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

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

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

ABSTRACT

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

Keywords

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

1. INTRODUCTION

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

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

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

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

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

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

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

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

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

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

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

2. RESOURCE AWARE CO-SCHEDULING

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

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

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

2.1 The Terrible Twins Approach

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

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

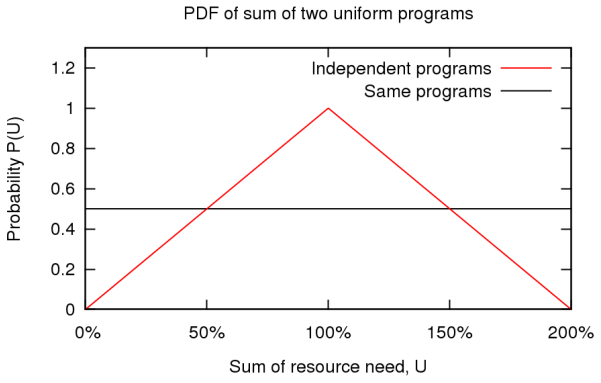

Figure 1: The probability density function (PDF) of the sum of the resource need of two co-scheduled programs.

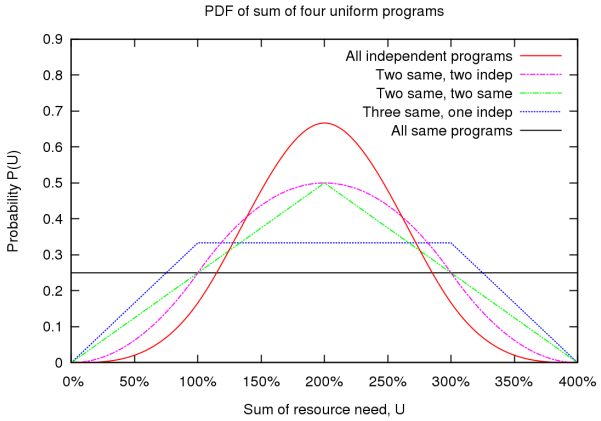

Figure 2: The probability density function (PDF) of the sum of the resource need of four co-scheduled programs where each program has a uniform resource need probability between 0% and 100%. Combining independent programs leads to a more centered PDF meaning that there is less risk for the total resource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

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

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

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

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

3. PROBABILISTIC ANALYSIS

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

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

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

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

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

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

4. EXPERIMENTAL SETUP

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

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

5. FULL-FACTOR EVALUATION

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

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

5.1 Full-Factor Measurement Results

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

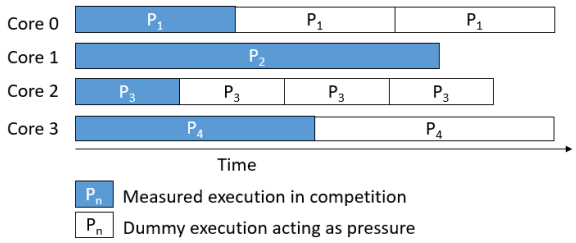

Figure 3: The methodology used when measuring the co-scheduling slowdown of programs P_1 to P_4. When the first instance of a program finishes, it is replaced with another instance of the same program. This procedure is repeated until the first instance of all programs have finished executing.

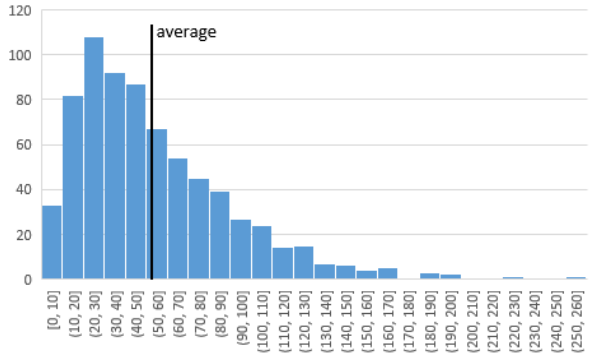

Figure 4: Histogram showing the measured slowdown distribution of all possible four-program co-schedules. The black line marks the average slowdown.

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

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

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

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

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

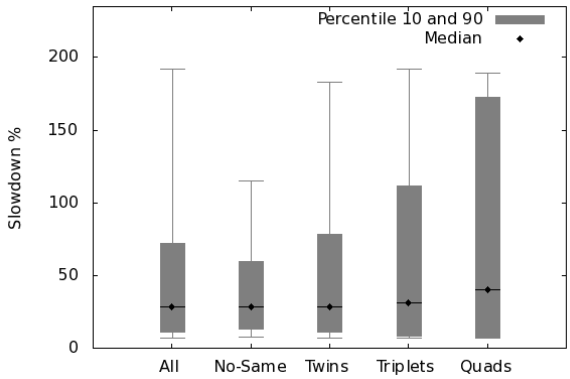

Figure 5: The filled box represents the co-schedules for each subset with slowdowns between the 10th and the 90th percentile. The dot and line in each box is the median slowdown and the end of the whiskers show the min and max slowdowns.

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

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

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

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

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

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

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

6. IMPACT ON SCHEDULING

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

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

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

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

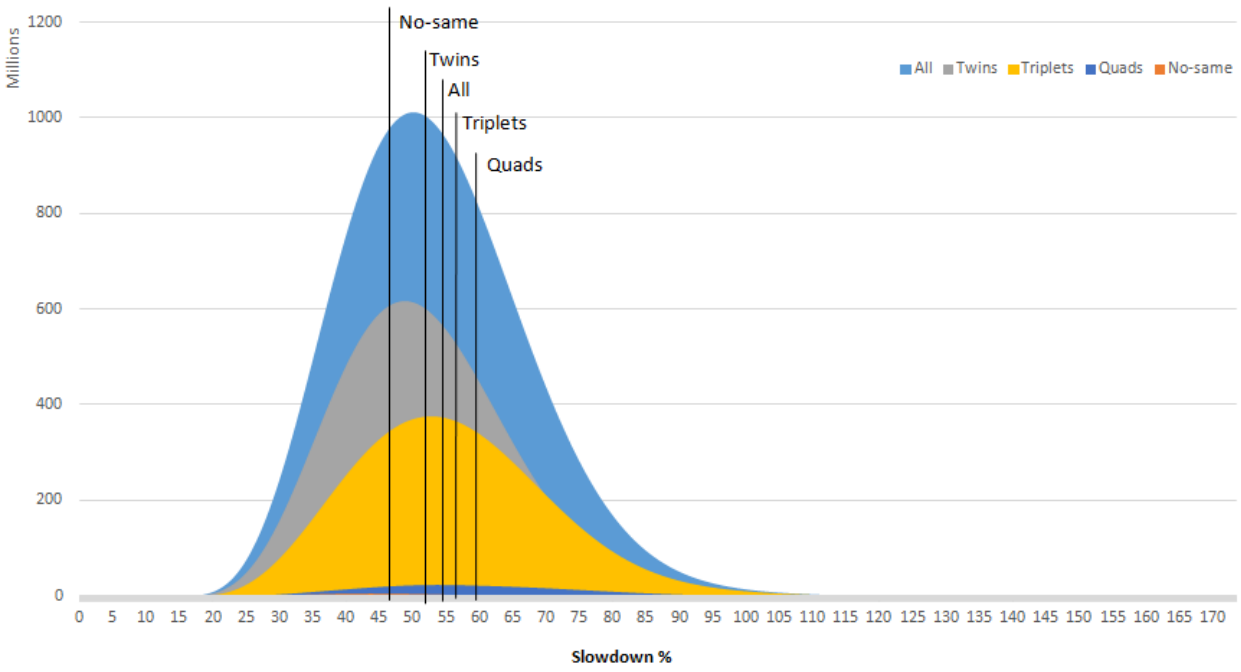

Figure 6: A histogram of all 351 billion evaluated schedules as well as the same subsets as in Table 2. The black lines mark the average slowdown of each subset. The bin size is 0.1.

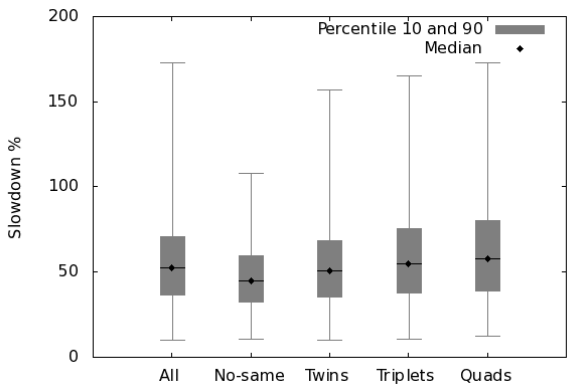

Figure 7: A box and whiskers plot of the same subsets as in Table 2. The filled box covers the 10th to the 90th percentile. The dot and line in each box is the median value and the end of the whiskers the min and max slowdowns.

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

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

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

7. CONCLUSION

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

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

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

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

8. REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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