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

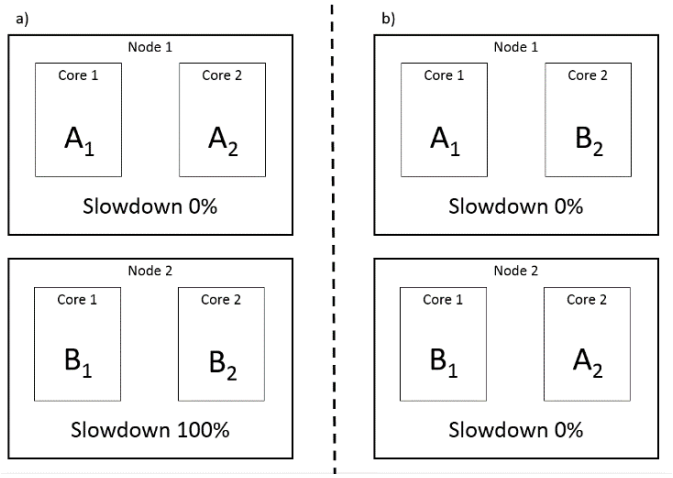

Figure 1: Example showing a bad allocation (a) and a good allocation (b) of instances of a program A that is computationally bound and B that is memory bandwidth bound.

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.

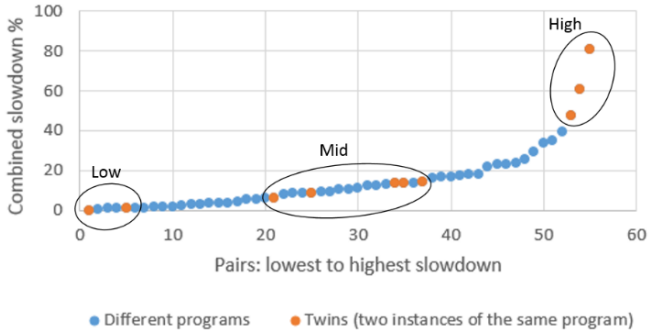

Figure 3: The combined slowdown (i.e., the sum of slowdowns) of the ten NPB programs pairwise co-scheduled in all different combinations. The pairs containing twins, i.e. two instances of the same program, have been marked in red.

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

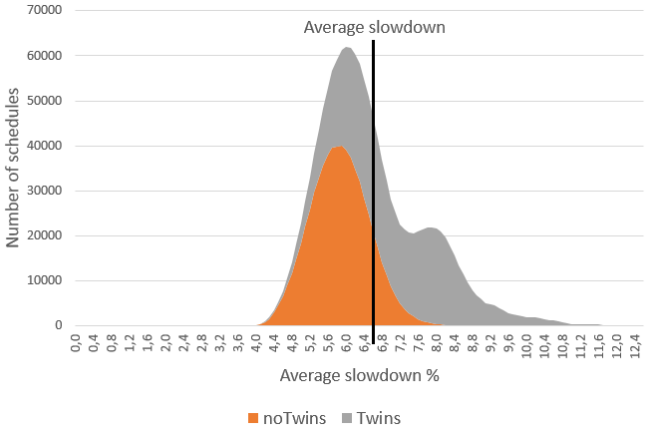

Figure 4: A histogram of the average slowdowns of all 1.4 million schedules resulting from the scheduling of two instances of each benchmark and the slowdown measurements from Figure 3. The gray area shows all schedules and the orange area is the subset where all co-schedules containing twins have been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

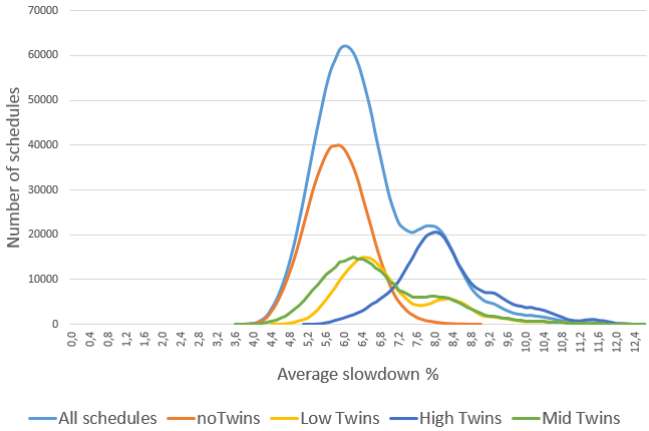

Figure 5: A histogram showing the same data as in Figure 4 with the addition of the subsets: Low twins, High twins, and Mid twins, illustrating that each group of twins (low, mid, or high) makes the slowdown worse compared to when all twins are removed (noTwins).

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.