP applications.

4.2 Results

Figures 3a and 3b show the execution time of the HP application and the LP applications respectively for the selected co-execution scenarios, normalized to when each application runs alone on the system; the last bars show the geometric mean.

The results with the No QoS configuration show that the lack of a QoS mechanism can severely harm the performance of the HP application due to the contention in the cache and the memory bandwidth with the LP applications, as was also shown in Section 2.2. This performance degradation can reach up to 90% and is on average 59%, compared to when running alone.

CT-QoS preserves QoS for the HP application by allowing them to execute on average by 84% compared to when running alone. However, CT-QoS achieves that QoS for the HP application at the cost of significantly sacrificing the performance of the LP applications and underutilising the shared resources. The reason is that CT-QoS pessimistically allocates 19 ways of the LLC for the HP application in a static fashion, even when less ways and hence cache space would have provided similar performance to the HP application. Indeed, CT-QoS degrades the performance of the LP application by 96% compared to when running alone.

Our proposed mechanism DCP-QoS enjoys almost the same QoS benefits that CT-QoS provides for the HP application, while increasing the performance of the LP applications too. More specifically, DCP-QoS achieves high performance for the HP application, by 80% on average compared to when running alone, and ensures QoS close to that of CT-QoS by less than 5%. In addition, DCP-QoS enables higher cache utilisation, increases the overall throughput, and improves the performance for the LP applications by 5× compared to the CT-QoS approach (from 4% to 22% compared to when running alone). Overall, DCP-QoS provides a good trade-off between QoS and utilisation, and enables the co-location of high and low priority workloads on the same server.

5. CONCLUSIONS AND FUTURE WORK

In this paper we experimented with the cache monitoring and partitioning facilities provided by modern multicore processors to test their efficacy in a co-scheduling scenario where a high priority process co-exists with a number of low priority ones. We devised a dynamic cache allocation scheme that monitors the execution behavior of the high priority process and adapts the cache size according to its demands. In this way, we were able to validate that it is possible to maintain the QoS levels of high priority processes to those of full cache allocation, while at the same time significantly boosting the system's throughput by providing more resources to the low priority processes.

We intend to extend our approach to more complex execution scenarios involving different mixtures of processes and priority levels, and combine the allocation decisions with workload characterization schemes to further improve its QoS and throughput.

6. ACKNOWLEDGMENTS

This research has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement no 732366 (ACTiCLOUD).

7. REFERENCES

[1] Intel RDT Software Package. https://github.com/01org/intel-cmt-cat.

[2] Intel Resource Director Technology. https://www-ssl.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html.

[3] M. Bhadauria and S. A. McKee. An approach to resource-aware co-scheduling for CMPs. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 189–199, New York, NY, USA, 2010. ACM.

[4] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[5] S. Blagodurov, S. Zhuravlev, and A. Fedorova. Contention-aware scheduling on multicore systems. *ACM Trans. Comput. Syst.*, 28(4):8:1–8:45, Dec. 2010.

[6] A. D. Breslow, L. Porter, A. Tiwari, M. Laurenzano, L. Carrington, D. M. Tullsen, and A. E. Snavely. The case for colocation of high performance computing workloads. *Concurrency and Computation: Practice and Experience*, pages 232–251, 2013.

[7] C. Delimitrou and C. Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 77–88, New York, NY, USA, 2013. ACM.

[8] C. Delimitrou and C. Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*, pages 127–144, 2014.

[9] C. Delimitrou and C. Kozyrakis. Hcloud: Resource-efficient provisioning in shared cloud systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*, pages 473–488, 2016.

[10] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 335–346, New York, NY, USA, 2010. ACM.

[11] A. Haritatos, G. Gourmas, K. Nikas, and N. Koziris. A resource-centric application classification approach. In *Proceedings of the 1st COSH Workshop on Co-Scheduling of HPC Applications, COSH@HiPEAC 2016, Prague, Czech Republic, January 19, 2016.*, pages 7–12, 2016.

[12] A.-H. Haritatos, G. Goumas, N. Anastopoulos, K. Nikas, K. Kourtis, and N. Koziris. LCA: A memory link and cache-aware co-scheduling approach for CMPs. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 469–470, New York, NY, USA, 2014. ACM.

[13] J. L. Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[14] A. Herdrich, R. Illikkal, R. Iyer, D. Newell, V. Chadha, and J. Moses. Rate-based qos techniques for cache/memory in cmp platforms. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 479–488, New York, NY, USA, 2009. ACM.

[15] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer. Cache QoS: From concept to reality in the Intel® Xeon® processor e5-2600 v3 product family. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 657–668, 2016.

[16] R. R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. R. Hsu, and S. K. Reinhardt. Qos policies and architecture for cache/memory in CMP platforms. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2007, San Diego, California, USA, June 12-16, 2007*, pages 25–36, 2007.

[17] A. Jaleel, W. Hasenplaugh, M. K. Qureshi, J. Sebot, S. C. S. Jr., and J. S. Emer. Adaptive insertion policies for managing shared caches. In *17th International Conference on Parallel Architecture and Compilation Techniques (PACT 2008), Toronto, Ontario, Canada, October 25-29, 2008*, pages 208–219, 2008.

[18] A. Jaleel, H. H. Najaf-abadi, S. Subramaniam, S. C. Steely, and J. Emer. Cruise: Cache replacement and utility-aware scheduling. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 249–260, New York, NY, USA, 2012. ACM.

[19] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 220–229, New York, NY, USA, 2008. ACM.

[20] K. Kumar Pusukuri, R. Gupta, and L. N. Bhuyan. ADAPT: A framework for coscheduling multithreaded programs. *ACM Trans. Archit. Code Optim.*, 9(4):45:1–45:24, Jan. 2013.

[21] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-up: increasing utilization in modern warehouse scale computers via sensible co-locations. In *MICRO*, pages 248–259, 2011.

[22] A. Merkel, J. Stoess, and F. Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 153–166, New York, NY, USA, 2010. ACM.

[23] K. J. Nesbit, M. Moretó, F. J. Cazorla, A. Ramírez, M. Valero, and J. E. Smith. Multicore resource management. *IEEE Micro*, 28(3):6–16, 2008.

[24] K. Nikas, M. Horsnell, and J. D. Garside. An adaptive bloom filter cache partitioning scheme for multicore architectures. In *ICSAMOS*, pages 25–32, 2008.

[25] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO*, pages 423–432, 2006.

[26] S. Srikantaiah, M. T. Kandemir, and M. J. Irwin. Adaptive set pinning: managing shared caches in chip multiprocessors. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008*, pages 135–144, 2008.

[27] L. Tang, J. Mars, and M. L. Soffa. Contentiousness vs. sensitivity: improving contention aware runtime systems on multicore architectures. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 12–21, New York, NY, USA, 2011. ACM.

[28] Y. Xie and G. H. Loh. Pipp: Promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 174–183, New York, NY, USA, 2009. ACM.

[29] H. Yang, A. D. Breslow, J. Mars, and L. Tang. Bubble-flux: precise online qos management for increased utilization in warehouse scale computers. In *The 40th Annual International Symposium on Computer Architecture, ISCA'13, Tel-Aviv, Israel, June 23-27, 2013*, pages 607–618, 2013.

[30] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. *SIGPLAN Not.*, 45(3):129–142, Mar. 2010.

Co-scheduling on Upcoming Many-Core Architectures

Simon Pickartz
Institute for Automation of Complex Power Systems
RWTH Aachen University
Aachen, Germany
spickartz@eonerc.rwth-aachen.de

Jens Breitbart
Bosch Chassis Systems Control
Stuttgart, Germany
jens.breitbart@de.bosch.com

Stefan Lankes
Institute for Automation of Complex Power Systems
RWTH Aachen University
Aachen, Germany
slankes@eonerc.rwth-aachen.de

ABSTRACT

Co-scheduling is known to optimize the utilization of supercomputers. By choosing applications with distict resource demands, the application throughput can be increased avoiding an underutilization of the available nodes. This is especially true for traditional multi-core architecture where a subset of the available cores are already able to saturate the main memory bandwidth.

In this paper, we apply this concept to upcoming many-core architectures by taking the example of the Intel KNL. Therefore, we take a memory-bound and a compute-bound kernel from the NAS Parallel Benchmarks as example applications. Furthermore, we examine the effect of different memory assignment strategies that are enabled by the two-layered memory hierarchy of the KNL.

Keywords

HPC; Co-Scheduling; Many-core architectures; Energy efficiency; Memory Hierarchy

1. INTRODUCTION

Supercomputers will considerably change on their path to exascale systems. The growth in size will be accompanied by an increasing complexity of the compute nodes in terms ascending core counts and multi-level memory hierarchies. These changes put high demands on the application developers. They do not only have to ensure scalability on the inter-node level for up to thousands of nodes, but also need to hand-tune their code for the full exploitation of resources on the intra-node level. This trend is especially challenging for legacy codes which are rarely adapted to the peculiarities of every upcoming generation of supercomputers.

Co-scheduling is one approach for an optimization of the system's utilization when running non-highly tuned codes. As opposed to an exclusive allocation of nodes to applications, the goal is to find suchlike with diverse resource demands to share a set of cluster nodes. Thereby, the individual job experiences a slow down compared to its exclusive execution but the overall runtime may be reduced compared to their serialized execution.

Intel's recently introduced many-core architecture Knights Landing (KNL) presages potential characteristics of upcoming supercomputers. It is a single chip possessing up to 72 cores exposing four Hardware Thread Contexts (HTCs) each. In contrast to common Xeon CPU cores, they are based on Intel's Silvermont micro-architecture which was originally designed for mobile processors such as Intel's Atom. However, the large amount of cores per chip and their 512 bit-wide Single Instruction Multiple Data (SIMD) units reveals a strong potential for highly parallel and vectorized codes. Furthermore, the KNL provides an additional level in the memory hierarchy by the introduction of on-die high bandwidth memory. This is put close to the processor and provides a bandwidth superior to the normal DRAM at the expense of slightly increased latency at lower bandwidth utilizations.

In this paper we investigate the viability of co-scheduling on many-core architectures by taking the example of the Intel KNL. Therefore, we use two kernels of the NAS Parallel Benchmarks (NPBs) suite for the conduction of both performance and energy measurements. Thereby, we want to estimate if our considerations made in previous works [2] apply to upcoming many-core architectures as well. In contrast to these studies, the additional level in the memory hierarchy of the KNL enables further co-scheduling scenarios, e. g., two applications may be executed concurrently using the different memory levels.

This paper is structured as follows: the next section presents the Intel KNL while discussing the different cluster modes and memory configurations. Section 3 gives an introduction to the NPBs and provides information on the runtime characteristics of the two kernels EP and CG which have been used for the evaluation of our work. After presenting a comprehensive evaluation in terms of performance results and energy consumption in Section 4, we discuss related work in Section 5 before concluding the paper in Section 6.

2. INTEL KNIGHTS LANDING

The Intel KNL is the second generation many-core processor of Intel's Xeon Phi product line. In contrast to its predecessor, the KNL is a self-boot processor that features binary compatibility to the mainline Xeon Instruction Set Architecture (ISA) [9]. The processor is divided into 36 tiles connected by a 2D-mesh which implements a YX-routing. Each tile posses two cores based on Intel's Silvermont micro-

COSH-VisorHPC 2017 Jan 24, 2017, Stockholm, Sweden

ISBN 978-3-00-055564-0
DOI: 10.14459/2017md1344415

architecture which have been adapted especially for High Performance Computing (HPC), e. g., each core provides four HTCs. There are two memory controllers on either side of the chip providing three channels respectively for memory bandwidths of up to 90 GiB/s. Apart from the well-known DDR4-DRAM, the KNL additionally provides up to 16 GiB of Multi-Channel DRAM (MCDRAM) which is 3D-stacked on-package memory. This memory type targets at high bandwidths of around 400 GiB/s, however in contrast to DDR4 it shows slightly higher latencies. The MCDRAM is directly connected to the mesh by using 8 memory controllers.

For the evaluation we used the Intel Xeon Phi 7210 possessing 64 cores and a total of 256 hyperthreads. The cores are clocked at 1.3 GHz. Besides the 16 GiB of MCDRAM the system is equipped with 96 GiB of DDR4 memory. We disabled the turbo mode for the avoidance of fluctuations in the measurements.

2.1 Cluster Modes

The 2D interconnect is used for the implementation of cache coherency among the cores. It can be configured in three different so-called cluster modes on the BIOS level. The all-to-all mode is the most general and allows for unbalanced memory distributions across the two DDR4 memory controllers [5]. Therefore, it does not implement an affinity between the tiles, directories, and the memory by uniformly hashing all memory addresses across the distributed directories.

In contrast, the quadrant mode divides the chip into four virtual quadrants which realize an affinity between the distributed directories and the memory. In this mode, the memory addresses are hashed to the directories which belong to the same quadrant as the memory. However, this is transparent to the software which sees a one-socket system exposing 64 cores.

Finally, the Sub-NUMA Clustering (SNC) exposes the quadrants as individual NUMA domains to the software. Hence, NUMA-aware applications can profit from lower latencies by reducing memory accesses to remote quadrants. For all measurements presented in this paper we activated the SNC mode.

2.2 Memory Configurations

The 3D-stacked MCDRAM can be configured in three different ways. Just as with the cluster modes, the configuration can only be performed by changing the BIOS settings accordingly. In the so-called flat mode, the memory extends the physical address space of the system and is explicitly exposed as NUMA domain to the software. Thus, in the quadrant cluster mode one additional NUMA domain shows up while there are four in the SNC mode.

Alternatively, the MCDRAM can operate in the cache mode in which it acts as last-level cache for the DDR4 memory in transparency to the software. Finally, in the hybrid mode, a portion of the MCDRAM acts as cache while the remainder can be used in flat mode. The measurements presented in this paper where conducted by using the stacked memory in flat mode.

3. NAS PARALLEL BENCHMARKS

The NPBs [1] is a set of computing kernels intended for the performance evaluation of supercomputers. They represent common kernels of computational fluid dynamics applications and offer different problem classes. Therefore, the benchmarks are well-suited for the evaluation of a wide range of cluster sizes.

For the evaluation of our work we chose the two kernels EP and CG. Furthermore, we chose Class C as problem size depicting a reasonable size for a workstation test system like the one used for our work. The CG kernel computes the approximation to the smallest eigenvalue of a large sparse matrix [8]. This benchmark is characterized by irregular memory accesses and communication and therefore likely to be limited in performance by the available memory bandwidth. EP computes statistics from Gaussian pseudo random numbers. As the name suggests, there are neither dependencies between the individual work items nor does the benchmark dependent on the available main memory bandwidth.

As we aim at applying co-scheduling to many-core systems, the combination of these two kernels is a reasonable choice. They perfectly meet the requirement of distinct resource demands and should complement each other in a co-scheduling scenario.

4. EVALUATION

This section presents an evaluation of the two kernels EP and CG from the NPBs suite. We start with an analysis of their scalability and power consumption using DDR4 memory and MCDRAM respectively. In the second part we discuss the performance and energy results obtained from co-scheduling the two kernels in different configurations. Each meter point was captured by executing the respective applications for 15 min in a loop and averaging the results afterwards. For a reduction of the captured meter points we chose a step width of 4 threads.

4.1 Application Scalability

The scalability results of the CG and the EP kernel are presented for both memory types in Figure 1. The speedup is computed based on the best sequential execution. This was for both kernels a single thread running on DDR4 memory since this provides slightly better latencies than the MCDRAM. The EP kernel should not be sensitive to certain pinning strategies and we therefore performed a compact pinning on the core level, i. e., we started to fill up the physical cores one by another before using the HTCs. In contrast, we used a scatter pinning on the core level for the CG kernel due to its memory-bound characteristics.

At least for small thread counts, the speedup curves of the EP kernel do not differ when using either of the memory types (cf. Figure 1b) since the kernel is compute-bound. However, for rising thread counts we can observe a continuously growing overhead of up to around 10 % when using traditional DDR4 memory (—△—). This might stem from the higher contention on the memory controllers and the mesh which can be handled more efficiently by the 8 controllers for the high-bandwidth memory in contrast to the two DDR4 memory controllers.

The CG kernel stronger benefits from the high-bandwidth memory. Already using 8 threads, its execution on MCDRAM starts to outperform the results obtained on the DRAM. For thread counts greater equal to 40 threads, we observe a performance benefit of 10 % to 20 %. However, for both memory types the kernel does not make efficient use of additional HTCs. This can be explained by the fact

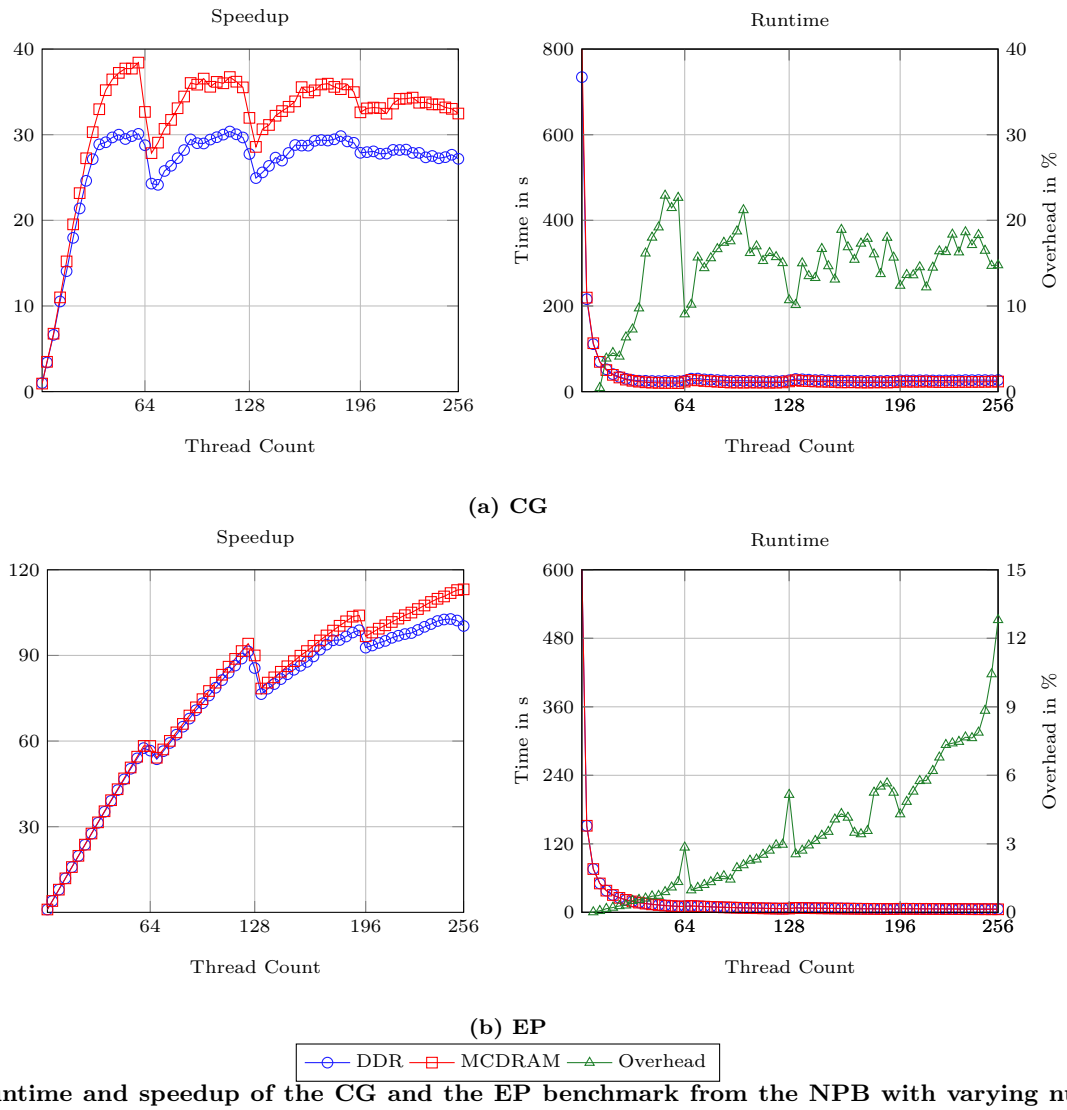


Figure 1: Runtime and speedup of the CG and the EP benchmark from the NPB with varying number of threads. The CG benchmark has been pinned in accordance with the scatter strategy while compact pinning on a core granularity was used for EP.

that the KNL employs out-of-order cores. In contrast, Ramachandran et al. could observe a constant performance increase for the CG kernel on its predecessor which was still based on an in-order architecture [6].

Figure 2 shows the power consumption of the two computing kernels which was captured by both using the Running Average Power Limit (RAPL) counter [3, 4] and a node-external Power Distribution Unit (PDU). The latter is a MEGWARE[1] Clustsafe unit which gauges the complete system power consumption including the power supply. Unfortunately, the RAPL counter available on the KNL only provide energy information of the memory and the whole CPU package but do not allow for a measurement of the energy consumed by the cores only.

The EP kernel exhibits a nearly constant growth of the power consumption except for the spikes at the HTC boundaries which correspond to the performance results discussed above. This is also true for the energy efficiency measured in mega-operations per watt (—— / - - -). The measurements reveal that the usage of the on-package memory result in a slight decrease of the overall power consumption (- - - / ——) which matches with the values obtained for the package and the memory controllers. The latter decreases by around 4 W to 6 W for larger thread counts while the power consumption of the package only increases by 1 W to 3 W.

The power consumption of the CG kernel increases as well for rising thread counts, however logarithmically in this case. Although we see a performance optimum at around 60 threads, the optimal power efficiency is already reached at around 36 and 40 threads when using DDR4 memory and MCDRAM respectively. As the energy efficiency for the CG kernel is improved by up to 20 % when using MCDRAM, we should obtain best co-scheduling performance when running it on the MCDRAM. In contrast, EP is not sensitive to the assigned memory type and therefore we might see performance benefits on its execution using DDR4 memory in a co-scheudling scenario.

[1] http://www.megware.com/.

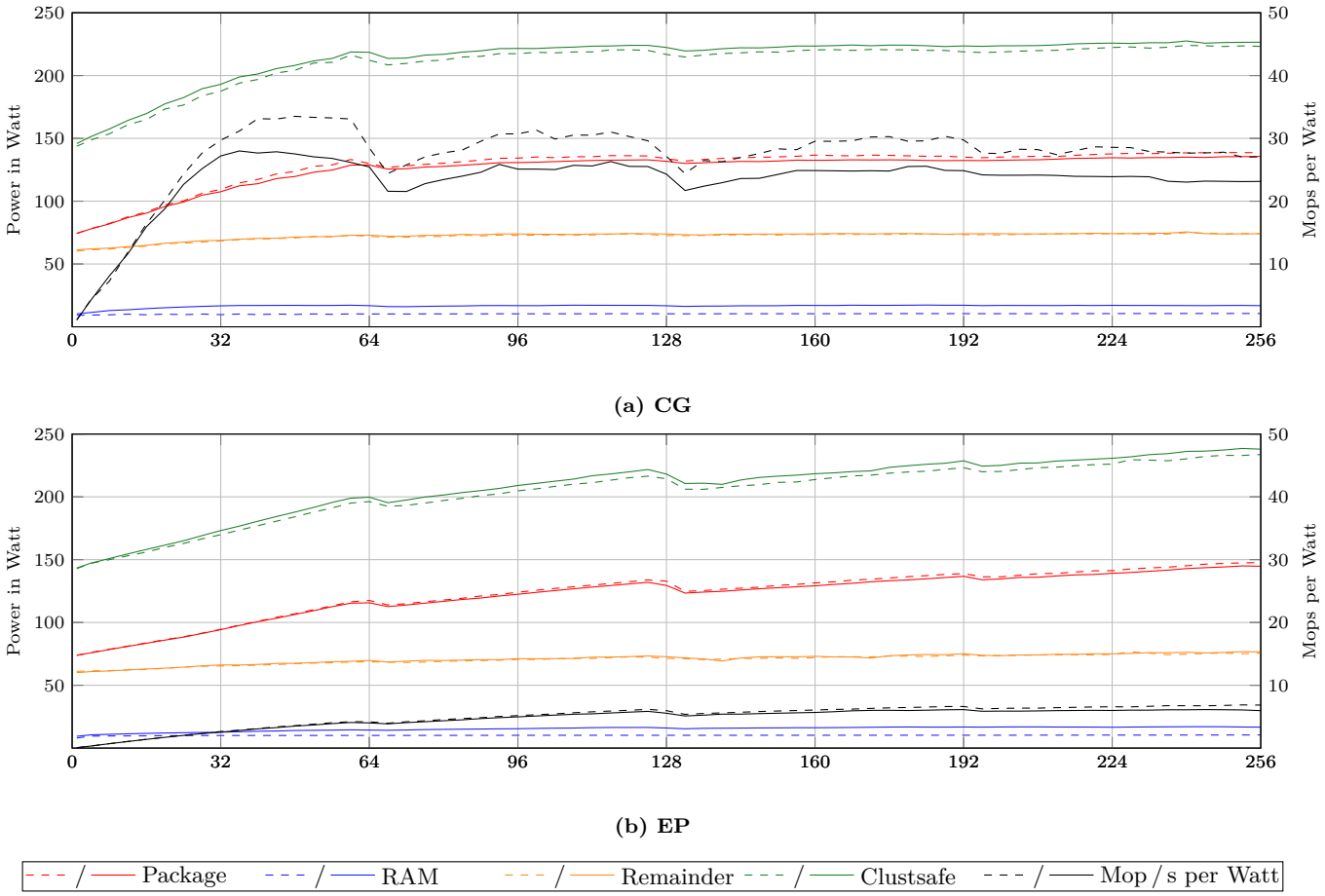


Figure 2: Power consumption (left y-axis) and power efficiency (right y-axis) of CG and EP using different thread counts respectively. The solid lines represent application runs using the DDR4 memory while the dashed lines show the results when using the on-package MCDRAM.

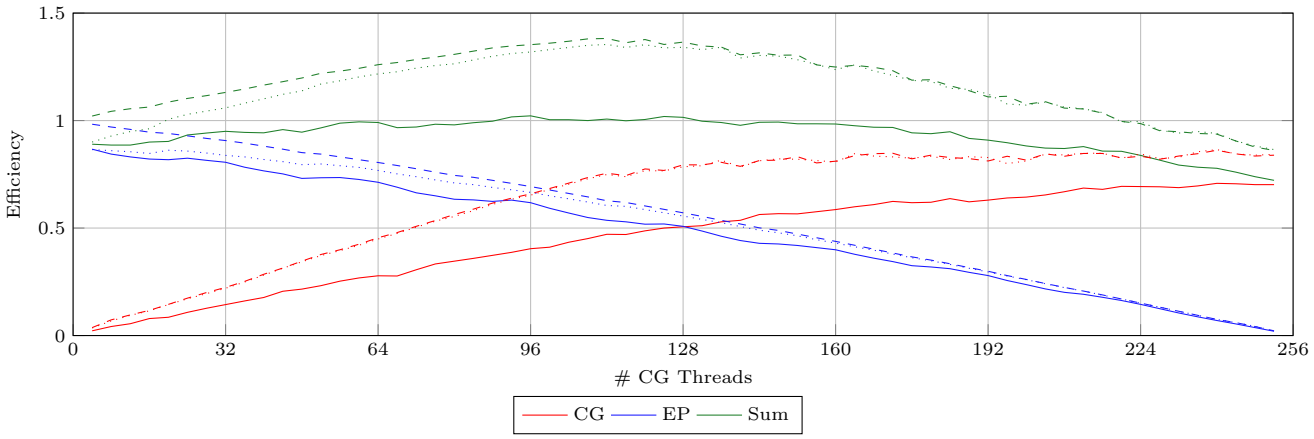
4.2 Co-scheduling Applications

An analysis of co-scheduling the CG and the EP kernel is presented in Figure 3 with respect to the applications' efficiency and power consumption. The former is computed based on the most efficient exclusive application run, i. e., using 60 threads for CG and 256 threads for EP, while assigning the MCDRAM respectively.

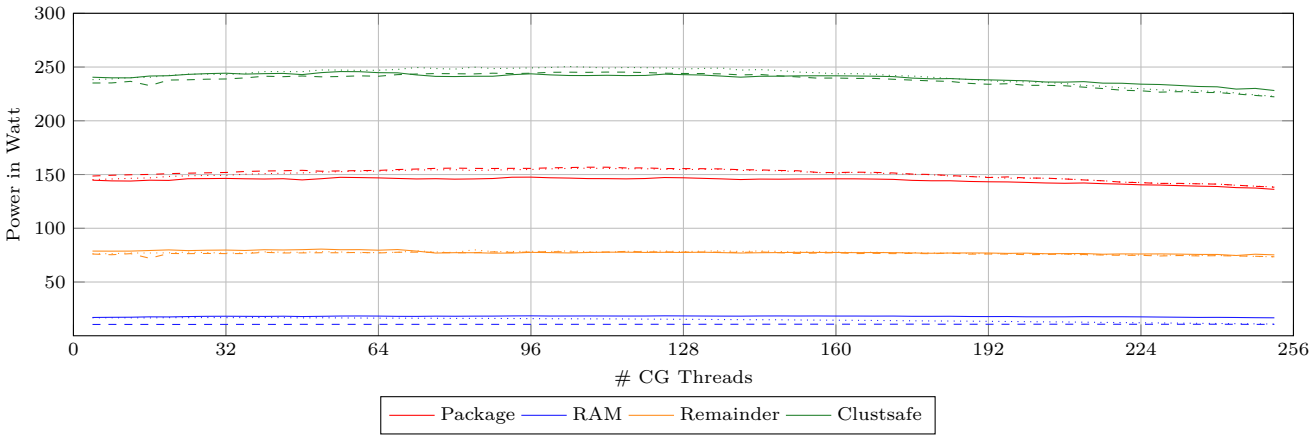
The solid lines represent application runs in which both kernels only allocate memory from the DDR4 memory. In this case, we do not expect larger benefits of co-scheduling both applications as the CG kernel strongly profits from the exection on MCDRAM (cf. Section 4.1). All the more surprising that we can observe a total efficiency greater than one for certain configurations.

In contrast, the dashed lines correspond to applications runs using only the MCDRAM for both kernels. In this scenario we can observe an increase of the overall efficiency by up to 38 % when assigning 112 to CG and the remaining cores to EP. Interestingly, around twice as much CG threads are necessary for a saturation of the efficiency in the co-scheduling case (instead of only 60 threads in the exclusive case). We expect this to be the result of a high contention within the on-die mesh caused by EP threads. There is only little influence on the power consumption when switching the memory types. The power consumption of the package increases when allocating the memory purely on the MCDRAM while that of the RAM decreases likewise. This is expected behavior since the former belongs to the package power domain. However, both CG and EP experience with 26.4 s and 8.5 s respectively a significant reduction of their performance compared to the exclusive cases. Since EP achieves almost the same efficiency on DDR4 memory and MCDRAM, i. e., it is only about 3 % faster on the MCDRAM when using 144 threads, one could assume that the efficiency of the co-scheduling scenario can be further increased when assigning distinct memory resources to each application.

The results obtained when running CG exclusively on the MCDRAM while EP uses DDR4 memory are represented by the dotted curves. However, in contrast to our expectations this strategy does not allow for an improvement of the overall efficiency. We expect that this is again caused by contentions within the on-die mesh as discussed before. Although this scenario realizes an exclusive assignment of the available resources, i. e., the two different memory types, the interconnect is still a shared resource and becomes the bottleneck. However, further research is necessary for a validation of these assumptions, e. g., a more sophisticated pinning of the threads could reduce the average path length to the memory controllers. The deviation from the MC-



(a)



(b)

Figure 3: Co-scheduling scenario of CG and EP: (a) the application efficiency which is computed based on the most efficient exclusive application run and (b) the power consumption for each application run. The solid, dashed, and dotted lines represent runs using DDR4 memory, MCDRAM, and DDR4 for EP and MCDRAM for CG respectively. The x-axis shows the number of threads used by the CG benchmark, the HTCs used by the EP benchmark can be computed via 256 minus the number of CG threads.

DRAM-only case for a high number of EP threads can be explained by its sensitivity to the memory type for high thread counts (cf. Section 4.1).

5. RELATED WORK

Simultaneous scheduling of different applications is common in the area of server and desktops systems. The hardware is designed with multiple HTCs for this type of simultaneous scheduling. However, most compute centers do not offer support for co-scheduling and rather assign the nodes explicitly to HPC applications.

A more common approach for the handling of underutilized nodes is frequency scaling. In doing so, the frequency is dynamically adapted which implicitly results in a reduction of the power consumption. Such an approach is obviously not able to increase the throughput but rather targets at an improved energy efficiency of the HPC systems. Wang et al. [10] discuss a scheduling heuristic reducing the overall system power consumption via Dynamic Voltage Frequency Scaling (DVFS). The Adagio [7] tool analyzes the time spent in blocking MPI function calls and decreases the CPU frequency accordingly. Thereby, the energy efficiency of the HPC system can be improved.

Energy efficient scheduling algorithms are also developed for task scheduling where a task graph representing a program is allocated and ordered on multiple processors. DVFS has been employed for a reduction of the energy consumption of the generated schedules, hence running the processors at heterogeneous speeds. Sinnen et al. discuss task scheduling algorithms [11] and show their potential to improve the energy efficiency on current CPUs.

6. CONCLUSION

In this paper we analyzed the benefit of co-scheduling on upcoming many-core architectures by taking the example of the Intel KNL. Furthermore, we examined the influence of different memory assignment strategies that are enabled by the additional layer in the memory hierarchy provided by the KNL. Our results show that co-scheduling can be valuable for manycore architectures as well, i. e., we could increase the

overall efficiency by up to 38 %. Against our expectations, we could not further improve the performance by dedicating the MCDRAM exclusively to the memory-bound application while serving the other from DDR4 memory. We assume that a high contention in the on-die mesh is the reason for the observed behavior.

For future work we plan a comprehensive analysis of the effects from different pinning strategies to confirm our assumptions. Therefore, we will use a set of micro-benchmarks to determine key figures such as latencies and maximum per-tile bandwidth.

7. ACKNOWLEDGMENTS

This research and development was supported by the Federal Ministry of Education and Research (BMBF) under Grant 01IH13004B (Project FAST). Furthermore, we want to thank MEGWARE who provided us with a Clustsafe to perform the energy measurements.

8. REFERENCES

[1] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *Int. Journal of High Performance Computing Applications*, Sept. 1991.

[2] J. Breitbart, J. Weidendorfer, and C. Trinitis. Case Study on Co-scheduling for HPC Applications. In *2015 44th International Conference on Parallel Processing Workshops*, pages 277–285, Sept 2015.

[3] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le. RAPL: Memory power estimation and capping. In *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, pages 189–194, Aug 2010.

[4] M. Hähnel, B. Döbel, M. Völp, and H. Härtig. Measuring energy consumption for short code paths using RAPL. *ACM SIGMETRICS Performance Evaluation Review*, 40(3):13–17, 2012.

[5] J. Jeffers, J. Reinders, and A. Sodani. *Intel Xeon Phi Processor High Performance Programming Knights Landing Edition*. Morgan Kaufmann Publishers Inc, 2016.

[6] A. Ramachandran, J. Vienne, R. V. D. Wijngaart, L. Koesterke, and I. Sharapov. Performance Evaluation of NAS Parallel Benchmarks on Intel Xeon Phi. In *Proceedings of the 2013 42Nd International Conference on Parallel Processing*, pages 736–743. IEEE Computer Society, 2013.

[7] B. Rountree, D. K. Lownenthal, B. R. de Supinski, M. Schulz, V. W. Freeh, and T. Bletsch. Adagio: Making DVS Practical for Complex HPC Applications. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 460–469. ACM, 2009.

[8] S. Saini, J. Chang, R. Hood, and H. Jin. A Scalability Study of Columbia using the NAS Parallel Benchmarks. Technical report, Aug 2006.

[9] A. Sodani. Knights landing (KNL): 2nd Generation Intel Xeon Phi processor. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–24, Aug 2015.

[10] L. Wang, G. von Laszewski, J. Dayal, and F. Wang. Towards Energy Aware Scheduling for Precedence Constrained Parallel Tasks in a Cluster with DVFS. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 368–377. IEEE Computer Society, 2010.

[11] A. Zaliwski, O. Sinnen, and S. Lankes. Evaluating DVFS scheduling algorithms on real hardware. In *5th International Workshop on Power-aware Algorithms, Systems, and Architectures (PASA 2016), held in conjunction with 45th International Conference on Parallel Processing (ICPP 2016)*, pages 273–280, Aug 2016.

On the Applicability of Virtualization in an Industrial HPC Environment

Tilman Küstner
Technische Universität München, Germany
Tilman.Kuestner@in.tum.de

Carsten Trinitis
Technische Universität München, Germany
Carsten.Trinitis@in.tum.de

Josef Weidendorfer
Technische Universität München, Germany
Josef.Weidendorfer@in.tum.de

Andreas Blaszczyk
ABB Corporate Research Center, Switzerland
Andreas.Blaszczyk@ch.abb.com

Patrik Kaufmann
ABB Corporate Research Center, Switzerland
Patrik.Kaufmann@ch.abb.com

Marcus Johansson
ABB Corporate Research Center, Sweden
Marcus.Johansson@se.abb.com

ABSTRACT

Virtual machines and, to a lesser extent, operating system level virtualization (also referred to as containers) allow for creation of tailored execution environments. This allows software installations to only have to target one environment, making deployment much easier for different platforms. For this reason the technique is used nowadays in most cloud systems.

In this paper we present an industrial setting in which a simulation code should be available on both different HPC cluster environments as well as on standalone workstations. To make deployment easier, we propose the use of virtualization and describe our prototype. Furthermore, we discuss benefits and challenges.

Keywords

HPC; Electric Field Simulation; Virtualization; Containers

1. INTRODUCTION

The introduction of PC-based clusters in the late 1990s has been recognized as a major step to widely enable High Performance Computing (HPC) in industry. Clusters allowed affordable parallel computing of technical applications typically performed with typically tens or hundreds of processors. The reduction of computation times from a few days or overnight to the range of minutes contributed to close integration of numerical simulations into the industrial design environment. For designer teams distributed across the world a reasonable environment has been provided by installing central HPC-clusters dedicated to specific applications which can be accessed through the corporate network. This type of operation has started at the beginning of the 2000s and successfully continues until today [4].

However, with the growing number of applications and users it turned out that the operation of dedicated cluster servers is not always economical. In some cases, where the high availability of computational resources is required and the clusters are dedicated for a single application, the utilization of hardware may be reduced to less than 5-10 %. Furthermore, due to limitations of data transfer rates across intercontinental corporate networks the dedicated servers need to be replicated in order to provide the requested compute capacity worldwide. This generates significant maintenance effort related to porting applications to different hardware platforms and operating systems. All these experiences created an evident need to rearrange the available resources in such a way that applications can be run on any hardware without maintaining a dedicated environment. In addition, the cost accounting according to usage is requested. The concept of virtualization addressed in our paper has been recognized as promising to fulfill the industrial needs.

2. BACKGROUND

The technology behind system virtualization is known since the 70-ties [11], and it is used since that time e.g. by IBM for their main frame systems up to now. However, virtualization on cheap mass-market computer systems only became available in the first years of the current century, when the increased performance of x86 systems made this approach feasible [12]. Early implementations employed workarounds for issues with x86 such as filtering the instruction stream or avoiding problematic instructions at all [1]. The new interest triggered processor vendors to implement new execution modes and hardware features for faster virtualization, reducing overheads in every new processor implementation. This guided the path to the huge market around cloud services today. By being able to efficiently virtualize at the border of hardware and software of a regular x86 system, users can bundle their software in complete system installations including their choice of an operating system and run these installations remotely. The costly maintenance and administration, including transparent backup and repair of failed components, can be outsourced to specialized hardware providers. Concentrating the hardware needs of large parts of the IT industry results in huge consolidation benefits, and competition in low prices. Private clouds, which actually are the old way for a company to run its own server hardware, only makes sense today in limited scenarios such

COSH-VisorHPC 2017 Jan 24, 2017, Stockholm, Sweden

ISBN 978-3-00-055564-0
DOI: 10.14459/2017md1344416

as with specific demands for the security of used data.

While there are a lot of benefits of virtualizing a full system, it is quite resource intensive. Often it is enough to provide an own view to a file system encapsulating a software installation as well as being able to control resource usage by means of regular operating system features. This lighter way of a virtual environment for a group of processes running on top of a given OS is called *containers*. An OS provides own name-spaces for resources such as mounted file systems, open files and network connections, or process IDs to processes inside containers. For the mainline Linux kernel, this is only available since a few years using so-called CGroups[1]. However, there is a large number software available for the administration of containers, with the most known being Docker [10]. As the added functionality for containers mostly consists of adding indirections to system calls, they do not change the performance of HPC code [13, 7].

The HPC community is quite conservative about virtualization solutions because of its eventual overheads. The additional latency introduced by virtualization layers (even in hardware) can disturb the performance of carefully tuned codes. Furthermore I/O hardware often is controlled directly from user level, bypassing the operating system for faster communication and synchronization. The latter complicates virtualization. However, the clear benefits to HPC users, reducing the dependencies to the execution environment by bundling complete software installations, are recognized by HPC computer centers. It should be clear from the above discussion that container solutions are preferred. As the complexity of controlling a lot of resources is not needed in an HPC context (such as restricting network access), software recently is developed such as Singularity[2] or PRoot[3], focusing mostly on a separate file system view.

From the users' point of view, it is convenient to be able to develop his/her HPC code in exactly the execution environment as provided by the HPC compute center, especially as workstations or even laptops run on multi-core processors nowadays, representing already a parallel environment. Furthermore, one can expect that there will be no issues around software packages traditionally provided by the compute center environment, as software dependencies are packaged within a container. One exception here are HPC libraries (such as MPI) able to directly talk to network hardware such as Infiniband, resulting in a dependence of the container to special hardware. If the user knows that performance of his/her application does not depend much on communication, s/he may decide to package a generic MPI implementation. In this case, the container could be embedded into a VM image and easily run also on cloud services, bringing additional benefits such as higher availability and scalability in exchange to spent costs for these systems. However, people regularly check cloud systems for their fitness for HPC codes, and often they fail due to the unknown hardware and connectivity between allocated compute resources [9]. Nevertheless, tailored services for HPC are created by cloud providers[4] which may getting better in the future. Since recently, cloud providers offer increased node performance by adding accelerators such as GPUs. Lower required node counts result in lower reliance on fast communication and synchronization, making such cloud systems more attractive to HPC.

[1]Committed in 2007 by Paul Menage and Rohit Seth.
[2]http://singularity.lbl.gov
[3]http://proot.me
[4]https://aws.amazon.com/de/hpc

3. APPLICATION

As a good candidate for the virtualization approach we consider a numerical program for computation of electrostatic fields based on the boundary element method (BEM) [8]. This ABB in-house program (POLOPT) has become very popular in dielectric design of power devices and is used worldwide by designers of switchgears, transformers as well as other high voltage equipment manufactured at ABB. It is well integrated with CAD-systems, enables automatic 3D-optimization [5] and can be efficiently used for evaluation of electric discharges [3], see application example in Figure 1.

The parallelization of our BEM-application is based on the de-facto message passing standard MPI [2]. According to the implemented master-slave model, the workers (slaves) perform the time consuming process of building the fully populated BEM matrix independently from each other and store the corresponding part of the matrix in local memory. An iterative GMRES solver is executed on the master node and performs parallel matrix vector multiplication by referencing the matrix parts residing on the slaves. This simple algebraic parallelization scheme turned out to be very efficient for up to 100-200 processors (cores). Within this range, the efficiency is almost independent from the structure and performance of the underlying communication network. This feature provides a good foundation for flexible virtualization.

4. APPROACH

4.1 Traditional Approach

Traditionally in high performance computing the software running the simulation (i.e. POLOPT in our case) has to be deployed and configured on each machine or compute cluster it is supposed to run on. This is the job of the system administrator, who has o adjust the users' environment accordingly such that the user can submit his or her parallel simulation run. Up until now, this has been accomplished within ABB using the Simulation Toolbox environment. The Simulation Toolbox platform has become well established in power device design [5]. However, setting up a compute cluster running the Simulation Toolbox including CAD- and numerical simulation- and optimization tools requires a non negligible administrative and maintenance effort as well as full root access to the server system.

4.2 Container/VM based Approach

Cloud computing has become a wide area of research within the last ten years. With powerful server farms becoming more and more affordable, users of high performance computing (mainly database driven[5][6]) have started to run their code in the cloud. This approach is becoming increasingly popular for simulation applications as well, e.g. CAD vendors (e.g. Autodesk[7] or PTC[8]) as well as CAE vendors like

[5]http://www.ptc.com/services/cloud/solutions
[6]http://www.ptc.com/services/cloud/solutionsapplications
[7]http://www.autodesk.com/360-cloud
[8]http://www.ptc.com/services/cloud/solutions

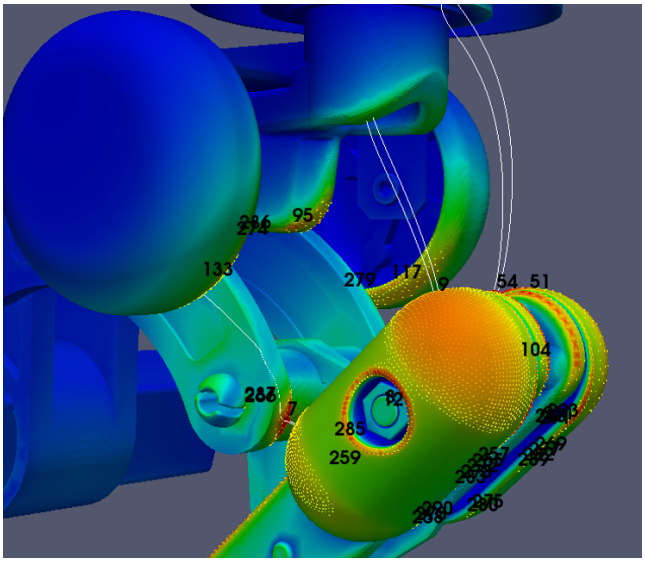


Figure 1: An example of 3D electric field computation (colors correspond to field strength) with evaluation of critical spots (denoted by black numbers) and discharge path (white trajectories) showed in Paraview-based visualizer of the ABB in-house tool Virtual High Voltage Lab [3].

e.g. SimScale[9] are offering cloud based solutions for their code. In addition, research projects aim at providing a comprehensive simulation environment within a cloud based architecture. CloudSME[10], an international research project funded by the European Union, falls into this category.

The abovementioned cloud based approaches are mainly driven by virtualization or container based technologies. Generally speaking, if communication is an issue, a container based approach should be preferred over a virtual machine base approach, as communication between virtual machines is slower than between containers. Although POLOPT's performance is not as communication critical as that of other simulation programs, we decided to use a container approach for an initial implementation. For this reason, the idea behind this paper is to port and adapt our POLOPT code to Docker technology [10]. When moving to a container based implementation, the following advantages and disadvantages need top be taken in to account:

- A container based approach has less hardware dependency than a traditional approach, i.e. the installation does not need to be reconfigured when hardware parameters change.
- A container based approach is "hardware agnostic", i.e. hardware specific features can be ignored from the users' point of view.
- A container based approach provides the well known cloud advantages such as:
  - The possibility of outsourcing the computations yielding no system administration efforts.
  - The possibility to generate specific usage statistics,
  - automatic backup,
  - increased reliability, and
  - better scalability.
- Customizable architecture, i.e. adjust resources according to requirements,
- A uniform development system for all platforms, which makes code development simple as only one version needs to be maintained.
- A container based approach might not be as efficient if the efficiency and performance of the code depend on specific hardware such as e.g. an Infiniband network.

As mentioned above, POLOPT's performance is not as dependent on the underlying network performance (POLOPT scales well even on TCP/IP based Ethernet [6]). Therefore we have decided to not use Infiniband hardware for communication and stick with TCP/IP Ethernet, which eliminated the abovementioned drawback of the container based approach.

5. PROTOTYPE

We want our prototype to be deployed in different use cases. For this reason, there actually are multiple parts to be used in the different scenarios.

Our prototype consists of the following parts:

- a container based on Debian including a POLOPT binary to be used for running as MPI task. This is compiled and linked with the Intel 17.0 compiler and Intel MPI. Thus, the container includes corresponding libraries. The TCP transport back-end is used for communication.
- a few adaptation scripts for our target HPC cluster environment, able to call a given job scheduler.
- a container with a HTTP server, a web page, as well as an Intel MPI installation with the mpirun command. The web page reachable via the HTTP server provides a portal page to (1) request the execution of a POLOPT solver run for given input data and (2) provides feedback to the user on currently executing solver runs including a progress report. For this, the container checks for two different scenarios. Either it runs on the login node, and has access to the adaptation scripts, or it is to be run on a standalone workstation. For the first, a CGI-script calls the adaptation scripts outside of the container which generates and submits a corresponding job script, first exporting the MPI installation outside of the container. For the latter, the compute container is launched first, and afterwards mpirun is called directly.

The scenario for the deployment on a regular HPC cluster is given in Figure 2, showing the different parts of our prototype described above. In contrast, Figure 3 presents the scenario where a user wants to run POLOPT on his/her

[9] http://simscale.com/
[10] http://cloudsme.eu/

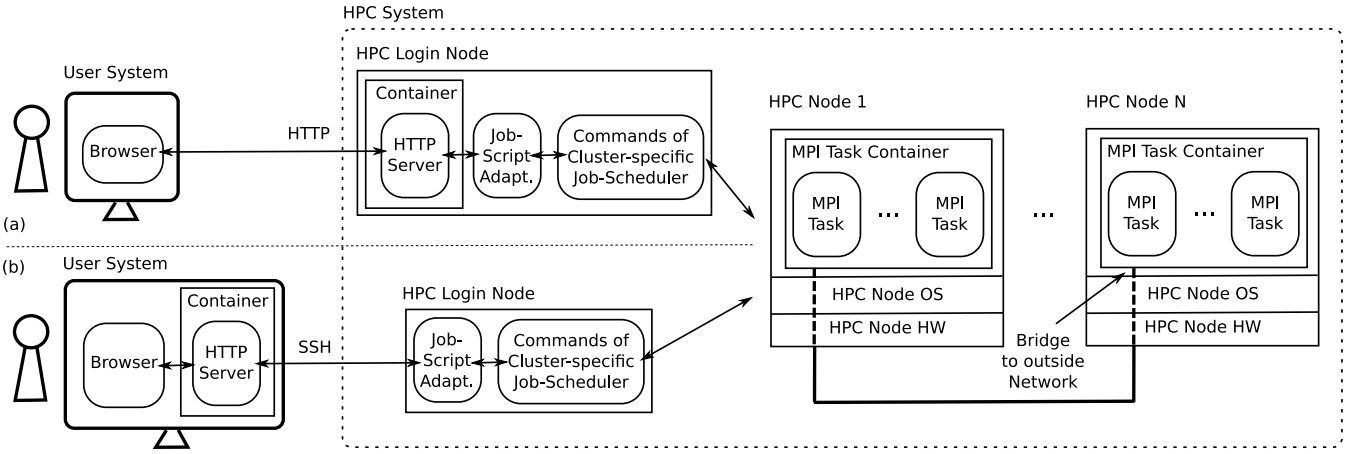


Figure 2: An HPC code using MPI tasks within containers on multiple nodes of an HPC cluster installation with a regular job scheduler. Depending on the HPC cluster provider, a front-end server starting and observing jobs may either run on the login node (a) or has to be installed on the user system (b). In either case, (small) adaption scripts to the job scheduler software used in the cluster have to be provided.

own workstation. This uses the system VM software Virtualbox[11] to be able to run on e.g. Windows or MacOS hosts.

We note here the challenges which had to be solved during development:

- We want MPI code using containers embedding the MPI tasks to be run on multiple nodes. For the MPI tasks to see each other, we decided to use bridged network interfaces to be able to access processes running inside of the containers on the various nodes, see Fig. 2.
- We had to make sure that the topology of compute nodes is passed through to the inside of our containers, as the MPI environment sees the container configuration e.g. regarding available CPUs and NUMA nodes. This is needed as MPI pins MPI tasks to logical CPUs, and any container configuration should keep the configured fixed binding.

First measurements using Docker containers have shown no performance loss on a single node. The sequential run showed exactly the same performance with and without Docker, which is important when comparing scalability, since slower sequential implementations are likely to improve scalability. Since this is not the case here, we ran the same relatively small input model (approx. 7100 unknowns) on a four core single node. In the final version of this paper, comprehensive measurements will be provided together with exact runtime numbers.
We expect the container based version to perform at full speed: For the reasons mentioned in section 4.2 POLOPT should scale well even with a TCP/IP based network as being used for our Docker based approach.

6. FIRST MEASUREMENTS

For the following first results, we run a small model with around 7100 unknowns, using one to four cores in different scenarios:

- direct execution.
- execution in Docker containers, with one container per MPI task[12]. Numbers shown are with containers already launched.
- execution within the system-VM VMware Workstation, using VTX. For the measurements, a guest VM configured with 4 CPU cores is running (with 1 GB memory).

The setup consists of a single system with an Intel Core i5-3450 quad-core processor (Ivy Bridge) with 3.1 GHz nominal clock rate, 6 MB L3 cache, and 8 GB main memory. This processor has support for hardware virtualization (Intel VTX with EPT). The operating system used on the host (and for the container scenario) is Ubuntu 16.04.01. As container solution, Docker 1.12.5 is used. As system-VM, VMware Workstation 12 Player with guest OS Debian 8.6 is used. POLOPT was compiled with the Intel compiler (ifort/icc 17.0.0) and Intel MPI 2017 was used.

Fig. 4 shows the absolute runtimes for the abovementioned scenarios. Results are averaged over 10 runs, respectively. To better understand scalability issues, Fig. 5 presents speedup numbers.

As can be seen, runtimes are quite similar. The VM solution is a little bit slower due to virtualization overhead. However, the speedup using the container approach is slightly worse than for both the native and system-VM approach. Further investigation of this phenomenon will be subject to future work.

7. DISCUSSION

Implementing the approach described above, we anticipate our simulation code to show the same performance with and without a container. Networking should not be a big concern, as POLOPT scales well on both Infiniband and

[11] `https://www.virtualbox.org`

[12] Another possibility is to have one container for all MPI tasks running on a compute node. This may be better for communication and is under investigation.

Ethernet. In general, simulation software which is not heavily reliant on a high performance network should suffer little slowdown in a virtualized environment. We do not expect a present job scheduling system to have specific extensions to run containers; instead, we provide adaptation scripts which allow to initialize the containers when a job is started and cleaning up afterwards.

In the case of using a cloud environment, there are both advantages and disadvantages. Most important is data security as CAD models and other data required by the simulation might be important business secrets. Data can be encrypted for transport on public networks but needs to be decrypted when running the actual computation. While newer processors have ISA extensions to limit access from OS- to user-level (e.g. enrypted data in DDR memory), such hardware deployed in a cloud does not naturally improve the consumer's confidence in the cloud provider.

Further, the customer has limited control over hardware, updates and monitoring of resources. Performance might be less predictable. If the simulation code depends heavily on inter-node communication, a high performance network is required. Also, for large amount of input/output data, a high performance parallel file system is desirable.

Using a cloud environment for simulation codes offers several advantages. Model data can be shared and versioned among engineers using a storage cloud. We hope to see higher flexibility and scalability as the same code (inside a container) can be be run on an engineer's laptop computer as well as a high performance cluster. The hardware on which the container starts can be chosen based on the simulation code and input data. For example, accelerator-aware codes can be launched on compute nodes with GPUs. Larger input models which require more main memory would start several containers on several nodes.

In a best case scenario, an industrial company does not need to buy and manage dedicated hardware. Flexible payment models, as offered by cloud hosting providers, also reduce costs. Better utilization of hardware capacity, better hosting and reduced energy consumption make this scenario also more environmentally friendly.

Deployment of new software and software updates is simplified. The workload of a system administrator is also reduced, as libraries and other prerequisites need not be installed on the system. Instead, they come packaged inside the container. This also reduces time and effort of end-users (engineers). They need less knowledge about accessing the compute cluster or the job scheduling system. Instead, they find a lean and clear web front-end, which shows number and progress of running jobs. Optionally, costs per job could be displayed to the end-user as well, in order to make the accounting more transparent.

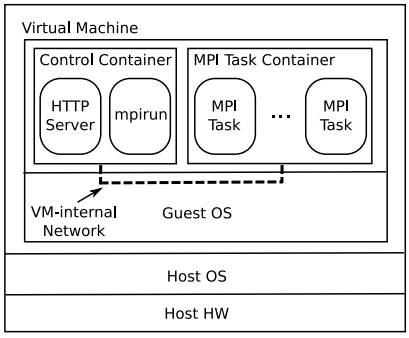


Figure 3: On a standalone workstation, the same container images as in the HPC cluster scenario can be used. Running them inside a VM allows independence from the OS of the workstation.

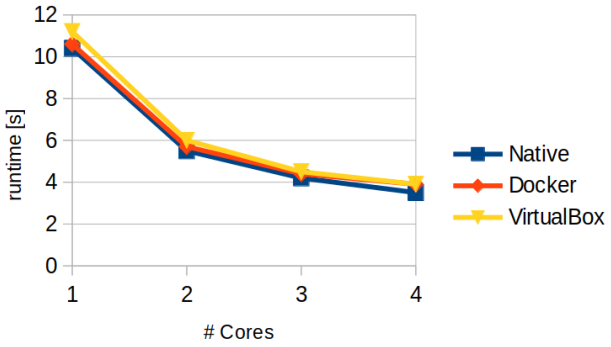


Figure 4: Absolute runtimes of a small POLOPT model in various scenarios, running on 1 up to 4 cores (MPI only).

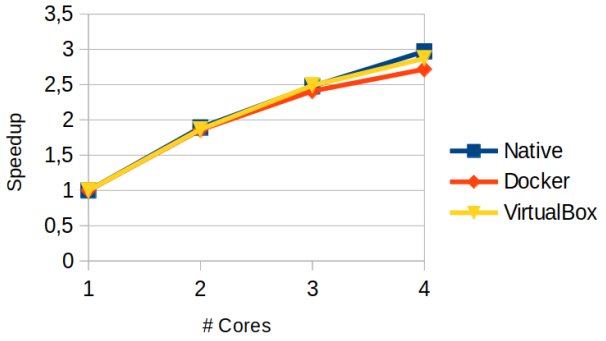


Figure 5: Speedups of a small POLOPT model in various scenarios, running on 1 up to 4 cores (MPI only).

8. CONCLUSIONS AND OUTLOOK

In this paper, we propose to enable the benefits of virtualization for industrial HPC codes. That is, simulation codes are deployed within containers and available for execution in different runtime environments such as HPC clusters and workstations. First measurements, using a relevant simulation code within ABB, show that packaging industrial codes in this way provides the end-user with reliable and fast computational resources even at peak times.

For future work, we will extend our measurements both to multiple nodes and other applications. For the latter, it is especially important to also look into more communication intensive applications.

If an industrial use case can risk to use cloud computing, we can assume that all execution environments use system VMs. In this case, approaches such as library-OSes which link applications and required OS functionality become interesting, as this can improve efficiency (no context switches between guest OS and guest user level). We will look into

such novel approaches in the future.

Acknowledgment

The authors would like to thank Anton Schreck for providing the prototype setup and first measurements.

9. REFERENCES

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, Oct. 2003.

[2] A. Blaszczyk, Z. Andjelic, P. Levin, and A. Ustundag. Parallel computation of electric fields in a heterogeneous workstation cluster. In *High-Performance Computing and Networking Conference*, volume 919 of *LNCS*, pages 606–611. Springer, 1995.

[3] A. Blaszczyk, J. Ekeberg, S. Pancheshnyi, and M. Saxegaard. Virtual high voltage lab. In *Scientific Computing in Electrical Engineering (SCEE) Conference*, Strobl, Austria, Oct 2016.

[4] A. Blaszczyk, H. Karandikar, and G. Palli. Net value! Low cost, high-performance computing via the Intranet. *ABB Review*, (1):35–42, 2002.

[5] A. Blaszczyk, J. Ostrowski, B. Samul, and D. Szary. Simulation Toolbox. In *ABB Review*, volume 3, 2013.

[6] A. Blaszczyk and C. Trinitis. Experience with PVM in an industrial environment. In *European Parallel Virtual Machine Conference*, pages 174–179. Springer, 1996.

[7] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*, pages 171–172. IEEE, 2015.

[8] N. D. Kock, M. Mendik, Z. Andjelic, and A. Blaszczyk. Application of 3-d boundary element method in the design of ehv gis components. *IEEE Electrical Insulation Magazine*, 14(3):17–22, May/Jun 1998.

[9] P. Mehrotra, J. Djomehri, S. Heistand, R. Hood, H. Jin, A. Lazanoff, S. Saini, and R. Biswas. Performance evaluation of amazon ec2 for nasa hpc applications. In *Proceedings of the 3rd Workshop on Scientific Cloud Computing Date*, ScienceCloud '12, pages 41–50, New York, NY, USA, 2012. ACM.

[10] D. Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), Mar. 2014.

[11] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, July 1974.

[12] M. Rosenblum. The reincarnation of virtual machines. *ACM Queue*, 2(5):34–40, 2004.

[13] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 233–240. IEEE, 2013.

Towards a Lightweight RDMA Para-Virtualization for HPC

Shiqing Fan
Huawei Technologies
German Research Center
shiqing.fan@huawei.com

Fang Chen
Huawei Technologies
German Research Center
fang.chen1@huawei.com

Holm Rauchfuss
Huawei Technologies
German Research Center
holm.rauchfuss@huawei.com

Nadav Har'El
cyllaDB
nyh@scylladb.com

Uwe Schilling
University of Stuttgart
HLRS Stuttgart
schilling@hlrs.de

Nico Struckmann
University of Stuttgart
HLRS Stuttgart
struckmann@hlrs.de

ABSTRACT

Virtualization has gained increasing attention in the recent High Performance Computing (HPC) development. While HPC provides scalability and computing performance, HPC in the cloud benefits in addition from the agility and flexibility that virtualization brings. One of the major challenges of HPC in virtualized environments is RDMA virtualization. Existing implementations of RDMA virtualization focused on supporting VMs running Linux. However, HPC workloads rarely need a full-blown Linux OS. Compared to traditional Linux OS, emerging *Library OSes*, such as *OSv*, are becoming popular choices as they provide efficient, portable and lightweight cloud images. To enable virtualized RDMA for lightweight library OSes, drivers and interfaces must be re-designed to accommodate the underlying virtual devices. In this paper we present a novel design, the *virtio-rdma* driver for *OSv*, which aims to provide RDMA para-virtualization for lightweight library OS. We compare this new design with existing implementations for Linux, and analyze the advantages of *virtio-rdma*'s architecture, its ease of migration to different operating systems, and the potential for performance improvement. We also propose a solution for integrating this para-virtualized driver into HPC platforms, enabling HPC application users to deploy their use cases smoothly in a virtualized HPC environment.

Keywords

Virtualization; virtIO; RDMA; HPC; Unikernel

1. INTRODUCTION

Para-virtualization has been commonly used in virtualized environments to improve system efficiency and to optimize management workloads. In the era of High Performance Computing (HPC) and Big Data use cases, cloud providers and HPC centers focus more on developing para-virtualization solutions of fast and efficient I/O. Due to the nature of high bandwidth, low latency and kernel bypass, Remote Direct Memory Access (RDMA) [3] interconnects play an important role for the I/O efficiency, and it has been widely deployed in HPC and data centers as an I/O performance booster. To benefit from these RDMA advantages in virtualized HPC environment, network communication supporting InfiniBand and RDMA over Converged Ethernet (RoCE) [4] must be enabled for the underlying virtualized devices.

There are a few existing solutions for RDMA virtualization, e.g. *vRDMA* [14] from VMware and *HyV* [13],a hybrid I/O virtualization framework for RDMA-capable network interfaces, from IBM. However, none of them is applicable for virtualization on HPC. *vRDMA* is only available for *VMware ESXi* guest, and it is not open source based and not free. *HyV* supports only for Linux kernel 3.13, and it relies heavily on Linux kernel drivers, which tightly couples Linux host and guest, excluding the usage of lightweight library OSes [8].

To enable such communications for virtualized devices with *OSv*, a lightweight, fast and simple library OS for Cloud, we designed a new para-virtualized frontend driver, *virtio-rdma*, for RDMA-capable fabrics. This solution aims to disrupt the overhead barrier preventing HPC Cloud adoption, enable HPC applications to run in virtual machines with a performance comparable to bare metal, and bring all the benefits from the Cloud. The *virtio-rdma* frontend driver is designed to support also shared memory communication for the virtual machines (VM) on the same host. By switching the protocols automatically in *virtio-rdma*, user application uses only the standard RDMA API to accomplish both inter-host and intra-host communications.

On the other hand, our design also includes a solution to use *OSv* and *virtio-rdma* in HPC environment. By extending *Torque* [5], necessary environment settings are configured to launch *OSv* and its components. The job submitting procedure is the same as on a normal HPC platform, except the job command line needs simple adoptions.

This paper is structured as follows: section 2 introduces the fundamental work for this design; section 3 presents the details of the *virtio-rdma*, including its components, capabilities and advantages; section 4 shows the basic integration solution for running HPC jobs with *OSv* and *virtio-*

COSH-VisorHPC 2017 Jan 24, 2017, Stockholm, Sweden

ISBN 978-3-00-055564-0
DOI: 10.14459/2017md1344417

rdma; section 5 concludes the paper and describes the current states of the implementation and future plan.

2. BACKGROUNDS

In this section, we introduce several key frameworks and projects which serve as the foundation for our RDMA para-virtualization.

Traditional Root I/O Virtualization (SR-IOV) shares RDMA devices between multiple virtual machines, it also introduces high development and maintenance costs. SR-IOV is strictly dependent on the choice of hardware devices, making migration difficult when switching to a new Network Interface Controller (NIC). Unlike SR-IOV, para-virtualization RDMA solutions are more flexible, and it uses standard APIs therefore it can be applied with different NICs. Moreover, it allows VMs with direct access to RDMA memory regions and avoids moving communication data around between host and guest.

2.1 Remote Direct Memory Access

Remote Direct Memory Access (RDMA) is the ability to directly access another system's memory without involving the CPU on that remote system. Two major technologies support RDMA: InfiniBand [1] and RoCE. InfiniBand provides support for native RDMA, providing highly efficient and low latency interconnect technology. For Ethernet based network connections, RoCE provides true RDMA semantics, serving as an efficient Ethernet solution today.

The RDMA verbs API [7], is an abstract interface to an RDMA enabled NIC (RNIC). It is a de facto standard software stack defined by *OpenFabrics Enterprise Distribution (OFED)*. OFED stack aims to develop open-source software for RDMA and kernel bypass applications. The interface provides access to the RNIC queuing and memory management resources and is normally implemented as a combination of the RNIC by the device vendors. Therefore, the level of abstraction for our RDMA para-virtualization will be the verbs API to be vendor independent. From an architecture point of view, a virtualized RDMA driver consist of a backend driver at host side which communicates with the actual physical device, and a frontend driver at the guest level. The communication between frontend and backend drivers are then carried out using Hypercalls.

2.2 OSv and virtio

Our implementation uses the *OSv* [10] operating system on each VM. *OSv* was designed and optimized specifically for running a single application on a virtual machine in the cloud. We note that the modern cloud provides features, such as isolation, hardware abstraction, and management, which were traditionally provided by operating systems. By not duplicating these features, *OSv* is simpler, smaller and faster than traditional operating systems such as Linux, and helps reduce the overhead of virtualization. It is basically derived from FreeBSD and has been largely re-implemented under C++11 standard. Each VM has a single copy of *OSv*, and each VM runs a single application.

OSv uses a single address space for the kernel and a single application's threads, and has been considered a *library OS* [8] or more recently, a *unikernel* [12]. By using the same page tables for all threads and the kernel, context switches are largely reduced. But unlike some other unikernels which support only a limited range of applications or hypervisors, *OSv* can run on most common hypervisors (KVM, Xen and VirtualBox), and run unmodified Linux applications; It also fully supports multi-threading, and VMs with multiple cores. Moreover, paging and memory mapping are supported in *OSv* via *mmap* API.

As *OSv* runs only in VMs, it does not need to implement drivers for a huge variety of real hardware such as network cards. Rather, it needs to support just a few para-virtual devices provided by the hypervisor. The KVM hypervisor uses the *virtio* [15] protocol as a framework for all low-overhead guest I/O; The the para-virtual network driver (*virtio-net*) and disk driver (*virtio-blk*) are implemented using this protocol. With virtio, the performance-critical data path has minimal overheads such as guest-host context switches, while the host retains full control over the management of the device.

The present work adds a new driver to *OSv*, *virtio-rdma*. This uses the same virtio protocol to allow the guest to efficiently use the host's hardware RDMA capabilities with minimal overheads (such as guest-host context switches) while still giving the host full control with whom each guest can communicate.

2.3 Torque

Torque [5] is a portable batch system, which is used in many HPC platforms, e.g. clusters at High Performance Computing Center Stuttgart (HLRS). It queues, schedules and executes compute jobs inside clusters.

In order to execute the same jobs inside VMs, *Torque* has to be extended and modified to set up suitable environment for starting the guest OS and initializing the

drivers that to be loaded for running the jobs.

3. THE NEW PARA-VIRTUALIZATION DESIGN

In this section, we describe the new *virtio-rdma* design comparing with the existing designs, how it is capable to work for OSv and several advanced features. Our new design is based on HyV, adapting and extending it where needed. It makes extensive use of hypercalls, to facilitate direct communication between guest and host.

The general idea of the design architecture is shown in Figure 1. The guest application uses *socket* or *RDMA*. The *virtio-rdma* frontend driver will decide which communication protocol to be used based on the location of the peers. For example, the communication between VMs on the same host will use shared memory as a short cut for better performance. The communication between VMs on remote host will use the virtual *RDMA* path.

3.1 Frontend Driver

Theoretically, there are several different possible design schemes for RDMA para-virtualization frontend driver. It can be implemented in different layers of the guest OS, i.e. drivers in guest user space, providers in guest kernel space or both.

HyV [13] follows the rules of using the *OFED* user and kernel modules as is, and replacing a few kernel modules with its own *virtio* drivers in the guest kernel space. As shown in Figure 2, we present an example of querying device verb call showing the differences between the standard path and the *HyV* path. The *libibverbs* library exposes the

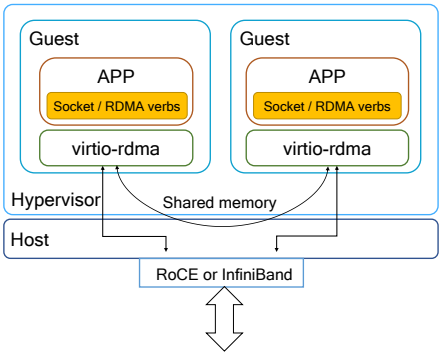


Figure 1: Architecture overview of *virtio-rdma*.

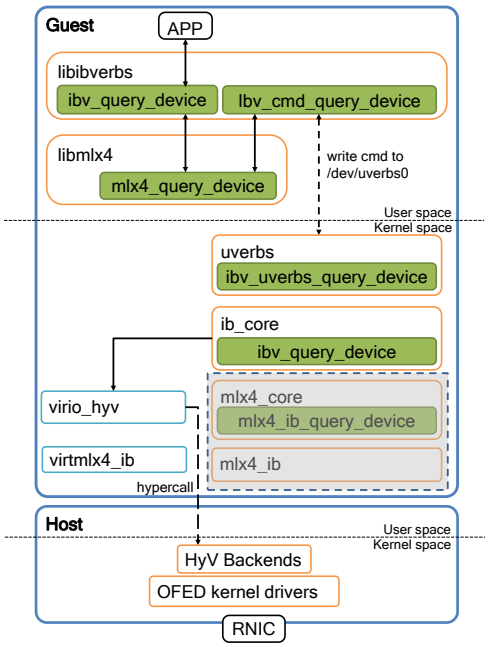


Figure 2: Workflow of calling *ibv_query_device* function using *HyV*.

verbs API to the application. The *libmlx4* library is the device provider in the user space, which is also known as a plugin to *libibverbs* library for Mellanox devices. For different RDMA devices, other plugin may be loaded. The *libmlx4* library provides necessary hardware information to prepare the verb command in *libibverbs* library. The uverbs kernel module exposes a uverbs device to the user space, which handles the request command written by the registered clients. The uverbs kernel module transforms the command to the kernel context and passes it to the lower level kernel module, i.e. *ib_core*, *mlx4_core* and *mlx4_ib*, to perform the real hardware operation. *HyV* replaces *lx4_core* and *mlx4_ib* kernel modules with its own *virtio* drivers, i.e. *virtio_hyv* and *virtmlx4_ib*, to send hypercalls to the host instead of manipulating the real hardware. The corresponding host driver (*vhost_hyv*) handles the request and perform the actual hardware operation with the local *OFED* support.

The *HyV* guest drivers abstract the kernel verb calls that are able to work with *OFED* kernel modules, thus it highly

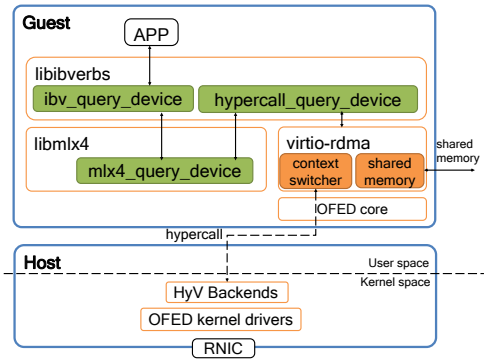


Figure 3: Workflow of calling *ibv_query_device* function using *virtio-rdma*.

relies on several unmodified *OFED* modules. When it comes to a library OS like OSv, implementing the same design become very difficult and even impossible due to the lack of *OFED* support. A library OS normally has only the minimum set of libraries that are required to support the targeting application, and there have no available support of *OFED* drivers for OSv or any other library OSes. Supporting the original *HyV* on *OSv* will involve extra work of porting the *OFED* modules and additional *FreeBSD* kernel implementations, which may end up to porting most of the *FreeBSD* kernel to *OSv*.

However, implementing similar frontend drivers as *HyV* on *OSv* is still possible with even a much simpler architecture. For a library OS, the libraries are compiled and sealed with the application together into fixed images that can be run directly with a hypervisor. The created images are constructed with single address space, that the need for transmitting data between user space and kernel space is not necessary. This specific feature of allowing direct access to hardware resources without context switches does not only improve the performance, but also allows us to get rid of the dependencies of the *OFED* kernel modules.

The new *virtio-rdma* driver is designed to work with minimum support of *OFED* kernel providers or drivers, as shown in Figure 3. We keep two user space libraries: *libmlx4* provides the basic hardware information, e.g. vendor ID, device ID and hardware specific parameter format; *libibverbs* exposes the verbs API to the user application, and compose and send the hypercall message to the host driver. Both of them have been simplified: only the necessary fundamental support is kept; the available implementation in *OSv* has higher priority to be reused and to replace the *OFED* implementation.

At moment *virtio-rdma* supports only InfiniBand devices by dynamically loading *libmlx4* plugin. But this can be extended by supporting more device providers in the guest, and replace *libmlx4* at runtime.

3.2 Backend Driver

On the host side, we take the advantage of using *HyV vhost* driver, and the communication contains the same data structure as *HyV* uses. The work on the host side involves only porting the *HyV* host drivers to the targeting Linux version[1]. The new frontend driver provides the sim-

[1] *HyV* was initially implemented for Linux Kernel version

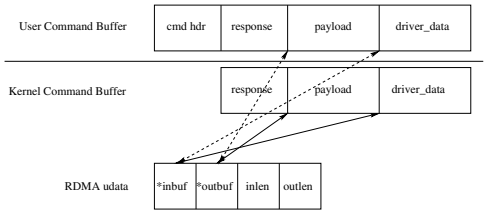
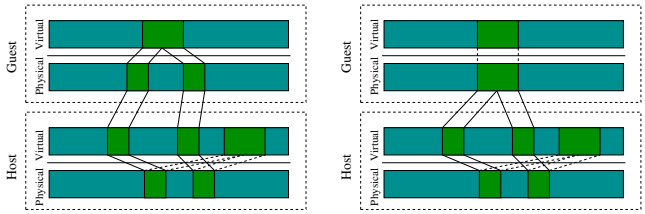


Figure 4: Context switch: command buffer and udata.



(a) Hyv **(b) virtio-rdma**

Figure 5: Comparison of memory mapping mechanisms between *Hyv* and *virtio-rdma*.

ilar hypercall functionality as *HyV*, but with entirely new implementation based on the *OSv* implementation, which is mostly C++11 standard based and the *virtio* API is defined differently as on Linux and FreeBSD.

3.3 Context Switch

The most significant buffer that requires frequent user to kernel switch is the *RDMA* command buffer, which contains a command header, response buffer, payload buffer and driver data. Normally, when calling a verbs API from user space, a user command buffer is initialilzed in *libibverbs* and copied to the kernel provider, which creates a *udata* structure based on the response and payload buffers. The *udata* is then pushed to the *RDMA* kernel call. After the call is finished, the kernel provider copies *udata* back to user space. *HyV* doesn't change this process on the Linux guest.

In order to be compatible with *HyV* backend driver for using *virtio-rdma*, the hypercall parameters, especially the user command buffer, have to be converted to the kernel format. As OSv uses single address space implementation, we are able to avoid many context switches for handling the *RDMA* command buffer, by diretlly mapping *udata* into user command buffer, as shown in Figure 4.

As we directly map the kernel objects to the user space in *OSv*, no kernel provider will be required, except a few *OFED* core definitions for handling the kernel objects such as *udata*. This also decreases the number of dependencies and also the number of library calls. Figure 3 shows the same example using *virtio-rdma* architecture. Instead of sending the uverbs command to *OFED* kernel drivers, it does the hypercalls directly with the help of *virtio-rdma*. The unnecessary libraries have been eliminated, which saves up to 5 library calls per verb command.

3.4 RDMA Memory Mapping

The RDMA memory regions, like queue pairs, completion queues and work requests, are directly shared between the guest and the host, and the InfiniBand hardware is able to operate on them via DMA. This is the fundamental rule followed by both *Hyv* and *virtio-rdma* to accelerate the data path and off-load the CPU.

When using *Hyv* guest driver with Linux, the kernel will assign memory with contiguous virtual address but potentially non-contiguous regions in physical space, as shown in Figure 5a. The guest driver needs to parse the physical address and save the starting physical address, offset, and size of each contiguous region (chunk) in a mapping list. The list will be passed to the host driver, where each chunk will be translated back to the host physical address space and then mapped with the InfiniBand hardware. At the end, the backend driver holds a contiguous virtual address of the same memory region.

However, for OSv, the core memory allocator works differently as on Linux. The allocated memory is always contiguous in both virtual and physical address spaces (Figure 5b). The memory translation is then made simpler: the guest driver takes care of the offsets in the first page and last page (all memory in between should be already contiguous and page aligned); it prepares the mapping list with physical memory of each chunk and pass them to the host; the backend driver will still map the guest memory into non-contiguous pages as it required.

The memory mapping process in *virtio-rdma* saves the effort of retrieving allocated pages of the the memory region, and finding contiguous pages by traversing and comparing page addresses.

3.5 Shared Memory Support

In this new design, *virtio-rdma* is capable for inter-host communication across the host through the hypercall path, it is also designed to support intra-host communication on the same host by supporting shared memory protocol. The VM and its application have no knowledge about the network topology, but VMs on the same host have the same *RDMA* device handle. Whether the communication should use shared memory or not is decided by comparing the device handles of peers.

The shared memory module in *virtio-rdma* is based on *ivshmem* [11], also known as *Nahanni*. It is implemented in the *virtio-rdma* driver, and it can be dynamically loaded when required. For example, when registering the *RDMA* memory, i.e. calling *ibv_reg_mr* verb API, the same memory region will be expose to be shared to other VMs on the same host by the shared memory module. Only if the communication peers has the same device handle, i.e. on the same host, the shared memory will be used for transmitting the data. The access permission is also protected by the *RDMA* communication rules, e.g. protect domain. For example, when calling *ibv_post_send* to post send to the VM on the same host,, we do not go futher with the *RDMA* stack, but rather use the implemented shared memory functions, which still follows the *RDMA* scenario: update the send request; tag the access permission of the send buffer in the queue pair to the other VM using the shared memory functions; update the completion queue by the frontend driver.

3.6 Support for socket

The purpose of supporting for *socket* in *virtio-rdma* is to accelerate the *socket* communication using the internel protocol, The *virtio-rdma* frontend driver needs to expose

3.13. Our project is targeting 3.18 for the host OS.

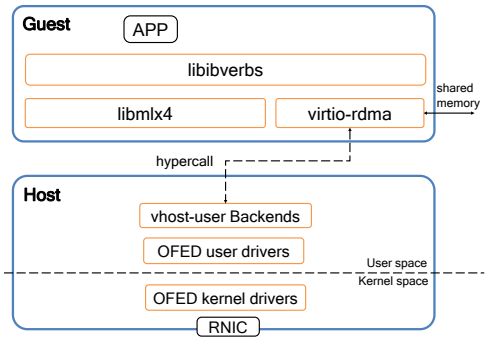


Figure 6: *Virtio-rdma* using *vhost-user* as the backend.

socket API to the guest application, and replace the *TCP* stack by virtual *RDMA* or shared memory. Two frameworks, *rsocket* [9] and *libvma* [16] have been evaluated and will be taken as the foundation of this work.

3.7 Improvement

One alternative and advanced way of doing the hypercalls in *virtio-rdma* is to compose hypercall message using the user context, which does not need the context switcher and OFED core definitions. However, this requires several changes on the host side, as shown in Figure 6: the backend driver should know about the user verbs, i.e. replacing the kernel verbs abstraction with user verbs API; the backend driver needs to work in user space. The key for this solution is *vhost-user* [6]. Converting the hypercalls with user verbs may simplify the design of *virtio-rdma* by omitting the user to kernel context switch and the OFED core definitions support. On the host side, it requires additional work to adjust vhost driver with *vhost-user* implementation.

4. HPC INTEGRATION

Our ultimate goal is to integrate *virtio-rdma* solution into a high performance cloud system, allowing for optimized network I/O performance. To boot a VM on a physical node in an HPC environment, the batch system requires certain adaptions and extensions. In this section, we describe the modification that needs to be carried out with resource manager and batch scheduler *Torque*, in order to run a job on a HPC system with *virtio-rdma* support.

Compared to traditional bare metal execution, a virtualized batch job's lifecyle is comprised the following:

- During the setup phase, the *prologue* [2] script generates metadata for customizing the VMs during boot by the help of *cloud-init* [15]. It also installs missing packages, creates the user account, and mounts shared file-systems available on the physical nodes. Further, the *prologue* script sets up the *virtio-rdma* and waits for the VMs to become available via SSH.
- Job execution is wrapped by another script that prepares the VM's environment variables for the applications. After preparing the VMs via SSH, the user job script is executed in the first virtual guest.
- In the tear down phase of a job's lifecycle, the *epilogue* [2] script cleans up all the instantiated VMs, including their *virtio-rdma* configuration.

The proposed workflow is completely transparent to the user, and carries out the same procedure for both jobs on VM and on bare metal node. Moreover, it skips the loading of extra modules, because in a virtualized environments applications images are packaged with all necessary dependencies.

5. CONCLUSION AND FUTURE WORK

In this paper, we present a design of a lightweight paravirtualized RDMA solution, called *virtio-rdma*, that implements a new frontend driver compatible with the *HyV* host driver with more advanced features, for example simple context switch and shared memory support. It is designed in a much simpler way than the original *HyV* architecture, due to the characteristics of *OSv* on the guest driver side. Our *virtio-rdma* approach supports *OSv* in its current state of development. However, we simplified the dependencies of *OFED* kernel modules and minimized support of *OFED* user libraries to provide API linkage to user application, allowing for future adaptation in any similar library OS *unikernels* or even Linux.

Furthermore, we proposed a concrete solution of adopting *virtio-rdma* in HPC environments with an extended *Torque*, allowing end-users to easily explore our para-virtualization solution without any prior domain knowledge.

Future work comprises the plan to evaluate and compare the I/O performance of the proposed implementation, on the extended virtualized HPC environments, to the traditional bare metal execution.

6. ACKNOWLEDGEMENT

This work is supported by the MIKELANGELO project (grant agreement No. 645402), which is co-founded by the European Commission under the Information and Communication Technologies (ICT) theme of the H2020 framework programme (H20202-ICT-07-2014: Advanced Cloud Infrastructures and Services).

7. REFERENCES

[1] Introduction to InfiniBand. http://www.mellanox.com/pdf/whitepapers/IB_Intro_WP_190.pdf.

[2] Prologue and Epilogue Scripts. http://docs.adaptivecomputing.com/torque/3-0-5/a.gprologueepilogue.php.

[3] RDMA Aware Networks Programming User Manual. https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf.

[4] RDMA over Converged Ethernet. https://en.wikipedia.org/wiki/RDMA_over_Converged_Ethernet.

[5] Torque Resource Manager. http://www.adaptivecomputing.com/products/open-source/torque.

[6] vhost-user. https://github.com/qemu/qemu/blob/master/docs/specs/vhost-user.txt.

[7] R. Bott. RDMA Protocol verbs Specification. *Igarss 2014*, (1):1–5, 2014.

[8] D. R. Engler, M. F. Kaashoek, and J. O'toole. Exokernel: An operating system architecture for application-level resource management. pages 251–266, 1995.

[9] S. Hefty. Rsockets. In *2012 OpenFabris International Workshop, Monterey, CA, USA*, 2012.

[10] A. Kivity, D. Laor, G. Costa, and P. Enberg. OSv — Optimizing the Operating System for Virtual Machines. *Proceedings of the 2014 USENIX Annual Technical Conference*, pages 61–72, 2014.

[11] C. Macdonell, X. Ke, A. W. Gordon, and P. Lu. Low-latency, high-bandwidth use cases for nahanni/ivshmem. 2011.

[12] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library operating systems for the cloud. *SIGPLAN Not.*, 48(4):461–472, Mar. 2013.

[13] J. Pfefferle, P. Stuedi, A. Trivedi, B. Metzler, I. Koltsidas, and T. R. Gross. A Hybrid I/O Virtualization Framework for RDMA-capable Network Interfaces. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '15, pages 17–30, New York, NY, USA, 2015. ACM.

[14] A. Ranadive and B. Davda. Toward a Paravirtual vRDMA Device for VMware ESXi Guests. pages 1–18, 2015.

[15] R. Russell. virtio: Towards a De-Facto Standard For Virtual I / O Devices. *ACM SIGOPS Operating Systems Review*, 42:95–103, 2008.

[16] M. Technologies. Mellanox's messaging accelerator. https://github.com/mellanox/libvma.

Author Index