Integrating Organic Computing Mechanisms into a Task-based Runtime System for Heterogeneous Systems

Thomas Becker[1]

Abstract: Modern computer architectures feature a high degree of parallelism and heterogeneity. They are deployed in different fields of application, which leads to different constraints and optimization goals. Additionally, the current and future system states have to be considered, making dynamic and proactive adaptations necessary. Organic Computing techniques offer a solution for this. A combination with runtime systems, which control execution and monitor the system, sensibly provides reduction in system complexity, efficient resource usage and the ability to dynamically adapt. To enable Organic Computing in runtime systems, we first study heterogeneous systems in different fields of application. As we identified dependability as a major concern, we study symptom-based fault detection, a light-weight technique to detect faults. We develop a mechanism based on rule-based machine learning to consider the identified requirements and constraints and dynamically balance contradicting optimization goals. Additionally, we present a scheduling mechanism to globally optimize several instances of a runtime system and show first results.

Keywords: Self-Organization; Task-based Runtime Systems; Heterogeneous Architectures.

1 Motivation

Modern computer architectures feature a high degree of parallelism and heterogeneity. Efficiently using a heterogeneous parallel architecture demands parallelizing applications. Additionally. the inclusion of different accelerators results in the need of using different programming models and deep platform knowledge. The goal should be to hide this complexity, but still enable the efficient usage of resources concerning makespan, energy consumption, heat dissipation, and system reliability. Additionally, the field of application's constraints have to be considered. For example, reaching deadlines, minimizing energy consumption and heat dissipation, and safety constraints, e.g. guaranteeing operational safety in the presence of faults in automobiles, are of great importance in embedded systems, whereas in high-performance computing and desktop computing the focus lies on maximizing throughput or minimizing the total makespan.

Besides adapting optimization goals to the field of application's constraints, they also have to be dynamically adapted to the current and prospectively possible system states and application requirements. This is because states and situations can occur that are not

[1] Karlsruher Institut für Technologie, Kaiserstraße 12, 76131 Karlsruhe Deutschland thomas.becker@kit.edu

doi:10.18420/inf2019_ws56

predictable at design time. So, e.g. in embedded systems with limited cooling capacities, load balancing to reduce heat is necessary when a certain temperature threshold is reached. However, the applications running in the system and the most sensible distribution of these applications at this point in time is not known at design time. Another important aspect is proactively avoiding disadvantageous or faulty system states.

In the literature, approaches to hide the complexity of heterogeneous systems exist. OpenCL [Opea] offers a uniform programming model, but still needs detailed hardware knowledge and is not able to dynamically adapt to new situations. This is also true for programming concepts like OpenMP [Opeb] and MPI [MPI], that support heterogeneous accelerators in their latest versions.

A different solution is offered by task-based runtime systems like HALadapt [Ki12] and StarPU [Au11]. They abstract application development from the underlying hardware via a library-based approach. Here, a user is able to define a specific functionality, e.g. a matrix multiplication, and provide different implementation versions. At runtime, these systems are then able to select a fitting pair of processing unit and implementation. However, the current state of the art only focuses on makespan minimization.

To overcome the challenges created by the deployment of heterogeneous systems in different application scenarios and fields of operation, it is necessary to dynamically compromise between contradictory optimization goals and proactively optimize the system. Self-organization (SO) is a way to provide a solution to these challenges. It is defined by Camazine et al. [Ca01] as follows:
SO is a process in which pattern at the global level of a system emerges solely from numerous interaction among the lower-level components of the system. Moreover, the rules specifying interactions among the system's components are executed using only local information, without reference to the global pattern.

However, a missing control mechanism can easily lead to undesired results. A concept that aims for controlled SO is Organic Computing (OC) [MST17]. Next to self-organisation, an important feature of OC is robustness, the ability of a system to become more resilient against disturbances and attacks from the outside [MST17]. To achieve these features, OC systems deploy an observer/controller architecture. The concept of this architecture is monitoring the current state of the system and the environment, analyzing this data, and making decision about the future system behavior based on this analysis. In the context of the aforementioned challenges, monitoring and analyzing the current and past states enables to draw conclusions about possible future system states. This is the basis to proactively trigger mechanisms that optimize the system and avoid undesired states. In total, OC enables the efficient usage of systems in different, dynamic fields of application because adapting to unknown system states gets possible.

Runtime systems already provide possibilities to capture system states by monitoring because they control task execution. Additionally, runtime systems enable dynamic adaptations

of the system. Thus, integrating the concept of organic computing into existing runtime systems is a logical solution to the aforementioned challenges. In total, the dissertation this work is based on is going to answer the following challenges:

- Research and analysis of heterogeneous parallel target platforms in different use cases and fields of operation to find occurring requirements and constraints for organic computing systems based on task-based runtime systems. Thereby, it is possible to derive sensible restrictions and configurations for a system to be created.
- A thorough study of methods and tools to capture and evaluate the system state by collecting monitoring data. Although much information has to be collected and processed, creating as little overhead as possible is of great importace.
- The exploration of the prediction of prospective system states, by among others analyzing past behavior, to enable proactively optimizing the system state. Hereby, past behavior is represented by information captured from the monitoring data.
- The study of methods to dynamically balance contradicting optimization goals based on both user inputs and alterations of the system state.

Ultimately, a core of functionality and mechanisms, which realize organic computing in a task-based runtime system and thereby enable a dynamic and proactive adaption of the system, shall be studied. This core is going to be integrated and implemented in an existing runtime system and then evaluated extensively.

The remainder of this work is organized as follows: Related work is discussed in Section 2. Section 3 introduces fundamentals necessary for this work. In Section 4, we present the concept of the dissertation. First selected results are outlined in Section 5. Finally, the paper concludes with an outlook into future work (Section 6).

2 Related Work

A well-known task-based runtime system for heterogeneous multicore architectures is StarPU [Au11]. StarPU abstracts the underlying hardware by offering offloadable tasks called *codelets*. For a codelet, the programmer can provide several different implementation versions. At runtime, StarPU selects the best performing version for each input size. In contrast to our work, StarPU solely considers the execution time.

Legion [Ba12] is a runtime system based on a data-centric programming model. It uses tasks as an abstraction of a unit of parallel execution and logical regions to support a relational model for data and has an index space and fields referred to as rows and columns. These regions can either be partitioned based on index space or spliced on their field space. A legion program executes a tree of tasks that are created recursively, thereby creating

parallelism. Each task works on a specific logical region. Legion also allows offloading, but the decision where to execute a task has to be made by the programmer.

In the literature, a variety of programming models that support heterogeneous architectures exists. The most well-known are OpenMP [Opeb], OpenCL [Opea] and MPI [MPI]. However, none of these supports the programmer in his decision where to execute a task.

The research community also developed new programming models for heterogeneous architectures. OmpSs [Du11] is a programming model based on the source-to-source compiler Mercurium and the runtime library Nanos++. It combines OpenMP and StarSs, whereby the execution model of StarSs, a thread-pool rather than OpenMP's fork-join parallelism, is used. OmpSs offers pragmas to parallelize sequential code and offload tasks to accelerators. Again, the programmer has to decide where to execute his tasks as offloading tasks has to be done explicitly.

3 Definitions and Fundamentals

3.1 Machine Learning

Machine learning can be split into supervised, unsupervised and reinforcement learning [Ma14]. Supervised machine learning tries to infer a function from labeled training data. The training data consists of pairs of feature vectors and a label. This label is predicted by the inferred function for future examples and can be a category or a real number. In the first case the problem is called classification and in the latter regression. The inferred function is called machine learning model.

In reinforcement learning, there is no label that defines the correct output for a given input. Instead, resulting actions are given a reward, which leads to a trial-and-error search for the most sensible action [SB18]. So, the goal of reinforcement learning is to maximize the given reward. In special cases, a selected action also influences all future rewards by affecting the next and thereby all following situations.

3.1.1 Rule-based Machine Learning

Rule-based machine learning (RBML) refers to machine learning methods that identify, learn and evolve a set of rules in contrast to other machine learning algorithms that create a singular model applied to all problem instances [WI95, UM09].

A common representative of RBML are learning classifier systems (LCS) [UM09]. LCS generally consists of two mechanisms, a discovery mechanism, which identifies new rules, and a learning algorithm, which improves existing rules through the acquisition of knowledge

resulting from past experience. Usually, LCS utilizes a genetic algorithm as discovery mechanism and reinforcement learning for rule improvement.

3.2 Faults

To define **faults**, **errors** and **failures**, we use the work of Salfner et al. [SLM10]:

- A **failure** refers to misbehavior that can be observed by the user. This means there may be something wrong inside the system, but as long this does not result in incorrect output there is no failure.
- An **error** is defined as the deviation of the system state from the correct state. Hence, an error may lead to the service failure of an system, but also can stay unnoticed.
- **Faults** are then the hypothesized cause of an error. This means that errors are manifestations of faults.

This work focuses on soft errors in hardware that constantly occur more often as they are caused by lowering the system voltage in the creation of energy-efficient products [SS02]. These are especially hard to detect because they are random and of temporary nature.

4 Approaches

4.1 Research of Heterogeneous Architectures in Different Fields of Operation

We analyzed the usage of heterogeneous architectures and corresponding task-based runtime systems in three different fields of operation and detected several requirements and constraints.

In a collaboration project called Envelope with TU Munich, RWTH Aachen and JGU Mainz as partners, heterogeneous architectures and the runtime system HALadapt are used in the context of high-performance computing (HPC). The analysis showed that dependability is a critical factor for future HPC systems as increasingly complex components are deployed in a continuously growing number. This leads to a rising failure rate of a single component. The large scale of HPC systems implies that traditional methods to increase system dependability like redundancy and constant periodic checkpoints reduce system efficiency and performance, and increase costs [Be08].

In a cooperation with Siemens, we used heterogeneous architectures and the runtime system EMB^2 [Sc15] in the context of non-safety-critical embedded systems with soft real-time constraints [BKS19]. A critical constraint for these systems is given by their dynamic nature.

Tasks can be triggered by recursion, signals or user interactions and therefore, not all task information is readily available. Hence, the runtime system and especially its scheduling mechanism has to be dynamic and adaptable.

A third project uses heterogeneous architectures and a task-based runtime system in the field of automotive systems in cooperation with Harman Automotive Systems. Automotive systems entail several requirements and constraints. For a subset of the functionality, safety constraints in the form of Automotive Safety Integrity Levels (ASIL) and deadlines have to be guaranteed. Therefore, this functionality has to be shielded from side effects potentially caused by e.g. operating systems or other tasks. Again, dynamics and adaptability are significant aspects as future automotive systems are not static and new situations and application updates can constantly occur.

A common overarching goal in all these projects is optimizing system efficiency and performance. A preferably optimal task mapping has to be found depending on the system state, situation, task characteristics and possible future states. Summarized, there are several contradicting optimization goals and constraints in all contemplated fields of operation. Additionally, these fields of operation are dynamic with changing situations, which puts different constraints and optimization goals in focus. So, methods that are able to dynamically and proactively adapt to these new situations and to weigh considered optimization goals accordingly are needed.

4.2 Dependability

In Section 4.1, we identified dependability and cost- and performance-efficient mechanisms to increase dependability as a major concern for future HPC systems. Possible solutions are proactive methods that are only used if a failure is bound to occur and thereby create less overhead. To enable them, a prediction model is necessary. A model can be created by collecting runtime data and evaluating the corresponding system state. The information can then be used as training data for machine learning algorithms. This is possible because faults manifest themselves as increasingly unstable performance-related behavior before escalating into failures. This means that it should be possible to predict upcoming failures by analyzing runtime data and finding patterns and correlations [WPN07]. This approach and follow-up mechanisms that allow the continuation of the system if a failure actually occurs are researched in the project Envelope. In particular, methods that migrate a process from a possibly failing computing node are studied, evaluated and combined with HALadapt.

In addition, the project includes research of symptom-based fault detection, a light-weight approach to detect faults during execution. Symptom-based fault detection is also based on the hypothesis that faults manifest themselves in performance-related behavior. Particularly, faults are detected by comparing values of performance metrics for a specific application with a database of its past behavior. If this comparison shows significant deviations, the

occurrence of a fault is assumed. As performance behavior is usually dependent on input data, the comparison has to be done for identical or at least similar input data.

4.3 Dynamics and Adaptability

The research mentioned in Section 4.1 showed that dynamic methods for system adaptation are critical in all fields of operation. Methods that are able to dynamically balance between contradicting optimization goals dependent on the current and possible future system states and situations are needed. This means that these methods require a database that stores past behavior, prediction models that are able to predict prospective system states, and machine learning algorithms that can create models to analyze current and past information to find a good balancing. Behavior can be monitored by performance counters and future behavior predicted by analyzing patterns in the execution. Examples are deterministic task sequences, tasks that usually are executed in sequence, or task combinations that lead to special side effects like high temperatures or unusual system loads. Monitoring can be easily done by and added to task-based runtime systems as they already control task execution.. Thus, additional functionality, e.g. accessing performance counters, can be executed before and after the actual task execution.

Balancing optimization goals is a multi-objective optimization problem best represented by a regression problem with multi-dimensional output variable $y = \{y_1, \ldots, y_n\}$ and input variable $x = \{x_1, \ldots, x_n\}$, where y represents the weights of the different optimization goals and x different metrics describing the current and possibly past system states.

RBML is a promising solution candidate as it is applicable to multi-objective problems and has features that compliment the need for adaptability and dynamics well. As there is no single model but rather a set of rules, it is easier to update these rules or only a relevant subset and thereby adapting them to changing situations. So, a extremely time-consuming new model training phase is not necessary to achieve adaptability. To learn, RBML uses either supervised or reinforcement learning. As creating a training set including enough labeled input data, which in this case means metric values combined with accordingly optimal balancing weights, seems impossible, reinforcement learning seems to be the better choice. A possible way to produce a reward in the context of task scheduling could be comparing the resulting schedule with a schedule created without a specific optimization goal weighing by a brute force or evolutionary algorithm. Another important aspect of RBML is that rules are usually in the form of **IF . . . THEN . . .** and therefore potentially human-readable. Especially in automotive system, traceability is critical as undesired system behavior hast to be obviated to guarantee safety-issues and quality of service.

Our approach for an analysis entity that balances the optimization goals can be seen in Figure 1. As input, we use current and past system behavior monitored by performance counters that also allows predictions about possible future system states. These values could be set in relation to available budgets for e.g. load, energy consumption or temperature. This allows to

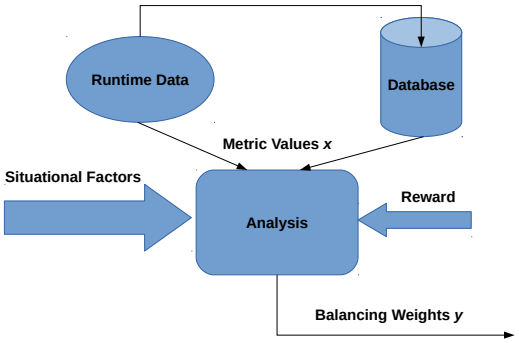

Fig. 1: Analysis entity to compute weights for the contradicting optimization goals

consider the influence of situational factors, e.g. has the battery to be used because the car is switched off, by adapting the available budgets accordingly. The actual analysis mechanism is then implemented with RBML that uses baseline schedules potentially computed by brute force or evolutionary algorithms as reward function.

4.4 Task Scheduling

After finding a balance between the optimization goals, the tasks ready-to-execute have to be scheduled according to this optimization function. As the systems we are focusing on are dynamic in nature, the scheduling algorithms also have to be dynamic and able to adapt during runtime. In our projects, we examined two different scenarios for task scheduling in a runtime system on user-level. In the first scenario, all tasks ready-to-execute are known by a single runtime system instance and so, there is one point where all information is gathered and available. In the second scenario however, the tasks are distributed over several runtime system instances. This means that a communication mechanism between the different instances is needed to coordinate the overall scheduling and to optimize the scheduling globally. To enable communication, global processing unit queues in shared memory are used [Ji09]. There, the runtime system instances can enqueue their scheduled tasks. Thereby, the global system state is updated and other instances can gather information about the availability of processing units. The problem here is the missing cooperation between the instances. Runtime instances just enqueue their tasks in their preferred queues without considering tasks of other instances whereby global optimization is not possible. Thus, our approach allows the redistribution of tasks when a new task is enqueued. When a new task is added, all processing unit queues are locked by a mutex and the adding instance is allowed to optimize the scheduling of all tasks already enqueued.

5 Results

This section contains selected results already obtained. Firstly, the study of symptom-based fault detection is presented. In this evaluation, we injected different faults and system interferences into benchmarks, a matrix multiplication and two Rodinia Benchmark Suite [Ch09] applications, using the fault injection library FINJ [Ne18]. FINJ controls the execution of benchmarks and fault-injection applications and offers a library of system interferences inspired by Tuncer et al. [Tu17].

The second section presents results for a predecessor of the mechanism introduced in Section 4.4. In the experiment, we scheduled two OpenMP applications, one memory and one compute-bound, using disjoint HALadapt instances in parallel. As benchmarks, we used MLEM, an application that uses the Maximum Likelihood Expectation Maximization (MLEM) algorithm [SV82] to reconstruct 3D images from data obtained from PET scanners, and a synthetic kernel (stressgen) that is based on the standard Linx tool stress[2]. MLEM consists of four separate kernels, two sparse vector matrix multiplications (forward and backward projection) implemented with OpenMP, and two sequential vector operations.

5.1 Symptom-based Fault Detection

We evaluated the concept of symptom-based fault detection on a modern computing system with two Intel Xeon E5-2650 v4 CPUs a 12 cores each and 128 GB with 2400MHz DDR4 SDRAM DIMM (PC4-19200). FINJ was used to control the execution of benchmarks and fault-injection applications on the same core.

In the experiments, we first created a database of performance-related metrics by executing all benchmarks 10 times without faults, collecting all available PAPI [Te10] events and filtering out events that exhibit unstable behavior. For every event included in the database, value limits were set to mark correct application behavior. Afterwards, all benchmark fault combinations were executed 10 times each. Again, the selected PAPI events were monitored and then compared to the database. If values varied significantly compared to the limits set previously, the occurrence of a fault was assumed. In Table 1, the results for a

Symptom	mMult w/o faults	mMult w faults		
	N = 300	N = 200	N = 100	N = 50
PAPI FP OPS	$54 \cdot 10^6$	$36 \cdot 10^6$	$18 \cdot 10^6$	$9 \cdot 10^6$
PAPI FML OPS	$27 \cdot 10^6$	$18 \cdot 10^6$	$9 \cdot 10^6$	$4.5 \cdot 10^6$
PAPI FADD OPS	$27 \cdot 10^6$	$18 \cdot 10^6$	$9 \cdot 10^6$	$4.5 \cdot 10^6$

Tab. 1: Results of the iteration number reduction for the matrix multiplication benchmark

reduction of loop iterations while computing a matrix multiplication can be seen. Originally,

[2] https://people.seas.harvard.edu/~apw/stress/

a multiplication of two 300×300 floating-point matrices were computed, leading to a total of $54 \cdot 10^6$ floating-point operations (FP OPS), $27 \cdot 10^6$ multiplications (FML) and $27 \cdot 10^6$ additions (FADD). We then reduced the number of outer loop iterations to $N = 200, 100$ and 50. The results show that the instructions counter are very precise as they precisely count the performed floating-point operations executed in the program. Therefore the fault is easily detectable as the fault changes the number of executed instructions.

Table 2 shows the results for the combination of SRAD and the interference benchmark dial, which creates load in the ALU by performing floating-point operations. SRAD is a diffusion method for ultrasonic and radar imaging applications based on partial differential equations used to remove noise. The additional load lead to significant increases in the L2 instruction cache misses (ICM). These correlate with the decrease in instructions cache hits (ICH) and an increase in L3 instruction cache accesses (ICA). All those symptoms were observable in 100 % of the execution runs. The additional data used for the computations lead to an increase in TLB data misses (DM) and a decrease in L2 data cache accesses (DCA). For

Symptom	SRAD w/o faults		SRAD w faults		
	∅	s	∅	s	occurr. ratio
PAPI L2 ICM	2088	154.6	2915.9	162.82	100 %
PAPI L2 ICH	20175.3	512.32	18633	169.35	100 %
PAPI L3 ICA	1950.6	101.63	2956.6	92.42	100 %
PAPI TLB DM	11899.7	1080.84	15885.1	2255.44	100 %
PAPI L2 DCA	19876232.64	2957162.47	10991900.1	14624.57	90 %

Tab. 2: Results of the combination of SRAD and dial

all symptoms, we computed the Welch's t-test [WE47], a statistical test that is used to test the hypothesis that two means belong to the same population. If that would be the case, the occurring symptom originated in correct behavior and not a fault. The results for the combination of SRAD and dial can be seen in Table 3. For all symptoms, the hypothesis could be rejected, which means that the symptoms were not caused by normal execution behavior. In total, we conducted 17 experiments and could find at least two symptoms with

Symptom	df	t	t_{crit}	p
PAPI L2 ICM	17.95	-11.66	-3.922	$8.26 \cdot 10^{-10}$
PAPI L2 ICH	10.94	9.04	4.437	$2.09 \cdot 10^{-6}$
PAPI L3 ICA	17.84	-23.16	-3.922	$1 \cdot 10^{-14}$
PAPI TLB DM	12.93	-5.04	-4.221	$2.31 \cdot 10^{-4}$
PAPI L2 DCA	9.00	9.50	4.781	$5.47 \cdot 10^{-6}$

Tab. 3: Results of the combination of SRAD and dial

occurrence ratios of 80 % in every experiment, which means we could detect all faults. A following analysis step showed that it was not possible to distinguish between fault classes by solely considering the occurred symptoms.

5.2 Co-Scheduling

The experiment [Be18] was conducted on *sk1*, a dual-socket system with two Intel Xeon Scalable Silver 4116 (Skylake-SP) processors with 12 cores each and 32 GB of RAM. Figure 2 shows the scaling behavior of both applications. The measurements confirm that MLEM is memory-bound and only scales up to six cores. For the experiment, we run 10

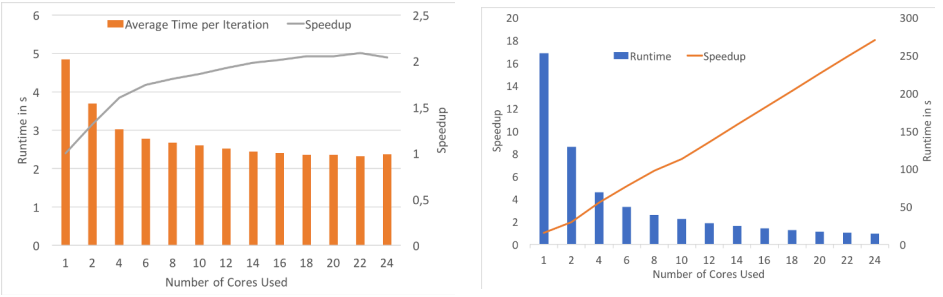

Fig. 2: Scaling behavior of both native OpenMP applications on sk1 without HALadapt

MLEM iterations. As the baseline, we use a combination of the fastest MLEM and stressgen executions which means both codes use all cores and are executed successively. This results in a total execution time of 40.44 s.

First, a profiling mechanism creates an application characteristic. The profiling executes the kernels, measures runtime and stores them in a database. For the OpenMP implementations, the mechanism varies the core number. To reduce the overhead, we implemented a scaling check and double the core number each time. The scaling check stops the profiling if adding cores no longer improves execution time. The scheduling mechanism inside a single HALadapt instance then works as follows:

1. The kernels included in the task graph are first sorted into a linked list respecting the given kernel dependencies and the user-defined call order. The kernels are then given to the scheduling mechanism in order of the sorted list.
2. For every kernel the mechanism selects the variant configuration with lowest predicted completion time. If an implementation candidate adds an additional core, the predicted execution time has to improve by at least the scaling factor else the candidate is discarded.

The shared queues enable the HALadapt instances to update the availability of the processing units. Further coordination was not yet used. This resulted in the following schedule: HALadapt maps the forward projection and the backward projection to the first four and five cores respectively. The stressgen kernel to the last 17 processing cores; leaving effectively two cores idle. Due to the missing coordination, the order in which the two task graphs are launched, matters. In the scenario in which the stressgen kernel was scheduled first

Tab. 4: Total execution time of the MLEM and stressgen co-scheduled in two instances

	Baseline	Two inst. sc.1	Two inst. sc.2
Total execution time	40.44 s	28.28 s	54.54 s
Speedup	1x	1.43x	0.74x

(scenario 2 in Table 4), it reserves 21 of 24 cores, leaving MLEM with only using the sequential kernels, increasing execution time by around 25%. If MLEM is started first, the schedule is as discussed above. In this scenario (cf. scenario 1 in Table 4), HALadapt is able to achieve a speedup of 1.43 compared to the baseline. The achieved speedup is reduced by the inclusion of the data setup and initialization in the measurements with HALadapt as a separate measurement of the kernels was not possible when two processes are scheduled. Additionally, a bug in the core binding module of HALadapt caused the problem of leaving two cores idle. Overall, the experiment shows that co-scheduling is able to achieve speedup when scheduling multiple processes but additional coordination next to shared processing queues is necessary.

6 Conclusion

This work focused on overcoming the challenges of modern computer systems in different fields of application by integration self-organization mechanisms into task-based runtime systems for heterogeneous parallel architectures. We identified the dynamic balancing of contradicting optimization goals as a major challenge for these systems. As a possible solution, we presented a new mechanism (cf. Figure 1) using rule-based machine learning with reinforcement learning. This mechanism uses information about the current, past and possible future system states, and situational factors to compute a weighting for a scheduling optimization function. Additionally, we presented first results for symptom-based fault detection, which uses performance counters to detect symptoms, and a user-level scheduling mechanism for disjoint processes. The results showed that both symptom-based fault detection and the scheduling algorithm have potential for future work and are beneficial to task-based runtime systems for heterogeneous parallel architectures. The results also showed that additional coordination is needed. Next to this, we are also going to extend symptom-based fault detection to GPUs and integrate it into HALadapt to automate the fault detection and use it to get a metric for dependability. In addition, the implementation and evaluation of the balancing mechanism in HALadapt is the major focus of our future work.

References

[Au11] Augonnet, Cédric; Thibault, Samuel; Namyst, Raymond; Wacrenier, Pierre-André: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience, 23(2):187–198, 2011.

[Ba12] Bauer, M.; Treichler, S.; Slaughter, E.; Aiken, A.: Legion: Expressing locality and independence with logical regions. In: SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. pp. 1–11, Nov 2012.

[Be08] Bergman, K.; Borkar, S.; Campbell, D.; Carlson, W.; Dally, W.; Denneau, M.; Franzon, P.; Harrod, W.; Hiller, J.; Karp, S.; Keckler, S. an d Klein, D.; Lucas, R.; Richards, M.; Scarpelli, A.; Scott, S.; Snavely, A.; Sterling, T.; Williams, R. S.; Yelick, K.; Kogge, P.: , ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems Peter Kogge, Editor & Study Lead, 2008.

[Be18] Becker, Thomas; Yang, Dai; Küstner, Tilman; Schulz, Martin: Co-Scheduling in a Task-Based Programming Model. In (Trinitis, Carsten; Weidendorfer, Josef, eds): Proceedings of the 3rd Workshop on Co-Scheduling of HPC Applications (COSH 2018). Manchester, United Kingdom, pp. 9–14, Jan 2018.

[BKS19] Becker, Thomas; Karl, Wolfgang; Schüle, Tobias: Evaluating Dynamic Task Scheduling in a Task-Based Runtime System for Heterogeneous Architectures. In (Schoeberl, Martin; Hochberger, Christian; Uhrig, Sascha; Brehm, Jürgen; Pionteck, Thilo, eds): Architecture of Computing Systems – ARCS 2019. Springer International Publishing, Cham, pp. 142–155, 2019.

[Ca01] Camazine, Scott; Franks, Nigel R.; Sneyd, James; Bonabeau, Eric; Deneubourg, Jean-Louis; Theraula, Guy: Self-Organization in Biological Systems. Princeton University Press, Princeton, NJ, USA, 2001.

[Ch09] Che, Shuai; Boyer, Michael; Meng, Jiayuan; Tarjan, David; Sheaffer, Jeremy W.; Lee, Sang-Ha; Skadron, Kevin: Rodinia: A Benchmark Suite for Heterogeneous Computing. In: Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC). IISWC '09, IEEE Computer Society, Washington, DC, USA, pp. 44–54, 2009.

[Du11] Duran, Alejandro; Ayguadé, Eduard; Badia, Rosa M.; Labarta, Jesús; Martinell, Luis; Martorell, Xavier; Planas, Judit: Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures. Parallel Processing Letters, 21:173–193, 2011.

[Ji09] Jiménez, Víctor J.; Vilanova, Lluís; Gelado, Isaac; Gil, Marisa; Fursin, Grigori; Navarro, Nacho: Predictive Runtime Code Scheduling for Heterogeneous Architectures. In (Seznec, André; Emer, Joel; O'Boyle, Michael; Martonosi, Margaret; Ungerer, Theo, eds): High Performance Embedded Architectures and Compilers. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 19–33, 2009.

[Ki12] Kicherer, Mario; Nowak, Fabian; Buchty, Rainer; Karl, Wolfgang: Seamlessly Portable Applications: Managing the Diversity of Modern Heterogeneous Systems. ACM Trans. Archit. Code Optim., 8(4):42:1–42:20, jan 2012.

[Ma14] Marsland, Stephen: Machine Learning: An Algorithmic Perspective, Second Edition. Chapman & Hall/CRC, 2nd edition, 2014.

[MPI] MPI: A Message-Passing Interface Standard , version 4.0. `https://www.mpi-forum.org/mpi-40/`.

[MST17] Müller-Schloer, Christian; Tomforde, Sven: Organic Computing – Technical Systems for Survival in the Real World. In: Autonomic Systems. 2017.

[Ne18] Netti, Alessio; Kiziltan, Zeynep; Babaoglu, Özalp; Sîrbu, Alina; Bartolini, Andrea; Borghesi, Andrea: FINJ: A Fault Injection Tool for HPC Systems. CoRR, abs/1807.10056, 2018.

[Opea] OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems, version 2.2. https://www.khronos.org/opencl/.

[Opeb] OpenMP - Application Programming Interface for Parallel Programming, version 5.0. https://www.openmp.org/.

[SB18] Sutton, Richard S.; Barto, Andrew G.: Reinforcement Learning: An Introduction. The MIT Press, second edition, 2018.

[Sc15] Schuele, Tobias: Embedded Multicore Building Blocks: Parallel Programming Made Easy. Embedded World, 2015.

[SLM10] Salfner, Felix; Lenk, Maren; Malek, Miroslaw: A Survey of Online Failure Prediction Methods. ACM Comput. Surv., 42(3):10:1–10:42, March 2010.

[SS02] Schiffmann, Wolfram; Schmitz, Robert: Technische Informatik 2. Springer Berlin Heidelberg, 01 2002.

[SV82] Shepp, L. A.; Vardi, Y.: Maximum Likelihood Reconstruction for Emission Tomography. IEEE Transactions on Medical Imaging, 1(2):113–122, Oct 1982.

[Te10] Terpstra, Dan; Jagode, Heike; You, Haihang; Dongarra, Jack: Collecting Performance Data with PAPI-C. In (Müller, Matthias S.; Resch, Michael M.; Schulz, Alexander; Nagel, Wolfgang E., eds): Tools for High Performance Computing 2009. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 157–173, 2010.

[Tu17] Tuncer, Ozan; Ates, Emre; Zhang, Yijia; Turk, Ata; Brandt, Jim; Leung, Vitus J.; Egele, Manuel; Coskun, Ayse K.: Diagnosing Performance Variations in HPC Applications Using Machine Learning. In (Kunkel, Julian M.; Yokota, Rio; Balaji, Pavan; Keyes, David, eds): High Performance Computing. Springer International Publishing, Cham, pp. 355–373, 2017.

[UM09] Urbanowicz, Ryan J.; Moore, Jason H.: Learning Classifier Systems: A Complete Introduction, Review, and Roadmap. J. Artif. Evol. App., 2009:1:1–1:25, January 2009.

[WE47] WELCH, B. L.: THE GENERALIZATION OF 'STUDENT'S' PROBLEM WHEN SEVERAL DIFFERENT POPULATION VARLANCES ARE INVOLVED. Biometrika, 34(1-2):28–35, 01 1947.

[WI95] Weiss, Sholom M.; Indurkhya, Nitin: Rule-based Machine Learning Methods for Functional Prediction. J. Artif. Int. Res., 3(1):383–403, December 1995.

[WPN07] Williams, A. W.; Pertet, S. M.; Narasimhan, P.: Tiresias: Black-Box Failure Prediction in Distributed Systems. In: 2007 IEEE International Parallel and Distributed Processing Symposium. pp. 1–8, March 2007.