Proceedings of the

First Workshop on Co-Scheduling of HPC Applications (COSH 2016)

Prague, Czech Republic, January 19, 2016
Co-located with HiPEAC 2016

Workshop Co-Chairs

Carsten Trinits and Josef Weidendorfer

(This page intentionally left blank)

Contents

(This page intentionally left blank)

Foreword

First Workshop on Co-Scheduling of HPC Applications (COSH 2016)

Prague, Czech Republic, January 19, 2016
Co-located with HiPEAC 2016

Welcome from the Organisers

Welcome to COSH 2016, the first workshop on Co-Scheduling of HPC Applications! The workshop is held in conjunction with the HiPEAC 2016 conference in the wonderful medieval city of Prague, Czech Republic. Holding the workshop for the first time, we received nine submissions from four different countries. Out of these, the programme committee selected six high quality papers for publication. In the process, each paper received three reviews. In addition, we very much appreciate a keynote by Christopher Dahnken of Intel Corporation on recent hardware support and latest developments in microprocessor technology within the co-scheduling context.
We are grateful for the outstanding job of our programme committee which managed to return all reviews within the very tight time constraints. We hope this workshop will lead to new insights and fruitful discussions around its relatively novel topic within the context of High Performance Computing.

Munich, December 2015

Carsten Trinitis, Josef Weidendorfer
Workshop Co-Chairs

Workshop Description

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.

Main topics

Exascale architectures, supercomputers, scheduling, performance sensors, energy efficiency, task migration

A Resource-Centric Application Classification Approach

Alexandros-Herodotos Haritatos
School of ECE, NTUA
aharit@cslab.ece.ntua.gr

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

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

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

ABSTRACT

In this paper we present a resource-centric application classification approach that monitors data flow along the path from main memory to the cores to locate spots of high resource utilization and potential resource contention. We designate three application classes, i.e. streaming applications, last-level cache sensitive applications and applications that restrict their activity either within the cores or in the private levels of the memory hierarchy. Our classification scheme can form the basis for a number of preliminary prediction models that are capable of predicting application interference with high accuracy.

Keywords

Application classification, contention-aware, prediction

1. INTRODUCTION

Chip Multiprocessors (CMPs) encapsulate several cores that share a number of critical resources such as memory links, cache memory and memory controllers. Applications running simultaneously may compete for these resources, leading to resource contention and eventually to performance degradation. Contention on shared resources may even victimize the threads of a single parallel application moderating, or even mitigating benefits of parallel execution. Beyond performance degradation that can be severe in many cases, performance instability is another critical issue especially in computing environments where performance guarantees need to be maintained. To address the problems created by resource contention, researchers have proposed modifications in hardware (e.g. cache partitioning [12]) or software (e.g. contention-aware scheduling [3, 10, 14]). In all cases, a classification scheme is employed that utilizes information regarding shared resource utilization [2, 3], application resource footprints [6, 7, 15] or co-execution behavior [4, 8].

COSH 2016 Jan 19, 2016, Prague, CZ

DOI: 10.14459/2016md1286948

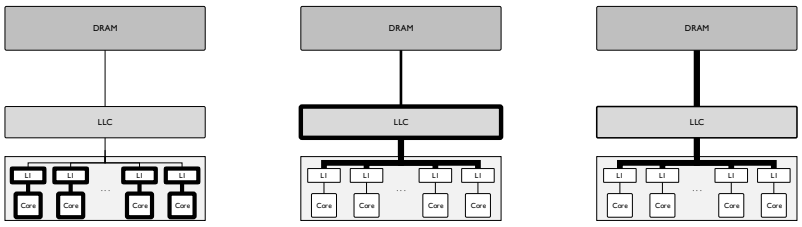
Contention-mitigating mechanisms build upon this knowledge to take better optimization decisions. For example, contention-aware schedulers rely on application classification that predicts interference of co-execution scenarios. Cache utilization patterns [6, 7, 15], LLC miss rate [3], memory link bandwidth [2, 10], contentiousness and sensitivity [14] have been proposed towards this direction. Ultimately, these schedulers aim at reversing the effects of contention on QoS, throughput [16] or energy consumption [10].

The accuracy of the classification scheme in the prediction of application co-execution penalties is one of the most critical factors for a co-scheduling framework. However, most of these schemes capture applications' activity in a limited part of the architecture, i.e. either memory link or last level cache (LLC). Thus, they cannot infer application utilization at each specific hardware resource. In this paper we present a classification scheme based on previous work [5] that inspects the entire memory hierarchy from main memory down to the compute cores and captures data flow and resource utilization. This information is utilized to understand application behavior and predict interference problems. In this way we are able to spot contention on both memory link and LLC. Our classifier distinguishes between three application classes: streaming applications; cache intensive applications; and applications that exhibit no significant activity on the shared resources of the system. We demonstrate interactions between applications from various classes are adequately predictable through simple prediction models, therefore could be applied in an optimized scheduling mechanism.

The rest of the paper is organized as follows: Section II presents our classification approach and Section III presents experimental results. In Section IV we discuss related work. Finally, Section V concludes the paper and discusses ideas for future work.

2. CLASSIFICATION

Our application characterization approach has the following objectives: a) to be capable of locating contention on both the shared memory link and LLC simultaneously, in order to more accurately capture the application's resource utilization pattern, b) to rely solely on information that can be collected at runtime from the existing monitoring facilities of modern processors (requiring no additional hardware support) in order to increase its applicability as much as possible, and c) to be sufficiently fast in order to be capable of supporting prompt decisions, required in dynamic execu-



(a) class N (b) class C (c) class S

Figure 1: Activity in application classes

tion environments where applications enter, exit and change behavior frequently.

2.1 Application classes

In our analysis, the following three application classes are relevant:

Class N: Applications that restrict their activity either within the core or in the private caches of the core. The members of this class create no contention to the shared system resources. The class includes applications with heavy computations, very small working sets or optimized data reuse that can be serviced by the private caches.

Class C: Applications with high activity on the shared LLC. This is a wide class including members with a combination of main memory access and LLC data reuse, or members with varying characteristics, such as those that operate on small data sets with heavy reuse, optimized code for the LLC (e.g. via cache blocking with a block size fitting the LLC), or latency-bound applications that make irregular data accesses and benefit a lot from LLC hits.

Class S: Applications of this class have a stable data flow on all links of the memory hierarchy. This class typically includes applications that perform streaming memory accesses on data sets that largely exceed the size of the LLC, or have either no reuse or large reuse distances Although they fetch data on the entire space of the LLC, they do not actually reuse them either because their access pattern does not recur to the same data, or because they have been swept out of the cache. No level of cache memories helps S applications accelerate their execution. Thus, they tend to pollute all levels of caches. Figure 1 indicates the activity spot of each class.

2.2 Classification method

Having defined the application classes, we need a concrete method to perform the classification using runtime statistics. The core idea is to inspect the data path from main memory down to the core to locate links with high utilization. We have focused only on the stream flowing towards the core, as we have empirically found that this direction concentrates the largest portion of contention. Figure 2 illustrates this idea.

Our classification method implements the decision tree shown in Figure 3. We follow a hierarchical approach in the classification. First, we look at the application activity in L1 cache. No activity in L1 means that data used from the entire path of memory are too few, indicating that the application's activity is restrained within the core. Applications that exhibit this attribute are classified as N. If we are unable to locate reuse at any level of the cache hierarchy, then the application has a streaming attitude and is marked as S. As cache reuse factor, we use the ratio $CR_i = \frac{B_{in_{i-1}}}{B_{in_i}}$. The rationale is simple: if data flow out of a cache towards the core with a much higher rate than they flow in, then we can safely assume that reuse is present. We empirically set a threshold of 2 to designate reuse. If there is cache reuse, reuse location needs to be examined. If reuse is higher in the private caches, then the application is classified as N,

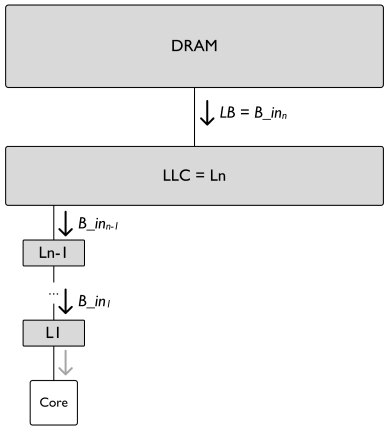


Figure 2: Inspected data flow in the memory hierarchy

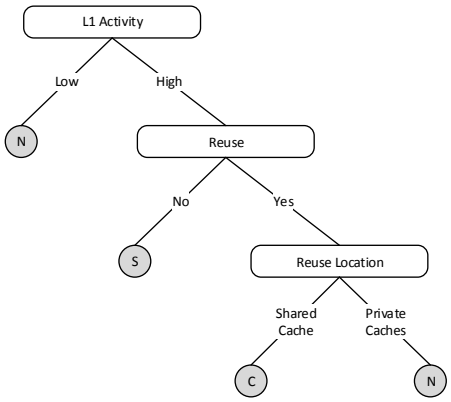


Figure 3: Decision tree for application classification

else the application is classified as C (as the dominant reuse is on the LLC).

2.3 Co-execution effects

Despite the fact that inside each class one may find applications with quite different execution patterns, the classes themselves can be used to capture the big picture of how applications access common resources and of co-execution interference between applications. In the following we denote xy the co-execution of an application from class x with an application from class y. We use $*$ as a wildcard for "any class". Here is what we expect from the co-execution of all combinations:

$N - *$: As applications from class N do not activate in any shared resource, this co-execution does not create interference with any of the applications.

$S - S$: Applications compete for the memory link. The contention pattern in this case indicates that the shared resource which in this case is the memory link bandwidth is divided (not necessarily equally) between the competing applications.

$S - C$: As S applications tend to pollute caches, C applications may suffer from the coexistence with S applications. In cases of high pace of data fetching their co-execution can be catastrophic for the C application. On the contrary, S applications having low pace of data fetching may cause no harm to C applications. The streaming nature of S applications causes data that are potentially heavily reused by C applications to be swept out of the LLC rapidly, enforcing them to access main memory. This contention pattern can lead to dramatic slowdowns for C applications. On the other hand, S applications suffer no severe penalty from co-execution.

$C - C$: This is the most difficult to predict co-execution. In the general case, we expect cache organization and replacement policies to be able to handle adequately high activity from different applications on the shared LLC. However, if both exhibit similar data access patterns, contention is expected to be high. To go deeper into the class and better understand possible interactions, one would require information on the data allocated to each application on the LLC and the access pattern. This would require either information from static analysis, or additional hardware support.

2.4 Interference prediction models

A successful classification model is the core of an accurate prediction model that will ultimately be used by resource management mechanisms such as contention-aware schedulers. Interference prediction is the most useful one, since it can be directly applied to scheduling policies. Interference effects are more complicated as the number of applications running simultaneously, increases. To confirm the validity of the expected co-execution effects, co-execution scenarios have to be simple. Therefore these scenarios consist of two applications competing for shared resources of the same package at the same time. There are six possible scenarios derived from our classification model, described next. We denote D_{X-Y} the degradation of application X when executed with application Y.

$S - S$: In this case both applications compete for the memory link. As the available bandwidth of the link is limited, applications need to share it. Even S applications that are running alone may have a small LLC reuse, this will be lost due to the high pace of LLC pollution, that their competitor causes. Because of this, we expect that the degradation will be expressed as:

$$D_{S-S} = \frac{BW_{max}}{BW^{in}_{mem \to LLC} + BW^{co}_{mem \to LLC}}$$

where BW is the bandwidth, in denotes the inspected application and co the co-executed one.

$S - C$: S applications invalidate C applications' cached data. We expect that as the bandwidth of S applications increases, the consequences will be more dramatic for C applications. The wide pallet of different memory access patterns demands observation on a large number of parameters. However, as an initial attempt, we experiment with a linear function of degradation with the bandwidth of the co-executed application. Thus, our prediction model is expressed as:

$$D_{C-S} = \alpha \times BW^{co}_{mem->LLC} + \beta$$

On the other hand, we expect S applications to suffer minimal pain, and we set $D_{S-C} = c_1$, with c_1 taking a value close to 1.

$C - C$: As cache hardware is expected to efficiently handle the conflicts in this case, degradation prediction is defined as a constant value taken from the average value of our experimental results, i.e. $D_{C-C} = c_2$

$N - S$, $N - C$, $N - N$: N applications do not affect and are not affected by other applications. In all of these cases all the degradation's predictions are taken as constants, i.e. $D_{S-N} = c_3$, $D_{C-N} = c_4$ and $D_N = c_5$, again with c_3, c_4 and c_5 taking values close to 1.

We calculate parameters α, β and $c_1 \ldots c_5$ with regression utilizing experimental data from the co-execution of our application test suite, which is presented in Section 3.2. We split our set in sliding windows of 80% training and 20% testing sets.

3. EVALUATION

3.1 Experimental Platform

The evaluation of the classification scheme is performed on an Intel® Xeon® CPU E5-4620. The architectural details are presented in Table 1. All the available hardware prefetchers are enabled whereas Hyperthreading and Turbo Boost are disabled. The platform runs Debian Linux 6.0.6 with kernel 3.7.10.

Cores	8
L1	Data cache: private, 32 KB, 8-way, 64 bytes block size Instruction cache: private, 32 KB, 8-way, 64 bytes block size
L2	private, 256 KB, 8-way, 64 bytes block size
L3	shared, 16 MB, 16-way, 64 bytes block size
Memory	64 GB, DDR3, 1333 MHz

Table 1: Processor details

In order to monitor the applications and acquire their profile, we employ hardware performance counters to collect performance data. Specifically, we use `UNHLT_CORE_CYCLES, INSTR_RETIRED, L1D.REPLACEMENT, L2_LINES.IN`. Furthermore, we use `OFFCORE_REQUESTS (0xB7, 0x01; 0xBB, 0x01)`.

3.2 Experimental Setup

In our evaluation, we populated all four application classes selecting benchmarks from a variety of multithreaded suites

Name	Source	Details	$B_{in_{3=LLC}}$ (GB/sec)	$CR_{llc} = \frac{B_{in_2}}{B_{in_3}}$	$CR_{l2} = \frac{B_{in_1}}{B_{in_2}}$	IPC	class
jacobi_l	polybench	large data set	10.321	1.50	1.02	0.559	S
stream	[9]	array size = 50000000, offset = 0, NTIMES = 10, kernel = TRIAD	11.646	1.00	1.03	0.724	
bt	NAS	class A	8.566	0.97	1.36	0.813	
fw	polybench	small data set	7.468	1.02	1.16	1.903	
Dynprog	polybench	custom data set	8.220	0.98	1.54	1.384	
Mvt	polybench	custom data set 1000	0.006	4.21	565.29	0.455	C
Mvt	polybench	custom data set 6000	2.274	0.92	8.31	0.343	
atax	polybench	large data set	2.426	0.82	8.61	0.336	
cg	NAS	class B	5.955	1.37	4.38	0.728	
syr2k	polybench	large data set	5.883	2.07	2.00	1.919	
gemver	polybench	large data set	2.873	0.82	7.23	0.440	
cholesky	polybench	large data set	0.156	0.85	195.56	1.876	
pchase	[1]	A pointer-chasing benchmark with data set fitting in LLC.	0.135	1.00	222.01	0.148	
lu_t	inhouse	Classic implementation of tiled LU decomposition with data set fitting in LLC.	0.009	8.67	178.91	1.635	
jacobi_s	polybench	small data set	0.534	1.15	35.17	1.209	
fw_t	inhouse	Implementation of tiled Floyd-Warshall algorithm from [13]. Data size fits in LLC.	0.063	9.92	34.45	2.181	
ep	NAS	class A	0.000	0.97	2644.41	0.746	N
mm_t	inhouse	Classic implementation of tiled Matrix Multiplication with data set fitting in LLC.	0.030	17.82	5.68	2.933	
blackscholes	PARSEC	large data set	0.186	0.65	5.63	1.732	

Table 2: Application suite

and inhouse implementations of classic kernels. Table 2 presents our application suite, together with key metrics used to classify each application according to our scheme. The test platform includes one shared LLC and two private caches (L_1 and L_2). In order to evaluate our classification scheme, we co-execute all possible pairs of applications, totally 361 pairs. We allocated half of the available cores to each application. If an application terminates, it is respawned in order to keep contention at the same level.

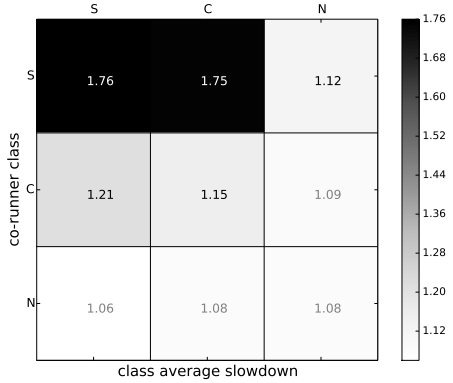


Figure 4: Average application slowdown due to co-execution at the class level. The horizontal dimension represents the slowdown imposed by each class, while the vertical dimension shows the slowdown suffered.

Figure 4 provides a class-level view of average slowdowns of each class measured. We may observe that our classification scheme is able to capture the general trend in interference penalties. In particular, we observe that whenever a N application is involved, the interference overhead is negligible. On the other side of the spectrum, the most contentious pairings arise in S-S and S-C collocations.

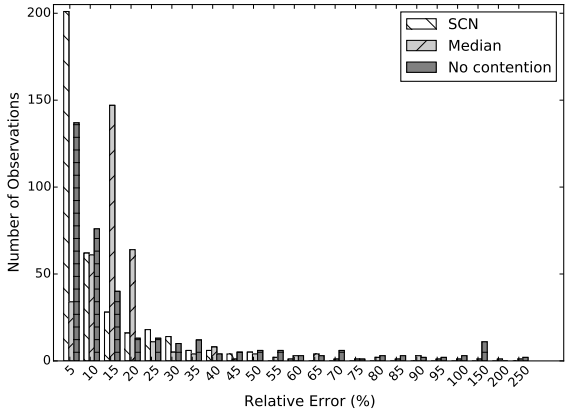


Figure 5: Relative errors for the SCN, 'median' and 'no contention' predictors.

Figure 5 shows the absolute values of the relative errors for all co-executions for our predictor (SCN) and two naive predictors, i.e. the 'median' predictor that always predicts the median of all cases (1.27) the 'no contention' predictor that neglects the effects of contention and always predicts a degradation factor of 1.05. Note that we favored the naive predictors by using the same training and testing sets (i.e. we did not split training and testing sets in 80% and 20% subsets as we did for SCN). We may notice that SCN achieved a prediction error of less than 5% for 55% of the co-executions and less than 30% for 94% of the co-executions. On the other hand, the naive predictors provided guesses

with significantly lower accuracies, i.e. the 'median' predictor had a prediction error of less than 5% for 9% of the co-executions and less than 30% for 89% of the co-executions, and the 'no interference' predictor had a prediction error of less than 5% for 38% of the co-executions and less than 30% for 80% of the co-executions.

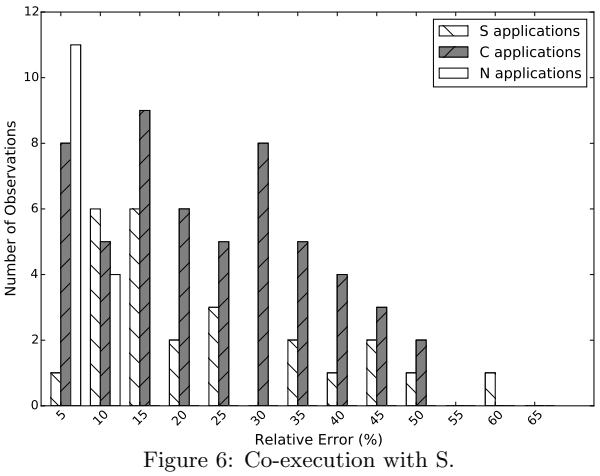


Figure 6: Co-execution with S.

Figure 6 shows the absolute values of the relative errors of SCN prediction for applications of all classes when they are co-executed simultaneously with S class applications. We may notice that despite their simplicity, the prediction models for C–S and S–S co-executions are able to capture the general degradation penalties and keep the prediction errors lower than 30% for 72% of the co-executions. On the other hand, as expected co-execution of S with N applications is predicted with high accuracy.

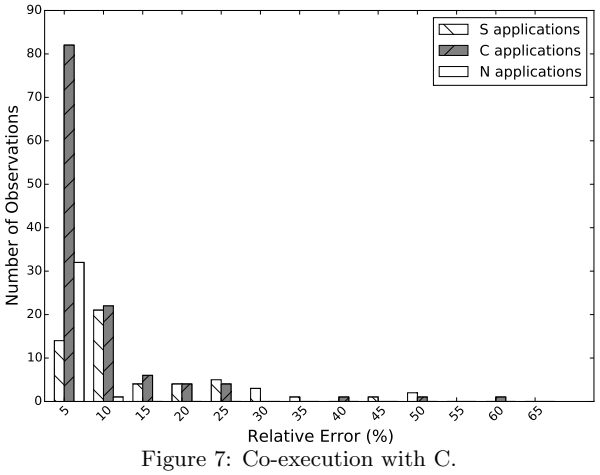


Figure 7: Co-execution with C.

Figure 7 shows the absolute values of relative errors of SCN prediction of all classes when executed with applications from the C class. Quite interestingly, our simple constant predictor for the C–C co-execution keeps errors lower than 30% for 97% of the co-executions, showing that in our experimental scenarios cache sensitive applications tend to share gracefully the shared LLC cache. We also note that S and N co-executions with C applications are accurately

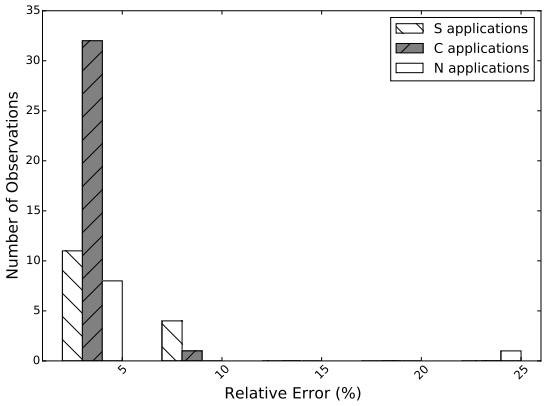


Figure 8: Co-execution with N.

predicted. Finally, Figure 8 shows the absolute values of relative errors of our predictor for applications of all classes when they are co-executed simultaneously with N class applications. Clearly, this is an easy to predict scenario since N applications indeed cause no substantial harm to the their co-executors, a fact that is captured by our predictor.

4. RELATED WORK

Various characterizations schemes have been proposed in the past. Lin et al. [7] proposed a scheme that partitions the LLC cache between two programs using cache coloring. The scheme is employing a program classification based on each program's performance degradation when running using a 1MB cache compared to running using a 4MB cache. Looking also at cache partitioning, Moreto et al. [11] proposed two metrics, namely $w_{P\%}$ and $w_{LRU(th_i)}$. The first metric refers to the number of ways needed by a benchmark to reach at least the P% of its maximum IPC, while the second one refers to the ways allocated to each thread by LRU when two benchmarks are executed together. Based on these two metrics applications were classified in one of three different classes.

Xie and Loh [15] proposed an animalistic approach of the application classification problem. All applications may belong to one of four classes, named Turtle, Sheep, Rabbits and Tasmanian Devils. Applications that do not make much use of the LLC are turtles, while applications that make use of the LLC but are not sensitive to the number of ways allocated to them belong to the sheep group. Rabbits are applications that are very sensitive to the ways allocated to them, and, finally, devils are the applications that make use of the LLC while having very high miss rates.

Jaleel et al. [6] categorize applications in four classes. Core Cache Fitting (CCF) are applications with a dataset that fits in the lower levels of the memory hierarchy and do not benefit from the shared level of cache. On the other hand, LLC Thrashing (LLCT) applications have a data set bigger than the available LLC. Under LRU, these applications degrade performance of any co-running application that uses the LLC. LLC Fitting (LLCF) applications require almost the whole LLC and their performance is severely impacted if there cache trashing occurs. Finally, LLC Friendly (LLCFR) applications are ones that even though they could improve their performance using more cache resources, they do not

suffer significantly when these resources are reduced.

Finally, Tang et al. [14] are employing two metrics for each application, namely Contentiousness and Sensitivity. Based on these, applications belong to one of the following four classes: 1) contentious and sensitive; 2) not contentious and insensitive; 3) contentious but not highly sensitive; or 4) not highly contentious but sensitive. Contentiousness and sensitivity are statistical numbers but, as authors claims, can be easily calculated through performance counters.

5. CONCLUSIONS AND FUTURE WORK

In this paper we presented a resource-centric application classification scheme that monitors data traffic across the entire memory hierarchy, using existing hardware monitoring mechanisms. We base a number of preliminary interference predictors on the classification scheme and evaluate them on a set of parallel applications. The prediction accuracy is very promising and sets a solid basis to support contention-mitigating mechanisms. As a future work, we intend to extend this work in the following directions: a) augment our prediction approach to minimize errors and mispredictions, b) apply this model in scenarios where more than two applications run concurrently c) apply this work to a contention-aware application scheduling environment.

6. ACKNOWLEDGMENTS

This research was funded by project I-PARTS: Integrating Parallel Run-Time Systems for Efficient Resource Allocation in Multicore Systems (code 2504) of Action ARISTEIA, co-financed by the European Union (European Social Fund) and Hellenic national funds through the Operational Program Education and Lifelong Learning (NSRF 2007-2013).

7. REFERENCES

[1] pChase benchmark. https://github.com/maleadt/pChase.

[2] Major Bhadauria and Sally 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.

[3] Sergey Blagodurov, Sergey Zhuravlev, and Alexandra Fedorova. Contention-aware scheduling on multicore systems. *ACM Trans. Comput. Syst.*, 28(4):8:1–8:45, December 2010.

[4] Christina Delimitrou and Christos 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.

[5] Alexandros-Herodotos Haritatos, Georgios Goumas, Nikos Anastopoulos, Konstantinos Nikas, Kornilios Kourtis, and Nectarios 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.

[6] Aamer Jaleel, Hashem H. Najaf-abadi, Samantika Subramaniam, Simon C. Steely, and Joel 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.

[7] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *International Symposium on High Performance Computer Architecture*, pages 367–378, 2008.

[8] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *MICRO*, pages 248–259, 2011.

[9] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[10] Andreas Merkel, Jan Stoess, and Frank 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.

[11] Miquel Moretó, Francisco J. Cazorla, Alex Ramírez, and Mateo Valero. Explaining dynamic cache partitioning speed ups. *IEEE Computer Architecture Letters*, 6(1):1–4, 2007.

[12] Konstantinos Nikas, Matthew Horsnell, and Jim D. Garside. An adaptive bloom filter cache partitioning scheme for multicore architectures. In *ICSAMOS*, pages 25–32, 2008.

[13] Joon-Sang Park, Michael Penner, and Viktor K. Prasanna. Optimizing graph algorithms for improved cache performance. *IEEE Transactions on Parallel Distributed Systems*, 15(9):769–782, 2004.

[14] Lingjia Tang, Jason Mars, and Mary Lou 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.

[15] Yuejian Xie and Gabriel Loh. Dynamic classification of program memory behaviors in CMPs. In *Proceedings of the 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2008.

[16] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Computing Surveys*, 45(1):4:1–4:28, December 2012.

Dynamic Process Management with Allocation-internal Co-Scheduling towards Interactive Supercomputing

Carsten Clauss
ParTec Cluster Competence Center GmbH, Munich
clauss@par-tec.com

Thomas Moschny
ParTec Cluster Competence Center GmbH, Munich
moschny@par-tec.com

Norbert Eicker
Jülich Supercomputing Centre
Forschungszentrum Jülich
n.eicker@fz-juelich.de

ABSTRACT

Heading towards exascale, the challenges for process management with respect to flexibility and efficiency grow accordingly. Running more than one application simultaneously on a node can be the solution for better resource utilization. However, we believe that this approach of co-scheduling can also be the way to go for gaining a degree of process malleability and dynamicity that can enable some kind of interactivity also in the domain of high-performance computing. In this paper, we present the recent advances made in this respect within ParaStation MPI, a high performance MPI library supplemented by a complete framework comprising a scalable and dynamic process manager. The paper presents four new scheduling policies, implemented in ParaStation MPI, for starting multiple MPI sessions concurrently and interactively within a single allocation of nodes. The features of these policies are detailed and evaluated by applying the Dynamic Job Scheduler Benchmark (djsb), a tool developed by the Barcelona Supercomputing Center especially for measuring interactivity and dynamicity metrics.

Keywords

Scheduling Policies, Co-Scheduling, Process Management, Interactive Supercomputing, High-Performance Computing

1. INTRODUCTION

Since the beginning of the pre-exascale era, there has been a rising demand for the support of interactivity and malleability also in the domain of high-performance computing. Such a support will allow supercomputer users to interact with their running applications, for example, in order to steer the progress of a simulation during runtime. It is widely believed that—besides some kind of a conceivable real-time interaction, for example, via graphical user interfaces for in-situ visualization—on large-scale supercomputers, such an interaction will primarily be conducted via additional applications to be started concurrently on the user's demand. (For proving this statement refer, for example, to the Technical Requirement Document [1] of Pre-Commercial Procurement (PCP) announcement issued by the Human Brain Project (HBP): In the context of the HBP it is foreseen to build up (pre-) exascale supercomputing systems featuring interactivity for large-scale brain simulations.[1])

According to this, each job will consist of multiple job steps (potentially divisible into primary and secondary ones) that may be launched interactively and that in turn can interact among each other. So, for instance, a user may run a large and long lasting simulation application, which then can interact during runtime with intermediately started auxiliary applications. Such secondary applications, which are then to be co-located with the primary application (either within its existing allocation or by requesting further resources) could then attach and interact with the long-running simulation in order to track and even govern its evolution. By co-locating the processes within the existing allocation, the linked applications can then take advantage especially of data and communication locality. Conceivable use case scenarios are, for example, visualization pipelines and the online post-processing of intermediate simulation steps as well as computational workflows and coupled codes for providing further input parameters during runtime of the primary simulation. Since such user interventions as well as the reactions made by the applications based on their interaction are not predictable, a dynamic and continuous sub-partitioning of the allocated resources is the consequence.

Such an allocation-internal co-scheduling may on the one hand aim at optimal system utilization. On the other hand, as user interactivity is also a matter of responsivity, the scheduling policy may focus on some kind of priorities. At this point, the above-mentioned demand for job malleability comes into play: Such a malleability comprises the question of the actual starting order of concurrently launched MPI sessions, the related question of a dynamic process-to-core assignment, the demand for the ability to reduce or increase the number of cores devoted to a certain MPI session, and the request for the possibility to suspend a whole MPI session on a temporary basis. Although these aspects are in the first instance relevant to the job-internal process management, at least the issue of reducing or increasing the number of processes and/or cores of an MPI session may also involve the system's higher-level resource manager.

For clarifying the different terms used in this paper, Figure 1 should illustrate the hierarchy of entities that have to be taken into account for the overall resource manage-

COSH 2016 Jan 19, 2016, Prague, CZ

DOI: 10.14459/2016md1286949

[1] www.humanbrainproject.eu

ment: The whole system is usually a *cluster* composed of *nodes*, while each node commonly features multiple *cores* resp. hardware threads. The user can request for a set of nodes/cores in terms of a *job allocation* for starting multiple parallel *applications* in terms of concurrent MPI *sessions* within.

Taken as a whole, the comprehensive management system then forms a three-tier hierarchy: At node level, the Linux scheduler manages the processes and threads, potentially governed by a predefined process pinning scheme. At job level, the local process manager handles the process-to-node/core assignment by starting, controlling and monitoring parallel MPI processes within the allocation devoted to the respective job. Finally, at cluster level, an outer resource manager maintains the different job queues of the batch system and performs the overall resource assignments granted by a job scheduler.

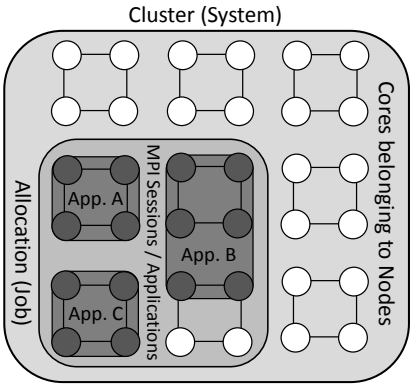


Figure 1: Naming of tiers in the system hierarchy

2. PARASTATION MPI

ParaStation MPI is an open-source MPI library, developed and supported by ParTec GmbH[2] under the umbrella of the ParaStation Consortium—an alliance consisting of ParTec, the University of Wuppertal, the Karlsruhe Institute of Technology (KIT), and the Jülich Supercomputing Centre (JSC). This MPI library has already proven to scale very well on large parallel production systems: In June 2009, ParaStation pushed the 3,288 node JuRoPA cluster at the JSC with more than 25,000 MPI processes to No. 10 in the world according to the Top 500 list. Furthermore, its successor, the JURECA system at the JSC with 49,476 cores, has just recently entered the November 2015 list at No. 50—again propelled by ParaStation [2].

The ParaStation MPI library (psmpi) is embedded into a complete framework for providing state-of-the-art cluster-based supercomputing [3]: Besides a robust and efficient cluster middleware, the ParaStation software suite also comprises sophisticated administration components like the ParaStation *ClusterTools* (for provisioning and maintenance), the ParaStation *HealthChecker* (for automated error detection and integrity checking) and the ParaStation *TicketSuite* (for analyzing and keeping track of issues).

psmpi is fully MPI-3 compliant [5], including support for the recent additions to the RMA interface, and also supports the MPICH-related Process Manager Interface (PMI) [4] as well as MPI-2 compliant dynamic process spawning—a feature long time neglected by other MPI implementations, but efficiently used by psmpi in the context of the Dynamical Exascale Entry Platform (DEEP) project [6], a project funded from 2011 to 2015 by the EU 7th Framework Programme.[3]

2.1 The Process Management System

The management facility of psmpi, called ParaStation Management (psmgmt), offers a complete process management system that can in turn be combined with an outer and more generic resource manager together with a batch queuing system plus job scheduler like TORQUE/MAUI or SLURM. The process management of psmgmt includes the creation of processes on remote nodes, control of the I/O channels of the remotely started processes, and the management of signals across node boundaries.

Since psmgmt knows about the dependencies between the processes and threads building a parallel session on a number of nodes of the cluster, it is able to take them respectively into account. That way, processes are no longer independent but form an entity in the same sense as the nodes are no longer independent computers but form a cluster of nodes as a self-contained system. This feature of psmgmt for handling distributed processes as a single unit plays an important role especially in the context of job control and allocation-internal scheduling—as it will be detailed later in this paper.

One important key to ParaStation's scalability is its efficient communication subsystem for inter-daemon messages. This subsystem, which uses the implementation of a highly-scalable Reliable Datagram Protocol (RDP), is used for resource monitoring as well as for launching and controlling the parallel processes by means of a network of ParaStation Daemons (psid). So, for example, this subsystem also performs process pinning, I/O forwarding and signal handling, and it ensures a proper resource cleanup after job termination.

2.2 Relation to the Resource Manager

Following a *one daemon per cluster node* concept, the daemon architecture is kept such generic that it can easily be extended by plugins for consolidating various services. Furthermore, this network of daemons also enables third-party services to piggyback their payload on the ParaStation communication subsystem. So, for instance, a TORQUE-related plugin (psmom) and a SLURM-related plugin (psslurm) efficiently replace the native daemons of these resource managers on the compute nodes in a ParaStation environment.

Figure 2 illustrates the orchestration between psmgmt and SLURM, as it is currently employed on the JURECA system at the Jülich Supercomputing Centre. As one can see, the psid together with its psslurm plugin plays the central role regarding process startup and job control on the compute nodes. SLURM itself is designed to operate even in heterogeneous clusters with up to tens of millions of processors and can accept thousands of job submissions per second with a sustained throughput rate of hundreds of thousand jobs per hour. Its direct linkage on JURECA to the network of distributed psids makes this orchestration between SLURM and ParaStation highly scalable and very flexible.

However, in case that the number of computing resources should actually be increased by MPI-2 compliant dynamic

[2]www.par-tec.com

[3]www.deep-project.eu

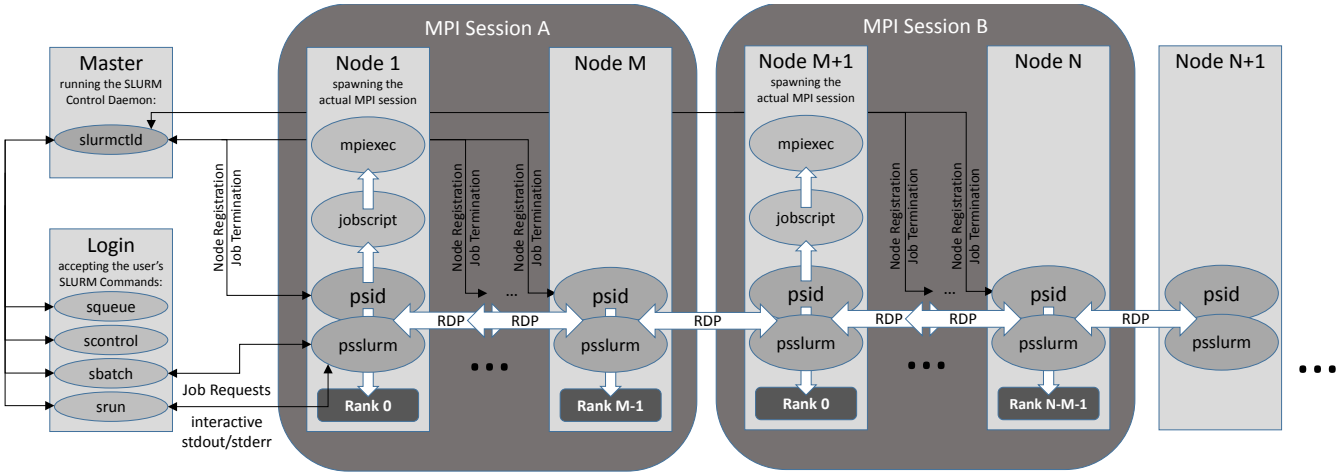


Figure 2: Orchestration between SLURM and ParaStation Management (psmgmt) via its psslurm plugin

process spawning during a job's runtime, the MPI layer would need the capability of requesting such a *post-allocation* of nodes from the resource management system. Such a request has then to be negotiated between the entities, but may very well also be rejected if the requested resources are currently not available. In fact, this feature has recently also been added to the ParaStation environment in the context of the above-mentioned DEEP project: The TORQUE server, which is used together with the MAUI scheduler in the DEEP project, has been enhanced by facilities to receive, queue and process new resource requests by applications via the ParaStation daemon subsystem during the runtime of a job. Moreover, the porting of these features to a SLURM environment, where the psslurm plugin for the psid and an additional plugin for the SLURM resource manager will jointly accomplish these tasks, is currently on-going work.

3. DYNAMIC PROCESS MANAGEMENT

As already stated in the introduction, dynamic process management at job level targets the malleability of MPI processes within an existing job allocation represented by a certain set of nodes currently assigned to a particular user or user group. A first degree of interaction between the user and concurrently running MPI sessions can be achieved by sending operating system signals to and between the respective parallel processes. However, such signals, as they are supported by all customary operating systems, are normally only valid in the context of the local node the operating system is running on. Hence, for supporting signal forwarding even across node borders, the respective middleware—thus, in our case, the local process manager—has to be capable of such a distributed signal handling.

3.1 Signal Handling and Process Pinning

In fact, psmgmt is already capable of doing so and natively takes care for the handling of process signals in a cluster-wide manner. This being so, some kind of runtime interactivity between the user and the started MPI sessions is already possible in this way. So, for example, sending a common `SIGTSTP` to ParaStation's `mpiexec` command would cause the whole respective MPI session to get suspended, whereas a `SIGCONT` can be issued to resume it later on.

Moreover, instead of a complete job preemption, even a temporal reduction of computing resources devoted to an MPI application is possible. According to this approach, the psids get signaled to perform a runtime adjustment of the rank-to-core pinning on each of the job's compute nodes. By means of such a re-pinning, some processes of a given job are moved within the respective nodes in such a manner that an appropriate fraction of oversubscribing for a certain group of cores is achieved.

The main advantage of this approach is that it is still transparent to the application. However, the temporal oversubscription will induce an operating system related scheduling overhead that might disturb the application's internal load-balancing scheme. Therefore, this method of re-pinning and oversubscribing should rather be a temporary measure in order to clear space, for example, for a short-running auxiliary application that is to be attached to a long-running simulation.

Based on these two approaches (suspend/resume and re-pinning of processes), new features for realizing job malleability and interactivity have been implemented recently within ParaStation. According to these approaches, the user can (for example, in context of an interactive SLURM session) launch multiple MPI applications in parallel and/or subsequently. As long as there are enough slots within the current allocation (according to the terminology of ParaStation, these *slots* are the hardware threads that can be assigned to processes resp. software threads), the psid will ensure via pinning that all slots are used exclusively by the assigned processes and threads. However, if the available slots get exhausted, allocation-internal scheduling policies come into play.

Moreover, the same applies to the case of MPI-2 compliant dynamic process spawning if a post-allocation of further nodes gets rejected by the resource manager. Although the initial started number of MPI ranks (this is the initial *world* size) may intentionally be smaller than the number of available slots within the allocation (this is the current *universe* size), hence leaving space in terms of free slots where new MPI processes can be spawned to, if all slots become populated, an oversubscribing or some other allocation-internal scheduling policy has to be applied for further spawn calls.

3.2 Allocation-internal Scheduling Policies

Currently, four of such policies for job- or allocation-internal co-scheduling are implemented in psmgmt: One that just lets the newly started processes wait for getting free slots, one that simply voids the previous exclusiveness of resources and thus allows for oversubscribing of slots, one that follows the suspend/resume approach in such manner that each subsequently started MPI session suspends its respective predecessor for getting free slots, and one that uses re-pinning for a temporal reduction of computing resources concerning the still running predecessor of the newly started application. Moreover, for most of these approaches further sub-policies are conceivable and to some extent already implemented in psmgmt.

All the four policies, as they are detailed in the following paragraphs, can currently be used by means of a wrapper script called `psmpiexec` that extends the common `mpiexec` command as commonly provided by psmpi. However, at this point it should be emphasized that both commands are more or less just user interfaces that can easily be replaced by others—so, for example, by a more SLURM-like `srun` frontend.

The Wait Policy.

According to this policy, any newly started MPI session that can no longer be scheduled into free slots has to wait until one or more of the previous ones gets finished so that enough slots become available again. As this policy still sticks to the original paradigm of preventing any oversubscription, mutual interferences between the sessions should almost be avoided. However, on the other hand, if the interaction between the sessions demand for a concurrent execution, this policy cannot safely be used.

The Surpass Policy.

This policy is based on the suspend/resume mechanism of psmgmt: Every time a new session gets launched within an allocation with already filled slots, the prior session(s) (these are the ones issued by preceding `psmpiexec` within the same allocation) get(s) automatically suspended until the successor becomes finished. The idea behind this policy is that the most recently started session should most probably be the one with the highest priority from the user's point of view. According to this idea, the user can start further sessions (for example for short running auxiliary applications) that then will surpass previously started, long-running ones.

The Overbook Policy.

When this policy is enabled and all free slots are exhausted, all the MPI sessions are run concurrently and in a competitive manner on the nodes and cores of the allocation. The question whether there should still be some kind of a pinning scheme in such an overbooked situation, or if all the processes should then be enabled to flow freely across the cores of their respective nodes, could be then considered as a further sub-policy.

The Sidestep Policy.

This policy is quite similar to the overbook policy. However, the difference is that here the processes of the already running applications are re-pinned in such a manner that the processes of the new session run on their cores exclusively. That means that, in a first instance, only those cores are overbooked where the processes of the preceding MPI sessions are pinned to.

The Spread Option.

Normally, ParaStation places all processes as compactly as possible (with due regard to any threads) onto the nodes. However, in cases where a small number of newly started processes are overloading an allocation already filled up with running applications, it might be beneficial to have the processes of the new session get started on the nodes as widespread as possible. This can be achieved by using an additional *spread* option, which is therefore meaningful together with the *overbook* or the *Sidestep* policy. Using the spread option, the hope is that the already running MPI sessions will not get as much affected by the additionally started processes as it would be the case if the latter were all started on one (or only a few) node(s) of the allocation.

4. EVALUATION OF THE POLICIES

The Dynamic Job Scheduler Benchmark (dsjb) is a tool developed by the Barcelona Supercomputing Center (BSC) for evaluation different scheduling solutions. Although its description[4] as well as its source code[5] are publicly available, this section initially gives a more detailed introduction into the respective benchmarking metric since the knowledge about this seems up to now not very widespread. After this introduction, this section presents some early results gained by applying this benchmark together with the new allocation-internal scheduling policies of psmgmt as detailed in Section 3.2.

4.1 Benchmark Description

The djsb has originally been written and released in the context of the Pre-Commercial Procurement (PCP) of the Human Brain Project (HBP). Its primary purpose is to allow for a performance comparison of the different resource management solutions proposed by different tenders during the PCP. In doing so, the benchmark focuses on *interactivity* and the *dynamicity* of the proposed job scheduling systems. The benchmark actually consists of multiple processes and threads performing the STREAM benchmark [8] in parallel—hence without any considerable communication.

The idea of this benchmark is to let two synthetic applications run concurrently within the same job allocation: one longer running "*simulation*" application and one shorter running "*analysis*" application, both to be modeled by the STREAM executable. The benchmark basically measures the runtime of each of both when they are started separately, as well as the runtime when they are executed concurrently. Based on these durations, the benchmark calculates some reasonable efficiency numbers (the so-called *Simulation/Analysis/Wait Coefficients*) and finally reports a *Dynamicity Ratio* as a product of those three coefficients.

Although the djsb focuses on interactivity, the synthetic applications are issued as MPI sessions automatically by a Python-based benchmarking script that models the hypothetical user of the job allocation by sporadically calling `mpiexec` for the short-running analysis application. Please note here that the document officially describing the bench-

[4] http://pm.bsc.es/~vlopez/files/djsb_doc.pdf
[5] http://pm.bsc.es/~vlopez/files/djsb.tar.gz

mark [7] uses a different nomenclature than we do within this paper: In the official djsb description, the term *job* refers to a single application (rather than to an allocation) and hence to the term of an MPI *session* according to the terminology used in this paper.

The actually measured parameters and performance metrics of the djsb are:

- *Wait Time:* Time that has passed between the session request issued by the "user" (this is the call of `mpiexec`) and the actual start of the respective application.
- *Execution Time:* Time that has passed between session start and its completion. This is hence the effective runtime of the application.
- *Response Time:* Time that has passed between session request by the "user" and its completion. This is hence the sum of Wait and Execution Time.
- *Slowdown:* Performance decrease in terms of the ratio between the actually measured times and the reference scenario where all sessions are run consecutively.
- *Expected vs. Real Slowdown:* While the Expected Slowdown is a pre-calculated value based on theoretical assumptions, the Real Slowdown is the actually observed one.
- *Efficiency Coefficient:* This is just the ratio of Expected and Real Slowdown. Higher values are better.

$$E = \frac{\text{Expected Slowdown}}{\text{Real Slowdown}}$$

- *The Wait Coefficient:* This value is calculated according to the following formula. (Please refer to the official djsb description [7] for a more detailed explanation of this.) Values close to or even greater than 1 are better.

$$W = \frac{\text{Wait Time in static case} + \text{Normalization Constant}}{\text{Wait Time in dynamic case} + \text{Normalization Constant}}$$

- *Dynamicity Ratio:* This is the product of the Efficiency Coefficients as measured for both synthetic applications (the long-running simulation and potentially several short-running analysis sessions)

$$D = E^{\,simulation} \cdot E^{\,analysis} \cdot W^{\,analysis}$$

4.2 Measured Benchmark Results

The benchmark results presented in this section were all obtained on an allocation with 4 nodes and 160 cores in total. The process/thread configuration chosen was as follows for all the benchmark runs:

	no threads	*with threads*
Simulation application	160 procs (40 per node)	32 procs (8 per node, 5 threads per proc)
Analysis application	32 procs (all on one node, or 8 per node with *spread* option)	8 procs x 4 threads (all on one node, no *spread* option used)

Without using the *spread* option, as detailed in Section 3.2, the chosen configuration would overbook the first node of the allocation with 32 analysis processes. In contrast, if the *spread* option is enabled, the 32 processes will be distributed across all 4 nodes of the allocation so that each node would then be overbooked by "only" 8 processes.

The following paragraphs show and briefly discuss the single coefficients and the overall dynamicity results that have been measured with this configuration for the different scheduling policies:

The Wait Policy.

	no threads	*with threads*
Simulation Efficiency:	1.04	1.03
Analysis Efficiency:	2.0	2.03
Wait Coefficient:	0.55	0.55
Dynamicity Ratio:	**1.15**	**1.15**

Since both applications are run within the allocation separately according to this policy, their efficiency (and hence their runtime seen individually) are quite good but the overall Wait Coefficient is relatively bad due to the long waiting time of the analysis application before it gets started. However, the overall measured Dynamicity Ratio is here greater than 1, what means that this policy improves the dynamicity and hence the anticipated capability for interactivity—at least with respect to the metric of the djsb benchmark.

The Surpass Policy.

	no threads	*with threads*
Simulation Efficiency:	0.86	0.86
Analysis Efficiency:	2.0	2.0
Wait Coefficient:	1.0	1.0
Dynamicity Ratio:	**1.72**	**1.73**

The main advantage of this policy is that long pending times of the analysis applications are avoided and that at least the efficiency of the analysis sessions should (and is) as good as in the case of the Wait policy. However, as the simulation application gets completely interrupted, its duration gets extended accordingly so that its efficiency is decreased in comparison to the Wait policy. Moreover, since the analysis sessions are usually not only shorter in runtime, but also smaller in the number of processors used, both policies (Wait and Surpass) may lead to a temporary under-utilization of the allocation.

The Overbook Policy.

	no threads	*with threads*	*spread option (no threads)*
Simulation Efficiency:	0.87	0.91	1.02
Analysis Efficiency:	1.23	0.89	1.02
Wait Coefficient:	1.0	1.0	1.0
Dynamicity Ratio:	**1.07**	**0.81**	**1.04**

In the case of this policy, the simulation as well as the analysis are run concurrently and in a competitive manner within the allocation. This usually means that the processes of the analysis sessions get started (and pinned) onto a subset of those cores where the simulation processes are already running on. Although this policy guarantees that there are no unnecessary idle times of cores during the benchmark's run,

the efficiencies of both the simulation application as well as the analysis application are liable to get impaired due to the temporary overload.

The Sidestep Policy.

	no threads	*with threads*	*spread option (no threads)*
Simulation Efficiency:	0.87	0.87	1.34
Analysis Efficiency:	1.61	1.5	1.13
Wait Coefficient:	1.0	1.0	1.0
Dynamicity Ratio:	**1.41**	**1.3**	**1.5**

These results show that this policy gains a very good Dynamicity Ratio, but when looking at all four policies it becomes clear that the Surpass policy gains the best results. However, it has to be emphasized, that especially the Surpass policy does not allow for an MPI-based interaction between both sessions via message-exchange due to the fact that the simulation session actually gets suspended during the runtime of the analysis application.

In fact, most of the efficiency coefficients are usually expected (at least in theory) to be in the range of [0,1] because this would represent the case when the applications share some of their resources at some point in time. On the other hand, the coefficients are greater than 1 when an application runs with more resources than expected—like in a session serialization.

However, according to the benchmark results we have measured and presented here, all four of the new allocation-internal scheduling policies would improve the dynamic behavior and thus the capability for interactivity. All in all, this indicates two facts for us:

1st: The metric of the djsb benchmark (here especially the calculation of the Expected Slowdown as well as the applying of some "magic" Normalization Constants) seems to be not very well balanced for all scenarios. However, it has to be emphasized that for the HBP-PCP, the benchmark scenarios are well-defined and differ from the process/thread configuration used for our measurements—it is most likely that the internal benchmarking parameters of the djsb are tailored to those configurations as given by the HBP-PCP.

2nd: Since the djsb totally neglects the actually required message-exchange between the concurrent sessions, the benchmark can only give a first hint for the interactive behavior of a system, but cannot really judge about complex interactivity scenarios as they are envisaged for future supercomputing systems. On the other hand, it is in the nature of things that benchmark scenarios have to tend to simplify things in order to make their results more conferrable.

5. CONCLUSION AND OUTLOOK

In this paper, we have presented recent advances made for ParaStation MPI (psmpi) and its process manger (psmgmt) with respect to co-scheduling and process malleability at job level. While co-scheduling is frequently associated with a means for better resource utilization, the approaches presented in this paper are primarily not so much *resource*-centric but rather *user*-centric, as they focus on *interactivity*. In doing so, four new policies for scheduling of concurrent MPI sessions within a single interactive job allocation have been presented and evaluated by means of the Dynamic Job Scheduler Benchmark (djsb). It turned out that (at least according to the metric used by the djsb) all four new policies help to improve the desired scheduling behavior towards malleability and interactivity. However, at the same time it became clear that the results of this benchmark are not quite meaningful when it comes to how concurrent sessions can actually interact between each other because the omits any communication metrics.

All in all, we believe that interactivity will become more and more important also in the domain of supercomputing and that a dynamic and malleable process management, as presented in this paper, is the first right step towards this challenge.

6. REFERENCES

[1] *HBP-PCP Technical Requirements concerning the R&D services on "Whole System Design for Interactive Supercomputing"*, Forschungszentrum Jülich, Human Brain Project, April 2014, online available: http://apps.fz-juelich.de/hbp-pcp/

[2] Erich Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer: www.top500.org – *Lists of World's TOP500 Supercomputers*, November 2015.

[3] ParTec Cluster Competence Center GmbH: *ParaStation Cluster Suite – product portfolio*, online available: http://www.par-tec.com/products/overview.html

[4] Pavan Balaji et al.: *PMI: A Scalable Parallel Process-Management Interface for Extreme-Scale Systems*, in Recent Advances in the Message Passing Interface: 17th European MPI User's Group Meeting, Springer Lecture Notes in Computer Science, pages 31–41, September 2010.

[5] The Message Passing Interface Forum: *MPI: A Message-Passing Interface Standard – version MPI 3.1*, printed at the High Performance Computing Center Stuttgart (HLRS), June 2015.

[6] Norbert Eicker, Thomas Lippert, Thomas Moschny, and Estela Suarez: *The DEEP Project – Pursuing Cluster-Computing in the Many-Core Era*, in Proceedings of the 42nd International Conference on Parallel Processing (ICPP), IEEE Computer Society Press, pages 885–892, October 2013.

[7] Marcal Sola and Victor Lopez: *Dynamic Job Scheduler Benchmark – HBP-PCP Benchmark Documentation*, July 2014, online available: http://pm.bsc.es/~vlopez/files/djsb_doc.pdf

[8] John D. McCalpin: *Memory Bandwidth and Machine Balance in Current High Performance Computers*, IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, pages 19–25, December 1995.

[9] Suraj Prabhakaran et al.: *A Batch System with Fair Scheduling for Evolving Applications*, in Proceedings of the 43rd International Conference on Parallel Processing (ICPP), IEEE Computer Society Press, pages 351–360, September 2014.

[10] Suraj Prabhakaran et al.: *A Batch System with Efficient Scheduling for Malleable and Evolving Applications*, in Proceedings of the 29th IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE Computer Society Press, pages 429–438, May 2015.

Detailed Characterization of HPC Applications for Co-Scheduling

Josef Weidendorfer
Department of Informatics, Chair for Computer Architecture
Technische Universität München
weidendo@in.tum.de

Jens Breitbart
Department of Informatics, Chair for Computer Architecture
Technische Universität München
j.breitbart@tum.de

ABSTRACT

In recent years the cost for power consumption in HPC systems has become a relevant factor. One approach to improve energy efficiency without changing applications is co-scheduling, that is, more than one job is executed simultaneously on the same nodes of a system. If co-scheduled applications use different resource types, improved efficiency can be expected. However, applications may also slow-down each other when sharing resources. With threads from different applications running on individual cores of the same multi-core processors, any influence mainly is due to sharing the memory hierarchy. In this paper, we propose a simple approach for assessing the memory access characteristics of an application which allows estimating the mutual influence with other co-scheduled applications. Further, we compare this with the stack reuse distance, another metric to characterize memory access behavior.

Keywords

Co-Scheduling; Memory Hierarchy; Application Characterization

1. INTRODUCTION

Improvements of computational power of HPC systems rely on two aspects: On the one hand, increased performance comes from an increased number of nodes[1]. On the other hand, node performance itself is expected to grow with newer hardware. Since quite some time, the latter can only be achieved by more sophisticated node designs. General purpose CPUs consist of an increasing number of cores with complex multi-level cache hierarchies. Further, the need for better power efficiency results in increased use of accelerators. Memory modules either have separate address spaces (accelerator vs. host) or are connected in a cache-coherent but non-uniform memory-access (NUMA) fashion. However, the latter makes it even more difficult to come up with high-performance codes as it hides the need for access locality in programs. All these effects result in a productivity problem for HPC programmers: development of efficient code needs huge amounts of time which is often not available. In the end, the large complex HPC systems may have a nice theoretical performance, but most real-world codes are only able to make use of just a small fraction of this performance.

Co-scheduling is a concept which can help in this scenario. We observe that different codes typically use (and are bound by) different kinds of resources during program execution, such as computational power, memory access speed (both bandwidth or latency), or I/O. However, batch schedulers for HPC systems nowadays provide dedicated nodes to jobs. It is beneficial to run multiple applications at the same time on the same nodes, as long as they ask for different resources. The result is an increased efficiency of the entire HPC system, even though individual application performance may be reduced.

To this end, the scheduler needs to know the type of resource consumption of applications to be able to come up with good co-scheduling decisions. HPC programs typically make explicit use of execution resources by using a given number of processes and threads, as provided by job scripts. Thus, we assume that co-scheduling will give out dedicated execution resources (CPU cores) to jobs. However, cores in the same node and even on the same multi-core chip may be given to different applications. In this context, co-scheduling must be aware of the use of shared resources between applications.

In this paper, we look at different ways to characterize the usage of the memory hierarchy by applications. Especially, the results should help in predicting mutual influence of applications when running simultaneously on the same node of an HPC system. First, we look at so-called stack reuse histograms. These histograms provide information on exploitation of cache levels and main memory. However, being an architecture-independent metric without time relation, they cannot include information about how much of a hardware resource is used. However, this is crucial to understand whether an execution gets slowed down when another application shares resources. Thus, secondly, we propose the explicit measurement of slowdown effects by running against a micro-benchmark accessing with a given memory footprint at highest access rate. We compare slowdown results with the reuse histograms of two specific real-world applications.

[1]A node is one endpoint in the network topology of an HPC system. It consists of general purpose processors with access to shared memory and runs one OS instance. Optionally, a node may be equipped with accelerators such as GPUs).

COSH 2016 Jan 19, 2016, Prague, CZ

DOI: 10.14459/2016md1286951

One is an example use-case of the numerical library LAMA [11], consisting of a conjugate gradient (CG) solver kernel. The other is MPIBlast [12], an application from bioinformatics, used to search for gene sequences in an organism database.

We plan to use the prediction for co-scheduling decisions. Within this larger context, an extension of the batch scheduling system[2] is planned which also takes online measurement from node agents into account (the node agent actually is an extension of `autopin` [10], a tool to find best thread-core bindings for multi-core architectures). The vision is that the system should be able to train itself and learn from the effects of its co-scheduling decisions. If there is no characterization of an application yet, the node agent could use the proposed micro-benchmark to gradually get more knowledge about the application for better co-scheduling. Use of the micro-benchmark with its known behavior allows us to obtain such knowledge which depends only on the application and node architecture. This is in contrast to observing slowdowns of arbitrarily running applications.

To get the reuse distance histograms of applications, we developed a tool based on Pin [13]. This tool allows the observation of the execution of binary code with the help of dynamic runtime instrumentation. This way, HPC codes with complex dependencies (in our case Intel MPI, Intel MKL, and Boost) can easily be analyzed without recompilation. Thus, the main contributions of this paper is detailed analysis of the relation of reuse distance histograms and slowdown behavior of applications triggered by a co-running micro-benchmark with one given reuse-distance.

In the next section, we present measurements about co-scheduling scenarios for the applications analyzed, providing motivation and a reference for our discussion. Then we shortly describe our Pin-based tool and the micro-benchmark. Finaly we present detailed slowdown figures for both the applications and the micro-benchmark itself.

2. CO-SCHEDULING MEASUREMENTS

As reference for later discussion, we first present some measurements of co-scheduling scenarios with MPIBlast and LAMA, as already shown in [3]. The system used is equipped with two Intel Xeon E5-2670 CPUs, which are based on Intel's Sandy Bridge architecture. Each CPU has 8 cores, resulting in a total of 16 CPU cores in the entire system. Both L1 (32kB) and L2 (256kB) caches are private per core, the L3 cache (20MB) is shared among all cores on a CPU socket. The base frequency of the CPU is 2.6GHz. However, "Turboboost" is enabled (i. e., the CPU typically changes the frequency of its cores based on the load of the system). However, we do not use Hyperthreading. All later measurements in this paper also were carried out on this system.

For both applications, typical input data is used which results in roughly the same runtimes if executed exclusively. Fig. 1 shows performance and efficiency of various co-scheduling scenarios. From the 16 cores available, the X axis shows the number of cores given to MPIBlast. The remaining cores are assigned to LAMA, respectively. Threads are alternatingly pinned to CPU sockets: e.g. for the scenario with 4 LAMA threads, two are running on each socket. The efficiency is given relative to the best dedicated runtimes of the applications. Note that MPIBlast must be run with at least 3 processes[3]. The best co-scheduling scenario (defined as highest combined efficiency of around 1.2) is with 11 cores given to MPIBlast and 5 cores to LAMA. This shows that LAMA and MPIBlast can benefit from being co-scheduled. In the referenced paper, we also showed energy consumption benefits are even higher. Section 5 will provide insights into why the positive effects are possible.

[2]Here we will look at Slurm.

3. REUSE DISTANCE HISTOGRAMS

For the effective use of caches, good temporal locality of memory accesses is an important property of any application. Temporal locality exists when a program accesses memory cells multiple times during execution. If such accesses are cached, following accesses to the same location can be served much faster, speeding up execution.

To better understand how efficiently an application can exploit caches, a precise definition of temporal locality for a stream of memory accesses is helpful. The *Stack Reuse Distance*, introduced in [1], is the distance to the previous access to the same memory cell, measured in the number of distinct memory cells accessed in between[4] (for the first access to an address, the distance is infinity). For a fully associative cache of size S with least-recently-used (LRU) replacement, a memory access is a cache hit if and only if its stack reuse distance is lower than or equal to S. Thus, if we generate a histogram of distances of all memory accesses from the execution of a program, we immediately can see from this histogram how many of these accesses will be cache hits for any given cache size: looking at the area below the histogram curve, this is the ratio of the area left to the distance signifying the cache size in relation to the whole area. Because the behavior of large processor caches (such as L3) is similar to the ideal cache used in the definition above, the histogram of stack reuse distances is valuable for understanding the usage of the memory hierarchy by a sequential program execution.

Figures 2 and 3 show histogram examples for sequential runs of the applications analyzed in this paper. Many accesses at a given distance means that a cache covering this distance will make the accesses cache hits. Looking e.g. at the three histograms in Fig. 2, where we marked the L3 cache size, it is obvious that even for a small run with 500^2 unknowns, for a large portion of accesses, LAMA has to go to main memory.

The histogram cannot directly be measured with hardware support. In the following, we shortly describe our own tool able to get reuse distance histograms. It maintains an exact histogram taking each access into account. Due to that, the runtime of applications is on average around 80 times longer compared to native execution.

[3]MPIBlast uses a two level master-worker scheme with one process being a "supermaster" and at least one other process being a master. Both supermaster and master distribute work to at least one worker.

[4]Papers from architecture research sometimes define the term *reuse distance* of an access from the previous access to the same memory cell as being the number of accesses in-between. This gives a time-related distance different to our definition here. A reuse distance in this paper is always the stack reuse distance.

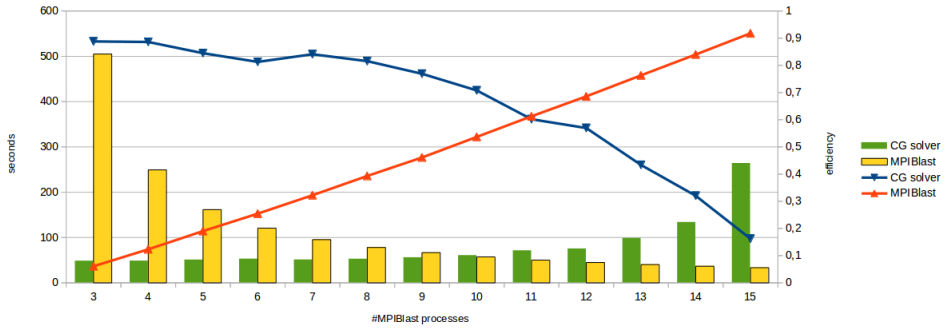


Figure 1: Runtimes and efficiency of co-scheduling scenarios. The core count used by MPIBlast is given on the X axis, other cores are used for LAMA/CG solver (from [3]).

PinDist: A Tool for Deriving Reuse Distances

With the use of Pin [13] we developed a tool that is able to observe all memory accesses from the execution of a program to obtain its reuse distance histogram. For this, Pin is dynamically rewriting binary code in memory directly before execution similar to just-in-time (JIT) compilers. Our tool maintains a stack of accesses to distinct memory blocks of size 64 bytes according to recency of accesses. For each access observed, the corresponding entry for the accessed block is moved to the top of the stack, and the depth where the block was found — this is the distance of the access — is aggregated in the histogram (on first access, we create a new entry using distance infinity). More precisely, we use distance buckets, allowing for a faster algorithm as given in [9]. As suggested in [15], we ignore stack accesses which can be identified at instrumentation time.

PinDist is available on GitHub[5].

4. DISTGEN: CONTROLLED MEMORY ACCESS BEHAVIOR

DistGen is a micro-benchmark written to produce executions exposing given stack reuse distances (and combinations). For this, it reads the first bytes of 64-byte memory blocks in a given sequence as fast as possible. For example, to create two distances, it allocates a memory block with the size of the larger distance. The smaller distance is created by accessing only a subset of the larger block, containing the required number of memory blocks corresponding to the smaller distance. Depending on the required distance access ratio, the smaller and larger blocks are alternately accessed. The expected behavior easily can be verified with our PinDist tool.

DistGen can run multi-threaded, replicating its behavior in each thread. It can be asked to perform either streaming access or pseudo-random access, prohibiting stream prefetchers to kick in. The later is not used in this paper, as it reduces the pressure on the memory hierarchy and does not provide any further information for our analysis. Further, it can be configured to either do all accesses independent from each other, or via linked-list traversal. The latter enforces data dependencies between memory accesses, which allows measuring worst-case latencies to memory. In regular intervals, DistGen prints out the achieved bandwidth, combined across all threads. DistGen is provided in the same GitHub repository as PinDist.

In the following, we used DistGen to simulate a co-running application with a given, simple memory access behavior: streamed access as fast as possible using exactly one specified reuse distance. To ensure occupying available memory bandwidth, we run DistGen on one CPU socket with four threads.

[5] `https://github.com/lrr-tum/reuse`

5. RESULTS

First, we analyzed LAMA and MPIBlast by extracting the reuse distance histogram from typical executions. In all histograms, we marked L2 and L3 sizes. All accesses with distances smaller than L2 size are not expected to influence other cores as L1 and L2 are private. In the range between L2 and L3 size, co-running applications may compete for cache space, and due to benefiting from reuse (accesses in this range are hits due to L3), slowdowns are expected if data is evicted by the co-running application. All accesses with distances larger than L3 size go to main memory, and need bandwidth resources which also is shared by all cores on a CPU, and thus a potentially another reason for slowdown.

Reuse Distances

The CG solver from LAMA is running sequentially in 3 configurations solving a system of equations with different number of unknown. Figure 2 shows the resulting histograms with markers for L2 and L3 cache sizes. It is interesting to observe spikes with heavy access activity at 3 distances which move upwards in the same fashion with higher number of unknowns. The solver does a sparse-matrix vector operation and multiple vector operations per iteration.

1. The large distance corresponds to the total size of the matrix in memory in CSR (compressed sparse row) format.
2. The middle spike corresponds to the vector length (e.g. 8 million doubles for the 1000^2 case).
3. The lower spike comes from re-referencing parts of the vector in the SpMV operation due to the sparsity structure (the example uses a 2D 5-point stencil).

In all LAMA cases, stream-accessing the sparse matrix cannot exploit caches and results in heavy memory access needs. From measurements we observed that LAMA performance does not improve beyond 8 threads with dedicated hardware. The reason is that 4 LAMA threads on each

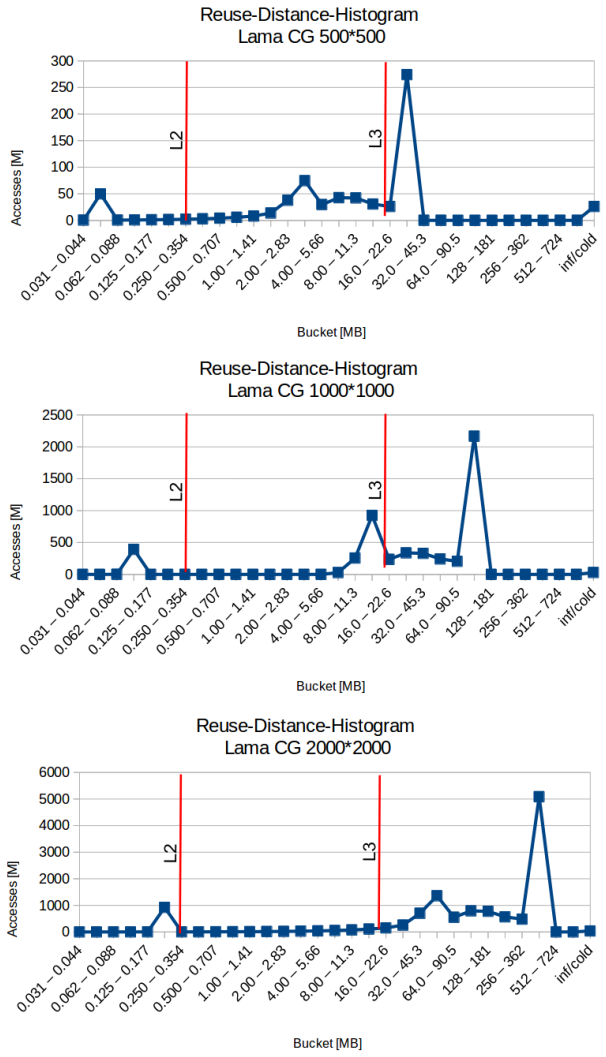


Figure 2: Reuse Distance Histogram for LAMA with 500^2 (top), 1000^2 (middle), and 2000^2 (bottom) unknowns.

CPU socket obviously saturate the available bandwidth to memory. Thus, LAMA performance is memory bound at that point.

Figure 3 shows two histograms of one MPIBlast run. The master MPI-task only has one spike, most probably corresponding to buffer sizes for reading input data and distributing data to workers. The supermaster process is doing nothing in this configuration and therefor not shown. Further, we show the histogram of an MPI worker (we run this with 32 MPI tasks in total, resulting in 30 worker processes). Histogram of all workers are similar. Apart from quite some activity below and around L2 cache size, there is a spike at the distance of L3 size. However, from measurements, we did not really see much traffic to main memory. The solution to this mystery lies in the fact that we do not show the number of accesses for the bucket with the lowest distances (from 0 to 31K). For almost all reuse histograms shown, the spike "skyrockets" the visualized range. E.g. for each MPIBlast workers, it is more than 15 billion accesses.

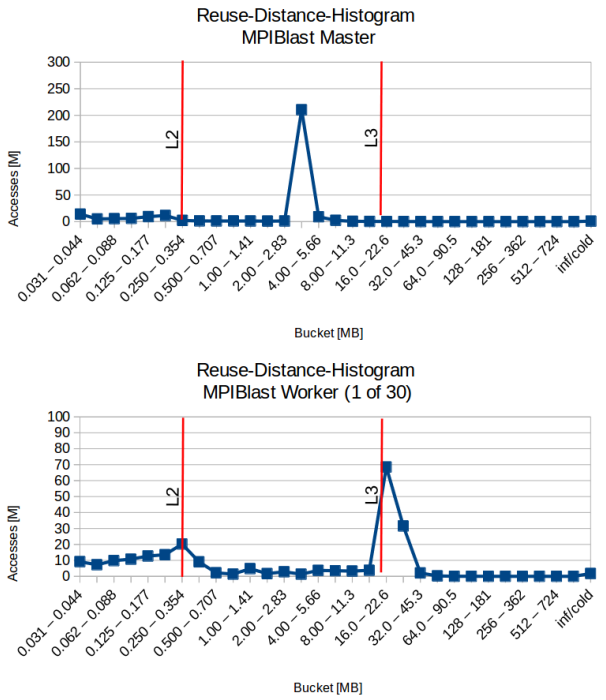


Figure 3: Reuse Distance Histogram for MPIBlast. Master (top) is distributing chunks of workload to workers (bottom) which all have the same histogram.

This shows a drawback not only of our visualization, but for such histograms in general. Even if 100% of accesses were represented by area in the histogram, we cannot see the frequency of accesses. The histogram may hint at memory boundedness, but the frequency of accesses may be so low that the hint is misleading. For more details, we have to look at real measurements showing the influence in co-running applications.

Slowdown Behavior

To predict the slowdown of applications being co-scheduled, we need to co-run it with a benchmark for which we know the behavior. Measurements are done using one CPU socket. DistGen is always running with 4 threads. That is, the 1 MB memory usage point in the figures actually means that each thread is traversing over its own 256 kB. While in this point mostly private L2 is used by DistGen, due to the strict inclusiveness property of the L3 cache in Intel processors, this still requires 1 MB space from L3.

Figure 4 shows the performance of the LAMA CG solver while being co-scheduled with DistGen with different distances. The reuse distance histograms predicted that the CG solver for both 500^2 and 1000^2 unknowns partially use L3 cache, whereas with 2000^2 unknowns there is hardly any benefit for L3 accesses. This can be seen clearly in Fig. 4. The performance with 2000^2 unknown gets severely reduced once DistGen starts consuming main memory bandwidth, whereas with 500^2 and 1000^2 unknowns we already see a performance degeneration when DistGen starts to consume L3 cache. Furthermore, the maximum overall performance hit is higher with 500^2 and 1000^2 unknowns as they benefited

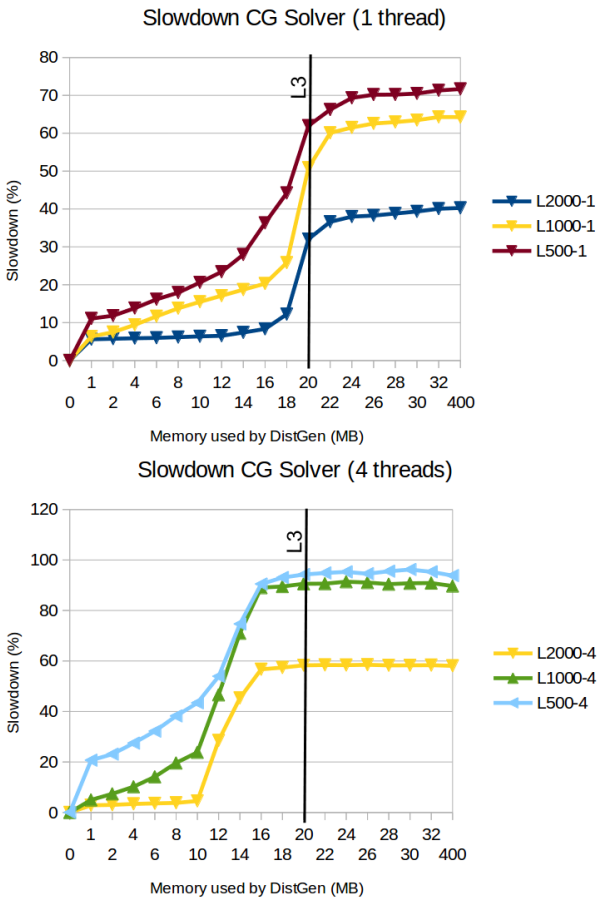


Figure 4: Slowdowns perceived by LAMA with 1 (top) and 4 threads (bottom), running against DistGen with 4 threads.

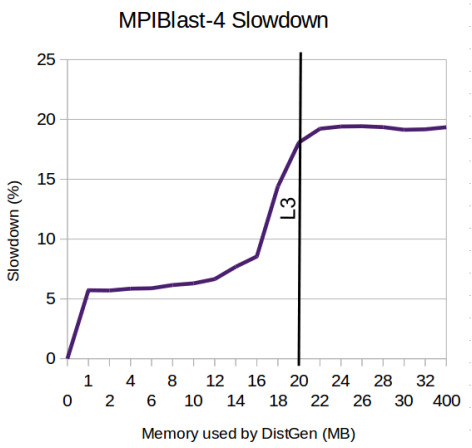


Figure 5: Slowdowns perceived by MPIBlast with 4 threads, running against DistGen with 4 threads.

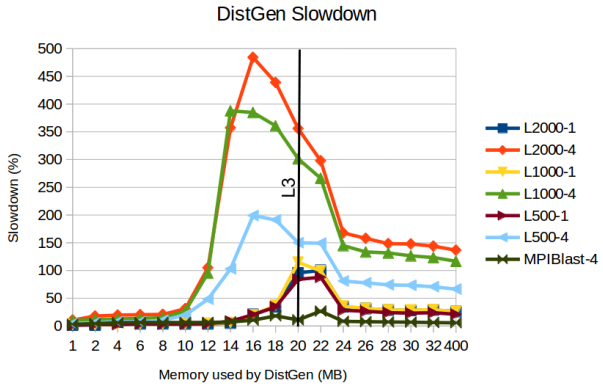


Figure 6: Slowdowns perceived by our micro-benchmark DistGen when running against various applications.

from L3 cache. The maximum overall performance hit is higher when using 4 threads compared to 1 one thread. This results from the fact that a single thread cannot consume the whole bandwidth provide by a CPU socket, whereas 4 threads can. Interestingly, the maximum slowdown with four CG solver threads is already reached with 16 MB DistGen usage. This shows the mutual influence between applications. We attribute this to the CG solver threads evicting cache lines from DistGen such that DistGen starts to use main memory bandwidth.

Figure 5 shows the performance of four MPIBlast processes when being co-run with DistGen with different distances. Again, the results of this measurement closely resembles the ones shown in the reuse distance histogram. MPIBlast mostly relies on its private L2 cache and therefore hardly reacts to DistGen consuming L3 cache. Once DistGen consumes main memory bandwidth we see a slowdown of MPIBlast, as it was predicted by the reuse distance histogram. We assume the initial 5% performance hit of MPIBlast when being co-run with DistGen to be the result of reduced CPU clock frequency. With four idle cores Intels Turboboost can notably increase the CPU clock frequency. But when DistGen is running, all 8 cores are active and the CPU temperature is increased leaving less opportunities to increase the CPU frequency. Overall, the maximum performance hit of MPIBlast ($\leq$ 20%) is far lower than that of the CG solver ($\geq$ 90%). We cannot obtain this information from the reuse distance histograms.

Figure 6 shows the performance of DistGen when being co-run with the various applications. We can gather almost the same information from these figures as we did from the previous ones, but our tool reacts much more apparent (up to 500%). All variations of the CG solver slow down DistGen when it uses main memory bandwidth, whereas MPIBlast hardly results in a slowdown. The single threaded CG solver requires less resources compared to the versions using 4 threads, where the slowdown perceived by DistGen peaks already at 16 MB. This confirms our assumption from above that DistGen is forced to go to main memory at this point.

Overall, we observe that the performance of DistGen when being co-run with an unknown application can provide valuable insights into the other application. Such information will definitely be useful to automatically determine if applications benefit from co-scheduling.

6. RELATED WORK

The stack reuse distance histogram has shown to be very

helpful in analysing memory usage and hinting at tuning opportunities [2]. There are quite some papers suggesting fast methods for its determination in software-based tools, as exact measurement is impossible using hardware. However, authors of [6] propose a statistical approximation using hardware measurements which is extended to multi-cores in [16]. We note that these methods, being statistical, only work well with regular memory usage patterns. None of the papers use the reuse distance histogram in the context of analyzing co-scheduling behavior.

Characterizing co-schedule behavior of applications by measuring their slowdown against micro-benchmarks is proposed by different works. MemGen [5] is focussing on memory bandwidth usage, similar to Bandwidth Bandit [7] which is making sure not to additionally consume L3 space. Bubble-Up [14] is very similar to our approach in accessing memory blocks of increasing size. However, we vary the number of threads co-run against our benchmark.

7. CONCLUSION

In this paper, we studied various ways of a-priori analysis of applications for suitability to improve system throughput via co-scheduling. Reuse distance histograms combined with slowdown measurements proved very useful in this context. We will use these methods in a modified job scheduler.

To avoid slowdown effects of co-running applications on the same multi-core CPU, recent hardware (some versions of Intel Haswell-EP CPUs) allows to configure L3 cache partitions for use by subsets of cores on the chip [8]. Instead of avoiding specific co-schedulings, one can dynamically configure resource isolation to avoid slowdown effects. In [4] it was shown that this can be helpful. We will extend our research in this direction.

Acknowledgments

The work presented in this paper was funded by the German Ministry of Education and Science as part of the FAST project (funding code 01IH11007A).

8. REFERENCES

[1] B. T. Bennett and V. J. Kruskal. LRU stack processing. *IBM Journal of Research and Development*, 19:353–357, 1975.

[2] K. Beyls and E. D'Hollander. Platform-independent cache optimization by pinpointing low-locality reuse. In *Proceedings of International Conference on Computational Science*, volume 3, pages 463–470, June 2004.

[3] J. Breitbart, J. Weidendorfer, and C. Trinitis. Case study on co-scheduling for hpc applications. In *International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems (SRMPDS 2015)*, Beijing, China, 2015.

[4] H. Cook, M. Moreto, S. Bird, K. Dao, D. A. Patterson, and K. Asanovic. A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. *SIGARCH Comput. Archit. News*, 41(3):308–319, June 2013.

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

[6] D. Eklov and E. Hagersten. StatStack: Efficient modeling of LRU caches. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE Internat ional Symposium on*, pages 55–65, March 2010.

[7] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten. Bandwidth bandit: Quantitative characterization of memory contention. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pages 1–10, Feb 2013.

[8] Intel. Improving real-time performance by utilizing cache allocation technology. white paper. White Paper, April 2015. Document Number: 331843-001US.

[9] Y. H. Kim, M. D. Hill, and D. A. Wood. Implementing stack simulation for highly-associative memories. *SIGMETRICS Perform. Eval. Rev.*, 19(1):212–213, Apr. 1991.

[10] T. Klug, M. Ott, J. Weidendorfer, and C. Trinitis. autopin – automated optimization of thread-to-core pinning on multicore systems. In P. Stenström, editor, *Transactions on High-Performance Embedded Architectures and Compilers III*, volume 6590 of *Lecture Notes in Computer Science*, pages 219–235. Springer Berlin Heidelberg, 2011.

[11] J. Kraus, M. Förster, T. Brandes, and T. Soddemann. Using LAMA for efficient amg on hybrid clusters. *Computer Science-Research and Development*, 28(2-3):211–220, 2013.

[12] H. Lin, P. Balaji, R. Poole, C. Sosa, X. Ma, and W.-c. Feng. Massively parallel genomic sequence search on the Blue Gene/P architecture. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–11. IEEE, 2008.

[13] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[14] J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. Contention aware execution: Online contention detection and response. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 257–265, New York, NY, USA, 2010. ACM.

[15] M. Pericas, K. Taura, and S. Matsuoka. Scalable analysis of multicore data reuse and sharing. In *Proceedings of the 28th ACM International Conference on Supercompu ting*, ICS '14, pages 353–362, New York, NY, USA, 2014. ACM.

[16] D. L. Schuff, M. Kulkarni, and V. S. Pai. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 53–64, New York, NY, USA, 2010. ACM.

Terrible Twins: A Simple Scheme to Avoid Bad Co-Schedules

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

Co-scheduling processes on different cores in the same server might lead to excessive slowdowns if they use a shared resource, like the memory bus. If possible, processes with a high shared resource use should be allocated to different server nodes to avoid contention, thus avoiding slowdown. This paper introduces the simple scheme of avoiding to co-schedule twins, i.e., several instances of the same program. The rational for this is that instances of the same program use the same resources and they are more likely to be either low or high resource users − high resource users should obviously not be combined, but a bit non-intuitively, it is also shown that low resource users should also not be combined in order to not miss out on better scheduling opportunities. This is verified using both a statistical argument as well as experimentally using ten programs from the NAS parallel benchmark suite. By using the simple rule of forbidding twins, the average slowdown is shown to decrease from 6.6% down to 5.9%, and the worst case slowdown is lowered from 12.7% to 9.0%, indicating a considerable improvement despite having no information about any programs' resource usage or slowdown behavior.

Keywords

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

1. INTRODUCTION

Processes executing on different cores in the same server typically share many of the server's resources such as, for example, caches, buses, memory and storage devices. When co-scheduled processes have to share a resource their execution is typically slowed down compared to if they would have had exclusive access to that resource [11, 14]. In one study [12] two co-scheduled programs even experienced a super-linear slow-down due to memory traffic interference, i.e. the execution times were more than doubled. In such cases it would be more efficient to run processes sequentially, both in terms of execution time and throughput.

The contention for shared resources has implications for job scheduling in large cluster or cloud systems where tens or hundreds of different programs should be allocated to thousands or tens of thousands of cluster or cloud nodes. Ideally, job scheduling should be done in a way that avoids combining jobs that compete for the same resources, thus minimizing the slowdown caused by resource sharing. Current research focuses on the obvious question of what information a scheduler needs in order to minimize the slowdown caused by resource competition between co-scheduled processes. Many studies are based on the idea that unless the slowdown [3, 9] or the resource utilization [1, 6] of co-scheduled processes can be fairly well estimated, it will not be possible for the scheduler to make an informed decision. Thus, the scheduling becomes pure guesswork and as a result the performance suffers. In this paper however, we show that slowdowns might actually be avoided despite having no knowledge of program characteristics.

It is common knowledge that co-scheduling programs with a high degree of resource usage has a negative impact on performance. However, the co-scheduling of two instances of a purely computationally bound program might also have a negative impact on the overall system performance; given that there are other programs that could have benefited from being co-scheduled with these programs. Hence, a co-schedule consisting of two computationally bound programs, albeit the fact that the programs do not experience any slowdown, should be considered to be a *bad co-schedule*.

In this paper we propose a simple scheme based on an observation from [4] where we noticed that, among the overall worst schedules examined, there was an over representation of schedules where a program was co-scheduled with another instance of itself. The scheme is based on the idea that 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.

In summary, we make the following contributions:

- We show that co-scheduling two computationally bound programs has a negative effect on the overall performance, and should be considered bad, although the programs themselves are not slowed down.
- We prove that co-schedules consisting of **twins**, i.e., several instances of the same program, are over represented among co-schedules with low and high slowdowns. That is, they are more likely to be considered as *bad*.

COSH 2016 Jan 19, 2016, Prague, CZ

DOI: 10.14459/2016md1286952

- We show that by using the simple scheme of disallowing a program to be co-scheduled with another instance of itself, we avoid many **bad** co-schedules and manage to do so without any knowledge of the programs' resource usage nor slowdown behavior.

Based on an experimental evaluation using ten programs from the NAS parallel benchmark suite, we find that our simple scheme effectively reduces the number of bad co-schedules. In Section 2, we first introduce the basic principles for how to co-schedule processes that have a low or high resource usage. Then, in Section 3, we describe the simple scheme and a statistical argument for why the simple scheme also should be a good scheme. Section 4 presents the experimental results before concluding the paper with discussions and conclusions, in Sections 5 and 6, respectively.

2. CO-SCHEDULING CAVEATS

When processes are co-scheduled on the same node in a cluster or cloud system they have to share the node's resources. It is obvious that a high level of resource sharing slows processes down compared to when executing alone on the same node, since they interfere with each other. It is not equally obvious that a low level of resource sharing can be a problem. This is best illustrated by an example where we try to schedule two instances of a computationally bound program, A, with two instances of a memory bandwidth bound program, B, where the execution speed of program B is limited by memory access contention. The four program instances can be co-scheduled in two different ways as shown in Figure 1. In the example we assume a simple processor without frequency scaling or some other advanced technique.

In Figure 1a, where two instances, A_1 and A_2, of A are co-scheduled on node 1, the slowdown for both A_1 and A_2 is 0%. The program instances B_1 and B_2 however, will both experience a slowdown of 100%, and require twice the time to complete their execution compared to when executing alone on the same hardware. Thus, the average slowdown for all processes in Figure 1a is 50%.

In Figure 1b, where A_1 is co-scheduled with B_1 and A_2 is co-scheduled with B_2, the slowdown of A_1 and A_2 is still 0%. Since program A never share any resources with any other program its slowdown will always be 0%. Turning to B_1 and B_2 we can conclude that since they do not share any resources, both B_1 and B_2 have exclusive access to memory resources, and their slowdowns are 0%. Hence, the average slowdown in Figure 1b is 0%.

From this illustration we can deduce **two principles**:

1. Co-scheduling programs which use the same resource should be avoided, especially if the level of resource usage is high.

2. Programs with no or very low resource usage should not be co-scheduled with other programs that have no or low resource usage. There might be much to gain by co-scheduling these programs with other high resource usage programs.

Conceptually, if the resource usage of all programs are known, a scheduler could use this information to avoid co-scheduling two programs that place a high load on any resource. It could also easily avoid co-scheduling two programs

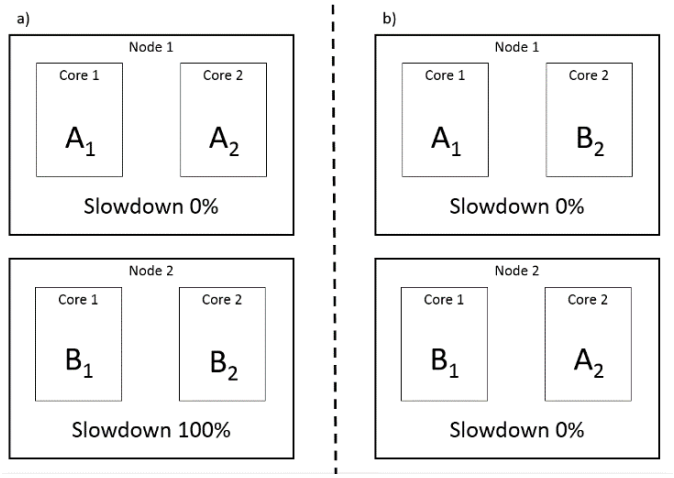


Figure 1: Example showing a bad allocation (a) and a good allocation (b) of instances of a program A that is computationally bound and B that is memory bandwidth bound.

with a low degree of resource usage, thus reaping the benefits of co-scheduling them with programs that have a higher degree of resource usage. However, in the next section we present a simple scheme that manage to avoid some of the bad co-schedules without this prior knowledge of the programs' resource usage.

3. A SIMPLE SCHEME TO AVOID BAD CO-SCHEDULES

The two principles presented in the last section and the fact that we earlier have observed, in [4], that co-schedules containing several instances of the same program are over represented among the worst co-schedules are the foundation for the **simple scheme** we propose:

- Avoid co-scheduling several instances of the same program.

The rationale behind this is that if a program utilizes a resource to a high degree, then, co-scheduling several instances of the same program will violate Principle 1, since we know that they all utilize the same resource. Likewise, co-scheduling several instances of the same computationally bound program will violate Principle 2.

One might argue that the resource usage of most programs is not extreme, that the described cases above will only apply to a few programs and that co-schedules containing several instances of the same program are indistinguishable from other co-schedules. This is not true as we will now show, first statistically and then experimentally. Co-schedules containing several instances of the same program are more likely to violate the two principles than other co-schedules.

To explain why instances of the same program might be over represented among bad co-schedules, we use the following statistical argument. Let us assume that a program or job J_i has a random resource utilization of X_i where X_i is a uniformly distributed random variable between 0 and 1, i.e., between 0% and 100% resource usage. Multiple jobs $J_1, J_2, ..., J_n$ will then have the resource usages

Figure 2: The probability distribution functions of the sum of uniform random variables: a) two jobs and b) four jobs. When combining different jobs (green curve) it is more probable that the combined resource use is centered around the average. In comparison, combining the same jobs (red curve) will have a greater probability of generating a lower or higher combined resource use.

$X_1, X_2, ..., X_n$ where all X_i are independent uniform random variables.

When co-scheduling two or more jobs, the combined resource usage will be the sum of the resource usages of the individual jobs. For two jobs J_i, J_j, we get the combined resource usage X_{i+j} as:

$$X_{i+j} = \begin{cases} X_i + X_j & \text{if } i \neq j \\ X_i & \text{if } i = j \end{cases}$$

This means that we obtain different probability distributions when combining two jobs depending on which jobs we combine. Combining independent jobs results in a **uniform sum** distribution while combining the same type of jobs preserves the original uniform distribution. This is illustrated in Figure 2 where we see that the sum of same jobs has a uniform distribution (red curve) and the sum of independent jobs has the uniform sum distribution (green curve), centered more around the average. Increasing the number of combined jobs would give us a distribution increasingly similar to the normal distribution due to the central limit theorem. This can be seen in Figure 2b which shows the distribution when combining four jobs.

In practice, the combined resource usage cannot really exceed 100% but the derivation above is valid also for lower ranges of resource use. Also, as long as jobs are independent, regardless of the actual underlying distribution, the central limit theorem will still give us a higher probability of evening out shared resource use when combining independent jobs compared to when combining the same dependent jobs. This means that combining instances of the same program often leads to a comparatively low or high resource usage and thus should be avoided.

4. EXPERIMENTAL EVALUATION

To evaluate our proposed simple scheme, we rely on benchmark execution time measurements and scheduling simulations. We first, in Section 4.1, co-schedule different pairs and measure the combined slowdown (sum of the two programs' slowdowns) to verify that co-scheduling several instances of the same program really tend to produce lower and higher combined slowdowns. Then, in Section 4.2, based on a scheduling scenario we evaluate the possible impact of the simple scheme on the overall throughput and performance of a system.

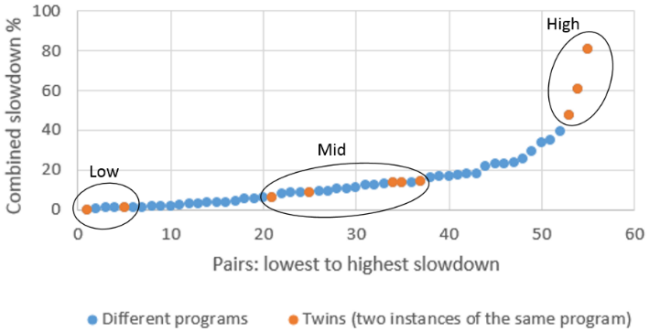


Figure 3: The combined slowdown (i.e., the sum of slowdowns) of the ten NPB programs pairwise co-scheduled in all different combinations. The pairs containing twins, i.e. two instances of the same program, have been marked in red.

The workload used in all experiments is the ten serial benchmarks of the Numerical Aerospace Simulation (NAS) parallel benchmark suite (NPB) reference implementation [10] designed at NASA. The NPB benchmark suite is a collection of five kernels, three pseudo programs, and two programs applicable to the area of computational fluid dynamics (CFD). A description of the NAS-parallel benchmarks is given in Table 1.

The evaluation was carried out on a computer equipped with an Intel Q9550 processor running CentOS Linux 5.10. The Yorkfield Q9550 processor has four cores and a 2-way split second/last-level (L2) cache architecture where two cores share the first L2-cache and the remaining two cores share the second. During these experiments the NPB programs were executed in pairs of two on the two cores sharing the L2.

4.1 Co-Scheduling Slowdowns in NPB

Using the ten NPB programs we co-scheduled all different program pairs and measured the combined slowdown of both programs. The combined slowdown is calculated as the sum of each individual slowdown for each program in the pair. This resulted in the 55 different co-scheduled pairs plotted in Figure 3. As can be seen the combined slowdown ranges from virtually no slowdown (0.04%) up to 80.6%. The average combined slowdown for all co-schedule pairs is 14.15%.

The pairs containing **twins**, i.e. two instances of the same program, are marked in red. As can be seen in Figure 3, the twins seem to be a bit over represented in the low and high areas of the distribution. This is in line with the statistical arguments made in Section 3.

4.2 Avoiding Twins: Impact on Performance and Slowdown

To evaluate the validity of the simple scheme, we used the benchmark data from the previous section to enumerate all possible ways in which two instances of each benchmark can be scheduled on a cluster of ten dual-core nodes. Thus, we

Table 1: A summary of the ten NAS-parallel benchmarks (NPB) [10] used in the experiments.

Abbr.	Type	Description
BT	Pseudo program	Block Tri-diagonal solver
CG	kernel	Conjugate Gradient, irregular memory access and communication
DC	Data movement	Data Cube
EP	kernel	Embarrassingly Parallel
FT	kernel	Discrete 3D fast Fourier Transform
IS	kernel	Integer Sort, random memory access
LU	Pseudo program	Lower-Upper Gauss-Seidel solver
MG	kernel	Multi-Grid on a sequence of meshes, memory intensive
SP	Pseudo program	Scalar Penta-diagonal solver
UA	Unstructured computation	Unstructured Adaptive mesh, dynamic and irregular memory access

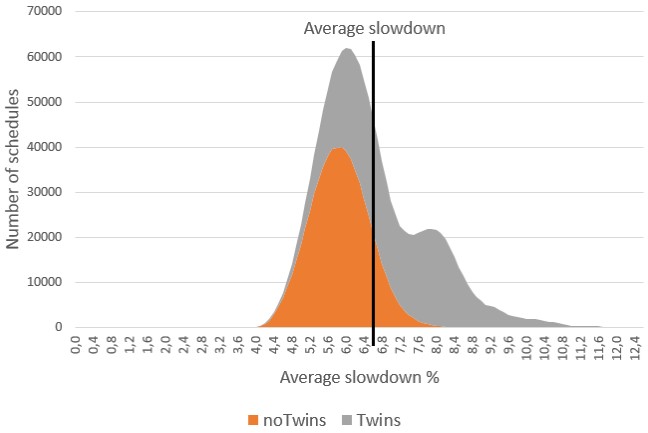


Figure 4: A histogram of the average slowdowns of all 1.4 million schedules resulting from the scheduling of two instances of each benchmark and the slowdown measurements from Figure 3. The gray area shows all schedules and the orange area is the subset where all co-schedules containing twins have been removed, showing that removing twins lowers both the average as well as the maximum slowdowns. The bin size is 0.1.

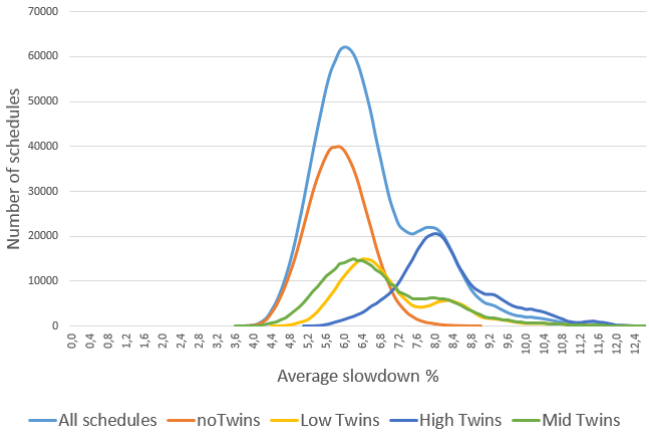


Figure 5: A histogram showing the same data as in Figure 4 with the addition of the subsets: Low twins, High twins, and Mid twins, illustrating that each group of twins (low, mid, or high) makes the slowdown worse compared to when all twins are removed (noTwins).

have 20 jobs to allocate to 20 cores. This results in a total of 1,436,714 different combinations.

The simulation results show that all different combinations exhibit an average slowdown ranging from 3.7% to 12.7%. The average of all combinations was 6.6%. This means that given this job mix and a job-scheduler that randomly allocates jobs to cores, the average slowdown would, over time, converge towards 6.6%. Thus, any scheme worth using has to improve on that percentage in order to be beneficial. Figure 4 shows a histogram of the full 1.4 million combinations of allocations, the black line marks the average slowdown of the entire population. The orange area (noTwins) consists of all schedules that do not contain any twins, i.e. pairs that include two instances of the same program, while the larger grey area (Twins) consists of all schedules that do include at least one twin.

When looking at Figure 4 it becomes quite obvious that disallowing twins will increase the overall performance. However, removing all twins does not only lower the average slowdown from 6.6% to 5.9% it also lowers the worst case slowdown from 12.7% to 9.0%. Furthermore, the execution time of some program instances are decreased quite significantly since the three worst performing pairs with a combined slowdown of ∼47%, ∼60% and ∼80% are all twins (the High pairs in Figure 3). Despite the fact that most schedules containing twins are bad, there are a few schedules containing twins that outperform most no-twin schedules. These are visible as the thin grey area along the left face of the orange noTwins area in Figure 4. It is unfortunate that these schedules are removed, although without more in-depth information we cannot know in beforehand which schedules to keep. These schedules mostly contain twins from the low and mid areas of Figure 3.

To further validate the hypothesis that combining programs which have a low resource usage has a negative impact on the overall performance we divided the twins into three groups: low, middle and high according to their placement in Figure 3. As Table 2 and Figure 5 show, all three twin groups have an average slowdown that is higher than that of the no-twins schedules. Furthermore, all schedules with a slowdown between 9.0% and 12.7% contain twin schedules.

As expected, the schedules in the high twins group, which has a high resources usage, should have much higher slow-

Table 2: Average, minimum and maximum slowdown in percent of the total execution time for all possible combinations as well as for the subsets containing schedules with no twins, only twins, or the low-, mid- and high twins. The noTwin schedules have the lowest average and worst slowdowns of all groups. Hence, removing the twins from the schedules increases the performance.

	All	No Twins	All Twins	Low Twins	Mid Twins	High Twins
Number of schedules	1436714	669734	766980	305062	381989	421743
Best	3.66%	3.81%	3.66%	4.47%	3.66%	5.18%
Average	6.59%	5.92%	7.18%	7.05%	6.83%	8.10%
Worst	12.66%	9.01%	12.66%	12.66%	12.66%	12.66%

down than those in the other groups, consistent with the earlier stated principle, Principle 1, in Section 3. Principle 2 stated that also the low twins should affect the overall slowdown negatively because there might be much to gain by co-scheduling them with other, high resource usage, programs. We can see that this is the case by looking at Figure 5, which shows that the schedules in the low twins group have higher slowdowns than both the no-twins and mid twins group. Hence, we can conclude that that the two principles are valid from our results. The mid twins group, although better than both the low and high groups, still performs worse than the no-twins group.

In conclusion we can see that all schedules containing twins, and not only the low and high twins, are much more likely to affect the slowdown negatively than a schedule without twins.

As an interesting side note, when all processes are co-scheduled as twins the average slowdown is 12.3% which is the $16^{(}th)$ worst schedule out of 1.4 million. Although the all-twins schedule is not the very worst way in which to schedule processes it is definitely one of the worst and it illustrates the risk of putting only twins together. Furthermore, twin pairs from all three groups are represented in the very worst schedule, which has a slowdown of 12.7%.

5. DISCUSSION AND RELATED WORK

The scheme presented in this paper is based on simple heuristics that will avoid some of the worst co-schedules and thus increase the performance of cluster or cloud systems. The benefit of the simple scheme is that its rule is easy to implement in a job-scheduler and no measurements or instrumentations are needed. However, this has to be balanced against the fact that it will not be possible to constantly find the best possible ways to co-schedule programs to reach the optimal performance.

Currently, much related research is investigating methods to find the best co-schedules by avoiding slowdown caused by sharing of resources such as; caches [2], memory buses [7], network links [8, 13] and co-scheduling slowdown [5]. The common denominator for these methods is that they all measure one or several aspect of a program's resource usage and the measurements are then used by the scheduler to decide which programs to co-schedule. In [4] as well as [15] the efficiency of several different methods are compared. Some of these methods are more accurate than our simple scheme. On the other hand, they are more complex to implement and most of them require that the programs are executed and characterized before scheduling them, or require access to hardware performance counters or both.

The evaluation done in this paper, uses two core nodes and single process programs. However, increasing the number of cores or processes of a program should not change the underlying principles. Nevertheless, further studies might be motivated to examine the scheduling of parallel processes on a larger number of cores and maybe using other workloads and scheduling scenarios as well.

The simple scheme can be used by itself or it can be seen as complementary to more complex methods. By combining different methods it might be possible to reduce the number of combinations to evaluate, thus reducing the complexity of the problem.

6. CONCLUSION

This paper introduces *Terrible Twins*, a simple scheme aimed at avoiding bad co-schedules by not co-scheduling **twins**, i.e., several instances of the same program. This simple rule is based on the observation that the twin co-schedules have a statistical distribution that makes them more likely to have a lower or higher combined resource usage compared to other co-schedules. And as shown, both co-schedules with low or high resource usage will hurt the overall system performance. The experimental results show that by using the simple heuristic of forbidding twins, the average and worst case slowdowns were decreased when co-scheduling 20 jobs on 10 double-core nodes.

7. REFERENCES

[1] C. D. Antonopoulos, D. S. Nikolopoulos, and T. S. Papatheodorou. Realistic workload scheduling policies for taming the memory bandwidth bottleneck of smps. In *International Conference on High Performance Computing*, pages 286–296, Berlin, Heidelberg, 2004. Springer-Verlag.

[2] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *International Symposium on High-Performance Computer Architecture*, pages 340–351, Washington, DC, USA, 2005. IEEE Computer Society.

[3] 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.

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

[5] 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.

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

[7] D. Field, D. Johnson, D. Mize, and R. Stober. Scheduling to overcome the multi-core memory bandwidth bottleneck. Hewlett Packard and Platform Computing White Paper, 2007.

[8] 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.

[9] 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.

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

[11] 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.

[12] D. Xu, C. Wu, and P.-C. Yew. On mitigating memory bandwidth contention through bandwidth-aware scheduling. In *International Conference on Parallel architectures and compilation techniques*, pages 237–248, New York, NY, USA, 2010.

[13] C.-T. Yang, F.-Y. Leu, and S.-Y. Chen. Network bandwidth-aware job scheduling with dynamic information model for grid resource brokers. *The Journal of Supercomputing*, 52(3):199–223, 2010.

[14] 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.

[15] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Comput. Surv.*, 45(1):4:1–4:28, Dec. 2012.

Impacts of Virtualization on Intra-Host Communication

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

Jens Breitbart
Department of Informatics
Chair for Computer Architecture
Technical University Munich
Munich, Germany
j.breitbart@tum.de

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

ABSTRACT

The exploitation of the whole computing power of current supercomputers is only achieved by few High-Performance Computing (HPC) applications. We do not expect this issue to be automatically solved with future hardware. One technique to achieve excellent system utilization is *co-scheduling*, where at least two applications with divers resource requirements share the resources of one compute node. Co-scheduling enabled by Virtual Machine (VM) migration is able to improve runtime and energy consumption of HPC applications. In this paper we investigate the impact of full-virtualization on the performance of intra-node communication between VMs for various VM counts. Our analysis reveals that compute-bound applications can achieve up to 97 % of the native performance when executed within 16 VMs while communication intensive operations such as collectives suffer from increased latencies by a factor up to 16. The results can be used as decision-making guidelines for the scheduling system to find suitable solutions for the overall system performance.

Keywords

Virtualization, Intra-Host, Co-scheduling, MPI, HPC

1. INTRODUCTION

We notice that common HPC applications are only capable of exploiting a fraction of the compute power offered by today's supercomputers. Although, some highly tuned applications are able to get close to the systems's peak performance, most applications are limited by a single resource, e. g., I/O or memory bandwidth. This characteristic is not expected to change with upcoming hardware, rather will the increasing gap between computing power and I/O performance [8] result in even more applications not being able to utilize all available resources.

One of the main goals of compute centers is the maximazation of the overall system utilization as it allows more scientists to conduct their research. A way to achieve this goal without manually tuning all applications used at a system, is *co-scheduling* [6], i. e., running two or more applications with divers resource demands in parallel on an overlapping set of nodes. Co-scheduling may benefit from adaptations to the schedule during the runtime as jobs come and go, each exhibiting individual resource demands. Such a dynamic schedule can only be realized if it is possible to move already running processes across nodes of the system. In previous studies we have investigated different techniques enabling such migrations and found full-virtualization, e. g., VMs based on KVM, to provide a good trade-off between performance and flexibility in terms of a greater application range [16].

However, identifying the correct size of a VM , i. e., the amount of Virtual CPUs (VCPUs), is non trivial. On the one hand, large VMs limit the flexibility of the scheduler as it can only migrate whole VMs from one node to another. Small VMs on the other hand may result in the execution of multiple VMs with the same application on the same host system, which slows down communication between the processes of the application. This slowdown is due to the fact that processes running natively on the host or within the same VM can communicate via shared memory, whereas inter-VM communication is typically handled via (virtual) network interfaces. In this paper we investigate the impact of full-virtualization on the performance of intra-host communication. Therefore, we compare the performance of different benchmarks and an example HPC application executed natively to their execution in one or more VMs running on the same host system.

Our results show that the influence of intra-host inter-VM communication on the applications' performance is highly dependent on their characteristics. Compute-intensive benchmarks achieve up to 97 % of the native performance when executed within multiple VMs on the same host. However, communication bound applications are slowed down by up to 26 % in our studies. From a microbenchmark analysis we could conclude that especially collective operations would benefit from a locality-aware communication layer. Here, the latency was increased by a factor of up to 16.

This paper is structured as follows: The following three sections give a detailed overview of the hard- and software setup used for our experiments. Section 5 presents the performance analysis results of selected benchmarks and applications. Before concluding the paper (Sect. 7), we discuss related work in Sect. 6.

COSH 2016 Jan 19, 2016, Prague, CZ

DOI: 10.14459/2016md1286953

2. HARDWARE

We ran all tests on a two-socket NUMA node equipped with Intel IvyBridge CPUs (E5-2650 v2) clocked at 2.6 GHz. Each CPU possesses 8 cores resulting in a total of 16 CPU cores in the entire system. One CPU core has support for two hardware thread contexts (HTC, often called Hyper-threading), i. e., a total of 32 HTCs are provided by the whole system. An HTC has its own set of registers but shares the instruction pipeline and both L1 and L2 caches with the second HTC of the same core. The instruction pipeline has dedicated hardware for floating point, integer, and SIMD instructions, which can be co-issued with various constrains. The L3 cache is shared among all CPU cores of a CPU.

Our platform supports Intel's virtualization technologies VT-x and VT-d [17, 2]. The former extends the ring concept of the x86-architecture by the so-called *non-root* execution mode. This allows for guest Operating Systems (OSs) to run at Ring 0—the most privileged level which is typically used for OS kernels—but with certain restrictions. As OSs are usually written with the assumption of having full control over the hardware, their execution at a different privilege level might result in unexpected or faulty behavior. The non-root execution mode gives a guest OS the impression of owning the hardware while still allowing the host kernel to intercept operations that should not be permitted to the guests. This hardware assisted virtualization provides computing power within VMs close to that of native execution.

However, in contrast to the processor and memory, the virtualization of I/O devices is still a challenge. Each interaction of a guest system with its devices requires I/O operations. For common hardware such as standard Gigabit Ethernet Network Interface Cards (NICs) it is possible to emulate these devices in software. However, this approach fails for high-performance networks, e. g., InfiniBand (IB), only reaching about 50 % of the native performance [13]. To overcome these performance penalties, Intel introduced the VT-d extensions providing guests direct access to the real hardware. It avoids expensive guest-to-host transitions every time the guest accesses its devices by granting direct access to the respective control registers. However, this technique by itself does not enable virtualized HPC clusters as one device can only be passed to one guest at a time. Hence, each VM communicating over IB would require an individual Host Channel Adapter (HCA).

This problem was identified by the Peripheral Component Interconnect Special Interest Group (PCI-SIG) proposing Single Root I/O Virtualization (SR-IOV) as an extension to the PCIe standard [10]. SR-IOV allows for hardware supported I/O device multiplexing by introducing two PCIe function types: Physical Functions (PFs) and Virtual Functions (VFs). The latter are a pared-down version of the PF providing all PCIe capabilities necessary for data movement. The compute node used for our evaluation is equipped with a two-port Mellanox ConnectX-3 IB adapter with support for SR-IOV. It allows for the creation of up to 16 VFs, i. e., the adapter can be attached to 16 VMs at a time.

3. KERNEL-BASED VIRTUAL MACHINE

We used Kernel-based Virtual Machine (KVM) as virtualization solution to perform our evaluation [15]. This hypervisor implements full-virtualization for the x86 architecture based on Intel's VT-x extension described in the previous section or AMD's virtualization extension (AMD-V). A VM is an ordinary processes from the hypervisor's point of view and can be treated like any other process running on the system. Similar to real hardware, it can be equipped with multiple VCPUs which are mapped onto threads of the KVM process representing the VM.

In contrast to other hypervisors, e. g., Xen [4], KVM only implements the necessary components for the virtualization of the CPU and the main memory. Other parts of the computer system have to be emulated in software. Therefore, KVM is usually deployed in conjunction with the user-space emulator QEMU [5].

The virtualization of I/O devices is facilitated by means of the Intel VT-d extensions and SR-IOV. The former allow for the pass-through of PCIe devices to VMs both prior the boot time and during the runtime of the VM (*hot-plugging*). This is an important feature for VM migration in HPC environments, as VMs cannot be migrated with an attached pass-through device, but pass-through devices must be unplugged from a VM prior migration. With SR-IOV it is possible to attach the same physical PCIe devices to multiple VMs at a time. Support for both SR-IOV and hot-plugging allows for near native IB performance, but also enables the scheduler to migrate VMs for the optimization of the overall system utilization.

Modern HPC systems are typically built by using NUMA systems. Software running on top should be NUMA-aware for a good exploitation of such architectures, i. e., threads and processes should be scheduled such that remote memory access is avoided as far as possible. KVM brings NUMA-awareness in terms of NUMA topologies that can be defined for the guests running on top. Therefore, it is possible to comprise one or more VCPUs in so-called *cells* which are recognized as NUMA domain by the guest system. Furthermore, memory policies can be imposed to these cells. Thereby restrictions to memory placement can be defined enabling the creation of a complete reflection of the host's topology to that of the guest.

4. TEST APPLICATIONS

This section briefly introduces the benchmarks and test application that have been used for the evaluation of intra-host inter-VM communication. They have been built with the Intel compiler and were executed by using Intel MPI.

4.1 Microbenchmarks

For the analysis of key figures assessing the communication performance of intra-host MPI communication among multiple VMs, we used both a self-written benchmark and a selection of the low-level benchmarks from the Intel MPI Benchmarks (IMB) [1]. The self-written benchmark[1] determines point-to-point latency and bandwidth by exchanging messages between two processes in a *PingPong* pattern [1]. From the IMB we use a set of benchmarks for the evaluation of the performance of MPI collective operations. We select the following five collective operations:

barrier as it is an indispensable collective operation for the synchronization of a set of MPI processes.

[1] https://github.com/RWTH-OS/mpi-benchmarks

broadcast is a common parallel building block, e. g., used at the beginning of solver phases, when a dataset must be send from the master process to all processes of a job.

allreduce is an operation combining partial results spread among different MPI processes into a final result, e. g., it is used at the end of a solver phase.

allgather only collects the partial results without combining them.

alltoall is an extension to allgather allowing for the distribution of distinct data to the receiving processes and typically involves the most communication.

The exact implementation these benchmarks within Intel MPI is unknown, however these collectives are typically implemented using a communication tree for a reduction of the required messages. The mapping of the tree onto MPI processes and there nodes or VMs (in our case) can influence the performance of the collective operation.

4.2 NAS Parallel Benchmarks

The NAS Parallel Benchmarks (NPBs) [3] is a suite of different computing kernels that are commonly used by large-scale fluid dynamics applications. It offers varying problem classes suiting different cluster sizes. For our tests we chose Class C, depicting a reasonable size for a small test system like the one used for our work.

Originally, the suite contained eight different benchmarks comprising five computing kernels and three so-called pseudo applications. From the kernels we chose FT and CG. The first computes a discrete 3D Fourier Transformation. This is a communication intensive kernel exhibiting an all-to-all communication pattern. CG computes the approximation to the smallest eigenvalue of a large sparse matrix by using a conjugate gradient method. This benchmark is characterized by irregular memory accesses and communication. Furthermore, we evaluated the three pseudo applications BT, LU, and SP which are basically solvers for equation systems.

As our work considers applications that are partly migrated among nodes in a cluster as a result of co-scheduling, we used the MPI implementation of the presented kernels. However, the three pseudo application exist as *multi-zone* version [18], as well. These solve the equation systems on loosely coupled discretization meshes and are intended for the evaluation of hybrid parallelization approaches. The OpenMP+MPI implementation solves the individual zones in parallel in accordance with the shared-memory paradigm while the exchange of boundary values between these zones is performed by means of message-passing. Therefore, they are ideal application benchmarks for our test scenario as many HPC applications exploit HPC systems by applying different parallelization approaches at the same time.

4.3 MPIBlast

We use a slightly modified version of MPIBlast 1.6.0[2] as a real world application for our analysis. Using MPI-only, it is a parallel version of the original BLAST (Basic Local Alignment Search Tool) algorithm from computational biology for the heuristical comparison of local similarities between genome or protein sequences from different organisms.

[2]http://mpiblast.org/

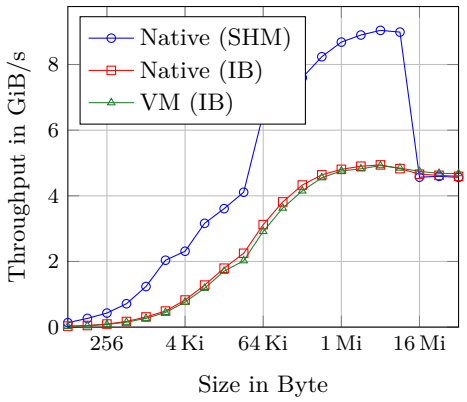


Figure 1: Throughput Overhead

Due to its embarrassingly parallel nature using a nested master-slave structure, MPIBlast allows for perfect scaling across tens of thousands of compute cores [7]. The MPI master processes hand out new chunks of workload to their slave processes whenever previous work gets finished. This way, automatic load balancing is applied. MPIBlast uses a two-level master-slave approach with one so-called super-master responsible for the whole application and possibly multiple masters distributing work packages to slaves. As a result, MPIBlast must be always run with at least 3 processes of which one is the super-master, one is the master, and one being a slave. We used only one master for all our benchmarks and communication mostly only happens between the master and the slave processes. The data structures used in the different steps of the BLAST search typically fit into L1 cache, resulting in a low number of cache misses. The search mostly consists of a series of indirections resolved from L1 cache hits, allowing for a good overlapping of different searches on the 2 HTCs of one core. Our modified version of MPIBlast is available at GitHub[3]. In contrast to the original MPIBlast 1.6.0 we removed all `sleep()` functions calls that were supposed to prevent busy waiting. On our test-system, this resulted in underutilization of the CPU. Removing sleeps increased performance by about a factor of 2. Furthermore, our release of MPIBlast updated the Makefiles for the Intel Compiler to utilize inter-procedural optimization which also resulted in a notable increase in performance.

5. EVALUATION

We measured the results of our self-written benchmark for the following three scenarios to understand the performance penalties when running MPI processes within multiple VMs:

native (SHM) shared memory communication on the host

native (IB) communication using IB on the same host

VM (IB) communication between processes residing in different VMs using IB with SR-IOV

For native (SHM) the latency between two MPI processes running on the same CPU socket is 0.27 µs. This value drastically increases to 1.37 µs for native (IB). However, for VM (IB) the overhead of the virtualization layer and SR-IOV

[3]https://github.com/jbreitbart/mpifast

Table 1: MPI Barrier (Latency in µs)

Native	1 VM	2 VMs	4 VMs	8 VMs	16 VMs
2.05	2.10	8.07	19.44	9.10	13.58

is hardly notable. The additional latency for VM (IB) of 0.02 µs is rather small. These results are similar when measuring the throughput between two MPI processes (cf. Fig. 1). The maximum bandwidth of VM (IB) is only at around 45 % of that in native execution.

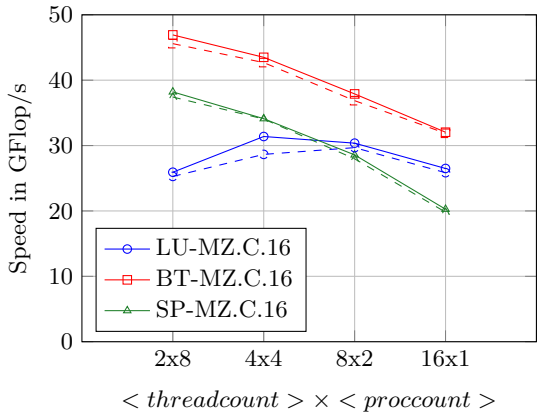


Figure 2: NPB-MZ Native (solid) vs. VM (dashed)

5.1 Collectives

To assess the impact of multiple VMs on communication intensive applications, we investigated a set of collective operations, namely: barrier, allgather, alltoall, allreduce, and broadcast (cf. Tab. 1 and Fig. 4). All benchmarks were started with 16 MPI processes each pinned to an individual core. The results present a scalability study over 2, 4, 8, and 16 VMs, i. e., 8, 4, 2, and 1 processes per VM respectively. Each VM has been equipped with one of the VFs that are available on our test system. As a result, inter-VM communication is using IB as transport, whereas ranks running on the same VM communicate via shared-memory segments. In all cases a compact pinning of ranks to VMs and cores has been performed, e. g., in the scenario with 2 VMs Ranks 0 to 7 have been started on one VM while the remaining ranks resided in the second.

Using more than one VM clearly increases the latency of the barrier by a factor of around 4. This is due to IB communication that takes place between some processes. However, adding more VMs has a moderate influence on the latency with additional 5.51 µs for 16 VMs running on the same host, i. e., instead of having a mixture of shared-memory (intra-VM) and IB (inter-VM) communication, all processes synchronize over IB. The peak of 19.44 µs in the case of 4 VMs might arise from a suboptimal distribution of ranks to VMs. The latency in this scenario can be improved to 10.68 µs by using a scatter pinning which. This results in a different communication scheme as the communication tree is most probably distributed differently across the VMs. However, we have to investigate that point in more detail to get a clearer picture.

The results of the other collective operations (cf. Fig. 4) reveal that their execution within more than one VM often results in significantly increased latencies. For example the broadcast operation is throttled by a factor of 1.5 when the 16 processes are distribted across two VMs compared to native (SHM) for small messages. This factor increases to 6.3 for the 16-VM case. However, for the 2- and 4-VM case the latencies converge at least for large messages of 4 MiB. The *Alltoall* operation within 16 VMs is throttled by a factor of 16 for small messages. However, again for larger messages this discrepancy decreases to a factor of around 4.5.

From the presented microbenchmark analysis it can be concluded that the execution of a communication bound MPI job on multiple VMs running on the same host can impose important performance degradation. This is mostly true for applications exchanging small-size messages and should be considered when taking any scheduling decisions.

5.2 Applications

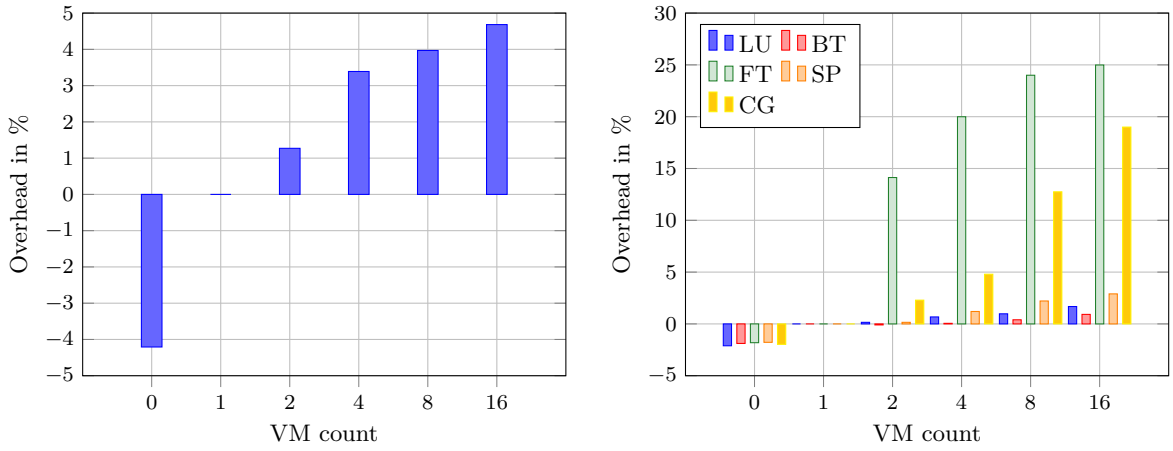
The previously shown microbenchmarks suggest that co-scheduling using VMs on HPC clusters result in significant performance penalties if the MPI library does not come along with efficient intra-host inter-VM communication. In this section we evaluate the performance hit for the applications described in Sect. 4.

In the first test case, 32 MPIBlast processes run equally distributed on a different number of VMs. All VMs run on the same compute node and use InfiniBand as inter-VM communication channels. Figure 3a shows the performance differences between the various configurations. The usage of 0 VMs means that all processes run natively on the host system communicating over shared-memory segments. As MPIBlast is a compute-bound application and the communication channel does not constitute its bottleneck, the performance differences between the configurations are rather small.

Figure 3b shows the results of a similar test scenario, in which the MPI-versions of the NPBs were divided equally over VMs. Overall, the usage of multiple VMs is not a performance drawback decreasing the performance of the pseudo applications by only 1 % to around 3 % for the 16 VM case related to the 1-VM case. The slower communication interface between VMs in comparison to native execution is only clearly noticeable in the more communication intensive kernels CG and FT.

The next test case is the *multi-zone* version of the NPB. They were started on one compute node for native (SHM) and within multiple VMs on that node for VM (IB), i. e., the process count equals the VM count for the latter case. However, the relationship between processes and threads changed between every run. Figure 2 shows that the best performance can be achieved if the benchmark uses more processes rather than threads. The usage of a message passing interface reduces the number of side effects such as *False Sharing* and contention on synchronization primitives. Furthermore, the memory allocation strategy is simplified. The processes are bound to a single NUMA node and always allocate the memory on its node guaranteeing local memory accesses. In the case of using one process and 16 threads, applications have to use NUMA-aware allocation strategies to achieve best performance [11].

In the case, that the benchmarks are running natively on the host system, a shared memory region is used for inter-process communication. If the MPI processes run within different VMs, a shared memory interface is missing and IB is used as communication channel for the inter-VM com-



(a) MPIBlast (32 Processes) (b) NPB Class C (16 Processes)

Figure 3: Overhead when running MPI-only applications across multiple VMs compared to the execution within one VM.

munication. Figure 2 reveals that the performance degradation by using multiple VMs is small. Consequently, the performance of the multi-zone version of NPB does not depend on small messages (smaller than the cache size), which clearly is more efficient by using a shared memory interface (cf. Fig. 1).

6. RELATED WORK

Overall there has been not much research on intra-host inter-VM communication. Typically, studies focus either on the comparison of different virtualization solutions in general [19], or they investigate the impacts of I/O virtualization on inter-node communication [14, 12].

Zhang et. al proposed a design of a locality-aware MPI library [21, 20]. Their implementation extends MVAPICH2 [9] by a *locality detector* enabling communication over shared-memory segments between processes residing in different VMs on the same host. Focusing on the performance benefits of Inter-VM Shared Memory (IVShmem) over SR-IOV communication they perform a comprehensive performance evaluation of inter-VM communication using either of the two mechanisms.

7. CONCLUSION

This paper explores the applicability of virtualization as driver for co-scheduling applied to HPC. We estimate the impact of the VM size on the performance of HPC applications by conducting scalability studies over different VM counts but with a fixed amount of processes, i. e., the varying VM granularity has a direct influence on the ratio between shared-memory and IB communication. Depending on the application's characteristics, a scheduler might decide to host multiple VMs of the same job on one node without taking high performance losses.

However, especially the latency of collective operations suffer from the IB communication channel between VMs. Therefore, we plan to work on locality-awareness of the MPI layer. This should not only comprise inter-VM communication over shared-memory but also adoptions of the communication channels to dynamic re-schedules that might occur during runtime.

8. ACKNOWLEDGMENTS

This research and development was supported by the Federal Ministry of Education and Research (BMBF) under Grant 01IH13004B (Project FAST).

9. REFERENCES

[1] Intel MPI Benchmarks. Technical report, Intel Corporation, 2014.

[2] Intel Virtualization Technology for Directed I/O. Technical report, Intel Corporation, 2014.

[3] D. H. Bailey et al. The NAS Parallel Benchmarks. *Int. Journal of High Performance Computing Applications*, Sept. 1991.

[4] P. Barham et al. Xen and the Art of Virtualization. *SIGOPS Oper. Syst. Rev.*, 2003.

[5] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, 2005.

[6] J. Breitbart et al. Case Study on Co-Scheduling for HPC Applications. In *Proc. Int. Workshop Scheduling and Resource Management for Parallel and Distributed Systems (SRMPDS 2015)*, Sept. 2015.

[7] A. Darling et al. The design, implementation, and evaluation of mpiBLAST. In *Proc. of ClusterWorld*, 2003.

[8] J. Dongarra. Impact of Architecture and Technology for Extreme Scale on Software and Algorithm Design. In *Euro-Par 2010 – Parallel Processing*, Aug. 2010. Euro-Par 2010 Keynote.

[9] W. Huang et al. Design of High Performance MVAPICH2: MPI2 over InfiniBand. In *Proc. 6th IEEE Int. Symp. Cluster Computing and the Grid*, CCGRID '06, 2006.

[10] Intel LAN Access Division. PCI-SIG SR-IOV Primer. Technical Report 2.5, Intel Corporation, Jan. 2011.

[11] S. Lankes et al. Node-Based Memory Management for Scalable NUMA Architectures. In *Proc. 2nd Int. Workshop Runtime and Operating Systems for Supercomputers (ROSS 2012) in conjunction with 26th Int. Conf. Supercomputing (ICS 2012)*, 2012.

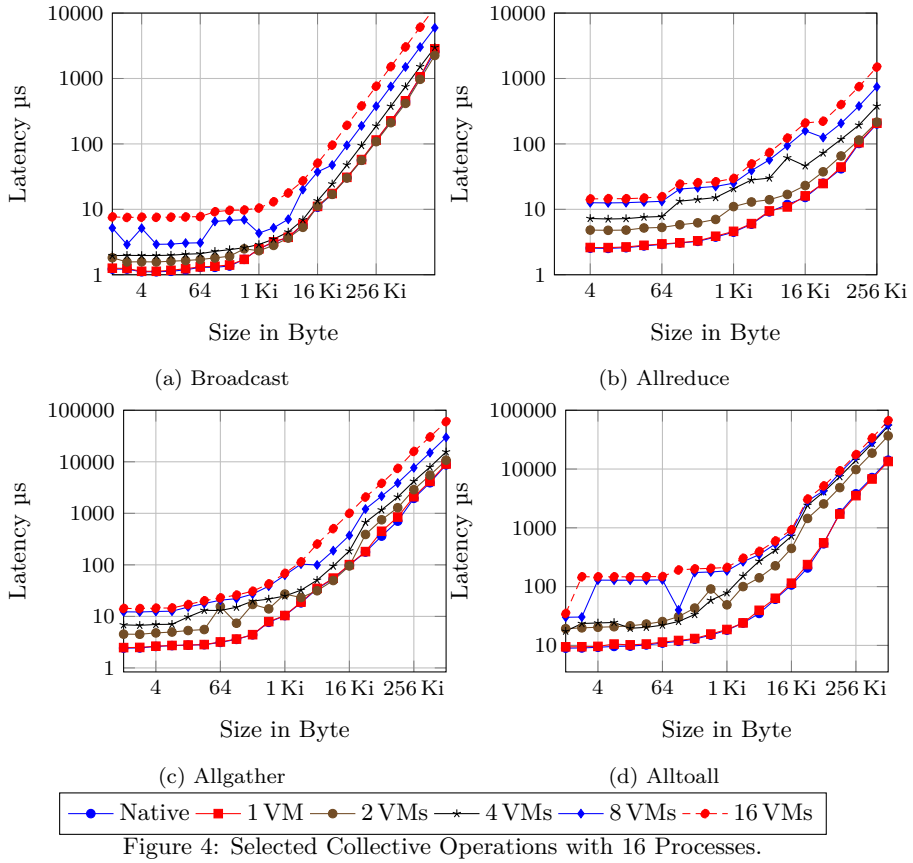


Figure 4: Selected Collective Operations with 16 Processes.

[12] J. Liu et al. High Performance VMM-Bypass I/O in Virtual Machines. In *USENIX Annual Technical Conference, General Track*, 2006.

[13] Y.-M. Ma et al. InfiniBand virtualization KVM. *CloudCom*, 2012.

[14] M. Musleh et al. Bridging the Virtualization Performance Gap for HPC Using SR-IOV for InfiniBand. *IEEE CLOUD*, 2014.

[15] L. Nussbaum et al. Linux-based virtualization for HPC clusters. In *Proc. Linux Symposium*, July 2009.

[16] S. Pickartz et al. Migration Techniques in HPC Environments. In *Euro-Par 2014: Parallel Processing Workshops*, Lecture Notes in Computer Science. 2014.

[17] R. Uhlig et al. Intel Virtualization Technology. *Computer*, 38, May 2005.

[18] R. F. Van der Wijngaart and H. Jin. Nas parallel benchmarks, multi-zone versions. *NASA Ames Research Center, Tech. Rep. NAS-03-010*, 2003.

[19] A. J. Younge et al. Analysis of Virtualization Technologies for High Performance Computing Environments. In *Cloud Computing (CLOUD), 2011 IEEE Int. Conf.*, 2011.

[20] J. Zhang et al. Can Inter-VM Shmem Benefit MPI Applications SR-IOV Based Virtualized Infiniband Clusters? In *Euro-Par 2014 Parallel Processing*. Jan. 2014.

[21] J. Zhang et al. High performance MPI library over SR-IOV enabled infiniband clusters. In *2014 21st Int. Conf. High Performance Computing (HiPC)*, 2014.

Impact of the Scheduling Strategy in Heterogeneous Systems That Provide Co-Scheduling

Tim Süß, Nils Döring, Ramy Gad, Lars Nagel, André Brinkmann
Zentrum für Datenverarbeitung
Johannes Gutenberg University Mainz
Mainz, Germanz
{t.suess, doeringn, gad, nagell, brinkman}@uni-mainz.de

Dustin Feld, Eric Schricker, Thomas Soddemann
Fraunhofer SCAI
Schloss Birlinghoven
Sankt Augustin
{dustin.feld, eric.schricker, thomas.soddemann}@scai.fraunhofer.de

ABSTRACT

In recent years, the number of processing units per compute node has been increasing. In order to utilize all or most of the available resources of a high-performance computing cluster, at least some of its nodes will have to be shared by several applications at the same time. Yet, even if jobs are co-scheduled on a node, it can happen that high performance resources remain idle, although there are jobs that could make use of them (e. g. if the resource was temporarily blocked when the job was started). Heterogeneous schedulers, which schedule tasks for different devices, can bind jobs to resources in a way that can significantly reduce the idle time. Typically, those schedulers make their decisions based on a static strategy.

In this paper, we investigate the impact if a heterogeneous scheduler allows modifications of the strategies at runtime. For a set of applications, we determine the makespan and show how it is influenced by four different scheduling strategies. A well-chosen strategy can result in a speedup of more the 2.5 in comparison to other strategies.

Keywords

Scheduling, Scheduling strategies, Heterogeneous systems

1. INTRODUCTION

For several years now, multi-core processors equipped with powerful vector units are the standard in almost all parts of the computing world. They are in cell phones, notebooks, desktop computers, servers and supercomputers. Additionally, GPUs and other architectures (Xeon Phi, FPGA, digital signal processors) are used in combination with normal processors to speed up suitable parts of an application. These *accelerators* mostly operate on separate memory spaces which requires time-consuming copy operations when the architecture is changed during a program run. At the moment, it seems as if this will not change in the foreseeable future. All these hardware architectures have in common that they only offer their performance benefits if developers write code for them and if they are able to exploit their inherent parallelism. Code for accelerators can be created using OpenCL and domain-specific languages (DSLs).

In almost all systems, a large fraction of the accelerator hardware will be frequently idle and not optimally used. This happens when

1. none of the concurrently executed programs on a computer can make use of a provided accelerator.
2. programs do not provide codes for the accelerators available.
3. a program cannot use its preferred resource because it is temporarily blocked by another application. The application may then be started on a less favorable resource. However, once a better resource becomes free, the program cannot be moved to this resource.

When the first situation occurs in a cluster environment, it can be solved by moving jobs between nodes or by already taking resource requirements into consideration during scheduling. If a resource is oversubscribed by multiple jobs on one node while the same resource is undersubscribed on another node, jobs can be migrated to balance the utilization. The second situation can obviously be avoided by providing codes for all concerned resources. Typically, a separate version of the program is needed for each resource. If multiple codes are available, the most suitable free resource can be chosen during runtime.

To tackle the third situation, it must be possible for a program to start its computation on one resource and move to another one later. Also a scheduler is required which manages the resources, assigns tasks to resources and migrates tasks. This way it can prevent resource oversubscription.

However, if the *scheduling strategy* (the algorithm which decides when a computation is started or migrated) is static, it cannot exploit program-specific information about the computations behavior which could be provided by the program developers.

VarySched is a scheduler that allows the scheduling of computations (denoted as *tasks*) on heterogeneous resources. An application must register itself at the scheduler by implementing an interface. The interface requires only the set of codes for the different resources (denoted as *kernels*) and

COSH 2016 Jan 19, 2016, Prague, CZ

DOI: 10.14459/2016md1286954

a ranking of these kernels. The ranking can correspond to performance, accuracy, energy consumption etc. We denote such a set of kernels as a *kernel collection*. The scheduler receives collections, chooses one of their kernels and schedules it to an available resource. This is similar to the behavior of the *Grand Central Dispatch* resource scheduler [1]. In contrast to the Grand Central Dispatch, VarySched allows to change the scheduling strategy. It even allows that an application provides its own strategy in form of a simple Lua script.

In this work, we evaluate the impact of different scheduling strategies on the makespan of programs. Four different types of strategies are tested:

long-term scheduler: allows to estimated the finishing time of a job by fixing the order of execution on one device.

short-term scheduler: compared to the long-term scheduler this scheduler provides short reaction times.

banking-based scheduler: is an extension of the short-term scheduler with an additional resource budget.

constraint-based scheduler: similar to the banking-based scheduler but with a different computation for the resource budget.

In addition, we determine the overhead caused by the different scheduling strategies and the costs for exchanging the scheduling strategy.

The paper is organized as follows: In Section 2, we discuss different techniques related to our scheduler. In Section 3, we describe the relevant parts of the scheduler and the scheduling strategies. After that we evaluate different aspects and components of our infrastructure in Section 4. Section 5 concludes the paper summarizing the results and giving an outlook on future work.

2. RELATED WORK

To leverage computer's full potential, jobs must utilize all available resources and the resources must be used in parallel, but not necessarily parallel within a single application.

Recently, quite some work has been published on the challenges of addressing exhaustive multi-core usage and heterogeneous scheduling. Nevertheless, so far there seems to be no published approach tackling the problem from all possible angles at the same time. Some of the approaches solely focus on the multi-threaded application support like DAGuE [4], Elastic Computing [13], or StarPU [2]. Others address the problem of multi-application thread scheduling like ADAPT [8], but are limited to the CPU-side of the problem. All of them have in common, that substantial code changes may be necessary to exploit the hardwares' full potential like in StarPU [2] or are even mandatory to make the system work (e.g. in DAGuE [4]).

However, some ideas are similar to ours. StarPU, e.g., deals with codelets, which are similar to what we address as kernel tasks. For calculating an optimal schedule, DAGuE and StarPU rely on Directed Acyclic Graphs (DAGs) to determine an optimal schedule, e. g. by utilizing task-graphs. Hence, the code developer needs to introduce the interdependencies of his tasks explicitly in those approaches.

Sun et al. have shown how a task queuing extension for OpenCL, providing a high-level, unified execution model coupled with a resource management facility can improve the performance within a heterogeneous environment [11]. Anyway, this approach is solely based on OpenCL and does not allow for the use of external code generators or other ways of utilizing its scheduling system.

The *Grand Central Dispatcher* (GCD) [1] solves this problem by applying a more fine-grained scheduling strategy. Instead of considering the program as a whole, it individually schedules sub-tasks (like functions) which must be marked in the program. The scheduled jobs are executed asynchronously which allows for an energy-efficient and effective utilization of all resources. The GCD's scheduling, however, is application-centric and has no global view for which reason the quality of the schedules is, as a matter of principle, limited. The queue, representing the jobs' priorities, has to be manually defined within the application using the `dispatch_set_target_queue`-function. Another drawback of the GCD is its restricted configurability which further restricts the decision-making process.

Beisel et al. [3] presented a resource-aware scheduler capable of distributing tasks among different hardware resources like VarySched. In contrast to VarySched, the scheduler uses always the same, static scheduling function.

3. SCHEDULING STRATEGIES

The VarySched scheduler is used to evaluate the impact of different scheduling strategies on the performance. VarySched is a newly developed task scheduler which will be published in the near future. In this section, we shortly describe the main features of VarySched as well as the scheduling strategies and applications that we use in our tests.

3.1 VarySched

VarySched consists of two parts, a daemon and an interface, which must be met by the applications that are to be managed by the scheduler. The scheduling daemon is not executed in the operating system's kernel space, but as a daemon with root privileges. Although this prevents the immediate cooperation with the Linux scheduler and the use of *cgroups*, it allows for more flexibility. Any user shall be able to submit a scheduler strategy with his programs and benefit from a better resource utilization.

Applications register their kernel collections at the daemon which determines when the kernels are executed. For this, the daemon requires a strategy, and VarySched even goes one step by allowing dynamic modifications of the strategy. Users can implement their own strategy as a Lua script. The scheduler can aim for different objectives, for example the reduction of makespan, energy consumption or heat production. The strategy can use every information that can be accessed from the Lua script. While the strategy can be flexible in Lua, the daemon is written in C++11 as well as the library used by the client. However, a C interface of the library is also provided to allow an easy use for C applications.

A messagebox system provides mechanisms for the communication between daemon and applications. There is one special messagebox to register kernel collections. After registration, each application gets its exclusive messagebox for further communications.

After an application has successfully registered, it is attached to one of the provided queues. There are different queues: one queue for each resource (denoted as *resource*

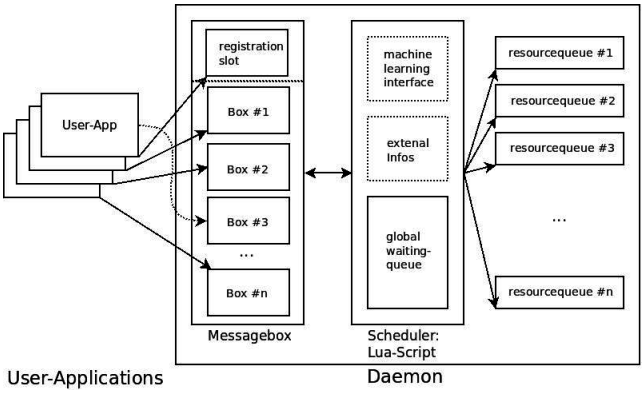


Figure 1: Architecture of VarySched. Applications register their kernels in the messageboxes. The scheduler takes the kernels, schedules them, and updates the kernels with the resource information.

queue) and one global queue. There are two possibilities to trigger scheduling decisions: 1) when a resource becomes free or 2) periodically calling a function. Ticks can be activated or deactivated and the time interval between two ticks can be adjusted. The scheduling algorithm, which has been implemented in the Lua script, decides which of the registered tasks is executed next and on what resource a specific kernel from the kernel collection is started. The application is informed about this decision via the associated messagebox (see Figure 1 for the schematic structure of VarySched).

VarySched provides mechanisms allowing dynamic modification of the strategy used. Triggered by a Unix signal, VarySched performs several steps:

- The Lua script containing the new strategy is loaded into a temporary buffer.
- The script is checked for being a valid Lua program.
- It is checked if the interface is implemented correctly.
- The old strategy is replaced by the new one and the queues are updated.

Note that all modifications are done while the daemon is running. The first three steps do not cause any runtime overhead because the tests are performed asynchronously in a parallel thread. The daemon is neither stopped nor paused nor must it be restarted. Additionally, there is no need to make a copy of the new queue as it can be passed directly to the new strategy. However, the new strategy can modify the queues as required.

The script can be an arbitrary Lua script that fulfills the daemon's scheduling strategy interface. Otherwise there are no limitations to the Lua program and therefore all Lua features can be used. Furthermore, arbitrary sources of information can be used in the codes if required to make a decision. Even external information sources, as from sensors or the internet, can be used. Thus, the target of the scheduling strategy can be arbitrary, as long as there is a path to the required information.

In our evaluation, the scheduling strategy depends on a *resource governor* (a system that predefines how much resources can be used) which has two levels: high and low. The governor can be used in the strategy to enable and disable resources. Thus, depending on the governor's state, tasks can either be executed on different resources in parallel or not.

3.2 Scheduling Strategies

In our tests, we use four scheduling strategies with different aims. We define two different governors (named low and high) which determine the type and the amount of resources that can be used.

3.2.1 Short-term Scheduler

The short-term scheduler aims for using all resources permanently. While it focuses on keeping all resources busy, the selection of a good kernel is secondary. It does not use the resource governor's state for the scheduling decisions.

At first, incoming tasks are placed in the global queue which is not associated to any resource. All resource queues contain only a single task that is processed instantly when it arrives. If a resource ρ becomes free, the scheduler traverses the global queue and searches for the first task that has the best performance on resource ρ (with respect to the strategy). This task is then scheduled on ρ and the current scheduling phase terminates. If there is no such task, the scheduler traverses the global queue again, searching for a task whose second preference is ρ, and so on.

3.2.2 Long-term Scheduler

The long-term scheduler aims to place all tasks on the resources they prefer most. Additionally, it tries to fill the queues such that the queues' work off requires similar time. Depending on the resource governor's state, the long-term scheduler masks different resources to stay unused. In our tests, if the governor's level is high, jobs can be scheduled on all resources; if the level is low, only the CPU cores can be used (e. g. for energy reasons).

The global queue contains only a single task t while the different resource queues can contain an arbitrary amount of task. The scheduler determines the length of the resource queue l_1 that t prefers the most. Then it determines the length of the resource queue l_2 that t prefers the second most. If $l_1 \leq \delta \cdot l_2$ (whereby δ is the performance factor between the resource that t prefers the most and the resource that t prefers the second most) t is schedule on its first choice. Otherwise the procedure is repeated with t's second and third preferences and so on until it reaches the least preferred resource.

3.2.3 Banking-based Scheduler

The banking-based scheduler assumes that each available resource has a limited budget of credits. Running a kernel on a resource costs a certain amount of credits. If a resource's budget suffices to bear the costs of a kernel, the respective amount of credits is removed from the budget and the task is scheduled to that resource. The scheduler starts with the most preferred resource and proceeds successively with the following resources. If no resource has a sufficient budget to take the task, the task stays in the global queue. The budget is refilled over time. After a certain amount of time, credits are added to the budget until the maximal budget limit is reached.

In our tests we start with a full budget of one hundred credits. Every five seconds 15 credits are added to the bud-

get if the resource governor is set to high, and ten credits are added if the governor is set to low. Running a task on the GPU costs ten credits; five credits are needed for all CPU cores, one credit for a single CPU core.

3.2.4 Constraint-based Scheduler

The constraint based scheduler assumes that every incoming job consumes resources denoted as credits and has an overall credit limit. The credit sum of concurrently running jobs must not exceed this predefined limit. The aim of this schedule is to provide a constant upper boundary for currently used resources. The resource queue of every device can hold only a single job. Incoming jobs are scheduled until all resources are used or the overall credit limit is reached. If one of these conditions is met, incoming jobs will be enqueued in the global queue until a resource has been freed and the free credits are sufficient.

In our tests, a task on the GPU costs nine credits, on all CPU cores six credits, and three credits on a single core. The credit limit is set to 18 if the governor is set to high and nine if it is set to low.

3.3 Test Environment and Applications

We tested two applications on a NVIDIA Jetson-TK1 system. The tests have been performed with two different resource governor states as described in Section 3.1. In our tests performing a matrix-matrix multiplication, we scheduled one hundred instances of the same application. For the LAMA application, we scheduled 25 instances.

Matrix-Matrix Multiplication.

Our first test application performs a matrix-matrix multiplication. The matrices are quadratic and contain 1024×1024 single-precision floating point values. The performed algorithm consists of three nested loops iterating over the two matrices. To generate parallel running codes automatically, we used Pluto-SICA [5, 6] and PPCG [12] to produce the resource-specific kernels of the kernel collections.

This scenario shows how VarySched can be used in combination with automatic code generators. It is not necessary to program the different code versions for multi-core CPU and GPU manually because in this example the code is sufficiently simple and, hence, manageable by the aforementioned tools.

A single matrix-matrix multiplication takes about 2.74 seconds on a single CPU core, about 1.11 seconds on all four cores, and about 3.69 seconds on the GPU.

LAMA Application.

LAMA [7] is an open source C++ library for building efficient, extensible and flexible solvers for sparse linear systems. A LAMA solver can be executed on various compute architectures without the need of rewriting the actual solver. LAMA supports shared and distributed memory architectures, including multi-core processors and GPUs.

For our tests, we use a conjugate gradient solver to solve an equation system which results from discretizing Poisson's equation with a 3-dimensional 27-points (and thus very sparse) matrix. The number of unknowns is $50 \cdot 50 \cdot 50 = 125000$. The CG algorithm is one of the best known iterative techniques for solving such sparse symmetric positive (semi-)definite linear systems [10]. It is therefore used in a wide range of applications (e.g. Computational Fluid Dynamics (CFD) or oil and gas simulations). The used kernel collection contains three kernels: one for a single CPU core, one using OpenMP, and one for the GPU and a single core.

This scenario shows that VarySched can schedule hardware-specific kernels whose functionality is provided by a library.

A single execution on the solver takes about 191.01 seconds on a single CPU core, about 103.19 seconds on all four cores, and about 41.78 seconds on the GPU.

Jetson-TK1.

The Jetson-TK1 is an ARM-based (Cortex-A15, four 32-bit cores, 2.3 GHz) system equipped with a 192-core Kepler GPU (GK20A). Additionally, the board provides 2 GiB main memory, shared and accessible by CPU and GPU [9]. We use two different Linux operating systems with different CUDA versions. For the matrix-matrix multiplications we use Ubuntu 14.04 and CUDA-6.5 and for the LAMA tests we use Gentoo and CUDA-6.0.

4. EVALUATION

In this section we evaluate the impact of the scheduling strategies (Section 3.2) on execution of the applications (Section 3.3) by running respectively 100 or 24 instances on one node in parallel. The experiments are performed for both governors and the quality of the schedules is measured by the makespan which is the time necessary to process all jobs.

The experiments are conducted as follows: All instances are started at approximately the same time in the beginning. One after another, the jobs register at the VarySched daemon and the scheduling strategy determines for each job which of their kernels is to be executed.

4.1 Matrix-Matrix Multiplication

The total execution time is displayed in Figure 2 for all matrix-matrix multiplications and both governors. An important observation is that the makespans of the constraint-based and the short-term scheduling strategy are almost the same when the governor is set to high (i.e. all resources can be used). Additionally, the makespan increases when the resource governor's setting is changed from high to low. For the short-term scheduler the makespan stays almost constant independently of the governor's state. This can be explained by the way this scheduler works. As it always tries to utilize all available resources, the governor's setting has no influence on the schedule.

The decreasing performance of the constraint-based and the banking-based scheduler in the low governor state can be explained by their credit-based approach. The performance of the long-term scheduler stays almost constant.

The available budget of the constrained-based scheduler and the banking-based scheduler depends on the governor's state. The constraint-based scheduler's credit limit is 18 in the high state and nine in the low state. The high state allows to utilize all available resources. The low state allows only the utilization of either three instances using a single core or two instances, one running a multi-threaded kernel and one a single-threaded kernel.

In this respect, the banking-based scheduler behaves differently as its performance is significantly smaller if the resource governor's state is lowered. This can be explained by how credits are added to the budget and when a task is scheduled. In both states, a constant number of credits is added every five seconds; 15 credits in the high state, 10 in

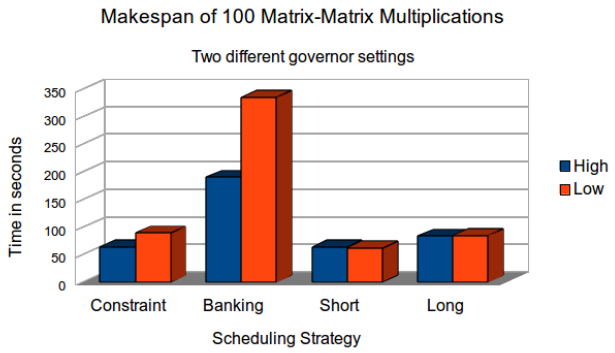


Figure 2: Makespan of one hundred instances of the matrix-matrix multiplication application with different governor states. The credit-based strategies are strongly influenced by the choice of the governor while the short-term and long-term scheduler are almost unaffected.

the low state. This frequency of incoming credits is too low to utilize all resources permanently because an instance of the application can be competely executed before the new credits arrive.

A task is scheduled when a sufficient amount of credits for a resource is available and, if the credits are only sufficient for the slowest resource, the kernel for this resource is chosen. Thus, 15 credits are sufficient for a GPU and multi-threaded kernel, ten credits are enough for one multi-threaded and one single-threaded kernel, and five credits are still sufficient for one multi-threaded or one single-threaded kernel. In general, the constraint-based scheduler's is multiple times faster than program execution using the banker-based scheduler.

The long-term scheduler achieves almost constant results for both resource governor settings. This can be explained by the fact that the last jobs to run are scheduled and executed on the slowest resource because there are some inaccuracies in the performance factors between the different resources.

But the increase of the makespan alone with different governors does not show the positive impact of co-scheduling. Since in most scheduling strategies the number of utilizable resources is reduced if the governor is lowered, the total execution time (the makespan) increases. However, this is even the case if the median of the applications' execution time decreases. Figure 3 shows that the runtime of a single matrix-matrix multiplication decreases if less resources can be used.

The increase of the single application's performance has two reasons: 1) The size of the matrices is small so that much of the time during the matrix-matrix multiplication is spent on copying data in the case of executing on the GPU. 2) The GPU versions also use CPUs a little bit and thereby influence CPU kernels. If there are no GPU kernels, then the cores are not shared.

When the governors are set to high, all resources are used and applications share and compete for these resources so that applications might block each other; while in the case, where the governors are set to low, less resources are used and applications are executed more sequentially.

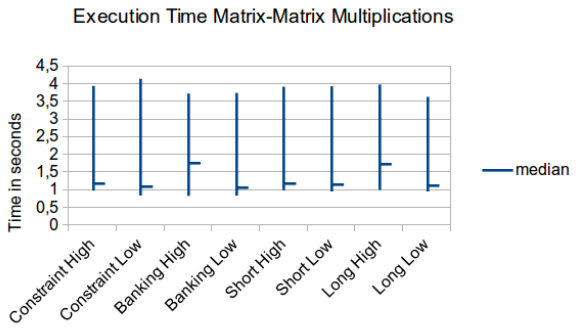


Figure 3: Runtime of matrix-matrix multiplication for different schedulers and governors.

4.2 LAMA Application

As in the previous section we first analyze the makespan of the LAMA application. When scheduling this application, the results for the short-term scheduler are similar to the ones of the matrix-matrix multiplication (see Figure 4). The short-term scheduler does not consider the governor's state and the application's preferences, thus all test runs have similar makespans.

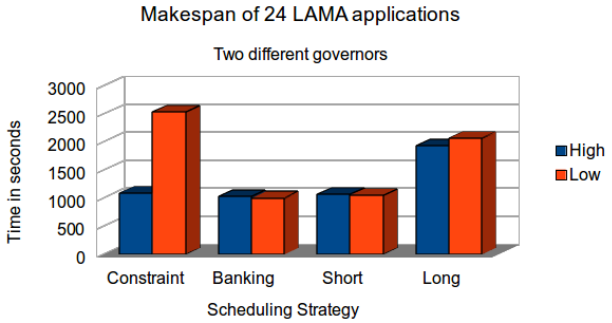


Figure 4: Makespan of one hundred instances of the LAMA application with different governor states.

The applications suffer the most if the constraint-based scheduler is applied and the governor is lowered. The total execution times more than double if the usage of the GPU is prohibited.

The banking-based scheduler behaves differently. While the makespan for the matrix-matrix multiplication increases when setting the governor's state to low, it is almost constant in case of the LAMA application. This can be explained by the required runtime for one application which is significantly higher than for the matrix-matrix multiplication. Due to the high runtime, the scheduler's budget can be refilled sufficiently fast, for which reason all resources can be used.

In case of the long-term scheduler, there is the same issues as for the matrix-matrix multiplication. The performance factors between the different resources are not well-adjusted and, thus, this scheduler achieves the worst results.

The median of the runtimes varies for three of the schedulers when changing the governors state (see Figure 5). While

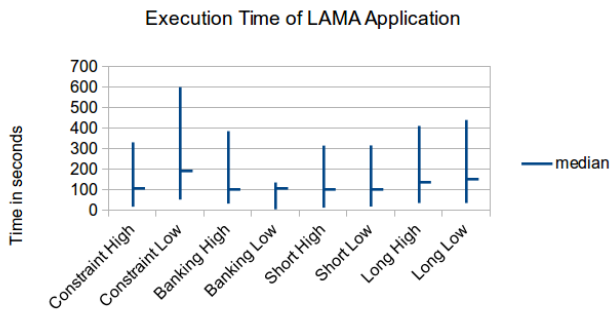


Figure 5: The runtime of the LAMA applications with different governors.

for the matrix-matrix multiplication the median only stays constant for the short-term scheduler, for the LAMA application it stays constant for the banking-based scheduler, too. This was expected from the previous results of the makespan. At maximum, using the banking-based scheduler is 2.5 times faster than using the constraint-based scheduler.

The median of the application runtime increases if the long-term scheduler is used. In comparison to the previous tests, the time required to copy the necessary data to the GPU can be compensated by the accelerated computation.

Findings: Co-scheduling can reduce the makespan of parallel executed applications. It has a positive impact on the systems performance, even in the case when the median runtime of a single application slightly decreases if number of used resources is increased.

5. CONCLUSION

In this paper we have shown that the scheduling strategy has a high impact on the makespan of co-scheduled applications when they are run on nodes with heterogeneous resources. In our experiments, we used VarySched, a resource scheduler that is specialized for such heterogeneous environments and that allows dynamic modifications of the scheduling strategy. We evaluated four different strategies using two applications and two resource governor settings. The results show that the application can be accelerated by a factor of up to 2.5 if the scheduler is chosen wisely.

Acknowledgments

This work was supported by the German Ministry for Education and Research (BMBF) under project grant 01|H13004A (FAST).

6. REFERENCES

[1] Apple. Grand Central Dispatch - A better way to do multicore. Technology Brief, 2009. http://opensource.mlba-team.de/xdispatch/GrandCentral_TB_brief_20090608.pdf.

[2] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice & Experience - Euro-Par 2009*, 23:187–198, 2011.

[3] T. Beisel, T. Wiersema, C. Plessl, and A. Brinkmann. Cooperative multitasking for heterogeneous accelerators in the Linux Completely Fair Scheduler. In *Proceedings of the International Conference on Application-Specific Systems, Architectures, and Processors*, pages 223–226, Piscataway, NJ, USA, 2011.

[4] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. DAGuE: A generic distributed DAG engine for High Performance Computing. *Parallel Computing*, 38(1-2, SI):37–51, 2012.

[5] D. Feld, T. Soddemann, M. Jünger, and S. Mallach. Facilitate SIMD-Code-Generation in the Polyhedral Model by Hardware-aware Automatic Code-Transformation. In A. Größliger and L.-N. Pouchet, editors, *Proceedings of the 3rd International Workshop on Polyhedral Compilation Techniques*, pages 45–54, 2013.

[6] D. Feld, T. Soddemann, M. Jünger, and S. Mallach. Hardware-Aware Automatic Code-Transformation to Support Compilers in Exploiting the Multi-Level Parallel Potential of Modern CPUs. In *Proceedings of the 2015 International Workshop on Code Optimisation for Multi and Many Cores*, COSMIC '15, pages 2:1–2:10, 2015.

[7] J. Kraus, M. Förster, T. Brandes, and T. Soddemann. Using LAMA for efficient AMG on hybrid clusters. *Computer Science - R&D*, 28(2-3):211–220, 2013.

[8] K. Kumar Pusukuri, R. Gupta, and L. N. Bhuyan. ADAPT: A Framework for Coscheduling Multithreaded Programs. *ACM Transactions on Architecture and Code Optimization*, 9(4):45:1–45:24, 2013.

[9] NVidia Corporation. Jetson TK1 Development Kit Specification - Version 01, 2014. http://developer.download.nvidia.com/embedded/jetson/TK1/docs/3_HWDesignDev/JTK1_DevKit_Specification.pdf.

[10] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003.

[11] E. Sun, D. Schaa, R. Bagley, N. Rubin, and D. R. Kaeli. Enabling task-level scheduling on heterogeneous platforms. In *GPGPU@ASPLOS*, pages 84–93, 2012.

[12] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gómez, C. Tenllado, and F. Catthoor. Polyhedral Parallel Code Generation for CUDA. *ACM Transactions on Architecture and Code Optimization*, 9(4), 2013.

[13] J. R. Wernsing and G. Stitt. Elastic Computing: A Portable Optimization Framework for Hybrid Computers. *Parallel Computing*, 38(8, SI):438–464, 2012.