Performance-centric scheduling with task migration for a heterogeneous compute node in the data center

Achim Lösch, Tobias Beisel, Tobias Kenter, Christian Plessl, and Marco Platzner
Paderborn University, 33098 Paderborn, Germany
Email: {achim.loesch, tbeisel, kenter, christian.plessl, platzner}@uni-paderborn.de

***Abstract*—The use of heterogeneous computing resources, such as Graphic Processing Units or other specialized coprocessors, has become widespread in recent years because of their performance and energy efficiency advantages. Approaches for managing and scheduling tasks to heterogeneous resources are still subject to research. Although queuing systems have recently been extended to support accelerator resources, a general solution that manages heterogeneous resources at the operating system-level to exploit a global view of the system state is still missing.**

In this paper we present a user space scheduler that enables task scheduling and migration on heterogeneous processing resources in Linux. Using run queues for available resources we perform scheduling decisions based on the system state and on task characterization from earlier measurements. With a programming pattern that supports the integration of checkpoints into applications, we preempt tasks and migrate them between three very different compute resources. Considering static and dynamic workload scenarios, we show that this approach can gain up to 17% performance, on average 7%, by effectively avoiding idle resources. We demonstrate that a work-conserving strategy without migration is no suitable alternative.

I. Introduction

Heterogeneous computing in the form of systems comprising different, often specialized computing resources has a long history. Integrating heterogeneous resources into one node or a system-on-chip is particularly wide-spread for embedded systems with tight performance and energy constraints. In the last decade we have also seen a rapid emergence of heterogeneous computing in the general purpose and high performance computing domains. Today, many of these systems employ heterogeneous nodes that use accelerators such as GPUs, Xeon Phi, or FPGAs in addition to CPUs. The driving force behind this rapid adoption is the opportunity to significantly improve the performance and energy-efficiency of applications by executing them on the most suitable computing resource.

Current usage scenarios for accelerator resources are almost exclusively focused on increasing the performance or energy efficiency of one specific application at a time. Thus, it is the application designer, who designates which compute steps of the program are executed on each specific resource, based on performance or efficiency targets. As long as the designer meets these goals, idle times of a specific compute resource are totally acceptable in such scenarios.

Typical data center or cloud computing environments are centered around the fundamentally different operating concept of consolidated execution of diverse workloads on a common pool of resources [1]. The additional operational goal for those systems is a reasonably high utilization of compute resources, which is a prerequisite for cost efficiency. When diverse workloads are to be distributed to heterogeneous resources, no longer the application, but a central instance must govern the assignment of individual compute steps to the resources. At the level of a single compute node with heterogeneous resources, this central instance is the operating system, which needs to be extended with a heterogeneous scheduler.

In this work, we present an execution environment around a heterogeneous scheduler in Linux user space that efficiently distributes diverse task sets to three resources, a general-purpose CPU, a GPGPU and an FPGA. We also present a programming pattern that allows task migration between resources through cooperative multitasking and show that in our scenario, it allows much more efficient utilization of heterogeneous resources than state-of-the-art approaches with run-to-completion execution models. Our scheduler supports different policies around performance or energy optimization and uses application characterization features that are generated offline. In this work, we focus on performance-centric scheduling.

An early version of the cooperative multitasking pattern and scheduler was presented in our previous work [2]. Some requirements of this previous pattern are now encapsulated into the new runtime library. The earlier scheduler mimicked the Linux completely fair scheduler CFS, and thus it was solely focused on a fairness-guided policy, which caused a high number of context switches and migrations. Therefore it relied on applications with minimal intermediate state for good performance. Also, the earlier work was limited to CPU and GPU scheduling.

II. Related Work

Existing work that investigates scheduling for heterogeneous resources is centered around run-to-completion concepts of tasks on non-CPU resources. The HEFT algorithm [3] presents scheduling strategies for real time systems equipped with GPUs and FPGAs with a special focus on dependency graphs. Similarly, StarPU [4] and DAGuE [5] are well-known frameworks relying on Directed Acyclic Graphs (DAGs) to compute a schedule and dynamically execute tasks on CPU and GPU resources for real-time environments. In contrast to those approaches, our work focuses on tasks without dependencies. Instead we also consider heterogeneous task migration, providing the scheduler with a larger decision space.

Published in: *Proc. Design, Automation and Test in Europe Conf. (DATE)*

Other approaches on heterogeneous scheduling focus on the criteria to base scheduling decisions on. Gregg et al. [6] use historical execution time data for choosing the best suited hardware at runtime, Choi et al. [7] extend this approach with estimated remaining execution times of already scheduled tasks. Wen et al. [8] use support vector machines classifiers based on static code features and runtime features to classify OpenCL kernels for CPU or GPU execution. All these just considered two execution resources and thus work with a simple speedup ratio. We present a way to extend this ratio to three or more targets, but, similar to Gregg et al. [6] rely on data from previous measurements.

Task preemption on FPGAs has been investigated since their use has been growing in the 2000s, in the form of performing full context saves and restores for multitasking [9] or of rolling back a task in favor of another task [10]. Checkpointing has been further examined in several works over the following years, proposing state machines for a well-defined and storable state [11], or adding on-chip monitoring for retrieving the context information [12] building upon the Berkeley Lab Checkpoint/Restart (BLCR) library [13]. Similar approaches have been transferred to the GPU domain. The authors of [14] also use the BLCR to write the state of an application process to a file without code modifications. In [15], checkpointing within CUDA programmed GPU kernels is enabled. None of these approaches have been applied to both GPU and FPGA accelerators at a time, and, just like our approach, they require the user to provide specialized kernels for the accelerators.

Approaches to transparent, compiler-based task migration in multi-ISA systems have been presented by DeVuyst et al. [16] and Venkat et al. [17]. In contrast to our work, they require shared memory regions, which are not commonplace in heterogeneous systems as of now.

III. Concepts

In this section, we first give some details on the scenario motivated in the introduction and then present the concepts to tackle its challenges.

A. Scenario

The platform we target in our work is a single compute node with heterogeneous resources that executes a diverse workload of different tasks, where each task is defined by its underlying application along with a set of input parameters. A huge task diversity is typical for cloud or data-center environments, where the combined demands of many users or customers are executed together on consolidated compute resources [18]. If different applications or tasks are optimized for the execution on different resources, this can lead to efficient utilization of all resources, but can also lead to undesired idle times of some resources.

Thus, we require that at least parts of the workload have implementations for several or all different resource types. These implementations, along with knowledge of their relative performance or efficiency, allow a central heterogeneous scheduler with an overview of the different resources to balance the workload between these resources in order to avoid idle times. This can result in better performance or energy efficiency on the system level, even if individual tasks are executed on non-optimally suitable resources.

In a dynamic execution scenario, any decision to execute a task on a non-optimal resource may turn out very disadvantageous, when better suited tasks for that resource arrive or when the best suited resource becomes available again. One way to revise such decisions is through migration of started tasks between different resource types. In the next subsection, we present our approach to prepare applications for such migration.

The concrete heterogeneous target platform investigated in this work consists of two *Intel Xeon E5-2609 v2* CPUs each with 4 physical cores, one *Nvidia Tesla K20c* GPU with 2496 shading units and one *Maxeler Vectis* accelerator that contains a configurable *Xilinx Virtex-6 SX475T* FPGA. Both accelerators are PCIe-cards that contain local memory (GDDR5 and DDR3 respectively) with independent memory space.

B. Programming Pattern for Cooperative Multitasking

In this discussion, we start with an application that has already been prepared to execute computational hot-spots on an accelerator or on multiple CPU cores using OpenMP. Typically, these compute phases are already encapsulated into some form of kernel code and for execution on accelerators without shared memory space, the required data transfers before and after kernel execution are specified. The left part of Figure 1 illustrates this situation for an application with GPU kernel. Note that in the Maxeler programming environment for FPGA accelerators, the host-code of an application looks very similar, but the specification of the FPGA-kernel itself is completely different.

Our programming pattern involves three structured code transformation steps and the inclusion of some boilerplate code to interact with the scheduler: first, by strip-mining the outer loop of the kernel function, that loop is split into a sequence of loop invocations with less iterations. The modified kernel in the right part of Figure 1 illustrates the invocation of one loop block with reduced iteration count. After the other upcoming transformations, between each of the kernel invocations, a scheduling decision can be performed. As second step, checkpoints are introduced. A checkpoint is a data structure that unambiguously describes the computation status between subsequent invocations of the strip-mined kernel loop. As illustrated in Figure 1, the checkpoint keeps track of the compute progress `t`, but the input and output data itself is just represented by pointers and needs not to be copied by default. As third transformation, the data transfers and the call to the compute kernel are encapsulated into individual functions that depend on a checkpoint parameter. Pointers to those functions are finally used in the boilerplate code (lower right part of Figure 1) that uses the scheduler interface to hand over control of the application execution to the global scheduler.

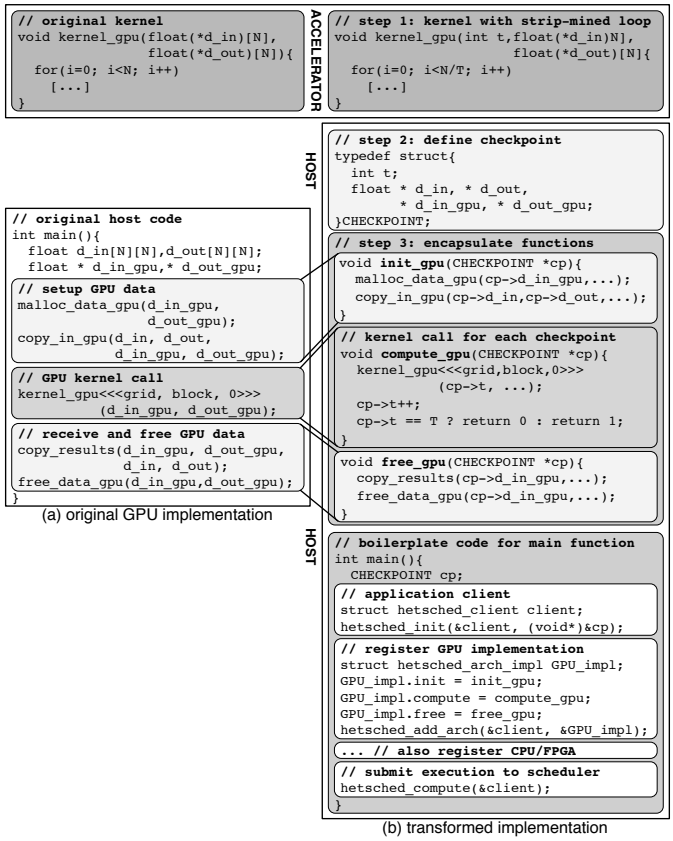


Fig. 1. Porting an application with GPU kernel (a) to the programming pattern for cooperative multitasking (b).

We applied this pattern to four different applications: (a) a Gaussian blur image processing algorithm, (b) a 2-dimensional heat transfer simulation, (c) a Markov chain steady state computation and (d) correlation matrix calculations. All of these applications have been prepared for parallel execution on CPU with *OpenMP*, for execution on a GPU using *NVidia CUDA* and for pipelined execution on an FPGA using a *Maxeler MaxJ* specification. After applying the programming pattern, execution on each resource can be interrupted at each checkpoint and migrated to any other resource using the encapsulated data transfer functions.

C. Application Characterization

In order to guide heterogeneous scheduling decisions, we characterize the suitability of each resource for the execution of each application with a set of input sizes. We measure execution times and energy consumption in advance with each task running exclusively on the system. Figure 2a illustrates the results of the execution time measurements for the correlation matrix calculation with different input data sizes. A typical pattern is that the CPU is fastest for small inputs and one or both of the accelerators, in this example only the GPU, take the lead for larger inputs.

From these results, we compute a scalar metric, the normalized affinity A, that expresses how suitable a specific resource R is for the execution of a particular task p. Since in contrast to the ratio between two fixed resources presented in [6] we investigate three different resource types, we express this ratio always with regard to the respectively best suited resource R_{opt} for each task. With the execution time t_{exe} as target metric, the corresponding performance affinity A_{perf} is computed as:

$$A_{perf}(p,R) = \frac{t_{exe}(p,R_{opt})}{t_{exe}(p,R)}$$

For the resource with lowest execution time for a task p, this affinity metric is maximal with:

$$A_{perf}(p,R_{opt}) = \frac{t_{exe}(p,R_{opt})}{t_{exe}(p,R_{opt})} = 1$$

For a resource R_d with for example double execution time, the affinity is smaller:

$$A_{perf}(p,R_d) = \frac{t_{exe}(p,R_{opt})}{2 \times t_{exe}(p,R_{opt})} = \frac{1}{2}$$

The affinity to resources which cannot execute a particular task corresponds to that of infinite runtime:

$$A_{perf}(p,R_{infeasible}) = \frac{t_{exe}(p,R_{opt})}{t} \underset{t\to\infty}{=} 0$$

D. Scheduler with variable policies

By using such affinities as guiding criterion for the scheduling decisions, the heterogeneous scheduler can easily be adapted to different optimization targets like raw performance, energy efficiency or energy delay product. Additional scheduling goals like combining performance or efficiency with fairness or responsiveness require scheduling criteria not covered in this work.

Independent of the concrete affinity used, after some simplifications, the basic principles of the scheduler as evaluated in this work can be summarized as follows:

- (A) When a task p arrives, enqueue it to the resource R with the highest affinity (Note: $A(p,R_{opt}) = 1$).
- (B) When a resource R has no tasks left and at any other resource there is at least one waiting task p, move the waiting task with the highest affinity to R to this resource (Note: $0 < A(p,R) \leq 1$).
- (C) When a resource R is executing a task p_r with non-optimal affinity and another task p_a with higher affinity arrives: $A(p_r,R) < A(p_a,R)$, preempt p_r at the next checkpoint and start executing p_a on R.
- (D) When a task p is executing on a resource with non-optimal affinity and a resource with higher affinity has no task left, migrate task p to that resource.

Additionally, the scheduler can heuristically prevent some migration decisions in order to avoid high migration overheads. Apart from this, for performance-centric scheduling with migration, as evaluated in this work, it is always beneficial to use all available resources. For energy-centric scheduling, it can be better to intentionally leave a resource idle, if no tasks with high affinity are available. We leave the exploration of this aspect for future work.

Our execution model requires exclusive usage of the accelerators GPU and FPGA, that is on these resources only one task can be running at a time. The CPU can work on more than one task concurrently with the underlying CPU scheduling performed by the Linux completely fair scheduler (CFS); the CPU also executes the heterogeneous scheduler itself, as well as the runtime library for tasks executed on the accelerators. To account for this, we perform additional load-balancing and prioritization that we cannot cover in detail here.

IV. Experiments

A. Benchmark Sets

We evaluate the functionality of our scheduler and the quality of the resulting schedules with three task sets. The task sets consist of 32 to 72 individual tasks covering a selection of the four applications that we prepared for heterogeneous migration between checkpoints: Gaussian blur (G), heat transfer simulation (H), Markov chain (M) and Correlation matrix (C). Table I presents an overview of the three task sets. In the list of tasks, we denote each task type with the abbreviation of the application name and in the subscript the input size, so M_{128} denotes a Markov chain task with an input size of 128. The colors designate for each task, on which resource it executes fastest (CPU, GPU, FPGA). We grouped the execution times t_{exe} on the most suitable resource R_{opt} into different bins and designed the task sets according two common principles:

- Following research on data-center workloads [1], [18], there are much more short running tasks then long running tasks, but the long running tasks account for the bigger part of the total execution times.
- When several task types fall into the same execution time group, we aimed for variety in both application types and most suitable resources R_{opt}.

We see that only in the group with execution times <1s, the CPU is the fastest available resource for several tasks. For tasks with longer execution times, always at least either the GPU or the FPGA are faster. *Set 1* and *Set 2* try to mimic the shape of the cumulative distribution function (CDF) plotted by Di et al. [18]. For *Set 1* we use a simple logarithmic distribution, whereas for *Set 2* we add additional long-running tasks to more closely match the heavy-tailed distribution suggested by further analysis in [18]. For *Set 3*, we aim at a bimodal distribution as suggested in [1] and use long running tasks from 4 different bins to retain some task diversity. For all task sets, we investigate two different scenarios. In the *static* scenario, all tasks of a set arrive at the same time as one common batch, whereas in the *dynamic* scenario, a new task is released every Ns. We repeat all tests with different random permutations of the sequence of arriving tasks.

B. Qualitative Result with Sample Schedule

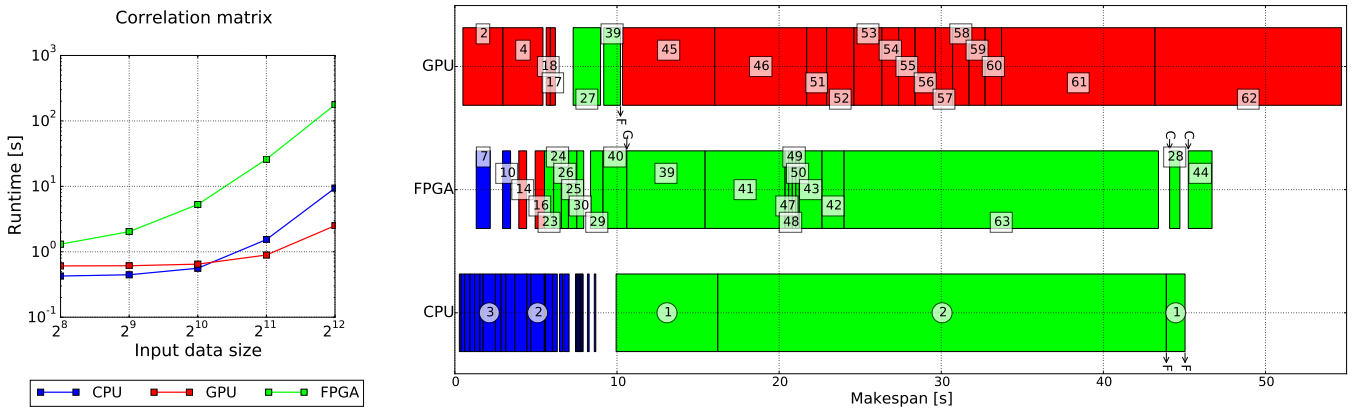
We first illustrate the functionality of the performance-centric policy our heterogeneous scheduler by analyzing a sample schedule when executing *Set 1* in the *dynamic* execution scenario with a delay of $0.25s$ between two consecutive job release times, depicted in Figure 2b. The blocks in the Gantt chart indicate which tasks are currently executing. The color denotes for which resource the task has the highest affinity. Numbers in small boxes indicate task IDs of tasks running on GPU or FPGA. On the CPU, several tasks can run concurrently. Here, numbers in circles indicate the number of concurrently running task.

TABLE I
Characterization of task sets used in the experiments. Colors designate affinity of tasks to CPU, GPU, FPGA. *For asterisked tasks, we disabled the (slow) FPGA implementations.

t_{exe}	*Set 1*: Logarithmic		*Set 2*: Heavy-tailed		*Set 3*: Bimodal	
	#Tasks	Task List	#Tasks	Task List	#Tasks	Task List
<1s	32	12x M_{128} 4x C_{256} 4x C_{512} 4x C_{1024} 4x G_{256} 4x G_{512}	32	12x M_{128} 4x C_{256} 4x C_{512} 4x C_{1024} 4x G_{256} 4x G_{512}	24	4x M_{128} 4x C_{256} 4x C_{512} 4x C_{1024} 4x G_{256} 4x G_{512}
1s-2s	16	8x M_{1024} 8x G_{1024}	16	8x M_{1024} 8x G_{1024}		
2s-4s	8	2x H_{2048} 2x C_{4096} 4x G_{2048}	8	2x H_{2048} 2x C_{4096} 4x G_{2048}		
4s-8s	4	2x M_{2048}* 2x G_{4096}	6	3x M_{2048}* 3x G_{4096}	2	1x M_{2048}* 1x G_{4096}
8s-16s	2	1x H_{4096} 1x C_{8192}*	4	2x H_{4096} 2x C_{8192}*	2	1x H_{4096} 1x C_{8192}*
16s-32s	1	1x G_{8192}	3	3x G_{8192}	2	2x G_{8192}
32s-64s			2	1x H_{8192} 1x M_{4096}*	2	1x H_{8192} 1x M_{4096}*
64s-128s			1	1x C_{16384}*		
Σ	63	= 24+20+19	72	= 24+26+22	32	= 16+9+7

Investigating the schedule, after a small initialization delay of the scheduler, the first tasks are released and CPU and GPU are starting to execute tasks that are most affine to these resources [principle (A) from Section III-D]. We see that up to around second 5, no task with highest affinity to the FPGA arrives, so the FPGA meanwhile executes two most CPU and two most GPU affine tasks [principle (B)]. In turn, around second 7, no most GPU affine task is left, so the GPU works consecutively on two FPGA tasks [again principle (B)]. In this example, many of the early arriving tasks are small, so system load is low and small gaps in the schedule occur in this phase.

Around second 10, with *task 45*, a more suitable task for the GPU arrives, so the running *task 39* is preempted for *task 45* [principle (C)] and continued on the FPGA once that becomes available. Around the same time, no most suitable tasks for the CPU are left. The CPU starts working on one, after some load-balancing two tasks that are best suited for the FPGA [again principle (B)]. When, around second 44, the FPGA finishes *task 63*, the last one from its own queue, *task 28* and then *task 44* are migrated from CPU to the better suited FPGA and are completed there [principle (D)]. Before these migrations, we see two small idle time spans on the FPGA. These result both from the migration overheads and from the time to wait for the next checkpoint on the CPU. A much smaller idle time span can be seen at the GPU, after the earlier mentioned migration of *task 39*. This illustrates that depending on the task, migration comes at a cost and can be problematic when



(a) (b)

Fig. 2. (a) Results of application characterization measurements: performance of correlation matrix calculation. (b) Sample schedule of *Set 1* in *dynamic* scenario with delay between job release times of $0.25s$.

used extensively as a method for fairness-centric scheduling.

Overall, the schedule shows that our scheduler can deal with a dynamic execution scenario and that the situations that lead to our scheduling principles actually occur in practice. It also illustrates that our scheduler is not a planning system, so individual long running tasks on a single resource, here *task 62* on GPU, can affect the overall makespan of a task set. In this regard, distribution of individual tasks over several different resources can become an interesting option.

C. Quantitative Evaluation of Migration Benefits

In order to assess the sustainability of our scheduling principles and the benefits of migration, we compare the execution of our heterogeneous scheduler with two different strategies that mimic state-of-the-art heterogeneous queuing systems with run-to-completion. In an *affinity-conserving* strategy, all tasks are executed on the resource with the highest affinity, which corresponds to our scheduling principle (A). In the presence of perfectly balanced task sets and job release sequence, this strategy is optimal. Table I illustrates that we roughly balanced the number of most affine tasks for each resource, but Figure 2b gives an indication that in terms on total runtime on the most affine resource, the CPU is less represented than GPU and FPGA. We consider such imbalances as typical for heterogeneous systems, which was one motivation for our work in the first place.

In an alternative *work-conserving* strategy, tasks are enqueued to the resource with the highest affinity, but when resources are idle, they execute waiting tasks with the next-best affinity from any other queue. This corresponds to our principles (A) and (B) and is similar to the runtime system presented e.g. in [8]. This strategy delivers good performance when no events occur that require migrations as outlined by our principles (C) and (D).

We compare both run-to-completion strategies to our heterogeneous scheduler for all three task sets, both in the *static* scenario and four *dynamic* scenarios, where tasks arrive every 0.25 to 1.0 seconds respectively. All strategies use the performance affinity A_{perf} as scheduling criterion. For each configuration, we performed tests with 10 different sequences of the tasks in the task set. We assess the makespan of each task set, as time from the release of the first task to the termination of the last running task.

TABLE II
SPEEDUP FACTORS OF OUR HETEROGENEOUS SCHEDULER WITH MIGRATION COMPARED TO RUN-TO-COMPLETION STRATEGIES.

Scenario	*static*	*dynamic*				mean
Delay [s]	0.0	0.25	0.5	0.75	1.0	
Speedup vs. *affinity-conserving* strategy						
Set 1	1.07	1.05	1.05	1.09	1.05	1.06
Set 2	1.17	1.17	1.12	1.15	1.12	1.14
Set 3	1.07	1.04	0.93	1.08	0.93	1.01
Speedup vs. *work-conserving* strategy						
Set 1	1.34	2.48	1.95	2.76	2.12	3.14
Set 2	2.40	2.03	2.28	1.80	2.02	2.01
Set 3	2.79	4.49	5.57	1.85	3.88	3.10

Table II summarizes the speedups of our heterogeneous scheduler with migration compared to the run-to-completion strategies. We first compare the speedup to the *affinity-conserving* strategy. In contrast to this strategy, our scheduler also uses non-optimal, otherwise idle resources. In the static scenario (delay 0s), our scheduler achieves speedups between 1.07x and 1.17x. In the dynamic scenarios, we see a mixed trend among the sets. For *Set 1* and *Set 2*, our scheduler achieves speedups in all scenarios with only minor differences to the static scenario. In the dynamic scenarios for *Set 3*, we see two speedups and two slowdowns, roughly balancing out each other.

Averaging over all 15 scenarios covered in the upper half of the table, we observe a speedup of 1.07x, that is our scheduler gains on average 7% performance by better utilization of all available resources. The maximal possible speedup in this comparison depends on the imbalance of the task sets and on the ability of non-optimal resources to execute tasks almost as fast as the optimal resource. On our hardware platform with three different resources, it can never exceed 3x.

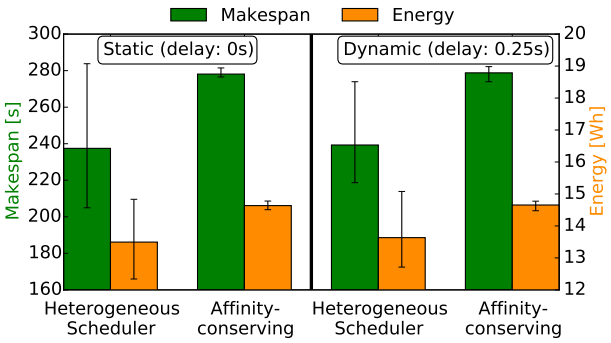


Fig. 3. Makespans and total energy used by our scheduler in comparison to *affinity-conserving* strategy for *Set 2*. Error bars denote minimal and maximal observed values. Note that both y-axes are cropped.

The comparison to the *work-conserving* strategy is quickly summarized: the attempt to utilize idle resources without the possibility to correct bad decisions through migration does not work. It is slower than our scheduler in every single scenario and on average, it causes a slowdown of 2.77x. We found some individual schedules from this strategy that are very good, on par with those of our scheduler, however many more show severe slowdowns, when long-running tasks end up on a non-optimal resource. Wen et al. [8] mitigate this shortcoming by starting the execution of the final task on both of their resources and aborting the slower one when the faster one is finished. However, that approach is only promising for the static scenario.

During task set execution, we also measure the total energy consumed by the entire compute node. In Figure 3, we present selected results by plotting next to each other the average makespans and total energy consumption during that makespan. We compare our scheduler to the *affinity-conserving* strategy and select two scenarios from *Set 2*, where our scheduler performs well in Table II. We see that when our scheduler can reduce the makespan sufficiently, even with performance-centric scheduling and with higher resource utilization, the total energy consumption can be reduced. In both scenarios from Figure 3, the energy savings are around 8%, roughly half of the gains in performance. On average over all scenarios from Table II, our scheduler induces a 3% higher energy consumption.

V. Conclusion

We have presented scenarios for the consolidated utilization of heterogeneous resources through a central scheduler. We have shown that compared to task execution purely on optimal resources, our scheduler can increase performance by on average 7% by avoiding idle times of resources. To achieve this result, task migration between resources is required to update decisions for execution on non-optimal resources, even though migration comes at a cost as our checkpoint-based approach. Even though we focused on performance optimization, in some scenarios total energy consumption was improved as a side-effect. In future work, we plan to assess the potential of energy-centric heterogeneous scheduling.

Acknowledgment

This work has been partially supported by the German Ministry for Education and Research (BMBF) under project grant 01|H11004 (ENHANCE) and the German Research Foundation (DFG) within the Collaborative Research Centre “On-The-Fly Computing” (SFB 901).

References

[1] A. K. Mishra, J. L. Hellerstein, W. Cirne, and C. R. Das, “Towards characterizing cloud backend workloads: Insights from google compute clusters,” *ACM SIGMETRICS Performance Evaluation Review*, no. 4, pp. 34–41, Mar. 2010.

[2] T. Beisel, T. Wiersema, C. Plessl, and A. Brinkmann, “Programming and scheduling model for supporting heterogeneous accelerators in Linux,” in *Workshop on Computer Architecture and Operating System Co-design (CAOS)*, Jan. 2012.

[3] H. Topcuoglu, S. Hariri, and M.-y. Wu, “Performance-effective and low-complexity task scheduling for heterogeneous computing,” *Trans. on Parallel and Distributed Systems*, no. 3, pp. 260–274, 2002.

[4] 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 and Experience*, no. 2, pp. 187–198, 2011.

[5] 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*, no. 1, pp. 37–51, 2012.

[6] C. Gregg, M. Boyer, K. Hazelwood, and K. Skadron, “Dynamic heterogeneous scheduling decisions using historical runtime data,” in *Workshop on Applications for Multi and Many Core Processors (A4MMC)*, 2011.

[7] H. J. Choi, D. O. Son, S. G. Kang, J. M. Kim, H.-H. Lee, and C. H. Kim, “An efficient scheduling scheme using estimated execution time for heterogeneous computing systems,” *The Journal of Supercomputing*, pp. 1–17, 2013.

[8] Y. Wen, Z. Wang, and M. F. P. O’Boyle, “Smart multi-task scheduling for opencl programs on cpu/gpu heterogeneous platforms,” in *Int. Conf. on High Performance Computing (HIPC)*. IEEE, Dec. 2014.

[9] H. Simmler, L. Levinson, and R. Männer, “Multitasking on FPGA coprocessors,” in *Int. Conf. on Field-Programmable Logic and Applications (FPL)*. Springer, 2000, pp. 121–130.

[10] K. Rupnow, W. Fu, and K. Compton, “Block, drop or roll(back): Alternative preemption methods for RH multi-tasking,” in *Symp. on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2009, pp. 63–70.

[11] D. Koch, C. Haubelt, and J. Teich, “Efficient hardware checkpointing: Concepts, overhead analysis, and implementation,” in *Int. Symp. Field Programmable Gate Arrays (FPGA)*. ACM, 2007, pp. 188–196.

[12] A. G. Schmidt, B. Huang, R. Sass, and M. French, “Checkpoint/restart and beyond: Resilient high performance computing with FPGAs,” in *Int. Symp. Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2011, pp. 162–169.

[13] P. H. Hargrove and J. C. Duell, “Berkeley lab checkpoint/restart (blcr) for linux clusters,” in *Journal of Physics: Conference Series*, no. 1. IOP Publishing, 2006, p. 494.

[14] H. Takizawa, K. Koyama, K. Sato, K. Komatsu, and H. Kobayashi, “CheCL: Transparent checkpointing and process migration of OpenCL applications,” in *Int. Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 2011, pp. 864–876.

[15] L. Shi, H. Chen, and T. Li, “Hybrid CPU/GPU checkpoint for GPU-based heterogeneous systems,” in *Parallel Computational Fluid Dynamics*, K. Li, Z. Xiao, Y. Wang, J. Du, and K. Li, Eds. Springer Berlin Heidelberg, 2014, pp. 470–481.

[16] M. DeVuyst, A. Venkat, and D. M. Tullsen, “Execution migration in a heterogeneous-ISA chip multiprocessor,” in *Int. Conf. on Architectural support for programming languages and operating systems (ASPLOS)*. ACM, 2012, pp. 261–272.

[17] A. Venkat and D. Tullsen, “Harnessing ISA diversity: Design of a heterogeneous-ISA chip multiprocessor,” in *Int. Symp. on Computer Architecture (ISCA)*, June 2014, pp. 121–132.

[18] S. Di, D. Kondo, and W. Cirne, “Characterization and comparison of cloud versus grid workloads,” in *IEEE Int. Conf. on Cluster Computing (CLUSTER)*. IEEE Computer Society, 2012, pp. 230–238.