Evaluating Dynamic Task Scheduling in a Task-Based Runtime System for Heterogeneous Architectures

Thomas Becker[1(✉)], Wolfgang Karl[1], and Tobias Schüle[2]

[1] Karlsruhe Institute of Technology, Kaiserstr. 12, 76131 Karlsruhe, Germany
{thomas.becker,wolfgang.karl}@kit.edu
[2] Siemens AG, Corporate Technology, 81739 Munich, Germany
tobias.schuele@siemens.com

Abstract. Heterogeneous parallel architectures present many challenges to application developers. One of the most important ones is the decision where to execute a specific task. As today's systems are often dynamic in nature, this cannot be solved at design time. A solution is offered by runtime systems that employ dynamic scheduling algorithms. Still, the question which algorithm to use remains.

In this paper, we evaluate several dynamic scheduling algorithms on a real system using different benchmarks. To be able to use the algorithms on a real system, we integrate them into a task-based runtime system. The evaluation covers different heuristic classes: In immediate mode, tasks are scheduled in the order they arrive in the system, whereas in batch mode, all ready-to-execute tasks are considered during the scheduling decision. The results show that the Minimum Completion Time and the Min-Min heuristics achieve the overall best makespans. However, if additionally scheduling fairness has to be considered as optimization goal, the Sufferage algorithm seems to be the algorithm of choice.

Keywords: Dynamic task scheduling · Heterogeneous architectures

1 Motivation

Today's computer systems are highly parallel and possess additional accelerators. Such complex heterogeneous architectures present many challenges to application developers. One of the most important questions developers are faced with is on which processing unit the execution of tasks of an application is most efficient, which may refer to best performance, lowest energy consumption or any other optimization goal. As many systems are dynamic in nature, meaning that they do not always execute the same tasks, and tasks start at unknown points in time, e.g., triggered by signals or user interactions, a static partitioning at design time is not able to optimize the system for all scenarios. To solve this problem, dynamic runtime systems may be employed, which abstract from the underlying system. The application developer simply defines his or her compute

M. Schoeberl et al. (Eds.): ARCS 2019, LNCS 11479, pp. 142–155, 2019.
https://doi.org/10.1007/978-3-030-18656-2_11

kernels representing specific functionality and is then allowed to either provide implementation variants himself or use implementation variants provided by e.g. a library. As dynamic runtime systems also take control of the execution, they can decide at runtime which implementation processing unit pair to use. To make such decisions, dynamic scheduling algorithms are needed. In the literature, a variety of different dynamic algorithms are described. Considering the fact that modern systems are used in a wide range of different scenarios and fields of application, the question remains which algorithm should be used in which scenario and which field of application. Therefore, the goal of this work is to study dynamic scheduling algorithms in several scenarios designed for heterogeneous parallel systems with an additional focus on characteristics of embedded systems, and thereby providing usage guidelines.

Hence, in this work, we evaluate selected dynamic scheduling algorithms in real-world scenarios. We utilize the Embedded Multicore Building Blocks (EMB^2), an open source runtime system and library developed by Siemens, which has been specifically designed for embedded applications, to operate the algorithms on a real system. In particular, we make the following contributions:

- We select six dynamic scheduling heuristics that we think are appropriate for the considered field of application.
- We extend the existing scheduling approach in EMB^2 with more sophisticated ones for heterogeneous systems.
- We evaluate these algorithms on a real system using a GPU as accelerator and investigate their behavior in terms of different metrics.
- We give guidelines which algorithms to choose.

The remainder of this paper is structured as follows: In Sect. 2, we briefly introduce the fundamentals of our work. The scheduling algorithms, EMB^2 and the extensions to EMB^2 are presented in Sect. 3. Section 4 describes the experimental setup and presents the results. Finally, we discuss related work (Sect. 5) and conclude with directions for future work (Sect. 6).

2 Fundamentals

2.1 Problem Statement and Task Scheduling

In the basic scheduling problem, a set of n tasks $T := \{t_1, \ldots, t_n\}$ has to be assigned to a set of m resources $P := \{p_1, \ldots, p_m\}$. Next to mapping a task t_i to a resource p_j, scheduling also includes the assignment of an ordering and time slices.

Scheduling problems are generally considered to be NP-hard [10]. As there is no algorithm that can solve all scheduling problems efficiently, there exist many different heuristics. These can be classified into static and dynamic algorithms. The main difference is that static algorithms make all decisions before a single task is executed, whereas dynamic algorithms schedule tasks at runtime. Hence, static algorithms have to know all relevant task information beforehand, while dynamic ones do not need full information and are able to adapt their behavior.

2.2 Optimality Criterion

The standard optimization criterion is the makespan, which is the time an application or a set of tasks spends in a system from start to finish. If several applications are scheduled simultaneously, only considering the makespan can lead to stalling one application in favor of the others. Therefore, it is sensible to also evaluate the algorithms regarding fairness.

A criterion that better reflects scheduling decisions for single tasks is the flow time F_i, which is defined as $F_i = C_i - r_i$, where C_i is the completion time and r_i the release time of a task t_i. Generally speaking, F_i is the time t_i spends within the system. So, the flow time is able to reflect how long a task is in the system before being executed and combines this with its execution time. As the two objectives efficiency and fairness are fundamentally at odds, Bansal et al. [2] suggest minimizing the l_p-norm of the flow time $\|F\|_{l_p}$ for small values of p. $\|F\|_{l_p}$ is defined as follows:

$$\|F\|_{l_p} = \left(\sum_i F_i^p \right)^{\frac{1}{p}}, \tag{1}$$

where p is a value chosen by the user.

3 Dynamic Scheduling Algorithms

This section presents the algorithms and the extensions to EMB^2. We selected these algorithms on the basis of their runtime overhead, scheduling decisions have to be made as fast as possible in dynamic systems, their implementation complexity, and their ability to work with limited knowledge about the set of tasks to be executed. These heuristics can be classified into immediate and batch mode. Immediate mode considers tasks in a fixed order, only moving on to the next task after making a scheduling decision. In contrast, batch mode considers tasks out-of-order and so delays task scheduling decisions as long as possible, thereby increasing the pool of potential tasks to choose from.

3.1 Immediate Mode Heuristics

Opportunistic Load Balancing (OLB). [8] estimates the completion time of the irrevocably scheduled tasks as a measure of load on a processing unit p_j. OLB then assigns a task t_i to the processing unit p_j that has the earliest completion time for its already assigned tasks.

Minimum Execution Time (MET). [7] maps a task t_i to the processing unit p_j that minimizes its execution time. The heuristic considers a task in isolation, not taking the actual load of the processing units in account when making a scheduling decision. Thus, this heuristic can easily lead to load imbalances if for all or most of the tasks a processing unit dominates.

Minimum Completion Time (MCT). [1] combines the execution time of a task t_i with the estimated completion time of the already assigned tasks of a processing unit p_j. In total, MCT predicts the completion time of a task t_i and assigns t_i to the processing unit p_j that minimizes the completion time of t_i.

3.2 Batch Mode Heuristics

Min-Min. [11] extends the idea of MCT by considering the complete set of currently ready-to-execute tasks. The heuristic then assigns the task t_i that has the earliest completion time to the processing unit p_j that minimizes the completion time of t_i. In general, the core idea is to schedule shorter tasks first to encumber the system for as short a time as possible. This can lead to starvation of larger tasks if steadily new shorter tasks arrive in the system.

Max-Min. [14] is a variant of Min-Min that is based on the observation that Min-Min often leads to large tasks getting postponed to the end of an execution cycle, needlessly increasing the total makespan because the remaining tasks are too coarse-granular to partition equally. So, Max-Min schedules the tasks with the latest minimum completion time first, leaving small tasks to pad out any load imbalance in the end. However, this can lead to starvation of small tasks if steadily new longer tasks arrive.

Sufferage. [14] ranks all ready-to-execute tasks according to their urgency based on how much time the task stands to lose if it does not get mapped to its preferred resource. The ranking is given by the difference between the task's minimum completion time and the minimum completion time the task would achieve if the fastest processing unit for this task would not be available. Tasks that do not have a clear preference for a processing unit are prone to starvation.

3.3 Implementation

We integrated the algorithms into EMB^2, a C/C++ library and runtime system for parallel programming of embedded systems.[1] EMB^2 builds on MTAPI [9], a task model that allows several implementation variants for a user-defined task. A developer defines a specific functionality, e.g., a matrix multiplication, and is then allowed to provide implementations for this task. MTAPI allows a developer to start tasks and to synchronize on their completion, where the actual execution is controlled by the runtime system. Thereby, the user has to guarantee that only tasks that have their dependencies fulfilled are started. Tasks are executed concurrently to other tasks that have been started and it is allowed to start new tasks within a task. The scheduling implementation of the current EMB^2 version distributes the task instances between heterogeneous processing units based on the number of already scheduled instances of the same task. For homogeneous

[1] https://embb.io/.

multicore CPUs, an additional work stealing scheduler [3,15] is used. As of yet, necessary data transfers for the accelerators are not considered separately. EMB^2 is designed and implemented in a modular fashion that allows developers to add further scheduling policies. However, a few extensions were necessary.

We added a general abstraction for processing units and grouped identical units in classes to allow a uniform treatment. Every unit is implemented using an OS-level worker thread. Workers corresponding to CPU cores are pinned to their respective cores but are assigned a lower priority than device workers.

Scheduling algorithms need task execution times to make sophisticated decisions. These can either be given by the user, an analysis step or predicted at runtime. In this work, we focus on dynamic systems which means static analyses are not possible. Therefore, we extended EMB^2 by a monitoring component that measures task execution times and stores them within a history data base with the problem size as key similar to the mechanism used in [13]. As data transfers are not yet considered explicitly in EMB^2, the execution times on accelerators include necessary data transfers. The stored data is then used to predict execution times of upcoming tasks to improve scheduling decisions. If there is already data stored for a particular task's implementation version and problem size, the data can be used directly. If there is data for a task's implementation version but with different problem sizes, interpolation is used to predict the execution time. If there is no data available at all, the runtime system executes a profiling run of this implementation version.

4 Experiments

To evaluate the scheduling heuristics, we considered a video-processing application using EMB^2's dataflow component, three benchmarks of the Rodinia Benchmark Suite [5], RabbitCT [19], and a benchmark with independent heterogeneous jobs. We chose them as they provide different characteristics, have sufficient problem sizes and thereby running time and possess an easily to parallelize kernel. We included benchmarks where the CPU outperforms the GPU, a benchmark, where the GPU strongly outperforms the CPU, and a benchmark where the difference between the GPU and CPU implementation is not as big. The independent heterogeneous jobs benchmark resembles dynamic systems as the task instances are started sporadically thereby adding a random component to the starting point of a task instance.

All experiments were executed ten times. For the single application benchmarks, we focus on the makespan because a user expects this to be optimized for a single application. We additionally evaluate the average flow time and the l_p-norm (Sect. 2.2) for $p = 3$ for the independent heterogeneous job benchmark. The following figures contain the average, the minimum and the maximum makespan of 10 evaluation runs as errorbars. We omitted the errorbars in the figure for the independent heterogeneous job benchmark to make it more readable.

4.1 Experimental Setup

The experiments were performed on a server with two Intel Xeon E5-2650 v4 CPUs a 12 cores each, an NVIDIA Tesla K80, and 128 GB a 2400 MHz DDR4 SDRAM DIMM (PC4-19200). The software environment includes Ubuntu 16.04.5, the Linux 4.4.0-138-generic kernel, glibc 2.23, and the nvidia-387 driver. EMB^2 was compiled with the GCC 5.4.0 compiler at optimization level-O3. The scheduling algorithms presented in Sect. 3 operate in the so-called pull mode in our experiments. In pull mode, the scheduler gets triggered iff at least one processing unit is idle. We chose this mode because it allows the scheduler to collect a set of tasks, which is needed to benefit from the batch mode heuristics.

4.2 Heterogeneous Video-Processing Application

The dataflow component of EMB^2 takes an arbitrary task graph describing the computation of a single data item, and parallelizes the computations over contiguous chunks of a data stream. They get submitted by a window sliding scheduler to the actual scheduler through reduction to fork-join parallelism while maintaining sequential execution of tasks. So, only tasks that are ready to execute are submitted to the actual scheduler. The application consists of a video-processing pipeline, performing the following steps:

1. Read and decode the next frame from an H.264-encoded video file. The corresponding process in the dataflow network is serial.
2. Convert the frame from the codec-native color space to RGB. This process is again serial because the conversion accesses a shared `libswscale` context. `libswscale` is a library that performs highly optimized image scaling and colorspace and pixel format conversion operations.
3. Apply the image transformation in two steps:
   (a) Perform a 3×3 box blur.
   (b) Cartoonify by performing a Sobel operator with a threshold selecting black pixels for edge regions and discretized RGB values for the interior. The Sobel operator consists of two convolutions with different 3×3 kernels followed by the computation of an Euclidean norm.
4. Convert the frame back from RGB to the codec-native color space.

The two image transformation operations have a CPU and GPU implementation. The cartoonify kernel has an average execution time of 165.97 ms on the CPU and 3.1 ms on the GPU for the *kodim23.png* test image by the Eastman Kodak Company. The box blur operation runs on average for 72.8 ms on the CPU and for 3.4 ms on the GPU. As input, we used a 30 s long test video encoded in 854:480 resolution with 30 fps at a bitrate of 2108 kb/s. The results are shown in Fig. 1. The best results are achieved by MCT, Min-Min, Max-Min, and Sufferage with MCT having the best results with an average of 10.3 s. OLB obtains a significantly worse result than the other algorithms with an average of 29.63 s because OLB does not consider task execution times, but rather just takes the next free processing unit, which in our implementation always starts with the CPU cores, and thereby only uses the, in this case slower, CPU.

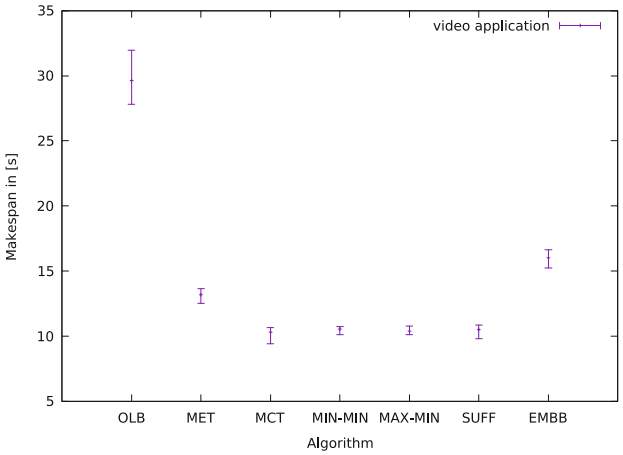

Fig. 1. Makespans for 10 runs of the video application benchmark

4.3 Rodinia Benchmark Suite

Hotspot3D iteratively computes the heat distribution of a 3d chip represented by a grid. In every iteration, a new temperature value depending on the last value, the surrounding values, and a power value is computed for each element. We chose this computation as kernel function for a parallelization with EMB^2 and parallelized it over the z-axis. The CPU implementation then further splits its task into smaller CPU specific subtasks. This is done manually and statically by the programmer to use the underlying parallelism of the multicore CPU and still have a single original CPU task that handles the same workload as the GPU task. For the evaluation, we used a $512 \times 512 \times 8$ grid with the start values for temperature and power included in the benchmark, and 1000 iterations. The average runtime on the CPU is 5.03 ms and 7.36 ms on the GPU.

Figure 2 shows the results of the Hotspot3D benchmark. Min-Min, OLB, MCT, Max-Min, and Sufferage all have an average of around 17 s with Min-Min having the lowest average of 16.94 ms by a very small margin compared to the group's highest average of 17.53 s by Max-Min. In this case, OLB benefits from the fact that it first distributes the load to the CPU. MET obtained the worst result because it does not consider the load of the processing units and just schedules all tasks to the fastest processing unit and so to the same CPU core.

Particlefilter is the implementation of a particle filter, a statistical estimator of the locations of target objects given noisy measurements, included in Rodinia. Profiling showed that *findIndex()* is the best candidate for a parallelization. *findIndex()* computes the first index in the cumulative distribution function array with a value greater than or equal to a given value. As *findIndex()* is called for every particle, we parallelized the computation by dividing the particles into work groups. The CPU implementation again further divides those groups into

subtasks. We used the standard parameters 128 for both matrix dimensions, 100 for the number of frames, and 50000 for the number of particles for the evaluation. The average task runtime on the CPU is 17.8 ms and 6.5 ms on the GPU. The results of the Particlefilter benchmark can be seen in Fig. 2. Here, the EMB^2 upstream algorithm got the best result with an average of 15.93 s where all other algorithms except OLB have an average of around 18 s. These results indicate that a distribution of tasks between the CPU and the GPU leads to the best result.

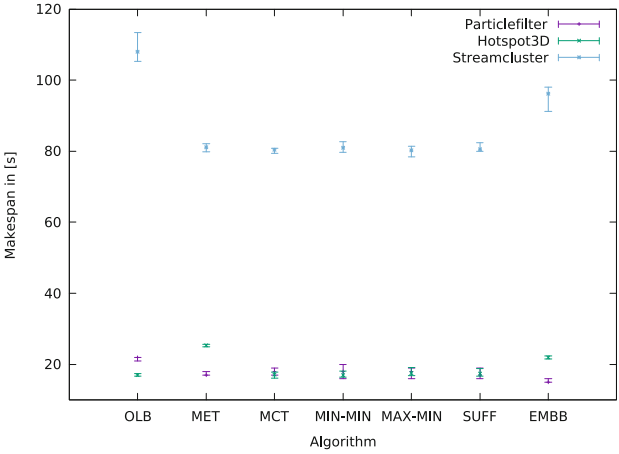

Fig. 2. Makespans for 10 runs of the Rodinia benchmarks

Streamcluster is taken from the PARSEC benchmark suite and solves the online clustering problem. For a stream of input data points, the algorithm finds a user given number of clusters. The main kernel of the algorithm *pgain()* computes if opening a new cluster reduces the total cost. In every iteration *pgain()* is called for each data point, so we parallelized the function by dividing the points into work groups. Again, the CPU implementation then further divides the work group into smaller chunks. We do not provide execution times as Streamcluster iteratively reduces the number of points considered, thereby varying in execution time. The results for the Streamcluster benchmark, see Fig. 2, show that all algorithms except OLB and the EMB^2 upstream version achieved an average makespan of around 80 s with Max-Min getting the best average by a small margin with 80.28 s compared to the second best average of 80.39 s by MCT and the group's worst average of 81.07 s by MET.

4.4 RabbitCT

RabbitCT is a 3D cone beam reconstruction benchmark framework that focuses on the backprojection step. It was created to fairly compare different

backprojection algorithms. In backprojection, each voxel is projected onto the projection data, then the data is interpolated and finally, the voxel value is updated. As this means that in every iteration the algorithm iterates over a 3D array, we parallelized the algorithm with EMB^2 by partitioning the volume by the z-axis. The CPU implementation then further partitions these chunks. We measured an average task runtime of 45.9 ms for the CPU and 97.7 ms for the GPU. RabbitCT provides an input data set which we used with a problem size of 512.

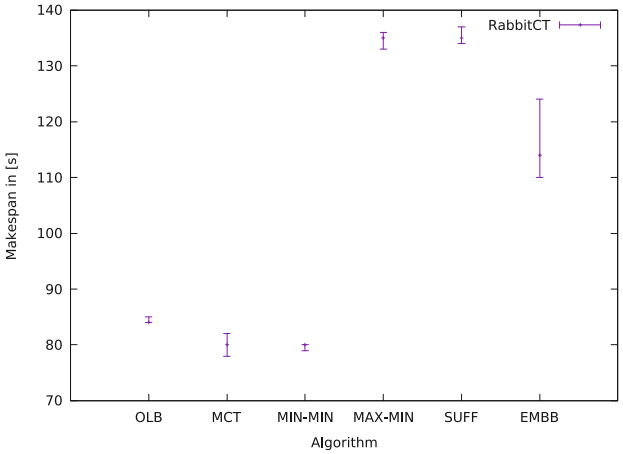

Fig. 3. Makespans for 10 runs of the RabbitCT benchmark

Figure 3 contains the results for the RabbitCT benchmark. We excluded MET as it was significantly worse then the other algorithms with an average of 400.17 s, thereby hiding details in the figure. MCT and Min-Min achieved the best results with MCT achieving an average makespan of 80.56 s and Min-Min achieving a slightly better average makespan of 80 s.

4.5 Independent Heterogeneous Jobs

Additionally, we evaluated the algorithms in a scenario with independent heterogeneous jobs. We chose three video-processing tasks that have both an OpenCL and a CPU implementation:

- J_1 **(Mean):** A 3×3 box blur.
- J_2 **(Cartoonify):** The cartoonify operation introduced in Sect. 4.2.
- J_3 **(Black-and-White):** A simple filter which replaces (R,G,B) values with their greyscale version $(\frac{R+G+B}{3}, \frac{R+G+B}{3}, \frac{R+G+B}{3})$.

All operations were applied to the *kodim23.png* test image. The three operations execute for 72.8 ms, 165.97 ms, and 11.4 ms on the CPU and 3.4 ms, 3.1 ms,

and 3.1 ms on the GPU. We used a sporadic profile to create task instances of these three jobs. New task instances were released with a minimum interarrival time of $\frac{1}{k}$ *secs*, where k is the parameter to control the load, plus a random delay drawn from an exponential distribution with parameter $\lambda = k$. By varying k, we can generate a range of different loads. The evaluation workload consists of 3000 tasks corresponding in equal proportions to instances of all three jobs. We conducted the experiment from $k = 500$ to 2000 with increments of 500. For this experiment, we measured the average makespan, the average flowtime and the l_3-norm. The EMB^2 upstream algorithm was excluded from the flowtime and l_3-norm measurements. In contrast to the other algorithms, which only schedule a new task iff at least one processing unit is idle, the EMB^2 upstream version always schedules a task as soon as it arrives in the system. Thereby, the time a task spends in the system is not really comparable to the other algorithms. The makespan results are shown in Fig. 4.

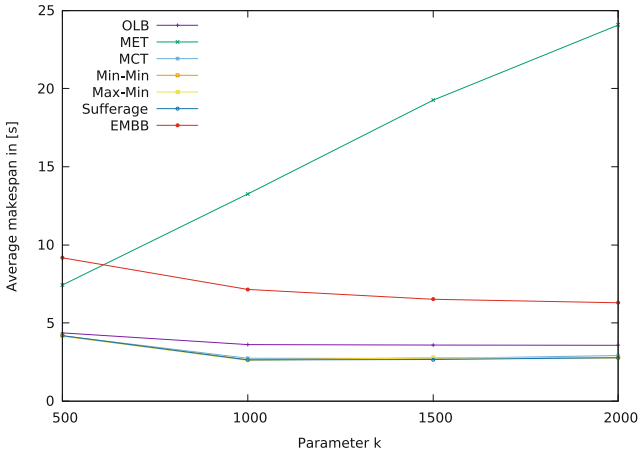

Fig. 4. Average makespan for 10 runs of the independent jobs benchmark

Here, Max-Min, Min-Min, MCT, and Sufferage nearly got the same results with Max-Min achieving the best results. Clearly, the worst results were obtained by MET. The figure of the average flowtimes (see Fig. 5) also show the best results for Max-Min, Min-Min, MCT, and Sufferage. However, for greater values of k there is a distinction between Max-Min and Sufferage, and Min-Min and MCT with the later two obtaining a worse average flowtime. Figure 6 shows the results for the l_3-norm. We excluded MET from the figure as its results were by far worse and so important details would get lost. Again, Sufferage and Max-Min got the best results. but this time for larger values of k Sufferage achieved better results.

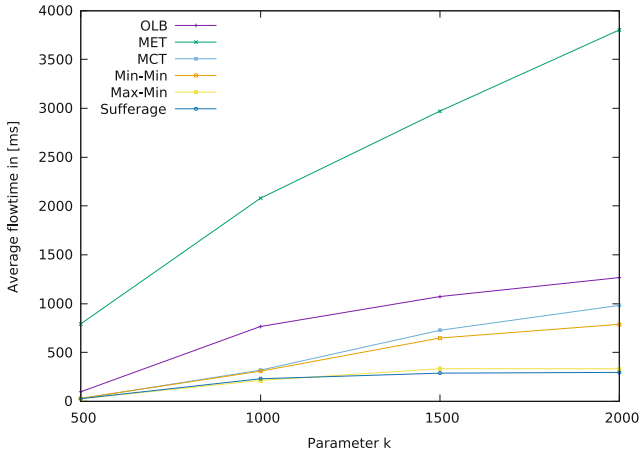

Fig. 5. Average flowtime for 10 runs of the independent jobs benchmark

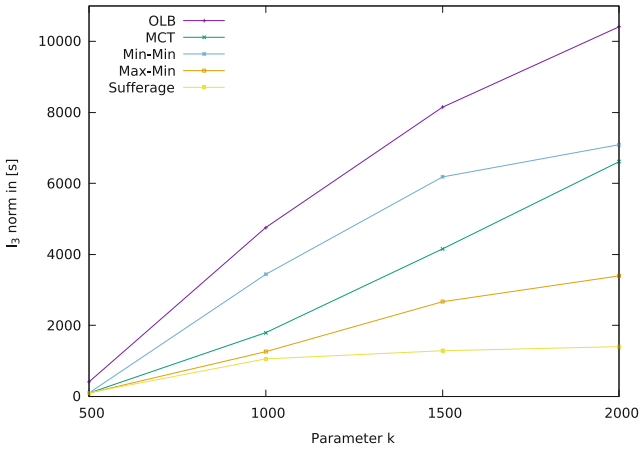

Fig. 6. Average l_3-norm for 10 runs of the independent jobs benchmark

5 Related Work

Task scheduling is a well-known research field which has lead to many heuristics for dynamic task scheduling. These can generally be classified into list scheduling heuristics [14,20], clustering heuristics [16], immediate mode heuristics [1,8,15], duplication scheduling heuristics [12] and guided-random-search-based algorithms including genetic algorithms [17,18], and swarm intelligence algorithms [6]. List scheduling heuristics sort all ready-to-execute tasks according to a priority criterion and then map the tasks to processing units in that order. In contrast, immediate mode heuristics assign a task to a processing unit as soon as it arrives. Clustering heuristics assume that communication costs are

a main factor of the total makespan. They try to minimize communication by clustering tasks and executing a cluster on a single processing unit. The goal of duplication scheduling is to reduce communication costs by executing key tasks on more than one processor, thereby avoiding data transfers. Contrary to the heuristic-based algorithms, guided-random-search-based algorithms try to efficiently traverse the search space by sampling a large number of candidates while also allowing temporary degradation of the solution quality. These algorithms are often only evaluated in simulations making it hard to judge their real world applicability. There also exist extensive studies that evaluate and compare different scheduling algorithms. Kim et al. [14] evaluate dynamic scheduling heuristics with independent tasks and task priorities. Braun et al. [4] compare eleven static scheduling heuristics that could also be used as batch-mode heuristics in a dynamic system. However, the heuristics are again evaluated in simulations only.

6 Conclusion and Future Work

In this work, we evaluated six heuristics. We integrated immediate and batch mode heuristics to see if it is possible to leverage sophisticated scheduling decisions in real-world scenarios. To evaluate the algorithms on a real system, we integrated them into EMB^2. The added heuristics and the EMB^2 upstream version were evaluated with six different benchmarks. In particular, we used a video-processing application, Particlefilter, Streamcluster and Hotspot3D of Rodinia, RabbitCT, and a benchmark consisting of three image filter jobs. As evaluation metric, we used the makespan for the application benchmarks. Additionally, we used the average flowtime and the l_3-norm for the independent jobs to measure fairness.

In five of six makespan-focused benchmarks, MCT and Min-Min achieved the lowest makespan or are within a 5% margin of the best makespan. The exception is Particlefilter where the best result is obtained by the EMB^2 upstream algorithm with a speed up of 11.6% to Sufferage. MCT and Min-Min still lie within a 17.9% and a 13.9% margin or a total difference of around 2.5 s. Max-Min and Sufferage also achieve the best or close to the best results in five out of six benchmarks but have a bigger outlier with the RabbitCT benchmark. Here, Max-Min and Sufferage have an makespan increase of around 70% or around 55 s. MET, OLB and the EMB^2 upstream algorithm constantly have worse results than the aforementioned ones. Considering the flowtime and the l_3-norm, Sufferage achieves the best results for the larger k values and is close to the best result for the smaller values. MCT and Min-Min both have increasingly worse results with larger values of k for both the average flowtime and the l_3-norm. In the worst case, the result increases by over 500%. So, in summary iff the focus only lies on the makespan, MCT or Min-Min seem to be the best choice with MCT being the significantly simpler algorithm. If fairness is an additional consideration, Sufferage seems to be the best choice. As future work, we want to consider task priorities, thus enabling soft real-time. The aforementioned starvation issues can also be improved by adding task priorities.

References

1. Armstrong, R., Hensgen, D., Kidd, T.: The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions. In: Proceedings of 1998 Seventh Heterogeneous Computing Workshop, (HCW 98), pp. 79–87, March 1998. https://doi.org/10.1109/HCW.1998.666547
2. Bansal, N., Pruhs, K.: Server scheduling in the Lp norm: a rising tide lifts all boat. In: Proceedings of the Thirty-fifth Annual ACM Symposium on Theory of Computing, STOC 2003, pp. 242–250. ACM, New York (2003). https://doi.org/10.1145/780542.780580
3. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM **46**(5), 720–748 (1999)
4. Braun, T.D., et al.: A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. J. Parallel Distrib. Comput. **61**(6), 810–837 (2001). https://doi.org/10.1006/jpdc.2000.1714
5. Che, S., et al.: Rodinia: a benchmark suite for heterogeneous computing. In: Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC), IISWC 2009, pp. 44–54. IEEE Computer Society, Washington DC, (2009). https://doi.org/10.1109/IISWC.2009.5306797
6. Elhady, G.F., Tawfeek, M.A.: A comparative study into swarm intelligence algorithms for dynamic tasks scheduling in cloud computing. In: 2015 IEEE Seventh International Conference on Intelligent Computing and Information Systems (ICICIS), pp. 362–369, December 2015. https://doi.org/10.1109/IntelCIS.2015.7397246
7. Freund, R.F., et al.: Scheduling resources in multi-user, heterogeneous, computing environments with SmartNet. In: Proceedings of 1998 Seventh Heterogeneous Computing Workshop, HCW 1998, pp. 184–199, March 1998. https://doi.org/10.1109/HCW.1998.666558
8. Freund, R.F., Siegel, H.J.: Guest editor's introduction: heterogeneous processing. Computer **26**(6), 13–17 (1993). http://dl.acm.org/citation.cfm?id=618981.619916
9. Gleim, U., Levy, M.: MTAPI: parallel programming for embedded multicore systems (2013). http://multicore-association.org/pdf/MTAPI_Overview_2013.pdf
10. Graham, R., Lawler, E., Lenstra, J., Kan, A.: Optimization and approximation in deterministic sequencing and scheduling: a survey. In: Hammer, P., Johnson, E., Korte, B. (eds.) Discrete Optimization II, Annals of Discrete Mathematics, vol. 5, pp. 287–326. Elsevier, Amsterdam (1979)
11. Ibarra, O.H., Kim, C.E.: Heuristic algorithms for scheduling independent tasks on nonidentical processors. J. ACM **24**(2), 280–289 (1977). https://doi.org/10.1145/322003.322011
12. Josphin, A.M., Amalarathinam, D.I.G.: DyDupSA - dynamic task duplication based scheduling algorithm for multiprocessor system. In: 2017 World Congress on Computing and Communication Technologies (WCCCT), pp. 271–276, February 2017. https://doi.org/10.1109/WCCCT.2016.72
13. Kicherer, M., Buchty, R., Karl, W.: Cost-aware function migration in heterogeneous systems. In: Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers, HiPEAC 2011, pp. 137–145. ACM, New York (2011). https://doi.org/10.1145/1944862.1944883

14. Kim, J.K., Shivle, S., Siegel, H.J., Maciejewski, A.A., Braun, T.D., Schneider, M., Tideman, S., Chitta, R., Dilmaghani, R.B., Joshi, R., Kaul, A., Sharma, A., Sripada, S., Vangari, P., Yellampalli, S.S.: Dynamically mapping tasks with priorities and multiple deadlines in a heterogeneous environment. J. Parallel Distrib. Comput. **67**(2), 154–169 (2007). https://doi.org/10.1016/j.jpdc.2006.06.005. http://www.sciencedirect.com/science/article/pii/S0743731506001444
15. Mattheis, S., Schuele, T., Raabe, A., Henties, T., Gleim, U.: Work stealing strategies for parallel stream processing in soft real-time systems. In: Herkersdorf, A., Römer, K., Brinkschulte, U. (eds.) ARCS 2012. LNCS, vol. 7179, pp. 172–183. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28293-5_15
16. Mishra, P.K., Mishra, A., Mishra, K.S., Tripathi, A.K.: Benchmarking the clustering algorithms for multiprocessor environments using dynamic priority of modules. Appl. Math. Model. **36**(12), 6243–6263 (2012). https://doi.org/10.1016/j.apm.2012.02.011. http://www.sciencedirect.com/science/article/pii/S0307904X12000935
17. Nayak, S.K., Padhy, S.K., Panigrahi, S.P.: A novel algorithm for dynamic task scheduling. Future Gener. Comput. Syst. **28**(5), 709–717 (2012). https://doi.org/10.1016/j.future.2011.12.001
18. Page, A.J., Naughton, T.J.: Dynamic task scheduling using genetic algorithms for heterogeneous distributed computing. In: 19th IEEE International Parallel and Distributed Processing Symposium, pp. 189a–189a, April 2005. https://doi.org/10.1109/IPDPS.2005.184
19. Rohkohl, C., Keck, B., Hofmann, H., Hornegger, J.: RabbitCT— an open platform for benchmarking 3D cone-beam reconstruction algorithms. Med. Phys. **36**(9), 3940–3944 (2009). https://doi.org/10.1118/1.3180956. http://www5.informatik.uni-erlangen.de/Forschung/Publikationen/2009/Rohkohl09-TNR.pdf
20. Topcuouglu, H., Hariri, S., Wu, M.Y.: Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Trans. Parallel Distrib. Syst. **13**(3), 260–274 (2002). https://doi.org/10.1109/71.993206