SimSo: A Simulation Tool to Evaluate Real-Time Multiprocessor Scheduling Algorithms

Maxime Chéramy, Pierre-Emmanuel Hladik, Anne-Marie Déplanche

SimSo: A Simulation Tool to Evaluate Real-Time Multiprocessor Scheduling Algorithms

Maxime Chéramy*, Pierre-Emmanuel Hladik* and Anne-Marie Déplanche†
*CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France
Univ de Toulouse, INSA, LAAS, F-31400 Toulouse, France
†IRCCyN UMR CNRS 6597, (Institut de Recherche en Communications et Cybernétique de Nantes), ECN,
1 rue de la Noe, BP92101, F-44321 Nantes cedex 3, France

***Abstract*—In this paper, we present SimSo, a simulator designed for the comparison and the understanding of real-time scheduling policies. This tool is designed to facilitate the implementation of schedulers in a realistic way. Currently, more than twenty-five scheduling algorithms are available in SimSo. A particular attention is paid to the control of the computation time of the jobs therefore introducing more flexibility, for instance by taking into account cache-related preemption delays. In addition, SimSo offers an easy way to generate the tasksets, to perform simulations and to collect data from the experiments.**

I. Introduction

Davis and Burns referenced more than thirty real-time multiprocessor scheduling algorithms in 2011 [11] and more than a dozen of new algorithms have emerged since then, e.g. [26], [22]. Such a large number of scheduling algorithms makes their evaluation and comparison difficult. The evaluation generally comes from theoretical analysis, simulation or an actual implementation, according to criteria that can include utilization bounds, success rates, number of preemptions, migrations, and/or algorithm complexity.

Our long-term objective is to compare the various schedulers with ease while taking into account the capacity of the hardware architecture (e.g. caches, dynamic frequency scaling, or system overheads) to have an effect on their performance. This effect is currently very difficult to evaluate using theoretical analyses such as schedulability tests or resource augmentation. On the other hand, while using a real system would seem to be a better approach, the effective implementation of a scheduler as an operating system component requires a substantial amount of time and the results are too specific to the system. As a consequence, we think that simulation could be a good compromise to efficiently evaluate scheduling algorithms.

This paper deals with SimSo, our tool to simulate multiprocessor real-time schedulers and that aims at facilitating the design of experimental evaluations. In a prior publication, some design choices regarding the simulation kernel have been presented [7]. More recently, we showed using SimSo how the use of the WCET could bias the evaluation of scheduling algorithms and how the impact of the caches could be integrated in the simulation [8]. As a consequence, the concept of *execution time model* was introduced.

Contribution. This paper presents SimSo and the main novelties that now enable to conduct large scheduling evaluations using it. It is indeed possible to automate the simulation of scheduling algorithms from the generation of the systems to the collection of the resulting data. The main task generators are now included and the number of available schedulers increased from five to more than twenty-five. Our methodology to automate the evaluation of multiple scheduling algorithms is described through an example.

Paper organization. The remainder of this paper is organized as follows: in Section II, related work is summarized. Section III presents SimSo, and Section IV shows how it can be used through an example. Finally, Section V provides some concluding remarks and envisages future work.

II. Related Work

Our work addresses the evaluation of the performance of scheduling algorithms using empirical measures. Empirical evaluations of scheduling algorithms focus on the overheads involved in scheduling decisions. The main studied causes of overheads are context switches, preemptions, migrations and computational complexity. Two approaches are typically considered to evaluate them. The first one is based on measured performance on a real platform with a dedicated operating system, e.g. the experiments done with LITMUS^{RT} [4], an extension of the Linux Kernel developed at the University of North Carolina, or the experimental work of Lelli et al. [20] on a dedicated implementation of Linux with RM and EDF multiprocessor schedulers. This method could also be conducted on a cycle-accurate simulated architecture with a real operating system as in [31]. The second approach is to use tools dedicated to the simulation of real-time systems. Most of these tools are designed to validate, test and analyze systems. MAST [16] proposes a set of tools to model and analyze distributed real-time systems with, for instance, feasibility tests or sensitivity analyses. MAST also includes a simulator, JSimMAST. Cheddar [28] proposes a GUI comprising a simulator, many feasibility tests and it is also used to simulate AADL models. RTSIM [5] is a collection of programming libraries for the simulation of real-time control systems. It is used in particular for experimenting new scheduling algorithms. The last version was published in 2007.

STORM [30] and YARTISS [6] are the closest tools to what we aim. They offer a simulator to conduct evaluation on scheduling algorithms with the possibility to easily join new scheduling policies. However, due to its time triggered simulation engine, STORM does not provide an efficient way to model the unit of time below a tick of simulation which is a significant limitation for us. YARTISS is certainly the most suitable tool to evaluate scheduling algorithms by considering overheads or hardware effects. However, we began the implementation of our tool in 2011, before YARTISS was published. Moreover, its design is focused on the study of energy consumption and customizing it for our needs would have been difficult.

III. SimSo

To facilitate the experimentation of scheduling algorithms, we thus propose a dedicated tool: SimSo[1], a real-time scheduling simulator designed to be easy to use as well as extend. This software is freely available under an open source license.

The design of SimSo has been driven by the components available in real systems so that practical issues regarding the implementation can be taken into consideration. Such issues would have been hard or even impossible to integrate into theoretical studies.

A. Architecture

The core of SimSo relies on SimPy[2], a process-based discrete-event simulation framework. The use of discrete-event simulation allows it to deal with short and long durations at the same cost. Its process-based nature offers a convenient way to express the behavior of the simulated components.

The characteristics of a system are modeled by a *Configuration* object that contains all the information about the system (tasksets, processors, duration, scheduler, etc). This object provides some methods to configure the system but also to save it into an XML file.

Figure 1 shows the main classes of SimSo and their mutual interactions. The design of SimSo is inspired by real systems: there are processors, tasks, jobs, timers, etc. Each of these objects simulates the behavior of the corresponding part on the system: *Tasks* release the jobs; *Jobs* emulate the execution of the task's code; *Timers* can launch a method on a processor at a given time; etc. The instances of *Processors* are actually the central part of the simulation because they simulate both a processor and the operating system executing on it. Each processor can execute a job or be interrupted to execute a method of the scheduler. Finally, the *Scheduler* object is not an active process. It could be considered as a part of the operating system and as a consequence, its methods are only called by the *Processors*.

The *Model* object is the conductor of the simulation. It takes as a parameter the *Configuration* object. When the *run_model* method is called, the objects described above are created and launched.

[1]SimSo: http://homepages.laas.fr/mcheramy/simso/
[2]SimPy: http://simpy.readthedocs.org/

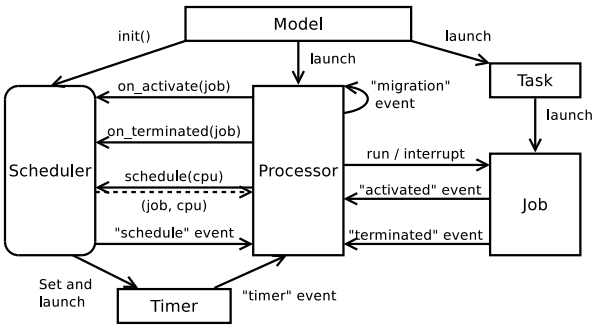

Fig. 1. Interactions between main class instances. *Processor*, *Task*, *Job* and *Timer* are *Process* objects and can have multiple instances.

The design of SimSo allows it to take into consideration various time overheads that occur during the life of the system. This includes direct overheads such as context-switches and scheduler calls (with fixed time penalties) but also indirect overheads with a simplified system of locks to forbid the parallel execution of a scheduler if needed. Such overheads are applied on the processor they are supposed to occur (e.g. the time spent in the scheduler is taken into account on the processor that called the scheduler).

We would also like to draw attention to the fact that the above-mentioned overheads only consume extra-time without changing the time used to execute the jobs. Indeed, as an example, these overheads do not take into account the possible cache misses that could slow down a job and increase its duration. This important aspect can also be taken into account by SimSo and is explained in section III-D.

B. Writing a Scheduler

The first requirement for the experimentation of a real-time scheduling policy is, undoubtedly, a way to specify the algorithm. This should be able to deal with any kind of online scheduler: global, partitioned, semi-partitioned, etc. Moreover, the implementation of a scheduler in a simulator should also be realistic in the sense that it should rely on mechanisms available on a real system. For instance, the choice of which processor should run the scheduler may have an impact on the performance or even the schedulability. Another example is the finite precision of the timers: this may introduce a tiny difference compared to the theoretical schedule and cause a major issue.

One of the advantages of using a simulator is to simplify the experimentation. Writing a scheduler should therefore be as easy as possible and rely on useful methods. We decided to use Python, a high-level language that benefits from a growing interest from the scientific community (e.g. the SciPy project). In practice, most of the schedulers that we have implemented contain less than 200 lines of code. The language is different to the one that would be used on a real implementation, however, this does not change the underlying algorithms and logic.

A scheduler for SimSo is a Python class that inherits from the *Scheduler* class and is loaded dynamically into the

simulator. The following methods must be implemented:

- **init:** The *init* method is called when the simulation starts, it is used to initialize the scheduler.
- **on_activate:** This method is called whenever a job is activated.
- **on_terminated:** This method is called when the execution of a job is done or when a job is aborted.
- **schedule:** This method returns the scheduling decisions. This method is called when a processor has been requested to take a scheduling decision. This request is usually done during a job activation, a job termination or by a timer.

As an example, figure 2 shows the source code of a global multiprocessor Earliest Deadline First scheduler[3].

```
from simso.core import Scheduler

class G_EDF(Scheduler):
    def init(self):
        self.ready_list = []

    def on_activate(self, job):
        self.ready_list.append(job)
        # Send a "schedule" event to the processor.
        job.cpu.resched()

    def on_terminated(self, job):
        # Send a "schedule" event to the processor.
        job.cpu.resched()

    def schedule(self, cpu):
        decision = None # No change.

        if self.ready_list:
            # Look for a free processor or the processor
            # running the job with the least priority.
            key = lambda x: (1 if not x.running else 0,
                             x.running.absolute_deadline if x.running else 0)
            cpu_min = max(self.processors, key=key)

            # Obtain the job with the highest priority within the ready list.
            job = min(self.ready_list, key=lambda x: x.absolute_deadline)

            # If the selected job has a higher priority
            # than the one running on the selected cpu:
            if (cpu_min.running is None or
                    cpu_min.running.absolute_deadline > job.absolute_deadline):
                self.ready_list.remove(job)
                if cpu_min.running:
                    self.ready_list.append(cpu_min.running)
                # Schedule job on cpu_min.
                decision = (job, cpu_min)

    return decision
```

Fig. 2. Code of a global multiprocessor Earliest Deadline First scheduler.

C. Available Schedulers

In order to check the ability to express a wide range of algorithms, we have already implemented more than 25 schedulers. The main uniprocessor schedulers, RM, DM, FP, EDF and M-LLF [24] are available. The DVFS schedulers Static-EDF and CC-EDF [25] are also available.

The library of schedulers provided with SimSo also includes a large variety of multiprocessor real-time scheduling algorithms, from partitioning to global ones.

The *partitioned* approach forbids migrations and necessitates a static allocation of the tasks to the processors. The schedulers P-EDF and P-RM are available (they use the Decreasing First-Fit assignment algorithm). Moreover, a dedicated class is provided in SimSo to offer the possibility to choose any uniprocessor scheduler and one of the available assignment algorithms (First-Fit, Next-Fit, Best-Fit, Worst-Fit, with or without an initial sorting). This class is intended to ease the development of a partitioned scheduler, but it is not mandatory.

On the other side, when migration is permitted, scheduling algorithms are referred to as *global*. A first category of global schedulers use a single list of active tasks and assign a priority to each task. For an architecture with m processors, the m jobs with the highest priority run in parallel. The following algorithms belonging to that category are available in SimSo: G-RM, G-EDF, G-FL [13], EDF-US [29], PriD [18], EDZL, M-LLF [24] and more recently U-EDF [22].

Baruah introduced the concept of fairness as a way to achieve optimality in terms of schedulability. SimSo provides such PFair schedulers with PD^2 and its work-conserving variant ER-PD^2 [1]. Subsequently, it was demonstrated that the fairness constraint could be released to only apply at the job boundaries and thus could reduce the number of preemptions and migrations. This led to the BFair and DP-Fair techniques. We have implemented such schedulers: LLREF [9], LRE-TL [15], DP-WRAP [21], BF [31] and NVNLF [14].

In order to reduce the number of migrations, some hybrid approaches, termed *semi-partitioned* approaches, combine the advantages of global and partitioned scheduling. At the present time, SimSo proposes three semi-partitioned schedulers: EDHS [19], EKG [2] and RUN [26].

D. Execution Time Model

When simulation is used to study the schedulability of a system, it is usual that the tasks meet their worst-case execution time at each job. However, the use of the WCET is in fact very pessimistic: the worst-case is an upper-bound that is hardly reached by the jobs, and it is even less likely that the jobs of all the tasks meet their WCET at the same time. As a consequence, we believe that the WCET approach should not be the only way to compare policies in terms of performance. It is non-realistic and gives an advantage to some scheduling policies that highly depend on the WCET. Relatedly, schedulers capable to take benefits from shorter computation times cannot be fairly evaluated. In [8], we give some experimental results that illustrate this fact.

Also, many scheduling evaluations only focus on the number of preemptions and migrations because they are the source of overheads. A preemption induces a system overhead due to the context-switching, but it may also increase the computation time of a job by causing extra cache misses. In fact, Mogul and Berg have shown that the Cache-Related Preemption Delays (CRPD) are more important than the system overheads. To increase realism, it is essential to integrate CRPD within the computation time of the jobs.

As a consequence of the two previous remarks, it is desirable to have the possibility to simulate a system with customized durations of jobs, depending on the purpose of the simulation. In SimSo this point is achieved with the *Execution Time Models* (ETM). An ETM is a class that determines the duration of the jobs during the simulation. Figure 3 shows

[3] A minor modification to this code would reduce the number of migrations by executing a job in the same processor than its previous execution.

the communication between a job and the ETM object (there is a single ETM object for all the jobs). The ETM object is informed by the jobs of any scheduling event. The job will use the *get_ret* method to get a lower bound of its remaining execution time and, when that time is up, the job calls that method again until it returns 0.

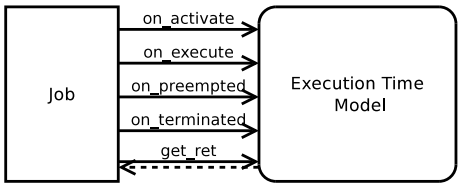

Fig. 3. Interface of any execution time model.

Several *Execution Time Models* are already available in SimSo. The simplest model consists of using the WCET of the tasks for their execution time. A second one uses a random duration for each job to meet a given average execution time (ACET). The ACET model uses a normal distribution defined by its mean, its standard deviation and is bounded by the WCET. Another model detects the preemptions and migrations and extends the WCET[4] of the job using fixed time penalties. Finally, a more complex model tries to simulate the state of the caches. In this latter model, the execution time of the jobs depends on the events that happen while they are active. This ETM is also interesting because it simulates the impact of shared caches and, as a consequence, it is impossible to know in advance when a job will end since it depends on external events.

These models can also deal with Dynamic Voltage and Frequency Scaling (DVFS). Indeed, when the speed of a processor is changed, the job that was running on it is preempted and resumed in order to inform the ETM and to reevaluate its remaining execution time. The current DVFS model simply considers that a job consumes its computation time proportionally to the speed of the processor. This is obviously a simplified assumption, but it is possible to implement more realistic ETM models to deal with DVFS.

Similarly, it should also be possible to add an energy consumption model.

E. Generation of Tasksets

A taskset is defined by the number of tasks, their utilization factor, periods, deadlines and the total utilization. Bini and Buttazzo showed how the random generation of the tasksets can bias the experimental results of some scheduling algorithms on uniprocessor [3].

For the multiprocessor case, several methods are used by the researchers to generate the tasksets. The most common algorithms are implemented in SimSo:

- Kato et al. use an approach inspired by the algorithm described by Ripoll et al. where tasks are appended to the taskset until the targeted total utilization is reached [19], [27]. The number of tasks is therefore variable.
- The algorithms *UUniFast-Discard* and *RandFixedSum* generate a taskset with a given number of tasks and a given total utilization [12]. At the present time, this methods seem to be the most efficient in generating tasksets with a weak bias.

These algorithms only generate a set of utilization rates and must thus be combined with a period generator. The following algorithms are made available in SimSo:

- *Uniform distributions in various fixed ranges:* Most evaluations use it and this is certainly an interesting way to study the influence of the periods, but it may not be relevant for realistic cases.
- *Log-uniform choice of periods [10]:* For a period range of 1-1000ms, the log-uniform distribution generates an equal number of tasks in each time band (1-10ms, 10-100ms, 100-1000ms) whereas a uniform distribution would generate 90% of the periods in the range 100-1000ms.
- *Random draw among a fixed set of values:* One could argue that in an industrial system, the periods are derived from the specifications, which are partly written by humans. Task periods are therefore more likely to be rounded.

Other period generators could also be added in the future. For instance, Goossens [17] suggested a method to reduce the hyper-period of the system by using periods that can be decomposed in a limited number of prime numbers.

F. Collecting Simulation Results

In order to evaluate scheduling algorithms, some data must be collected from the simulation. The literature proposes many measures, here is a non-exhaustive list of data that could be recovered:

Success rate: The ratio between the number of jobs that have exceeded their deadline and the number of jobs. It gives a performance indicator on the schedulability of a taskset.

Preemptions and migrations: Preemptions and migrations are a factor of overhead and many recent schedulers are focusing on their reduction. A distinction is made between job migration and task migration since they may have not the same implications.

Scheduler calls: The algorithm of a scheduler requires some time to determine which jobs should run on the processors. Some scheduling policies are known to make many scheduling decisions, and some require a significant amount of time to compute. Therefore, it is interesting to keep track of the number of calls to the various methods of the scheduler.

Normalized laxity: Lelli et al. proposed to measure the performance of a scheduler by computing the normalized laxity [20]. The laxity of a job is its relative deadline minus its response-time. The laxity of each job of each task is divided by the task period in order to obtain a normalized laxity. A greater normalized laxity is synonym of a better safety and better reactivity.

[4]In this case, the WCET is defined as the worst-case execution time without any interruption.

During the simulation of a system with SimSo, every significant events are traced. At the end of the simulation, a *Results* object is built to store these events and could be post-treated to compute measurements. Whereas this approach is actually heavier than just counting events such as the preemptions and migrations during the simulation, this provides more flexibility. Indeed, it is not necessary to modify the code of the simulator to add the computation of new measurements one did not think about. A set of methods are also available to ease the retrieval of usual metrics such as the ones mentioned above.

SimSo provides a graphical user interface that helps to configure a system and run it. That GUI is capable of displaying common measures such as preemptions, migrations, or execution times. It is also possible to display a gantt chart, which is very useful during the development of a scheduler. However, this GUI only shows the results for a single simulation.

G. Conducting an Evaluation Campaign

To conduct a large evaluation campaign, it is possible to use SimSo as a Python module. This way, a Python script can be written to automate the creation of systems, their simulation and the collection of the results. This choice was motivated by the fact that the studies can be very specific and a graphical user interface would be necessarily too frozen or too complex. On the other hand, using a script is much more flexible. Everything that is possible using the graphical user interface is also possible from a script.

IV. Example

This section illustrates the use of SimSo in conducting an experiment on scheduling policies. SimSo is used as a module for a Python script and the steps described below have been programmed.

This experiment focuses on the number of preemptions and migrations in function of the number of tasks, for various numbers of processors and load. The objective is to compare five schedulers: G-EDF, NVNLF, EKG[5], RUN and U-EDF.

A. Generation of the Configurations

The first step is to define the characteristics of the simulated systems. For this example, we have selected the following parameters:

- Number of tasks: 20, 30, 40, 50, 60, 70, 80, 90, 100
- Number of processors: 2, 4, 8
- System utilization: 85%, 95%

For each configuration (tasks, processors, utilization), twenty tasksets are generated using the methods offered by SimSo, leading to a total of 5400 systems ($9 \times 3 \times 2 \times 20 \times 5$). The RandFixedSum algorithm was used to determine the task utilizations and the periods were chosen randomly within a log-uniform distribution between 2 and 100 ms. The ACET Execution Time Model is used and, for each task, the expected value is set to 75% of the WCET and the standard deviation to 10% of the WCET. Each system is simulated on the interval of time 0-1000ms[6].

The *Configuration* objects were saved into XML files for potential reuse (it is interesting to repeat simulations on systems with atypical results in order to obtain a better understanding.).

B. Simulation and Collection of the Results

SimSo executed 5400 simulations which took approximatively 2 hours on an Intel Core i7 processor.

When a simulation is done, the number of preemptions and job migrations are extracted from the *Results* object built by the *Model* object. Preemptions caused by the system (e.g. the scheduler is called but no decision is taken) are not taken into account.

In order to facilitate the analysis, we stored the data in an SQLite3 database.

C. Analysis

From that database, another script draws the charts using matplotlib, a plotting library for Python. Each point is the mean of the twenty tasksets sharing the same parameters. The results for 8 processors and a system utilization of 95% are shown on Figure 4.

A few comments on the results are provided here as a complement to the figure. EKG generates a lot of migrations that could be easily avoided with a better choice of the parameter K or other improvements [23]. The results for NVNLF are getting better with more processors unlike the others. U-EDF could probably do better combined with clustering. With more than 20 tasks, RUN acts as a partitioned scheduler most of the time. G-EDF provides better results in terms of preemptions and migrations but a few jobs were aborted as a consequence of deadline misses. U-EDF and RUN could probably catch up with G-EDF with a work-conserving variant.

V. Conclusion

In this paper, we have presented SimSo, a simulation tool to evaluate the multiprocessor schedulers. Its objective is to facilitate the comparison of the numerous scheduling policies. To this end, we will conduct large campaigns of experiments with many scheduling algorithms using the same tasksets. This should allow us to reproduce numerous experiments in order to confirm or invalidate results. At the present time, more than twenty-five schedulers are available, showing that SimSo is capable of handling partitioned, global and hybrid scheduling approaches.

The architecture of SimSo, in particular the scheduling interface, was briefly explained. Particular care has been taken to keep a realistic scheduling interface so that practical decisions are not eluded. This has also enabled SimSo to take into consideration direct overheads such as the context-switches or scheduling decisions. Moreover, the computation time of the jobs is determined by a model that can be selected depending on the purpose of the simulation. Hence, the computation time

[5]The parameter K has been set to the number of processors.

[6]Unfortunately, the hyper-period for a set of 100 tasks with random periods is far too long to be considered (in years).

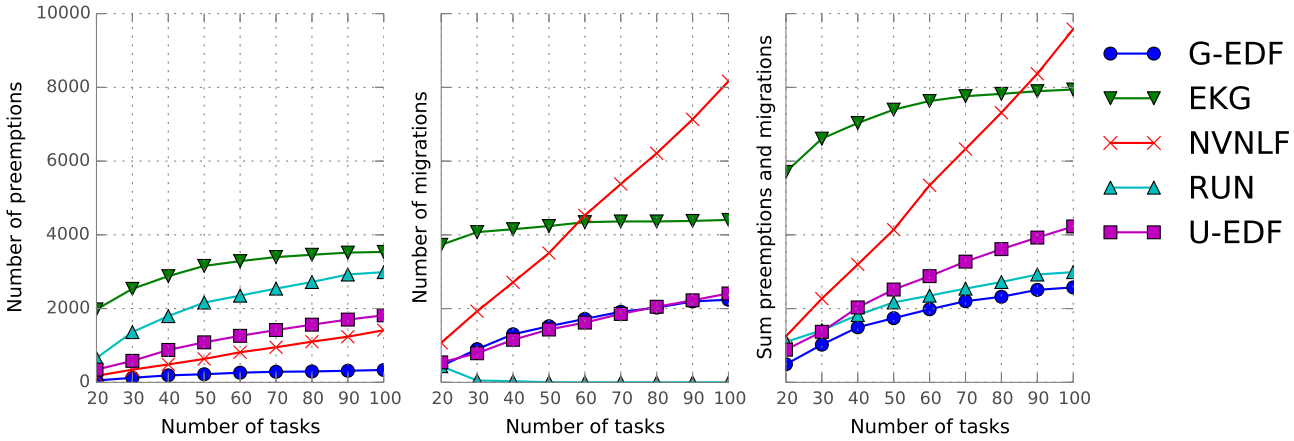

Fig. 4. Number of preemptions and migrations for a system with 8 processors and a (worst-case) total utilization of 95%. The simulation used random durations for the job computation time.

of a job can either be a static duration, a random duration, or even take into account cache-related preemption delays. Additionally, a small example shows the capability of SimSo to produce concrete results.

Future work includes an improvement of SimSo by introducing cache interferences in the simulation and introducing more complex task behaviors such as shared resources and precedence relations.

Acknowledgment

The work presented in this paper was conducted under the research project RESPECTED (http://anr-respected.laas.fr/) which is supported by the French National Agency for Research (ANR), program ARPEGE.

References

[1] J. Anderson and A. Srinivasan, "Early-release fair scheduling," in *Proc. of ECRTS '00*, 2000.

[2] B. Andersson and E. Tovar, "Multiprocessor scheduling with few preemptions," in *Proc. of RTCSA*, 2006.

[3] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Syst.*, vol. 30, no. 1-2, 2005.

[4] J. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. Anderson, "LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers," in *Proc. of RTSS*, 2006.

[5] A. Casile, G. Buttazzo, G. Lamastra, and G. Lipari, "Simulation and tracing of hybrid task sets on distributed systems," in *Proc. of RTCSA*, 1998.

[6] Y. Chandarli, F. Fauberteau, D. Masson, S. Midonnet, and M. Qamhieh, "YARTISS: A Tool to Visualize, Test, Compare and Evaluate Real-Time Scheduling Algorithms," in *Proc. of WATERS*, 2012.

[7] M. Chéramy, A.-M. Déplanche, and P.-E. Hladik, "Simulation of real-time multiprocessor scheduling with overheads," in *Proc. of SIMUL-TECH*, 2013.

[8] M. Chéramy, P.-E. Hladik, A.-M. Déplanche, and S. Dubé, "Simulation of real-time scheduling with various execution time models," in *Proc. of the WiP session of SIES*, 2014.

[9] H. Cho, B. Ravindran, and E. Jensen, "An optimal real-time scheduling algorithm for multiprocessors," in *Proc. of RTSS*, 2006.

[10] R. Davis and A. Burns, "Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems," in *Proc. of RTSS*, 2009.

[11] ——, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surv.*, vol. 43, no. 4, 2011.

[12] P. Emberson, R. Stafford, and R. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *Proc. of WATERS*, 2010.

[13] J. Erickson and J. Anderson, "Fair Lateness Scheduling: Reducing Maximum Lateness in G-EDF-Like Scheduling," in *Proc. of ECRTS '12*, 2012.

[14] K. Funaoka, S. Kato, and N. Yamasaki, "Work-conserving optimal real-time scheduling on multiprocessors," in *Proc. of ECRTS '08*, 2008.

[15] S. Funk and V. Nanadur, "LRE-TL: An Optimal Multiprocessor Scheduling Algorithm for Sporadic Task Sets," in *Proc. of RTNS*, 2009.

[16] M. Gonzalez Harbour, J. Gutierrez Garcia, J. Palencia Gutierrez, and J. Drake Moyano, "MAST: Modeling and analysis suite for real time applications," in *Proc. of ECRTS '01*, 2001.

[17] J. Goossens and C. Macq, "Limitation of the hyper-period in real-time periodic task set generation," in *Proc. of the 9th International Conference on Real-Time Systems (RTS)*, 2001.

[18] J. Goossens, S. Funk, and S. Baruah, "Priority-driven scheduling of periodic task systems on multiprocessors," *Real-Time Systems*, vol. 25, no. 2-3, 2003.

[19] S. Kato and N. Yamasaki, "Portioned EDF-based scheduling on multiprocessors," in *Proc. of EMSOFT*, 2008.

[20] J. Lelli, D. Faggioli, T. Cucinotta, and G. Lipari, "An experimental comparison of different real-time schedulers on multicore systems," *Journal of Systems and Software*, vol. 85, no. 10, 2012.

[21] G. Levin, S. Funk, C. Sadowski, I. Pye, and S. Brandt, "DP-FAIR: A Simple Model for Understanding Optimal Multiprocessor Scheduling," in *Proc. of ECRTS '10*, 2010.

[22] G. Nelissen, V. Berten, V. Nelis, J. Goossens, and D. Milojevic, "U-EDF: An Unfair But Optimal Multiprocessor Scheduling Algorithm for Sporadic Tasks," in *Proc of ECRTS '12*, July 2012.

[23] G. Nelissen, S. Funk, and J. Goossens, "Reducing Preemptions and Migrations in EKG," in *Proc. of RTCSA*, Aug 2012.

[24] S.-H. Oh and S.-M. Yang, "A modified least-laxity-first scheduling algorithm for real-time tasks," in *Proc. of RTCSA*, 1998.

[25] P. Pillai and K. G. Shin, "Real-time dynamic voltage scaling for low-power embedded operating systems," in *Proc. of SOSP '01*, 2001.

[26] P. Regnier, G. Lima, E. Massa, G. Levin, and S. Brandt, "RUN: Optimal Multiprocessor Real-Time Scheduling via Reduction to Uniprocessor," in *Proc. of RTSS*, 2011.

[27] I. Ripoll, A. Crespo, and A. Mok, "Improvement in feasibility testing for real-time tasks," *Real-Time Systems*, vol. 11, no. 1, 1996.

[28] F. Singhoff, J. Legrand, L. Nana, and L. Marcé, "Cheddar: A flexible real time scheduling framework," *Ada Lett.*, vol. XXIV, no. 4, 2004.

[29] A. Srinivasan and S. Baruah, "Deadline-based scheduling of periodic task systems on multiprocessors," *Inf. Process. Lett.*, vol. 84, no. 2, 2002.

[30] R. Urunuela, A.-M. Déplanche, and Y. Trinquet, "STORM a simulation tool for real-time multiprocessor scheduling evaluation," in *Proc. of ETFA*, 2010.

[31] D. Zhu, D. Mosse, and R. Melhem, "Multiple-resource periodic scheduling problem: how much fairness is necessary?" in *Proc. of RTSS*, 2003.