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

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

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

ABSTRACT

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

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

Keywords

Scheduling, Scheduling strategies, Heterogeneous systems

1. INTRODUCTION

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

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

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

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

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

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

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

COSH 2016 Jan 19, 2016, Prague, CZ
© 2016, All rights owned by authors. Published in the TUM library.
DOI: 10.14459/2016md1286954

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

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

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

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

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

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

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

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

2. RELATED WORK

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

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

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

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

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

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

3. SCHEDULING STRATEGIES

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

3.1 VarySched

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

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

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

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

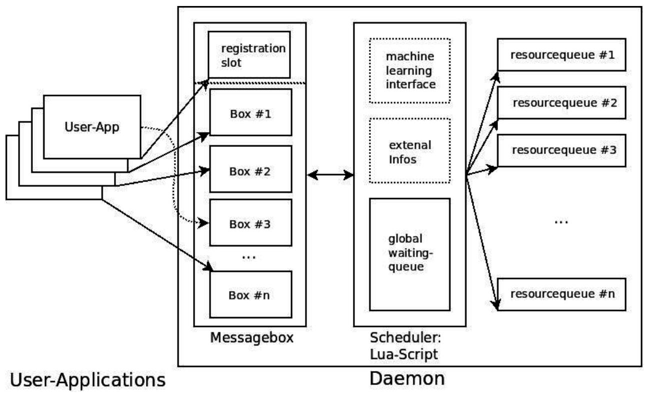

Figure 1: Architecture of VarySched. Applications register their kernels in the messageboxes. The scheduler takes the kernels, schedules them, and updates the kernels with the resource information.

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

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

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

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

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

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

3.2 Scheduling Strategies

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

3.2.1 Short-term Scheduler

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

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

3.2.2 Long-term Scheduler

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

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

3.2.3 Banking-based Scheduler

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

In our tests we start with a full budget of one hundred credits. Every five seconds 15 credits are added to the budget if the resource governor is set to high, and ten credits are added if the governor is set to low. Running a task on the GPU costs ten credits; five credits are needed for all CPU cores, one credit for a single CPU core.

3.2.4 *Constraint-based Scheduler*

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

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

3.3 Test Environment and Applications

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

Matrix-Matrix Multiplication.

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

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

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

LAMA Application.

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

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

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

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

Jetson-TK1.

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

4. EVALUATION

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

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

4.1 Matrix-Matrix Multiplication

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

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

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

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

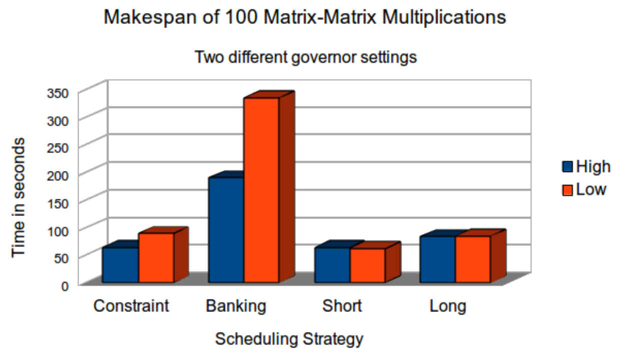

Figure 2: Makespan of one hundred instances of the matrix-matrix multiplication application with different governor states. The credit-based strategies are strongly influenced by the choice of the governor while the short-term and long-term scheduler are almost unaffected.

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

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

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

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

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

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

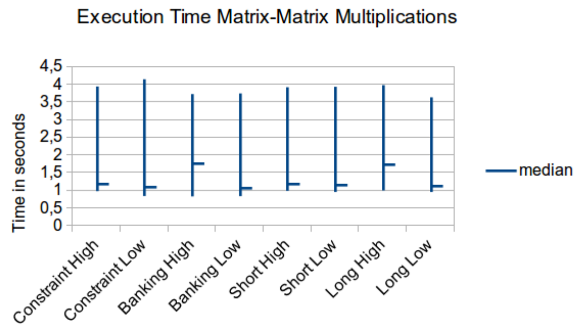

Figure 3: Runtime of matrix-matrix multiplication for different schedulers and governors.

4.2 LAMA Application

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

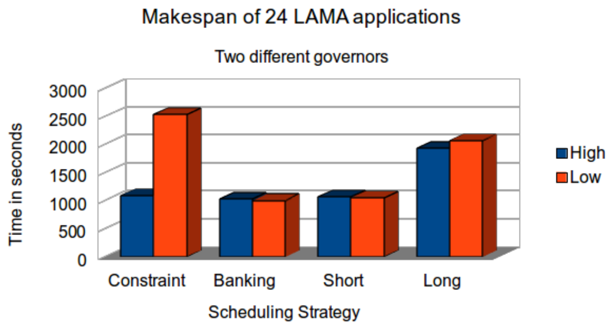

Figure 4: Makespan of one hundred instances of the LAMA application with different governor states.

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

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

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

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

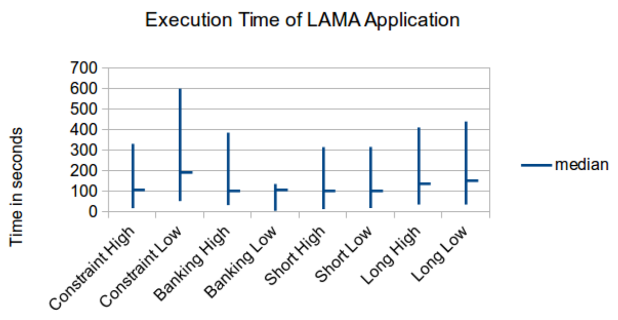

Figure 5: The runtime of the LAMA applications with different governors.

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

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

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

5. CONCLUSION

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

Acknowledgments

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

6. REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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