Evolving Scheduling Strategies for Multi-Processor Real-Time Systems

Frank Feinbube, Max Plauth, Christian Kieschnick and Andreas Polze
Operating Systems and Middleware
Hasso Plattner Institute
University of Potsdam, Germany
Email: {frank.feinbube, max.plauth, christian.kieschnick, andreas.polze}@hpi.de

***Abstract*—In recent years the multi-core era started to affect embedded systems, changing some of the rules: While on a single processor, Earliest Deadline First has been proven to be the best algorithm to guarantee the correct execution of priorized tasks, Dhall et al. have shown that this approach is not feasible for multi-processor systems anymore. A variety of new scheduling algorithms has been introduced, competing to be the answer to the challenges multi-processor real-time scheduling is imposing. In this paper, we study the solution space of prioritization-based task scheduling algorithms using genetic programming and state-of-the-art accelerator technologies. We demonstrate that this approach is indeed feasible to generate a wide variety of capable scheduling algorithms with pre-selected characteristics, the best of which outperform many existing approaches. For a static predefined set of tasks, overfitting even allows us to produce optimal algorithms.**

I. Introduction

Following the trends in the personal computing sector, many embedded systems are nowadays equipped with multiple processing units. These resources are used for non-critical tasks like entertainment systems and critical ones, where wrong timing is considered a failure. In real-time systems, the latter are traditionally studied using the preemptive task model. A task T arrives at time A in the system and is supposed to finish its execution by its deadline AD. Furthermore, with these critical tasks, it is usually assumed, that the worst case execution time C is known upfront. Tasks can either be occurring only once or periodically, where AD is also considered to be the time interval, after which the task arrives again. If a given set of tasks includes only periodic tasks, it is called a *periodic* task set; otherwise it is called *sporadic*.

A task scheduling algorithm is used to schedule these tasks onto p processors so that no task misses its deadline. This is usually realized by assigning priorities to the tasks. If the task set is known upfront, *static* scheduling algorithms can be used, assigning fixed priorities to the tasks. This is very efficient since the scheduling algorithm only needs to be executed once. If the task set is not known upfront and new tasks arrive during system runtime, *dynamic* scheduling algorithms need to be utilized. They reevaluate the priorities of all known tasks and are usually executed when new tasks arrive or at predefined time intervals during runtime.

Born at a time when resources for embedded systems were very restricted, traditional scheduling algorithms are rather simplistic, usually assigning priorities based on a single attribute: the deadline. As discussed in Section II, more sophisticated algorithms are required in multi-core scenarios. Ideally, an algorithm should be optimal, which means that it is capable of finding a feasible schedule whenever there exists one. While it has been proven that an optimal algorithm for multi-core scenarios cannot exist, a number of algorithms have been proposed that can schedule certain classes of task sets. In Section III we describe our approach to the problem. By applying genetic programming and state-of-the-art accelerator technologies, we were able to evaluate a vast variety of prioritization-based scheduling algorithms. As shown in Section IV our implementation can be used to find close-to-optimal algorithms tailored to task sets with specific characteristics.

II. Related Work

For *single processor* scenarios, optimal algorithms have been around for a long time [1]: Rate Monotonic Scheduling (RMS) [2] is an optimal static scheduling algorithm for periodic task sets. RMS prioritizes inverse proportionally to period lengths. Earliest Deadline First (EDF) [2] is an optimal dynamic scheduling algorithm for sporadic task sets. Each time a new task arrives, EDF prioritizes based on the deadlines of all tasks. Least Laxity First (LLF) [3] is also an optimal dynamic scheduling algorithm. The priority of each task is based on the difference of its remaining execution time and the time until its deadline is violated. Since this difference constantly changes during runtime, LLF shows strong *oscillation effects* as shown in Figure 1 leading to a huge amount of task switches. In practice, task switching in embedded systems comes with a performance overhead. Thus, there are variations of LLF such as Modified Least Laxity First (MLLF) [4] that try to reduce the oscillation effect.

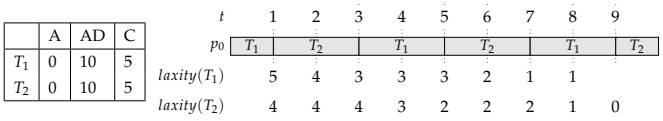

Fig. 1. Scheduling algorithms like Least Laxity First [3] show oscillating behavior where the priority is altered at each quantum.

In *multi-processor* scenarios, things get a little bit more complicated: Besides oscillation effects, task schedulers also have to cope with *Dhall's effect* and *pure global task sets*.

Dhall's effect is demonstrated in Figure 2. It describes the scenario where there are task sets which produce a very

low overall system utilization, but still miss a deadline when scheduled with traditional algorithms. A number of "hot fixes" to EDF and RMS were introduced that have been proven to circumvent the problem: e.g. EDF First Fit/Best Fit [5], Earliest Deadline Until Zero Laxity (EDZL) [6], and UMax algorithms [7], [8]. Although Dhall's effect is prevented, these scheduling algorithms only allow for low system utilizations: e.g. 35.425% for sporadic and 37.482% for periodic task sets [7], [8]. Since this is significantly lower than the 50% utilization, that is considered the actual limit [9], new approaches were evaluated. Lundberg has proven that by assigning task priorities based on the slack ($AD - C$) instead of the deadline, the acceptable utilization for sporadic task sets can be increased to 38.197% [10].

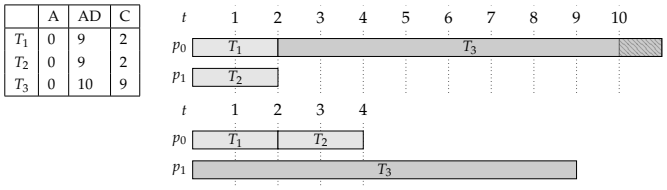

Fig. 2. This two-processor scenario with three tasks demonstrates Dhall's effect [2]. Although it is possible to schedule all tasks according to their deadline (bottom schedule), Earliest Deadline First (EDF) fails to do so (schedule on top).

A popular approach to multi-processor real-time scheduling is to statically allocate tasks to processors so that a task will never be migrated to another one. The alternative to this *partitioned* approach, is the *global* approach where each processor can execute each task and tasks will be migrated accordingly. Migrating tasks results in additional overhead, but it is the only way to handle *pure global task sets* as depicted in Figure 3.

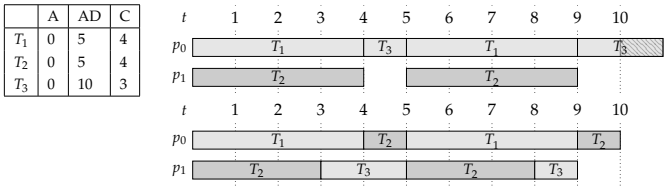

Fig. 3. This two-processor scenario with three tasks demonstrates Levin's pure global task sets [11]. Although it is possible to schedule all tasks according to their deadline (bottom schedule), it is impossible to do so by pinning tasks to a single processor (schedule on top).

There are algorithms that avoid *Dhall's effect* and are capable of scheduling pure global task sets while allowing utilizations of up to almost 100%. Proportionate Fair Scheduling [11], [12] and Dp-fair use a fluid scheduling model with fair task progress, which requires a reprioritization of all tasks at predefined time intervals. Largest Local Remaining Execution First (LLREF) [13] follows a similar model, but reprioritizes based on the laxity and execution time of the active tasks, instead of static time intervals. When it comes to theoretical maximal system utilization, these scheduling algorithms perform exceptionally well. However, depending on the frequency of the reprioritizations, they show oscillation effects and introduce significant scheduling overhead. This overhead is comprised of the execution time of the more complex scheduling algorithm itself, the overhead for switching the active tasks and the overhead for task migration between processors. Another restriction is that the aforementioned reprioritizing algorithms are only suitable for periodic task sets. Hong et al. [14] formulated the hypothesis that there is no optimal priority-driven algorithm for sporadic task sets. This hypothesis has been proven by Fisher [15].

A. Research Gap

In this work, we contribute to the field of real-time multi-processor scheduling by presenting an approach to:

- Identify novel algorithms by exploring the solution space for real-time scheduling algorithms.
- Create algorithms complying with desired characteristics such as the number of task migrations and maximal system utilization.

As a means to implement these goals, we use genetic programming to evolve real-time scheduling algorithms with pre-selected characteristics. Using genetic programming for the creation of our algorithms allows us to cover a wide variety of scheduling alternatives, thereby helping us to identify the attributes and functions that are most successful to reduce overheads while allowing for a solid system utilization. While being able to create optimal algorithms for many of the task sets we used in our evaluation, we were unable to identify an algorithm that is optimal for the general case. However, these findings harmonize with the proof of Fisher [15], which states that no optimal algorithm can exist for the general case. Running such a compute-intense simulation to identify suitable algorithms was only possible due to the performance of modern processor and state-of-the-art accelerator technologies.

We are not the first to apply genetic algorithms to the research area of scheduling algorithms for multi-processor systems. Hou et al. [16] and Greenwood et al. [17] used genetic algorithms and evolutionary strategies to generate heuristics for predefined task graphs. While demonstrating the feasibility of the approach, both studies focussed exclusively on task sets that are known upfront and created heuristics that, while useful in for multi-processor systems in general, did not consider real-time requirements.

Furthermore, there are existing studies that simulate scheduling algorithms to evaluate their qualitative and quantitative characteristics [18]–[20]. These approaches are sophisticated to gain insight into capabilities of a single selected scheduling algorithm, while our approach allows sift through a vast amount of scheduling algorithms to identify the interesting candidates for further examination.

To the best knowledge of the authors, we are the first to apply genetic programming for an exploration of the real-time scheduling algorithm solution space for arbitrary task sets.

III. Approach

Mathematical modeling of the task scheduling domain and proving the qualities of particular scheduling algorithms becomes increasingly complicated the more complex the scheduling algorithms are. Thus, the next best thing would be a simulation of all possible scheduling algorithms starting with a very

limited set of terms and functions and iteratively considering more, when the current complexity is exhaustively studied. Such an approach has to handle humungous state explosions with every additional variable and function. Evolutionary processes and genetic algorithms have proven to be ideal for these kinds of scenarios, since they confine unpromising states while iteratively exploring the more promising ones. [21]–[23] This section discusses the application of genetic algorithms to identify promising scheduling algorithms.

A. Architecture

The general architecture of our approach is depicted in Figure 4. We start by loading the three kinds of task sets, that we use as the workload for our simulation. The task sets are described in detail in Section III-B. Furthermore, we generate a number of initial prioritization schemes. Prioritization schemes form the core of our scheduling algorithms. They encapsulate everything that is needed to assign priorities to task sets. The generic task scheduler shown in Figure 5 will use these schemes to prioritize the tasks and then simply schedule them based on their priorities.

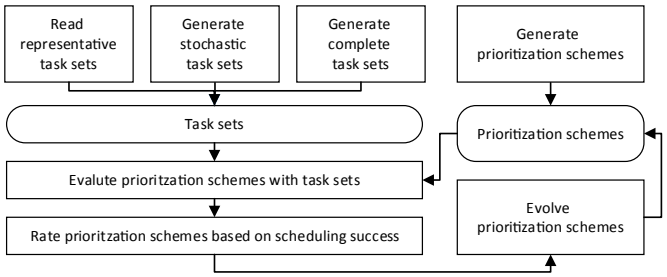

Fig. 4. Architecture: the evolutionary process iteratively refines the scheduling strategies using a variety of task sets.

```
1  for(runtime = 0;
2    runtime < simulationEnd && !missedDeadline(tasks);
3    ++runtime)
4  {
5    activeTasks = filterActive(tasks);
6
7    // this is exchanged with each prioritization scheme
8    prioritizationScheme->prioritizeTasks(activeTasks);
9
10   orderDescendantByPriority(activeTasks);
11   tasksToSchedule = selectFirst(activeTasks, processors);
12
13   simulateDiscreteStep(tasksToSchedule);
14 }
```

Fig. 5. The generic scheduler is the core of our implementation. In our implementation schedulers only differ in the way they assign priorities to tasks at any given point in time throughout the execution. This scheduling strategy is determined by the prioritization scheme.

The evolutionary process is conducted iteratively with the following consecutive steps: evaluation, rating, evolving. In the evaluation step, each prioritization scheme is used to schedule each of the task sets. It is monitored how many tasks switches and task migrations were required and how many of the task sets failed to be scheduled successfully, e.g. a deadline was missed. In the rating step, this information is used to assign a fitness value to each of the prioritization schemes. Based on the fitness value the well-known mechanisms of selection, mutation and crossover are applied to create the next generation of prioritization schemes. This process is repeated until candidates with fitness values that are sufficient to comply with our requirements have been found, e.g. prioritization schemes capable of scheduling all task sets successfully, or a predefined maximal runtime is exceeded.

By adapting the fitness rating accordingly, this architecture allows us to easily ensure that the scheduling algorithms comply with our requirements when balancing task migrations and maximal supported utilization.

B. Task Sets

The quality of the resulting prioritization schemes depends primarily on the task sets that are used for the fitness rating of the evolutionary process. We distinguish between three categories of task sets: *representative* task sets, *stochastic* task sets and *complete* task sets.

Representative task sets are a selection of tasks sets from the literature that is used to evaluate the capability of a prioritization scheme to handle the 'hard' cases. For single processor scenarios, we have task sets that can barely be scheduled by Rate Monotonic Scheduling (RMS), cases that RMS fails to schedule, but Earliest Deadline First (EDF) can schedule. In the multi-processor scenarios, we we extend these conventional task sets so that the workload increases according to the number of processors. Furthermore, we add task sets that show different effects discussed in Section II. Our set of representative task sets includes both periodic and sporadic task sets. Most of these task sets could be scheduled with a partitioning strategy, e.g. without task migration. Consequently, we added pure global task sets as described by Levin et al. [11] to complete our mix of representative task sets. An overview of aforementioned task sets and the ability of selected scheduling algorithms to find a feasible schedule is presented in Table I.

While representative task sets are well suited to remove prioritization schemes that fail to handle the problematic cases, *stochastic* task sets allow us to assess the overall scheduling performance by mitigating undesireable overfitting effects. To accomplish this, we generate a number of task sets with a pseudo-random generator based on a stochastic distribution.

Complete task sets are created by generating every possible combination of task distributions for a given number of processors and number of scheduling time slices (quanta). Since both representative task sets and stochastic task sets are included in complete task sets, they deliver the best quality for the evaluation. The drawback is, though, that the amount of task sets that have to be generated grows exponentially and renders computation unfeasible for all but very small amounts of processors and quanta. In our experiments, we studied complete tasks sets for up to 8 processors and quanta of up to 6 intervals, resulting in about 10^8 task sets.

C. Evolution

We represent each prioritization scheme as an abstract syntax tree (AST) that can be executed for a task to produce a priority. Figure 6 shows an example. The evolutionary process

TABLE I. CHARACTERISTICS OF OUR SET OF REPRESENTATIVE TASK SETS. THE RIGHT PART OF THE TABLE SHOWS WHICH PROCESSOR CONFIGURATIONS CANNOT BE SCHEDULED BY EXISTING SCHEDULING ALGORITHMS. 1, 2, 4, 8, 16 ARE THE NUMBERS OF PROCESSORS USED. CONFIGURATIONS MARKED WITH A * CAN ONLY PARTLY BE SCHEDULED. PLEASE NOTE THAT LEVIN'S PURE GLOBAL TASK SETS [11] CAN NEITHER BE SCHEDULED BY APPROACHES THAT APPLY A SIMPLE PARTITIONING, NOR BY APPROACHES THAT ARE SENSITIVE TO UTILIZATION.

	periodic	partitionable	Laxity-based	global EDF	EDF-US	EDZL
RMS3	✓	✓		2*		
RMS4	✓	✓		2* 4 8 16		4* 8* 16*
WikiEDF	✓	✓				
Partitioned	✓	✓	2* 4* 8* 16*	4* 8* 16*	2* 4* 8* 16*	4* 8* 16*
Dhall		✓		2 4 8 16	1*	
SlackDhall		✓	4* 8* 16*		1* 2* 4* 8* 16*	4* 8* 16*
Detail		✓		2		
Split		✓				
Interwoven		✓	2 4 8 16	2 4 8 16	1 2 4 8 16	2 4 8 16
Levin [11]	✓		2 4 8 16	2 4 8 16	2 4 8 16	2 4 8 16

of selection, mutation and crossover was realized according to the literature. [21]–[23] The initial population is generated purely randomly, with a restricted AST depth of up to 5.

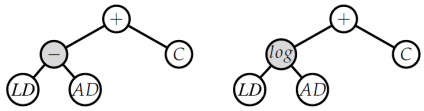

Fig. 6. Prioritization schemes are represented as abstract syntax trees. Mutations and crossovers are realized by varying and exchanging nodes. In this example $(LD - AD) + C$ mutates to become $log(LD, AD) + C$.

TABLE II. ATOMIC AND SIMPLE DERIVED TERMINALS, BASED ON THE CURRENT TASK AND THE SYSTEM.

x	random floating point values from -10.0 to 10.0
0, 1	constant values 0 and 1
m	number of processors
A	arrival time
RD	relative deadline (relative to arrival time)
C	capacity = worst case execution time
PT	amount of C that has already been executed
P	current task priority (starting with 0)
T	current point in time
AD	absolute deadline = A + RD
ST	slack = RD - C
L	remaining surplus time = (AD - T) - (C - PT)
U	utilization created by task = C / RD
LD	remaining execution time = C - PT
RU	remaining utilization = LD / (AD - T)

The nodes in the AST are the terminals listed in Table II. We distinguish between three types of basic terminals: numbers, system terminals, task specific terminals. System terminals comprise of the processor count and the time. Task specific terminals are deadline, worst case execution time and so forth. In addition to these, we provide a selection of derived terminals. These are not essential, since they would be generated by the evolutionary process anyway, but since they are the core of many of the popular scheduling algorithms like EDF [2] and LLF [3] we provided them, as well. Furthermore, the introduction of derived terminals improved the performance of the evolutionary process significantly. Please note that the resulting prioritization schemes do not consider the other tasks in the system, thereby guaranteeing a linear execution time of the represented scheduling algorithm.

The set of functions supported by our AST are: *addition, subtraction, multiplication, protected division, protected logarithm, exponentiation, check for equality, check for inequality, selecting the minimum, and selecting the maximum*. Checking for equality and inequality will produce either 1 for success or 0, allowing a combination with the other functions: AD * (L == 0).

The fitness of a prioritization scheme is rated according to multiple objectives [23]. A prioritization scheme is considered better than a similar one, if it can either schedule more task sets successfully or needs significantly less migrations for the scheduling. The impact of the objectives on the fitness functions can be configured by weights. For the selection process, we experimented with different population sizes. We observed that a tournament based selection process with 8 participants and a population size of 100 produced the best results.

In our experiments, we experienced overfitting effects [22], where the identified candidates were capable of scheduling all the task sets we trained them with. This is useful, if you want to use the approach, to find the perfect schedule for a specific task set. In the study of the solution space for scheduling algorithms, it is a hindrance, though, because overfitted prioritization schemes perform worse in the general case. To control overfitting, we created two distinct sets of task sets – the first to evolve the schemes and the second for the final evaluation. Furthermore, we applied randomizations and weighted function length negatively, since long functions tend to overfit more, than shorter ones.

D. Implementation and Performance Tuning

For the practical evaluation, we implemented the conceptual architecture presented in Figure 4. Fortunately, the repetitive steps of generation, evaluation and selection are suited for a parallel implementation. Our initial measurements indicated that the evaluation step is the predominant workload causing 99.99% of the overall execution time. As a consequence, all optimization efforts were directed at improving the efficiency of the evaluation step.

The time required for the evaluation process was greatly reduced using several optimization techniques: Using a stack-based representation of terms resulted in a decreased number of memory allocation operations compared to a tree-based data structure. At the same time, the stack-based structure

managed to increase the degree of data locality. Targeting the goal of data locality as well, an additional blocking method was applied to increased the amount of cache hits. Finally, we evaluated several strategies to vectorize our implementation. However, in contrast to the other optimizations, none of the vectorization strategies resulted in any significant performance improvements.

In addition to an x86_64 CPU-based implementation, we also created prototypes targeting Intel's Many Integrated Core (MIC) architecture exclusively as well as a hybrid version. The hybrid implementation applies an asymmetric load distribution scheme between the CPU and the MIC in order to maximize the execution speed.

The Xeon Phi accelerators based on the MIC architecture consist of 57-61 cores that are based on a modified P54C design. Unlike GPU compute devices, all cores of a MIC accelerator can act independently of each other. This property makes the MIC architecture a promising target for the parallel evaluation of diverse prioritization functions. Since the MIC architecture supports x86_64 instructions, the optimization we conducted improved the performance for both architectures.

IV. Evaluation

A. Qualitative evaluation

As described in Section III-C, we designed our implementation to assign fitness ratings based on weighted objectives. Figure 7 shows the impact of weighting migrations with 10%.

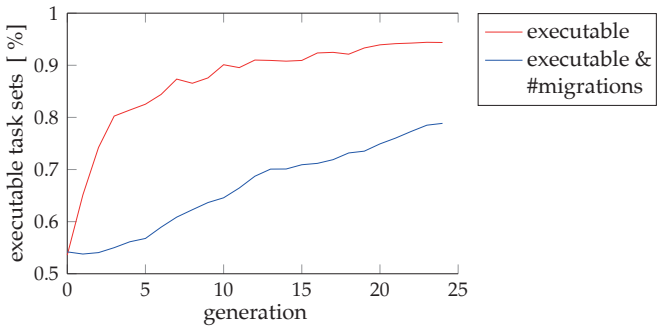

Fig. 7. Fitness ratings that are based on the number of executable task sets exclusively show a faster evolutionary progress, but introduce a considerable amount of task migrations.

A selection of the resulting prioritization functions is listed in Table III. In this example, *L/RU* was capable of scheduling all task sets, but required a substantial amount of task migrations. As another interesting candidate, *AD* reduced the number of migrations by a factor of 35.9, but failed with over 25% of the task sets. These examples show that even simple functions can handle the training task sets very successfully. Our second set of task sets proved to be greater challenge. We conducted elaborate simulation runs each with up to 200 generations. The most successful ones were capable of scheduling 83% of the task sets successfully. Some of them, such as *1 / L*, were capable of executing pure global task sets, but failed with others.

TABLE III. The quality of exemplary prioritization functions based on capability of scheduling task sets and the number of required task migrations.

function	# executable task sets		migrations / task set
L/RU	75	100 %	862
L	71	94.67 %	819
AD	56	74.67 %	24
$AD - 1.0$	56	74.67 %	24

Figure 8 and Figure 9 show which terminals and functions are most dominant. The terminals that are used by the state-of-the-art scheduling algorithms such as laxity *L*, remaining execution time *LD*, deadline *AD* are successful at surviving the selection process. Surprisingly, the processor count, that could be a mechanism to distinguish single-processor from multi-processor systems is only scarcely used for prioritization. The most prominent functions are basic arithmetic functions such as addition and multiplication as well as selecting the minimum and maximum. Functions allowing terminals to have strong influence on the results such as exponentiation and logarithm are only used rarely.

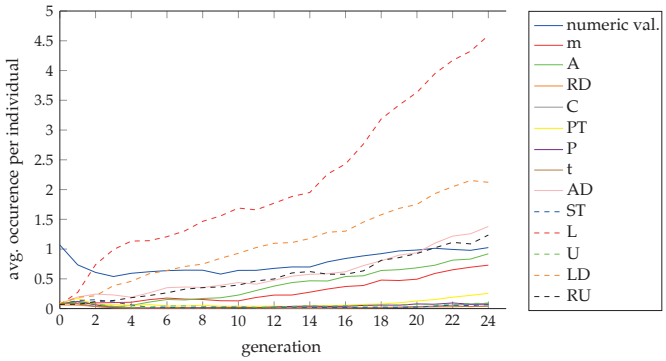

Fig. 8. Terminals with dynamic properties such as Laxity *L*, remaining execution time *LD* and remaining utilization *RU* were especially successful in the evolutionary process.

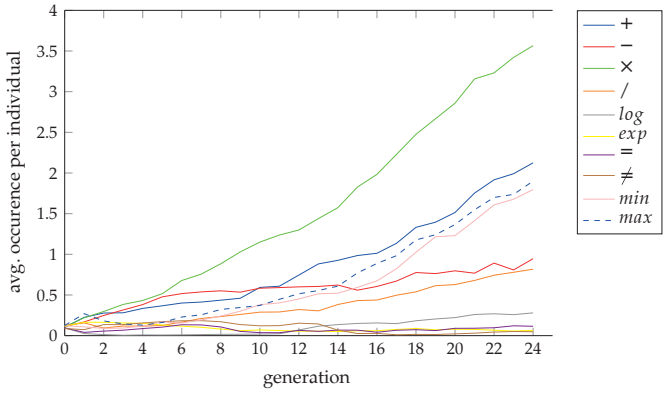

Fig. 9. In our evolutionary process arithmetic operations as well as minimum and maximum operations were predominant.

In the majority of our experiments, we found a vast amount of candidate prioritization schemes with interesting properties. However, a generic optimal solution was not found, concurring with the literature [11], [14], [15].

B. Performance evaluation

Our optimized implementation was able to retrieve valid prioritization functions for multiprocessor systems ranging from 1 up to 400 processors in a feasible amount of time. Benchmarks were performed in a test environment equipped with two Xeon E5620 processors, each containing 4 cores clocked at 2.40 GHz, and 24 GB of main memory. Furthermore, a Xeon Phi 5110P accelerator was employed, providing 8 GB of dedicated memory and 60 cores clocked at 1.053 GHz.

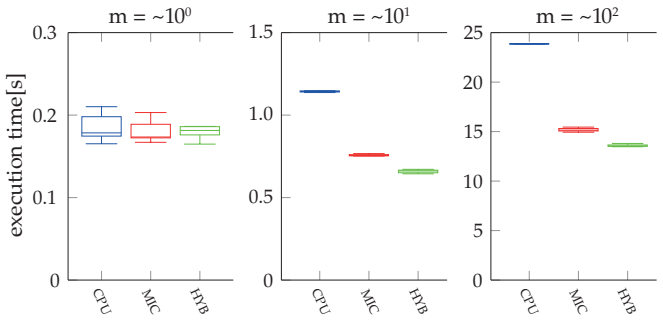

Fig. 10. Across all problem sizes for *m*, the *MIC* always outperforms the *CPU*. However, the hybrid approach *HYP* always provides an additional performance improvement on top of the *MIC* performance.

The measurements illustrated in Figure 10 demonstrate that even though evolutionary approaches require huge amounts of compute resources, modern CPUs empower us to accomplish the task in acceptable time. The first generation of MIC-based hardware accelerators allowed us to push the limit a little further by achieved speedup factors of 2 for $m = \sim 10^2$.

V. Conclusion

In this work we have studied the feasibility of genetic programming and the evolutionary process to explore the solution space of priority-based scheduling algorithms. We found that this approach is indeed helpful to identify the terminals and functions that are most dominant in promising prioritization schemes. Furthermore, we demonstrated that it is possible to weight desired characteristics like task migration and find optimal schedulers for static task sets by exploiting overfitting.

None of the scheduling algorithms that we generated, not even the most promising ones were capable to schedule all our task sets successfully. These findings harmonize with Fisher's proof [15] that no optimal priority-driven scheduling algorithm exists for arbitrary task sets.

Acknowledgement

This paper has received funding from the European Union's Horizon 2020 research and innovation programme 2014-2018 under grant agreement No. 644866.

Disclaimer

This paper reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains.

References

[1] Burns, A. and Wellings, A. *Real-Time Systems and Programming Languages*. Addison Wesley, 2001.

[2] Dhall, S. K. and Liu, C. L. *On a Real-Time Scheduling Problem*. Operations Research, 1978, Vol. 26, pp. 127-140.

[3] J. Y.-T. Leung, *A new algorithm for scheduling periodic, real-time tasks*. Algorithmica, vol. 4, no. 1-4, pp. 209219, 1989.

[4] S.-H. Oh and S.-M. Yang, *A modified least-laxity-first scheduling algorithm for realtime tasks*. Real-Time Computing Systems and Applications, 1998. Proceedings. Fifth International Conference on, pp. 3136, IEEE, 1998.

[5] J. M. López, M. García, J. L. Diaz, and D. F. Garcia, *Worst-case utilization bound for edf scheduling on real-time multiprocessor systems*. Real-Time Systems, 2000. Euromicro RTS 2000. 12th Euromicro Conference on, pp. 2533, IEEE, 2000.

[6] C. Seongje, L. Suk-Kyoon, and L. Kwei-Jay, *Efficient real-time scheduling algorithms for multiprocessor systems*. IEICE Transactions on Communications, vol. 85, no. 12, pp. 28592867, 2002.

[7] L. Lundberg, *Analyzing fixed-priority global multiprocessor scheduling*. Real-Time and Embedded Technology and Applications Symposium, 2002. Proceedings. Eighth IEEE, pp. 145153, IEEE, 2002.

[8] L. Lundberg and H. Lennerstad, *Guaranteeing response times for aperiodic tasks in global multiprocessor scheduling*. Real-Time Systems, vol. 35, no. 2, pp. 135151, 2007.

[9] A. Srinivasan and S. Baruah, *Deadline-based scheduling of periodic task systems on multiprocessors*. Information Processing Letters, vol. 84, no. 2, pp. 9398, 2002.

[10] L. Lundberg, *Slack-based multiprocessor scheduling of aperiodic real-time tasks*. Real-Time Systems, vol. 47, no. 6, pp. 618638, 2011.

[11] G. Levin, S. Funk, C. Sadowski, I. Pye, and S. Brandt, *Dp-fair: A simple model for understanding optimal multiprocessor scheduling*, Real-Time Systems (ECRTS), 2010 22nd Euromicro Conference on, pp. 313, IEEE, 2010.

[12] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, *Proportionate progress: A notion of fairness in resource allocation*. Algorithmica, vol. 15, no. 6, pp. 600625, 1996.

[13] H. Cho, B. Ravindran, and E. D. Jensen, *An optimal real-time scheduling algorithm for multiprocessors*. Real-Time Systems Symposium, 2006. RTSS06. 27th IEEE International, pp. 101110, IEEE, 2006.

[14] K. S. Hong and J. Y.-T. Leung. *On-Line Scheduling of RealTime Tasks*. IEEE Transactions on Computers, 41:1326-1331, 1992.

[15] N. W. Fisher, *The multiprocessor real-time scheduling of general task systems*. University of North Carolina at Chapel Hill, 2007.

[16] E. S. Hou, N. Ansari, and H. Ren, *A genetic algorithm for multiprocessor scheduling* Parallel and Distributed Systems, IEEE Transactions on, vol. 5, no. 2, pp. 113120, 1994.

[17] G. W. Greenwood, A. Gupta, and K. McSweeney, *Scheduling tasks in multiprocessor systems using evolutionary strategies* Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on, pp. 345349, IEEE, 1994.

[18] F. Singhoff, J. Legrand, L. Nana, and L. Marcé, *Cheddar: a flexible real time scheduling framework*, ACM SIGAda Ada Letters. vol. 24, pp. 1-8, ACM, 2004.

[19] F. Golatowski, J. Hildebrandt, J. Blumenthal, and D. Timmermann, *Framework for validation, test and analysis of real-time scheduling algorithms and scheduler implementations*, 13th IEEE International Workshop on Rapid Systems Prototyping, pp. 146-152. IEEE. 2002.

[20] G.A. Lloyd, *Comparing schedulability of global, partitioned and clustered multiprocessor platforms using empirical analysis*, 2010.

[21] T. Bäck and H.-P. Schwefel. *An overview of evolutionary algorithms for parameter optimization* Evolutionary computation, vol. 1, no. 1, pp. 123, 1993.

[22] J. R. Koza *Genetic programming as a means for programming computers by natural selection* Statistics and Computing, vol. 4, no. 2, pp. 87112, 1994.

[23] R. Poli, W. B. Langdon, and N. F. McPhee, *A field guide to genetic programming* Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk, 2008.