Partial Coscheduling of Virtual Machines Based on Memory Access Patterns

Anselm Busse
Technische Universität Berlin
Berlin, Germany
anselm.busse@tu-berlin.de

Jan H. Schönherr
Technische Universität Berlin
Berlin, Germany
jan.schoenherr@tu-berlin.de

Matthias Diener
Federal University of Rio Grande do Sul
Porto Alegre, Brazil
mdiener@inf.ufrgs.br

Philippe O. A. Navaux
Federal University of Rio Grande do Sul
Porto Alegre, Brazil
navaux@inf.ufrgs.br

Hans-Ulrich Heiß
Technische Universität Berlin
Berlin, Germany
hans-ulrich.heiss@tu-berlin.de

ABSTRACT

Virtualization is omnipresent in server environments. The scheduling of virtual machines is a challenging task because it is necessary to avoid differences in processing progress of the virtual CPUs, which otherwise can lead to a severe performance degradation. Coscheduling is a commonly used technique to solve this issue. With coscheduled virtual machines, all virtual CPUs are executed at the same time by the host. However, in a situation with virtual machines of arbitrary size, coscheduling of a whole virtual machine can lead to an under-utilization of the host. This situation occurs when the sizes of the virtual machines prohibit a scheduling where all cores of the host machines are used at every point in time.

In this paper, we show that this under-utilization can be reduced through partial coscheduling. Partial coscheduling uses sets that are not based on the size of the virtual machine but on the requirements of the load inside the virtual machine. We show through experiments with the Linux Kernel Virtual Machine (KVM) in combination with a coscheduling capable Linux kernel, that partial coscheduling can lead to an overall performance improvement compared to full coscheduling of complete virtual machines. The partial coscheduling approach requires knowledge about the relation between processes and threads inside the virtual machine, which is usually not available at runtime. To gather this information without modifying the guest, we propose an automatic algorithm based on the recent technique of Communication Detection through Shared Pages (SPCD), which detects the memory access behavior of applications inside virtual machines.

Our experiments show that partial coscheduling can improve the utilization of the host by reducing the waste of computation time caused by unnecessarily idle cores, thereby increasing the performance of virtual machines. In many scenarios, automated partial coscheduling can also increase the host utilization.

Categories and Subject Descriptors

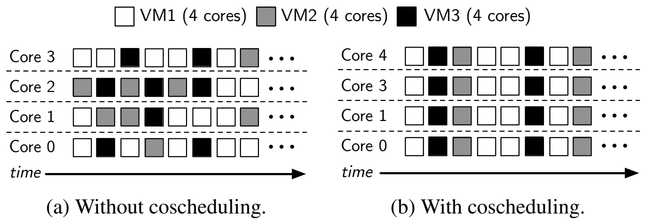
D.4.1 [**Operating Systems**]: Process Management—*Scheduling, Virtualization, Threads, Multiprocessing / Multiprogramming / Multitasking*

Keywords

Scheduling, Virtualization, Memory Sharing, Communication Detection, Coscheduling

1. INTRODUCTION

Virtualization is omnipresent in server environments for many reasons, such as improving resource utilization, reliability, and maintainability. Running virtual machines presents many challenges, mostly related to the fact that the virtualization should be transparent to the virtualized software while at the same time it has to share the host resources with other virtual machines. One of the critical areas in this context is the scheduling of the virtualized CPUs which has special requirements compared to ordinary application scheduling. Traditional operating systems or bare metal applications assume that every online CPU is available and will process data in a predictable way. Therefore, it can schedule the application based on this assumption and will introduce for example synchronization barriers according to the expected system behavior. For a virtual machine, this assumption does not hold, since it has to share the CPUs with other virtual machines. This becomes an issue when one virtual CPU produces more progress than another since this might lead to a situation where the virtualized application gets



(a) Without coscheduling. (b) With coscheduling.

Figure 1: Scheduling for a quad-core machine with three virtual machines VM1–VM3, each consisting of 4 virtual cores. Each virtual machine is represented by a different color.

stalled since it can not know about this behavior. This effect is well-known and can be handled by coscheduling [11]. It applies to both types of hypervisors as defined by Popek et al. [12], with the only difference that for a bare metal hypervisor the coscheduling can be enforced by the hypervisor itself, while in the hosted hypervisor case the underlying operating system has to support coscheduling.

As an example for coscheduling, consider a quad-core machine with three virtual machines or parallel applications with four cores or threads each. Without coscheduling (Figure 1a), the load on the CPUs advances unevenly and in case of synchronization events, processing on one CPU is stalled until the other CPU(s) are scheduled. Coscheduling (Figure 1b) avoids this issue since all loads that can potentially depend on another are scheduled at the same time. Coscheduling is a common technique used in hypervisors such as VMware vSphere [17] and our research shows that this is also feasible for hosted hypervisors by introducing coscheduling to the host operating system, such as Linux [13].

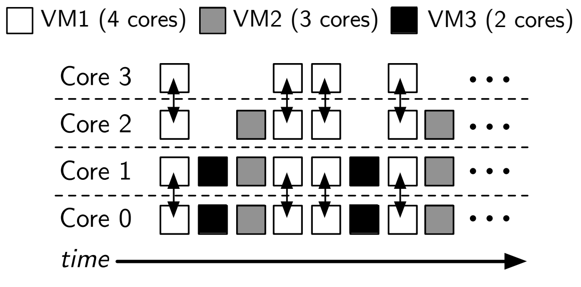
Traditional coscheduling works well as long as all the virtual machines have the same size as the host machine. However, this is not always the case. As an example, consider the situation depicted in Figure 2a. In this scenario, some cores of the host machine are idle when a smaller virtual machine is scheduled, since the other virtual machines can only be scheduled at the same time. However, the load inside the virtual machine might allow to break up the sets because not every CPU depends on every other CPU. This allows a better overall utilization of the host CPUs (Figure 2b). To utilize this observation, we propose a partial coscheduling of the virtual machines, that still has the benefits of coscheduling but is not as restrictive and results in a better overall system utilization.

However, acquiring the knowledge needed for a partial coscheduling is not trivial, since the virtual machine is a black box. An intrusive way to solve this issue could be an extended paravirtualization that allows the virtual machine to communicate the relations of CPUs to the host. This also requires modifications of the guest, which might not be easy to deploy. In this paper, we adapt *SPCD*, a technique for memory sharing detection presented by Diener et al. [7] to obtain the knowledge about CPU dependencies inside the virtual machine without modifying the guest. SPCD provides information about how many memory regions are shared between processes and threads. Based on the assumption that memory sharing also means a dependency and the fact that virtual CPUs are mapped to threads or processes in the hypervisor or host operating system respectively, SPCD can detect the relation between virtual CPUs. With this metric, we are able to automatically select sets of virtual CPUs that should be coscheduled.

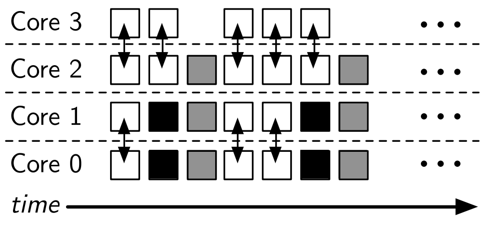
We make the following main contributions in this paper:

- We show that the utilization of a host can be improved by building coscheduling subsets based on the memory sharing behavior.
- We demonstrate that the memory access behavior inside the virtual machines can be detected through page sharing detection on the host.
- We propose an algorithm to detect sets based on the memory access behavior inside the guest.
- We show that the partial coscheduling of virtual machines can be accomplished without modifying the guest, and can outperform the full coscheduling approach.

The rest of the paper is structured as follows: In the next section, we will describe both the coscheduling implementation and memory sharing detection mechanism we used for our partial coscheduling approach. Section 3 introduces the algorithm used to detect the subsets inside the guest in order to create partial coscheduling groups. Section 4 describes the experimental setup. In Section 5, we will present and analyze our experimental results. Section 6 discusses related work. Finally, we present our conclusions in Section 7.



(a) One set per virtual machine (full coscheduling).



(b) Optimized sets (partial coscheduling).

Figure 2: Scheduling situation with coscheduling and three virtual machines VM1–VM3. VM2 and VM3 are smaller than the host machine (3 and 2 cores, respectively). Each virtual machine is represented by a different color. Arrows indicate interaction between virtual cores.

2. BACKGROUND

In this section, we will explain the Linux coscheduler and the page sharing detection mechanism we utilized to detect the memory sharing between processes and threads inside the virtual machine.

2.1 Coscheduling in Linux

For our evaluation, we used a modified Linux 3.8 kernel with integrated coscheduling support. This coscheduler was developed by Schönherr et al. and is described in [13]. The implementation is the result of an integration technique to realize coscheduling functionality in already existing operating system schedulers without disturbing their usual behavior. Especially for us, it retains the behavior in interactive settings and allows to freely mix coscheduled and non-coscheduled workloads, while still providing strong coscheduling guarantees.

Groups of tasks, which should be executed simultaneously, along with further configuration settings are specified via the `cgroup` interface of Linux. These groups do not have to span the whole machine: smaller groups are placed and balanced with the usual Linux rules. Additionally, it is possible to create and change these group definitions dynamically, so that they can be adapted to the detected situation.

2.2 Memory Sharing Detection

From the operating system point of view, a virtual machine in Linux is a parallel application that consists of several threads, executing one thread per virtualized core. To detect the memory sharing pattern of the virtual machine threads, we adapted an existing mechanism, *Shared Pages Communication Detection (SPCD)* [7], that performs memory sharing detection for parallel applications. SPCD uses page faults of parallel applications to detect memory accesses to shared memory regions. Page faults within the same region from different threads are considered communication events. As page faults only happen on the first access to a page during normal operation, SPCD introduces extra page faults during the execution of the parallel application by periodically clearing the page present bit of pages that can be shared. As these extra page faults do not represent missing information in the page table, they can be resolved with a low overhead without involving the normal kernel routines. SPCD produces a *communication matrix*, where each cell of the matrix contains the number of communication events between the pair of tasks. An example matrix is shown in Figure 3.

3. GROUP DETECTION

Based on the communication pattern detected with the mechanism presented in Section 2.2, we developed an algorithm to determine the optimal set size and therefore which virtual CPUs should be coscheduled.

For the algorithm, we define d as the dependency value between two subtrees. We compute d by defining X as set that contains every task of the one subtree and Y as the set that contain all the tasks of the other subtree. Let x and y be elements of X and Y respectively. We calculate the d values of two subtrees by accumulating the communication events (*CE*) of all pairs of tasks and weigh it with the total number of pairs:

$$d = \frac{\sum\limits_{\forall x \in X} \sum\limits_{\forall y \in Y} CE(x,y)}{|X| \cdot |Y|} \tag{1}$$

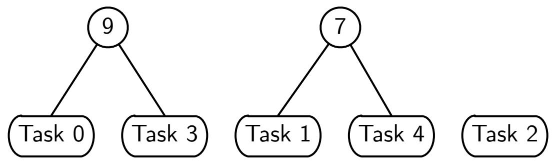
Based on the dependency values, we create a binary tree. We start with subtrees that only hold one task each. Therefore, the initial number of subtrees is equal to the number of virtual CPUs. Next, we calculate the dependency between every subtree. We then chose the pairs with the highest dependency value for new subtrees. This process is repeated until the final tree is created. For the communication matrix shown in Figure 3, Figure 4 depicts the steps of this algorithm. The leaves of the tree contain the virtual CPU ID and the internal nodes contain the dependency value d between the left and right subtrees. Leaves are considered to have a value of $d = 0$.

To determine the subsets that should be coscheduled, an algorithm walks through the tree and splits the tree if the dependency values of the subtrees are more than a certain factor t of the dependency values of the current tree (cf. Figure 4c). Each split results in a new coscheduled subset, which might be divided further when the algorithm advances.

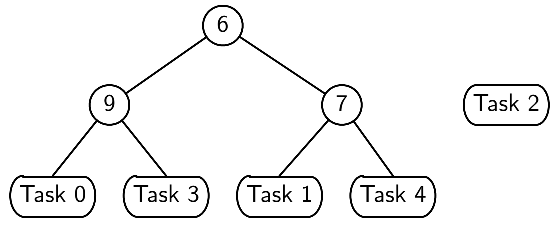
Even though the algorithm obviously has a high complexity, in practice it does not result in a significant performance degradation since the size of virtual machines is normally limited to less than a few hundred CPUs.

	Task 0	Task 1	Task 2	Task 3	Task 4
Task 0	\	6	1	9	6
Task 1	6	\	1	6	7
Task 2	1	1	\	1	1
Task 3	9	6	1	\	6
Task 4	6	7	1	6	\

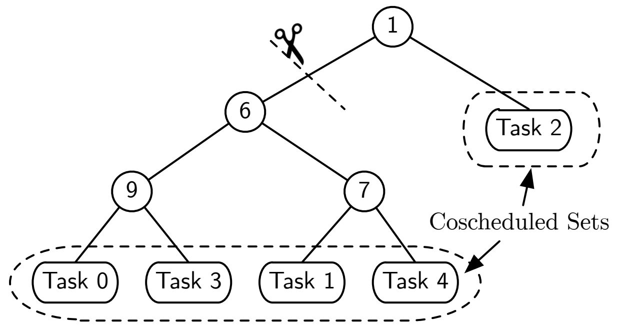
Figure 3: Example communication matrix for a group of 5 tasks. Each cell of the matrix contains the number of communication events (CE) for each pair of tasks. For example, there are 6 communication events between tasks 0 and 1.



(a) First step.



(b) Second step.



(c) Third step with group selection.

Figure 4: Example for building a dependency tree using the communication matrix shown in Figure 3. Leaves contain the tasks. Internal nodes contain the dependency value d. Subset selection was performed with $t = 1.5$.

4. EXPERIMENTAL SETUP

To evaluate our proposed partial coscheduling, we performed experiments on two different host machines. *Lynnfield* is a single socket machine with an Intel Core i7 860 CPU of the Lynnfield generation with simultaneous multithreading (SMT), running at 2.8 GHz with 8 GB of memory and 8 hardware threads. *Gainestown* is a dual socket machine with two Intel Xeon X5570 CPUs of the Gainestown generation with deactivated SMT, running at 2.93 GHz with 48 GB of memory and 8 hardware threads. Both

Table 1: Overview of the OpenMP version of the NAS Parallel Benchmarks (NPB) used in the evaluation.

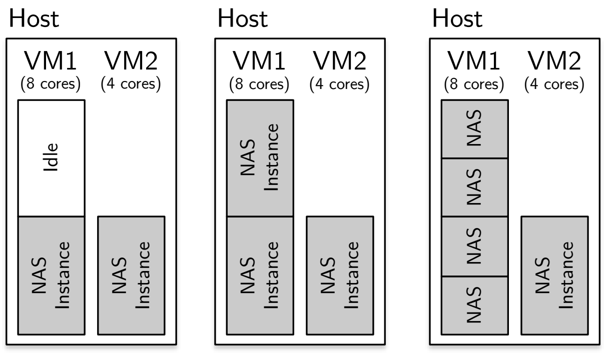
Name	Description	Focus
BT	Block Tridiagonal	Floating point performance
CG	Conjugate Gradient	Irregular communication
EP	Embarrassingly Parallel	Floating point performance
IS	Integer Sort	Integer performance
LU	Lower and Upper Triangular	Regular communication
MG	Multigrid	Regular communication
SP	Scalar Pentadiagonal	Floating point performance
UA	Unstructured Adaptive	Irregular communication

machines run Gentoo Linux and the Linux kernel (version 3.8) with the coscheduling implementation described in Section 2.1. We use the Linux Kernel Virtual Machine (KVM) [10] as the hypervisor. The virtual machines also run Gentoo Linux with kernel version 3.8.

For our experiments, we created two virtual machines (*VM1* and *VM2*) on each host. VM1 has the same number of CPUs as the host (8), while VM2 has half the number of cores (4). As workloads inside the virtual machines, we use the OpenMP implementation of the NAS Parallel Benchmarks (NPB) developed by Jin et al. [8] with the *B* input size. The NAS Parallel Benchmarks measure the performance of parallel systems based on various common scientific calculations and computing problems. An overview of the benchmarks is shown in Table 1. In the smaller of the two virtual machines (VM2), we run one NAS instance that uses all the cores of the virtual machine. In the bigger virtual machine (VM1), we studied three load scenarios, *single*, *double* and *quad*. The *single* scenario is a partial load with one NAS instance that uses only half the number of cores (Figure 5a). The *double* scenario runs two NAS instances at the same time, each using half of the virtual machine (4 cores). This scenario puts the virtual machine under full load (Figure 5b). The *quad* scenario uses four NAS instances at the same time, each using a quarter (2 cores) of the virtual machine (Figure 5c). We pin the threads of the NAS instances to a fixed CPU of the virtual machine.

With these scenarios, we first established a baseline by defining a coscheduled set for each virtual machine, which is traditionally done by hypervisors. We refer to this baseline as *full coscheduling*. Second, we measured the execution time of a partial coscheduling with static subsets build based on the prior knowledge, how the NAS benchmarks will occupy the virtual CPUs. We refer to this scenario as *Partial Static*. The third measurement was an automatic subset creation at runtime based on our partial set detection described in Section 3 with $t = 1.5$ for all benchmarks. A daemon in the background reevaluated the set composition every five seconds. SPCD was configured with the same configuration as described in the original paper [7]. We reset the communication matrix of the SPCD after starting the benchmark to avoid a false dependency detection from shared libraries and benchmark initialization. We refer to this scenario as *Partial Auto*.

We chose these experiments to show the potential of the partial coscheduling approach. The possible increase of system utilization strongly depends on the specific scenario. If, for example, we would have conducted our experiments with two virtual machines that both had the same size of the host machine, there would have been no room for improvements since the host would have been already fully utilized.



(a) *Single* scenario. (b) *Double* scenario. (c) *Quad* scenario.

Figure 5: The three load scenarios that were evaluated.

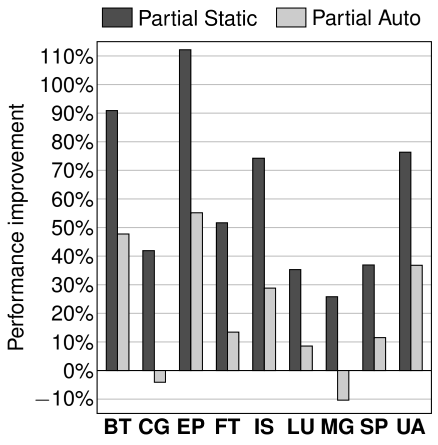
5. RESULTS AND EVALUATION

Figures 6 and 7 show the performance improvement of the static partial coscheduling and the automatic partial coscheduling with the runtime of the full coscheduling scenario as baseline. The execution times were calculated by accumulating the execution time of the NAS instance in the smaller virtual machine and the bigger virtual machine. For the *double* and *quad* scenarios, the runtime of the bigger virtual machine was determined by the average runtime of all NAS instances.

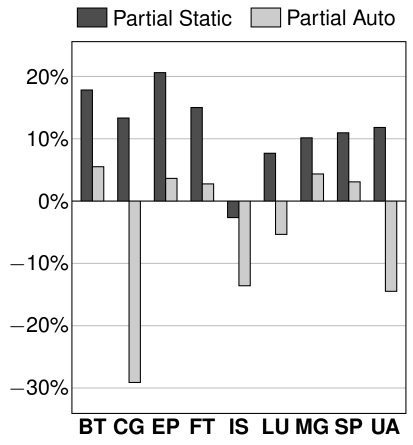
From the benchmark setup we expect a performance improvement of 33% for the *double* and *quad* scenario. This expectation is based on the fact that through the full coscheduling when the smaller virtual machine is scheduled half of the CPUs are not used and therefore wasted. Through the partial coscheduling, a part of the bigger machine can be scheduled in this slot, hence resulting in the performance improvement. For the *single* scenario, the theoretical performance improvement is 100%. Since half of the bigger virtual machine is idle, those virtual CPUs do not need to be schedule but instead the smaller virtual machine can be scheduled in this slot. However, the expected performance improvement will be lower, since one of the cores not occupied by the NAS benchmark will eventually run some other operating system thread and therefore make the scheduling of the whole subset necessary.

First, we want to discuss the results of the Gainestown machine. For the *quad* scenario, the results of the static partial coscheduling show results close to the expected performance improvements, except for the LU and MG benchmarks. Through the higher overall system load, resources like the cache are stronger congested. We presume that those benchmarks are more prone to those effects than the other benchmarks. Compared to the *quad* scenario, in the *double* scenario no result comes close to the expected performance gain. We assume that based on the granularity running a housekeeping task in the operating system leads to a higher performance loss than in the *quad* case. For the *single* scenario, only the BT and EP benchmarks come close to the expected results. Contention effects are much higher in this scenario since the average system load rises from only 50% to 100%. Furthermore, the housekeeping of both the host and guest operating system might have an impact, as discussed above.

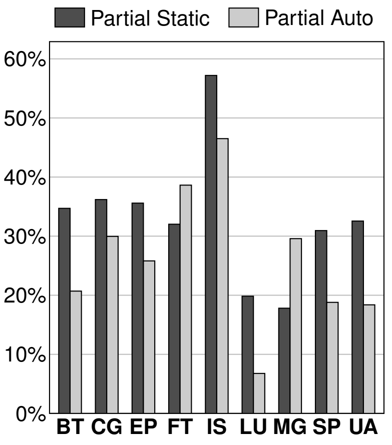
The automatic partial coscheduling shows lower performance improvements, since it needs some time to detect the relation between the virtual cores and introduces additional overhead from the memory sharing detection. However, most of the time it still achieves a better performance than the full coscheduling scenario. Major exceptions are the CG, UA, and IS benchmarks in the *double* scenario. For the IS benchmark, the runtime is too small for the au-



(a) Single.

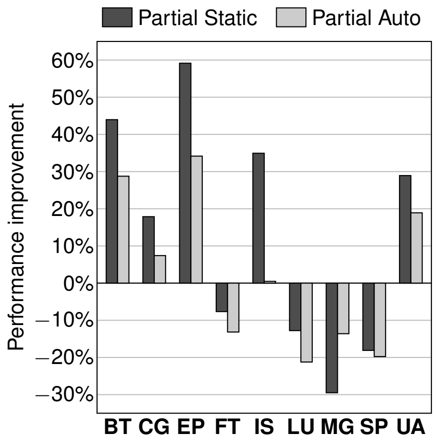


(b) Double.

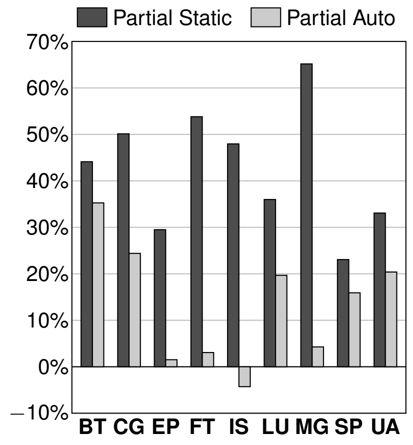


(c) Quad.

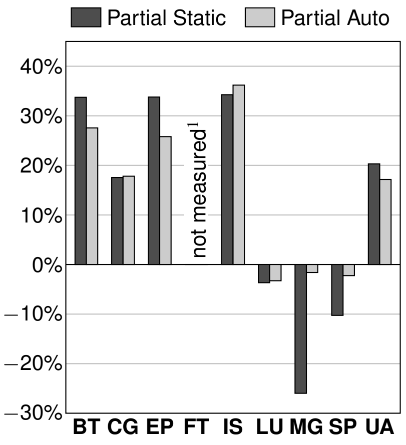
Figure 6: Performance improvement on the **Gainestown** machine, normalized to the baseline (full coscheduling).



(a) Single.



(b) Double.



(c) Quad.

Figure 7: Performance improvement on the **Lynnfield** machine, normalized to the baseline (full coscheduling).

tomatic set detection. For the former two benchmarks we examined the set detection and observed that it was not optimal in respect to the real relation of virtual CPUs. The reason is the parameter t that we had chosen for the detection algorithm. For those benchmarks, another value for the parameter would have resulted in a better placement. These results show that this parameter is crucial and it is challenging to find a value that fits for most problems.

The results are different for the Lynnfield machine. Even though the partial coscheduling shows an improvement compared to the full scheduling, it is not as predictable as with the Gainestown machine. In our opinion, the reason is that the experiments on the Lynnfield machine were conducted with SMT enabled. As previous research has shown, the NAS benchmarks are affected by resource contention and are strongly affected by the scheduling on SMT enabled machines [5]. As a result, these effects have to be taken into account when applying the static or automatic partial coscheduling approach to virtual machine scheduling on an SMT system.

[1]The FT benchmark for the *quad* scenario was omitted for the Lynnfield machine, since it would have used more memory then available on that machine.

6. RELATED WORK

Bai et al. [2] identify lock contention as a challenge for virtual-to-physical CPU scheduling, and present a full coscheduling solution for the Xen hypervisor. When running parallel benchmarks that need to synchronize often, their solution outperforms the default Xen scheduler. Benchmarks with little synchronization can not benefit from the full coscheduling and might even lose performance.

Uhlig et al. [16] also identify the lock contention issue. They propose a paravirtualization approach to cope with this challenge, where the guest operating system gives the host information about locking relations. Hence, the approach is applicable to kernel space locking. Since paravirtualization requires modification of the guest, they acknowledge that this is not always possible. To circumvent this issue, they propose to monitor transitions of the guest between user and system mode and assume that it is only safe to preempt if the guest is in user mode and therefore its advancement can not be hindered by a kernel lock.

Yu et al. [18] also uses the term partial coscheduling, however they use it in a different context. In their work they tackle the issue of parallel execution of two or more coscheduled sets. This feature is inherent to the coscheduling implementation we used for our experiments and also obviously necessary for our approach. The authors do not consider the problem of set optimization for one virtual machine.

VMware vSphere uses a relaxed coscheduling since version 3 [17]. It tries to reduce the skew between virtual CPUs and therefore minimize the effects of different speed of progress of the CPUs. This mechanism was also described in a recent patent [19].

The problem discussed in this paper can be compared to the challenge of scheduling parallel applications, where it is also beneficial to run them coscheduled [14]. However, in this case the determination of coscheduled sets is more easy, since it is most of the time the whole parallel application. Previous work already suggest scheduling algorithms for such a purpose [3, 4, 15].

The problem of determining feasible sets for coscheduling might also be solved by graph partitioning algorithms such as [1, 6, 9]. However, most of these algorithms are designed for offline partitioning of very large graphs. Since we perform the group detection during execution, we developed the much less complex algorithm described in Section 3.

7. CONCLUSIONS

In this paper, we have shown that it is possible to optimize the coscheduling of virtual machines with partial coscheduling based on the load inside the virtual machine both with and without prior knowledge of the load. The performance improvement with the chosen scenarios reached up to 100% with the static partial coscheduling and up to 50% with the automatic partial coscheduling. We utilized memory sharing patterns to determine the relation of threads and processes inside the virtual machine. Furthermore, we presented a tree-based algorithm to perform the coscheduling based on the memory sharing data. We conducted detailed measurements to show that the approach is feasible. We compared the coscheduling of complete virtual machines with a static coscheduling based on the knowledge of the virtual machines' load. We also showed that these sets can be determined by an automatic algorithm, that, even though not showing the exact same improvements, can often increase the overall performance.

For the future, we intend to improve the algorithm described in Section 3 to be also topology aware, since on larger machines non-uniform memory access (NUMA) effects can have an additional impact on the performance and must be taken into account for the coscheduling of the virtual machines.

8. REFERENCES

[1] K. Andreev and H. Räcke. Balanced graph partitioning. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 120–124, 2004.

[2] Y. Bai, C. Xu, and Z. Li. Task-aware based co-scheduling for virtual machine system. In *ACM Symposium on Applied Computing (SAC)*, pages 181–188, 2010.

[3] F. Broquedis, O. Aumage, B. Goglin, S. Thibault, P.-A. Wacrenier, and R. Namyst. Structuring the execution of OpenMP applications for multicore architectures. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–10, 2010.

[4] F. Broquedis, F. Diakhaté, S. Thibault, O. Aumage, R. Namyst, and P.-A. Wacrenier. Scheduling Dynamic OpenMP Applications over Multicore Architectures. In *International Workshop on OpenMP (IWOMP)*, pages 170–180, 2008.

[5] A. Busse, J. H. Schönherr, M. Diener, G. Mühl, and J. Richling. Analyzing resource interdependencies in multi-core architectures to improve scheduling decisions. In *ACM Symposium on Applied Computing (SAC)*, pages 1595–1602, 2013.

[6] A. Caldwell, A. Kahng, and I. Markov. Design and Implementation of the Fiduccia-Mattheyses Heuristic for VLSI Netlist Partitioning. In *Algorithm Engineering and Experimentation*, volume 1619 of *Lecture Notes in Computer Science*, pages 182–198. 1999.

[7] M. Diener, E. H. M. Cruz, and P. O. A. Navaux. Communication-Based Mapping Using Shared Pages. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, pages 700–711, 2013.

[8] H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. Technical report, Oct. 1999.

[9] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[10] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium*, pages 225–230, 2007.

[11] J. Ousterhout. Scheduling techniques for concurrent systems. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 22–30, 1982.

[12] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *ACM SIGOPS Operating Systems Review*, 7(4):121, 1973.

[13] J. H. Schönherr, B. Lutz, and J. Richling. Non-intrusive coscheduling for general purpose operating systems. In *International Conference on Multicore Software Engineering, Performance, and Tools (MSEPT)*, pages 66–77, 2012.

[14] J. H. Schönherr, B. Juurlink, and J. Richling. Topology-aware equipartitioning with coscheduling on multicore systems. In *IEEE International Workshop on Multi-/Many-core Computing Systems (MuCoCoS)*, pages 1–8, 2013.

[15] S. Thibault, R. Namyst, and P.-A. Wacrenier. Building portable thread schedulers for hierarchical multiprocessors: The bubblesched framework. In *Euro-Par 2007 Parallel Processing*, volume 4641 of *Lecture Notes in Computer Science*, pages 42–51. 2007.

[16] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards scalable multiprocessor virtual machines. In *Conference on Virtual Machine Research And Technology Symposium (VM)*, 2004.

[17] VMware, Inc. *VMware vSphere: The CPU Scheduler in VMware ESX 4.1*, 2010. White paper.

[18] Y. Yu, Y. Wang, H. Guo, and X. He. Hybrid Co-scheduling Optimizations for Concurrent Applications in Virtualized Environments. In *IEEE International Conference on Networking, Architecture, and Storage*, pages 20–29, 2011.

[19] H. Zheng and C. Waldspurger. Implicit co-scheduling of cpus, Aug. 28 2014. US Patent App. 14/273,022.