An Experimental Comparison of Different Real-Time Schedulers on Multicore Systems[1]

Juri Lelli(*), Dario Faggioli(*), Tommaso Cucinotta(§), Giuseppe Lipari(*)

(*) *Real-Time Systems Laboratory, Scuola Superiore Sant'Anna, Pisa (Italy)*
e-mail: {j.lelli, d.faggioli, g.lipari}@sssup.it
(§) *Bell Laboratories, Alcatel-Lucent, Dublin (Ireland)*
e-mail: tommaso.cucinotta@alcatel-lucent.com

Abstract

In this work, an experimental comparison among the Rate Monotonic (RM) and Earliest Deadline First (EDF) multiprocessor real-time schedulers is performed, with a focus on soft real-time systems. We generated random workloads of synthetic periodic task sets and executed them on a big multi-core machine, using Linux as Operating System, gathering an extensive amount of data related to their exhibited performance under various real-time scheduling strategies. The comparison involves the fixed-priority scheduler for multiprocessors as available in the Linux kernel (with priorities set so as to achieve RM), and on our own implementation of EDF, both configured in global, partitioned and clustered mode. The impact of the various scheduling strategies on the performance of the applications, as well as the generated scheduling overheads, are compared presenting an extensive set of experimental results. These provide a comprehensive view of the performance achievable by the different schedulers under various workload conditions.

Keywords: real-time scheduling, global EDF, multi-core systems, experimental evaluation

1. Introduction

Multi-processor and multi-core computing platforms are nowadays largely used in the vast majority of application domains, ranging from embedded systems, to personal computing, to server-side computing including GRIDs and Cloud Computing, and finally high-performance computing. In embedded systems, small multi-core platforms are considered a viable and cost-effective solution, especially for their lower power requirements as compared to a traditional single processor system with equivalent computing capabilities. The increased level of parallelism in these systems may be conveniently exploited to run multiple real-time applications, like found in industrial control, aerospace or military systems; or to support soft real-time Quality of Service (*QoS*) oriented applications, like found in multimedia, gaming or virtual reality systems.

Servers and data centres are shifting towards (massively) parallel architectures with enhanced maintainability, often accompanied by a decrease in the clock frequency driven by the increasing need for "green computing" [1][2].

[2] For example, various talks held at the Bell Labs Open Days on October, 2010 in Stuttgart highlighted such a need.

Cloud Computing applications promise to move most of the increasing personal computing needs of users into the "cloud". This is leading to an unprecedented need for supporting a large number of interactive and real-time applications, often involving on-the-fly media streaming, processing and transformations with demanding performance and latency requirements. These applications usually exhibit nearly periodic workload patterns which often cannot saturate the available computing power of a single (powerful) CPU. Therefore, there is a strong industrial interest in executing an ever-increasing number of applications of this type on the same system, node, physical CPU and even core, whenever possible, in order to minimise the number of needed nodes (and reduce both power consumption and costs).

In this context, a key role is played by real-time CPU scheduling algorithms for multi-processor systems, due to their potential impact on the performance experienced by the scheduled applications. These can be roughly categorised into *global schedulers* and *partitioned schedulers*. In global scheduling, the ready tasks with the highest priorities execute on the available processors at any time. This implies the need for dynamically *migrating* tasks among processors. On the other hand, in partitioned scheduling each task is statically allocated on one processor, according to a specific allocation algorithm, and tasks cannot migrate. *Clustered schedulers* reside somewhat in the middle, where the available processors are partitioned into clusters to which tasks are statically assigned, but in each cluster tasks are globally scheduled. In a multi-core system, the use of partitioned or clustered scheduling policies brings the additional problem of how to partition the tasks among cores or clusters of cores.

An orthogonal way to categorise schedulers is according to the way task priorities are assigned. If tasks' priorities never change during the task lifetime, we have a *fixed priority scheduler*, otherwise we have a *dynamic priority scheduler*. In this paper, we focus on the two most popular schedulers: Rate Monotonic (RM) priority assignment for fixed priority schedulers; and Earliest Deadline First (EDF) for dynamic priority schedulers.

The designer of a real-time system often needs to compare different available real-time scheduling strategies, in terms of their impact on the performance of the hosted applications.

Many recent papers compare different scheduling strategies (global/partitioned, and fixed/dynamic priority) from a theoretical point of view (see Section 2 for an analysis of related work). In these works different schedulers are compared with respect to their achievable overall utilisation, under the constraint of maintaining the task-set schedulable, assuming worst-case conditions for the tasks execution, i.e., the analysis is based on the Worst-Case Execution Times (WCET). Although appropriate for hard real-time systems, in soft real-time ones such approaches are at risk of neglecting (or merely reminding to WCET analysis for) many practical issues, such as the overhead of the scheduler, the increased execution times as due to migrations and increased cache misses, and the presence of variability of memory access times in Non-Uniform Memory-Access (NUMA) machines. These issues may have a great influence on the actual performance, especially as the number of cores increases.

For instance, partitioned schedulers typically present less overhead. However, in an open system where tasks may dynamically enter and leave the system, a static allocation strategy may lead to underutilised systems. On the other hand, global scheduling is more flexible as it automatically balances the load across all processors. In addition, global

dynamic priority schedulers (like EDF) are known to guarantee bounded tardiness as long as the total load does not exceed the system capacity [2, 3, 4]. However, global schedulers typically present a higher overhead; they cause migrations, which in turn may lead to a non-negligible increase in the tasks execution times.

Therefore, it is important to quantify such overheads in order to complement the theoretical properties of a scheduler with its practical performance figures. In this way, the designer can take a more informed decision on which scheduler to select for various application workload types.

1.1. Paper Contributions

In this paper the performance of partitioned, clustered and global variants of Rate Monotonic (RM) and Earliest Deadline First (EDF) scheduling algorithms in the Linux OS are compared. The experimental comparison is conducted on the Linux OS due to its wide applicability (with various kernel-level patches) in the domain of real-time systems.

We compare our own implementation of Global EDF (G-EDF) in the Linux kernel with respect to the fixed priority Linux scheduler (configured so as to realise RM). The goal is not to demonstrate the effectiveness of our scheduler, but rather to make a thorough performance comparison, and establish which scheduler performs better in different contexts. In order to precisely control the experiments, our methodology consists in generating sets of synthetic real-time tasks with various characteristics in terms of execution time and memory requirements and usage. The task set is then executed on a multi-core platform and the tasks' performance is measured. The focus is on the metrics typically of interest for developers and other people who investigate on performance issues, and not purely on schedulability analysis. Indeed, we consider the laxity (tasks should not complete too close to their deadlines), the number of migrations and context switches (they may potentially affect negatively the performance), and the number and type of cache misses (they have a direct impact on the application execution times and performance). Since we focus on the comparison among different CPU scheduling policies, in this paper we only consider independent tasks. The hardware platform is a AMD^R Opteron^{TM} 6168 with 48 cores (4 sockets with 12 cores for each processor).

Our implementation of G-EDF has been made available as open-source code. This, together with the details about the configuration of the experiments, allows to reproduce and verify all the results that have been included in the paper (see the Section 6). Also, this allows other researchers to perform independent investigations on partitioned, clustered and global EDF scheduling on Linux, as well as to develop new schedulers and concretely compare their performance with these policies. Last, but not least, this gives to any developer the possibility to try these policies for their real-time applications.

1.2. Paper Organisation

The remainder of this paper is organised as follows: in Section 2, the related work is briefly recalled. In Section 3, the background concepts needed to understand the remainder of the paper are introduced, and in Section 4 the modifications performed on the Linux kernel, to make it support global EDF scheduling, are sketched out. Sections 5

and 6 describe the methodology and report the results of the experimental evaluation phase, respectively. Finally, in Section 7 conclusions are drawn and possible directions for future work are envisioned in Section 8.

2. Related Work

The comparisons available in the literature between different real-time multiprocessor scheduling solutions are almost always conducted by measuring the percentage of schedulable task sets among a number of randomly-generated ones. For example, this has been done in [5, 6, 7]. These approaches often rely on schedulability tests or simulations, and they do not involve real tasks running on a real system, thus they cannot collect such run-time metrics as the actually experienced laxity/tardiness, cache misses, context switches and migrations. For example, the possibility for these approaches to properly account for overheads due to scheduling and migration are limited. Often, these overheads are assumed to have already been considered in the Worst Case Execution Time (WCET) of the tasks. However, the scheduling policy itself may impact the WCET figures (as due to how many times a task is preempted or migrated).

Some of the main theoretical properties of EDF and RM are analysed in [8], but the study refers only to uniprocessor systems.

In the field of WCET analysis, in [9] a method is proposed to bound the cache-related migration delay in multicores, while in [10, 11] the focus is on devising proper task interference models. On a slightly more practical basis, memory access traces of actual program executions have been used to feed cache architectural simulators in [12, 13], while in [14, 15, 16] some micro-benchmarks have been run on a Linux system in order to quantify the cache-related context switch delay in some specific situation (e.g., because of interrupt processing). Although being related to the present work, these studies concentrate on the effects that either migrations or preemptions have on task execution due to interference on shared caches in particular conditions. Instead, this paper aims at a more holistic and high-level comparison of some of the available solutions for multiprocessor scheduling.

In the domain of distributed real-time scheduling on heterogeneous Grids, various schedulers have been compared by considering applications comprising of direct acyclic graphs (DAG) of computations [17, 18] with end-to-end deadlines. Similarly, the investigation in the present work might be expanded by comparing the performance of the schedulers under DAG-based workloads. However, due to space reasons, we do not consider DAGs in this work, but we defer such further investigations to future research.

The line of research closer to the approach of this paper is the one carried out by the Real-Time Systems Group at University of North Carolina at Chapel Hill. By means of their $LITMUS^{RT}$ testbed [19], they conducted investigations on how real overheads affect the results coming out of theoretical analysis techniques. There are several works by such group going in this direction: in [19] Calandrino et al. studied the behaviour of some variants of global EDF and Pfair, but did not consider fixed-priority; in [20], Brandenburg et al. explored the scalability of a similar set of algorithms, while in [21] the impact of the implementation details on the performance of global EDF is analysed; finally, in [22, 23, 24] Bastoni et al. showed intents similar to the ones of this paper, concentrating on partitioned, clus-

tered and global EDF on a large multi-core system. In all these works, samples of the various forms of overhead that show up during execution on real hardware are gathered and are then plugged in schedulability analysis tests, making them more accurate. However, the final conclusions about the performance of the various scheduling algorithms are actually influenced by the accuracy of the best known schedulability tests, which are often quite conservative.

Instead, in this work the tasks have just been deployed under various scheduler configurations, and their obtained performance has been systematically measured. Therefore, the conclusions drawn in this paper are simply derived from the intrinsic timing behaviour of the tasks as exhibited during the performed experiments. Further considerations regarding specifically the comparison with [23] follow in Section 7.

3. Background

In this section a few background concepts about real-time multiprocessor scheduling are introduced for a better understanding of the rest of the paper.

In this paper, a real-time system is considered as a set of n real-time tasks $\{\tau_i, \ldots, \tau_n\}$ to be scheduled over a set of m identical unit-capacity processors $p_1, \ldots, p_m$. Each task τ_i follows the *periodic* task model: it activates periodically with an inter-arrival time of T_i, generating a sequence of *jobs*. Each job executes for at most a worst-case execution time (WCET) of C_i, and must complete within a relative deadline equal to the period T_i. The difference between the absolute deadline of a job and the time it finishes is called *laxity*, and it is normally positive (i.e., the job respected the deadline). However, the focus is on *soft* real-time tasks, where occasional deadline misses may be tolerable (usually leading to some kind of Quality of Service degradation). Therefore, the laxity may be negative as well (in such a case we also refer to its absolute value as *tardiness*). Each task has also a processor utilisation factor $U_i = \frac{C_i}{T_i}$. Finally, in this paper, the size of the data accessed by a task during its jobs is called the *working set size* (WSS).

This work focuses on the multiprocessor variants of both fixed and dynamic priority real-time scheduling (more details about theoretical results in this field can be found in [25, 26, 27, 28]). Moreover, *global*, *clustered* and *partitioned* policies are considered. The main advantage of global scheduling is in the simplicity of deployment, since the load is automatically balanced among all the available CPUs, while finding an optimal partitioning of the tasks is a bin-packing problem, which is NP-hard to solve. On the other hand, partitioned scheduling is immune from the overhead caused by migrations. Clustered approaches are used as a compromise between these two extremes.

4. Implementation of Global Scheduling in SCHED_DEADLINE

In the Linux kernel, scheduling decisions are implemented inside *scheduling classes*. Stock Linux comes with two classes, one for fairly scheduling best-effort activities (`SCHED_OTHER` policy) and one implementing fixed priority real-time scheduling (`SCHED_FIFO` or `SCHED_RR` policies), following the POSIX 1001.3b [29] specification. Recently, a new real-time scheduler has been made available for the Linux kernel in form of a new scheduling class. It is

called SCHED_DEADLINE [3] [30] and implements EDF scheduling with both hard and soft resource reservation capabilities [31, 32]. More precisely, SCHED_DEADLINE implements a variant of the Constant Bandwidth Server (CBS) algorithm [33], achieving *temporal isolation* among concurrently running tasks (more details are available in [30]). Each task in the OS belongs to one of the mentioned scheduling classes. From here on, a task whose scheduling class was set to SCHED_DEADLINE will be referred to as a SCHED_DEADLINE task.

We modified SCHED_DEADLINE, originally supporting only partitioned scheduling, to support also global and clustered scheduling [34]. The approach used for the implementation is the same used in the Linux kernel for the fixed-priority scheduler: a *distributed run-queue* is used, meaning that each CPU maintains a private data structure implementing its own ready queue and, if global or clustered scheduling is to be achieved, tasks are migrated among processors when needed. In more details:

- the SCHED_DEADLINE tasks of each CPU are kept into a CPU-specific run-queue, implemented as a *red-black tree*

shared among all the cores, and the cache hierarchy is on 3 levels, private per-core 64 KB L1D and 512 KB L2 caches, and a global 10240 KB L3 cache.

The hardware platform runs the Linux OS modified with the patch described in Section 4.

5.2. Task Structure

In order to compare the performance of the different scheduling strategies, it is important to precisely control the parameters of each experiment. To this end, a configurable program was developed to simulate the behaviour of a periodic real-time task. An experiment consists in running many concurrent instances of this program with different parameters.

The program parameters are the period, the expected execution time, the Working Set Size (WSS)[4], the CPU affinity[5] and the scheduler type and parameters (computation-time and period). In each periodic instance, the task accesses an array of 64 Bytes elements — i.e., equal to the size of a cache line (at all levels). Following the methodology in [22], the read-to-write ratio has been set equal to 4. The elements in the array are allocated contiguously and they have been accessed both sequentially and randomly, i.e. in the worst possible case with respect to data locality. In the following, we show results only for the sequential access case, however the ones for the random access case are similar in trends, but they exhibit higher numbers of cache misses. The number of elements in the array depends on the WSS of the specific round of experiments. In order to achieve actual execution times for the jobs that stay below the WCET figures (as computed by the task set generator described below), the time to access the elements of the array has been benchmarked. To impose worst-case conditions, this benchmarking phase was done with a number of tasks from 2 to 20 concurrently executing on the same CPU under `SCHED_FIFO` real-time scheduling, all of them accessing their entire WSS. In fact, each time a job is interrupted, it may resume on a different processor (due to migration); or it may resume after a long interval of time. In these cases, especially when the cumulative WSS of all the tasks exceeds the level-X cache size, it is likely that its data is not present in the cache and must be reloaded from the next level in the memory hierarchy.

5.3. Task Set Generation

The algorithm for generating task sets used in this paper works as follows. We generate a number of tasks $N = x \cdot m$, where m is the number of processors, and x is set equal to 2, 3 and 4. The overall utilisation U of the task set is set equal to $U = R \cdot m$ where R is 0.6, 0.7 and 0.8. To generate the individual utilisation of each task, the `randfixedsum` algorithm [35] has been used, by means of the implementation publicly made available by Paul Emberson[6]. The algorithm generates N randomly distributed numbers in $(0, 1)$, whose sum is equal to the chosen U. Then, the periods are randomly generated according to a log-uniform distribution in $[10ms, 100ms]$. The

[4] Our program activates periodically and accesses a fixed set of memory locations, which constitute the Working Set.

[5] This is the set of CPUs over which the task is allowed to run.

[6] More information is available at: `http://retis.sssup.it/waters2010/tools.php`.

(worst-case) execution times are set equal to the task utilisation multiplied by the task period. The WSS size w has been set equal to 16KB and 256KB.

The original `randfixedsum` algorithm has been used in such a way that the generated tasks resulted in a computation time higher than $C_{min} = 50\mu s$, per-task utilisation lower than U (see above) and whole task set utilisation of U itself. In fact, in the partitioned cases, we aimed at task sets able to be partitioned in a reasonably balanced way (e.g., avoiding too heavy tasks which would have forced some cores to be much more loaded than others). This was done by using the unmodified algorithm for generating tasks with a whole utilisation of m, then rescaling the resulting tasks to a whole utilisation of $U - C_{min}/T_{min}$ (with $T_{min} = 10ms$ being the minimum possible period), and in the end summing up C_{min} to all the obtained computation times.

For all the possible combinations of parameters x, R and w specified above, 3 task sets have been generated. For each task set and considered scheduling and allocation policy, an experiment of 60 seconds has been run.

5.4. Scheduling and Allocation

We compared two different classes of scheduling algorithms: fixed priority scheduling, with priorities assigned according to Rate Monotonic (RM, i.e. inversely proportional to the tasks' periods); Earliest Deadline First (EDF), where the priority of a task's instance is inversely proportional to its absolute deadline. In the used `SCHED_DEADLINE` scheduler, thanks to the CBS algorithm [33], the absolute deadline of each task is automatically updated by the kernel at every re-activation of the task, and set forward in time of one task period.

We also compared 3 different scheduling solutions: partitioned, clustered and global scheduling.

The reason for considering a clustered allocation is related to the underlying hardware architecture, in which different cores have non uniform access times to the system memory hierarchy. For instance, if cores that share some level of cache are placed in the same cluster, tasks running on them will likely find their working-set warmer than when migrated on some other completely unrelated core. Moreover, the interconnections between the various CPUs and the main memory banks of the system might entail different access latencies from a specific core to a specific set of RAM addresses.

Therefore, on the AMD^R 6168 eight clusters are considered, each containing the cores that are physically placed on the same node, i.e., that share their L3 cache. This configuration is instantiated on Linux by means of `cpusets`, a feature that makes it possible to create groups of processors and to confine tasks that belong to the set to only migrate among those CPUs (i.e., the *scheduling domain* with Linux terminology). Also, in the case of partitioned and clustered allocations, the memory allocated by tasks is *pinned* to the memory banks associated with the corresponding socket by using the same mechanism. Finally, partitioned scheduling is achieved by setting the *CPU affinity* of the tasks.

When constraining the migration capability of the tasks, i.e., in clustering and partitioning, tasks are assigned to cores so as to minimise the maximum total load on each core or group of cores. The optimum configuration has been computed off-line by solving an *integer linear programming* optimisation problem using the GNU Linear

Programming Toolkit (GLPK)[7]. This results in a load which is spread as evenly as possible across the cores. However, for the partitioned scheduling cases, due to the high number of tasks, it was not possible to solve such problem optimally, but rather a maximum solving time of 960 seconds has been used, and the best solution found in that time was used.

For each allocation algorithm, both EDF and RM scheduling strategies have been considered, for a total of 6 different configurations (see Table 1 for easy reference).

algo/type	**part.**	**clust.**	**glob.**
EDF	P-EDF	C-EDF	G-EDF
RM	P-RM	C-RM	G-RM

Table 1: Algoritms vs. scheduling solutions: possible configurations.

5.5. Performance and Overhead Evaluation

The metrics against which the listed scheduling solutions are evaluated in this work are the following (further summarized in Table 2):

- **task migration rate**, defined as the total number of migrations occurred during a test divided by the number of tasks, provides an estimation of how frequently a task migrated;
- **task context switch rate**, defined as the total number of context switches occurred among all tasks during a test, divided by the number of tasks; provides an estimate of how frequently a task was preempted;
- **normalised laxity**, an estimation of how big the laxity of the various tasks is, compared to the task period. For each job of each task, this is given by the actual laxity (relative deadline minus response-time) divided by the task period. The average of all the per-job values gives the normalised laxity of the task, while the average among the laxity of all the tasks in a task-set provides the normalised laxity for the task set;
- **measured utilization**, an estimation of the utilization as experimented by running each task set. For each job of each task, this is given by the actual execution time divided by the task period. The average of all the per-job values gives the utilization of the task, while the sum of the averages provides the measured utilization for the task set;
- **L1 miss rate** and **L2 miss rate**, defined as the total number of (L1 resp. L2) cache misses experienced by a job divided by the overall execution time of the job; similarly to the other metrics, per-task and per-set figures are obtained as well.

[7]More information is available at: `http://www.gnu.org/software/glpk/`.

The migration and context switch rates provide insightful information about the overhead imposed by a scheduling algorithm on the system. The laxity is representative of how precisely the actual behaviour of the tasks adheres with the one that could have been expected, since it provides direct information about the exhibited response-time of the tasks. We normalise the laxity to the period in order to put it in relationship with the timing periodicity (and deadline) of the task.

Metric	Abbreviation	Definition
task migration rate	Migr. Rate	$\frac{migrations\ during\ a\ test}{number\ of\ tasks}$
task context switch rate	Cont. Switch Rate	$\frac{context\ switches\ during\ a\ test}{number\ of\ tasks}$
normalized laxity	Norm. Laxity	$\frac{relative\ deadline - resp.\ time}{task\ period}$
measured utilization	Exp. U	$\frac{experimented\ exec.\ time}{task\ period}$
L1 & L2 miss rate	L1D(L2) Miss Rate	$\frac{L1(L2)\ cache\ misses\ per\ job}{job\ exec.\ time}$

Table 2: Evaluation metrics.

In addition to this, the duration (in clock cycles) of the main scheduling and migration related functions is recorded, in order to compare the complexity of these operations for the two algorithms in the various situations. These functions are the following:

- **enqueue_task**, which inserts a task that just activated (e.g., wake-up) in the ready queue of a CPU;
- **dequeue_task**, which removes a task that just deactivated (e.g. sleep) from the ready queue of a CPU;
- **pull_tasks**, which picks ready but not running tasks from remote runqueues if they could run on the calling CPU;
- **push_tasks**, which tries to send ready tasks that are not running on the calling CPU to some other CPU where they would execute.

The performance metrics are gathered by using standard time and timer-related of the *GNU C Library*[8], by the *Hardware Performance Counters* and related tools and libraries[9] Functions durations in cycles are obtained by directly instrumenting the functions inside the kernel.

[8]More information is available at: `http://www.gnu.org/software/libc`.

[9]More information at: `http://en.wikipedia.org/wiki/Hardware_performance_counter`, `http://icl.cs.utk.edu/papi/index.html` and `https://perf.wiki.kernel.org`.

6. Experimental Results

Running all the tests took several days, and yielded to an extensive set of experimental data. In this section, an excerpt of such data is reported. The full obtained data set (4.3GB in compressed form) is available for download from: `http://retis.sssup.it/people/jlelli/papers/JSS2012`. Statistics come from the results of running 3 different randomly generated task sets for each configuration in terms of scheduler, allocation policy, number of tasks and their WSS.

For example, one of the 3 task sets generated in the case of U=0.6 and 96 tasks is reported in Figure 1. Each point corresponds to a task with period and utilisation as read from the X and Y coordinates of the point. The higher density of points for lower periods is due to the logarithmic generation of the random periods.

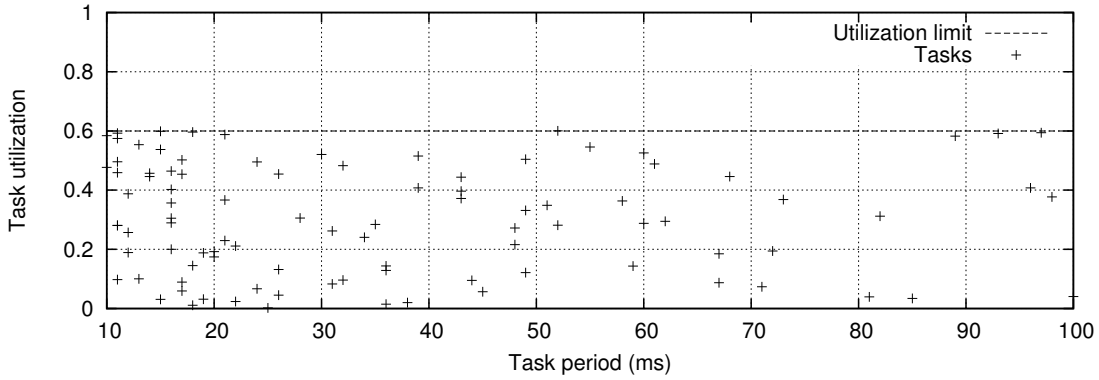


Figure 1: One of the used task-sets with U=0.6 and 96 tasks.

6.1. Running Each Task Alone

U	L1D	L2	L1D	L2
	WSS 16KB		WSS 256KB	
0.6	0.086	0.138	4.238	0.145
0.7	0.082	0.126	4.244	0.131
0.8	0.081	0.120	4.233	0.124

Table 3: Cache behaviour with tasks from the task-sets executed in isolation.

First, each task of each set has been executed in isolation at real-time priority (the exact real-time scheduling policy has no impact) in order to get its "reference" behaviour in terms of execution time and minimum cache misses. The obtained cache miss figures, averaged for all of the task sets, are reported in Table 3. The experimental variation of all the parameters is practically null due to the execution in isolation of each individual task, thus it has not been shown. As it can be seen, the cache-miss rates decrease when increasing the utilisation. In fact, longer jobs exhibit a better chance to access hot-cache data in their activations. The trends are similar for both cases of 16KB and 256KB WSS values.

6.2. *Impact of Scheduling*

Now for each configuration all the tasks in the task-set have been deployed concurrently on the platform. In the following, the focus is on the configurations with a WSS of 16KB. In all the shown results, the performance of EDF based and RM based policies are numerically and visually compared.

Table 4 reports the obtained normalised laxity, measured U, context switch rate, and migration rate for the various configurations: global (top table), clustered (middle table) and partitioned (bottom table) scheduling, both with EDF and RM policies. Each row corresponds to a different configuration with a total utilisation and number of tasks as indicated in the first two columns. The reported numbers constitute the average and standard deviation of the obtained metrics of interest averaged across all the tasks and the 3 task sets corresponding to each configuration.

U	#	Norm. Laxity		Exp. U		Cont. Switch Rate		Migr. Rate	
		G-EDF	G-RM	G-EDF	G-RM	G-EDF	G-RM	G-EDF	G-RM
	96	0.637 ± 0.00	0.637 ± 0.00	0.714 ± 0.00	0.714 ± 0.00	7177.260 ± 204.66	7116.566 ± 320.55	1859.906 ± 43.87	1558.247 ± 62.43
	144	0.742 ± 0.00	0.741 ± 0.01	0.755 ± 0.01	0.755 ± 0.01	6952.722 ± 351.99	6989.104 ± 205.90	2296.683 ± 182.73	1981.012 ± 225.60
0.6	192	0.790 ± 0.01	0.786 ± 0.01	0.806 ± 0.01	0.809 ± 0.01	7056.780 ± 297.99	7268.071 ± 297.61	3129.241 ± 132.60	2825.498 ± 289.64
		G-EDF	G-RM	G-EDF	G-RM	G-EDF	G-RM	G-EDF	G-RM
	96	0.586 ± 0.01	0.585 ± 0.01	0.813 ± 0.00	0.814 ± 0.00	6942.156 ± 207.28	7099.365 ± 220.57	2062.438 ± 136.11	1988.503 ± 136.62
	144	0.703 ± 0.01	0.700 ± 0.01	0.856 ± 0.00	0.859 ± 0.00	6956.086 ± 82.72	7159.338 ± 97.04	3158.907 ± 83.59	2865.542 ± 154.45
0.7	192	0.756 ± 0.01	0.560 ± 0.12	0.903 ± 0.01	0.909 ± 0.01	6689.106 ± 103.68	7112.031 ± 205.73	4365.604 ± 604.26	4201.139 ± 384.19
		G-EDF	G-RM	G-EDF	G-RM	G-EDF	G-RM	G-EDF	G-RM
	96	0.408 ± 0.08	0.057 ± 0.27	0.919 ± 0.00	0.921 ± 0.00	7212.948 ± 78.86	7549.163 ± 79.46	3329.632 ± 37.64	3471.920 ± 42.39
	144	0.246 ± 0.25	-1.277 ± 1.09	0.960 ± 0.01	0.964 ± 0.01	6447.051 ± 376.24	7209.424 ± 571.82	5261.287 ± 993.66	4944.606 ± 792.15
0.8	192	-17.268 ± 9.15	-10.330 ± 6.02	1.014 ± 0.01	1.016 ± 0.01	5879.139 ± 226.55	7146.064 ± 206.88	8088.148 ± 671.48	6886.774 ± 506.12

U	#	Norm. Laxity		Exp. U		Cont. Switch Rate		Migr. Rate	
		C-EDF	C-RM	C-EDF	C-RM	C-EDF	C-RM	C-EDF	C-RM
	96	0.658 ± 0.01	0.659 ± 0.01	0.656 ± 0.00	0.656 ± 0.00	6772.528 ± 131.22	6838.740 ± 194.84	1612.500 ± 57.60	1499.889 ± 61.64
	144	0.752 ± 0.02	0.749 ± 0.02	0.695 ± 0.01	0.695 ± 0.01	6500.192 ± 356.20	6698.641 ± 374.65	2038.775 ± 108.98	2029.683 ± 218.87
0.6	192	0.791 ± 0.02	0.787 ± 0.02	0.744 ± 0.01	0.746 ± 0.01	6550.771 ± 339.06	6979.425 ± 331.32	2600.418 ± 242.83	2571.901 ± 275.62
		C-EDF	C-RM	C-EDF	C-RM	C-EDF	C-RM	C-EDF	C-RM
	96	0.604 ± 0.02	0.603 ± 0.02	0.750 ± 0.00	0.749 ± 0.00	6478.545 ± 254.34	6684.986 ± 213.21	1844.708 ± 129.20	1860.281 ± 116.97
	144	0.706 ± 0.03	0.705 ± 0.02	0.792 ± 0.00	0.793 ± 0.00	6489.877 ± 302.18	6771.567 ± 248.40	2486.000 ± 98.06	2426.850 ± 86.98
0.7	192	0.750 ± 0.03	0.745 ± 0.03	0.838 ± 0.01	0.838 ± 0.01	6247.576 ± 86.31	6957.189 ± 127.15	2977.594 ± 160.47	3106.653 ± 148.02
		C-EDF	C-RM	C-EDF	C-RM	C-EDF	C-RM	C-EDF	C-RM
	96	0.427 ± 0.09	0.367 ± 0.11	0.850 ± 0.00	0.849 ± 0.00	6472.833 ± 156.42	7016.882 ± 119.31	2259.243 ± 40.09	2475.653 ± 14.71
	144	0.642 ± 0.05	0.220 ± 0.27	0.889 ± 0.01	0.890 ± 0.01	6045.720 ± 473.54	6751.257 ± 585.09	2995.255 ± 330.74	3135.801 ± 364.48
0.8	192	-0.017 ± 0.45	-0.470 ± 0.69	0.935 ± 0.01	0.935 ± 0.01	5834.167 ± 187.69	6868.012 ± 336.15	3766.528 ± 319.98	4011.521 ± 371.34

U	#	Norm. Laxity		Exp. U		Cont. Switch Rate		Migr. Rate	
		P-EDF	P-RM	P-EDF	P-RM	P-EDF	P-RM	P-EDF	P-RM
	96	0.599 ± 0.04	0.598 ± 0.04	0.656 ± 0.00	0.656 ± 0.00	5895.493 ± 238.37	6246.983 ± 259.01	0.000 ± 0.00	0.000 ± 0.00
	144	0.686 ± 0.05	0.684 ± 0.05	0.692 ± 0.01	0.693 ± 0.01	5575.343 ± 276.93	6006.002 ± 302.92	0.000 ± 0.00	0.000 ± 0.00
0.6	192	0.709 ± 0.06	0.707 ± 0.07	0.741 ± 0.01	0.741 ± 0.01	5560.694 ± 216.06	6132.670 ± 232.04	0.000 ± 0.00	0.000 ± 0.00
		P-EDF	P-RM	P-EDF	P-RM	P-EDF	P-RM	P-EDF	P-RM
	96	0.526 ± 0.05	0.525 ± 0.04	0.748 ± 0.00	0.748 ± 0.00	5464.538 ± 254.98	5882.622 ± 226.24	0.000 ± 0.00	0.000 ± 0.00
	144	0.617 ± 0.07	0.613 ± 0.06	0.789 ± 0.00	0.789 ± 0.00	5373.111 ± 154.76	5943.572 ± 42.84	0.000 ± 0.00	0.000 ± 0.00
0.7	192	0.633 ± 0.08	0.476 ± 0.21	0.834 ± 0.01	0.834 ± 0.00	5319.613 ± 122.86	6023.090 ± 219.59	0.000 ± 0.00	0.000 ± 0.00
		P-EDF	P-RM	P-EDF	P-RM	P-EDF	P-RM	P-EDF	P-RM
	96	0.226 ± 0.19	-0.254 ± 0.42	0.847 ± 0.00	0.849 ± 0.00	5519.233 ± 167.03	6252.608 ± 67.19	0.000 ± 0.00	0.000 ± 0.00
	144	0.537 ± 0.08	0.530 ± 0.07	0.885 ± 0.01	0.887 ± 0.01	5318.801 ± 417.72	5860.312 ± 431.40	0.000 ± 0.00	0.000 ± 0.00
0.8	192	-15.216 ± 8.93	-12.471 ± 6.76	0.929 ± 0.01	0.908 ± 0.03	4791.134 ± 184.04	5837.344 ± 170.43	0.000 ± 0.00	0.000 ± 0.00

Table 4: Statistics for the metrics of interest when WSS=16KB, under various configurations: global (top table), clustered (middle table) and partitioned (bottom table) scheduling, both with EDF and RM policies.

First, we focus on how the computation times changed in the various cases, then the performance of the various

scheduling algorithms from the application viewpoint.

Comparing the same configurations under EDF and RM, the obtained context switches under EDF are lower than under RM, with a tendency to have slightly lower average context switches at high overall utilisation (0.8). On the other hand, the migration rates of EDF based policies are significantly higher than the ones of RM based ones. However, the actual impact on the execution times of the tasks is quite limited, as it can be observed by the experimental U measured (and averaged over the 3 task sets available for each configuration) for EDF and RM based policies, which are basically equivalent.

Comparing different configurations, the context switch rate decreases with the load while, on the contrary, the migration rate increases (but decreases with the WSS, not shown). Also, note that the average experimental utilisation increases when switching from partitioned to clustered and then to global policies, as expected due to the presence of migrations. This is also confirmed when looking at the average execution times obtained for the individual tasks. For example, Figure 2 shows the ratio among the execution times obtained with global, clustered and partitioned EDF scheduling, for the task set with $U = 0.6$ and 96 tasks shown also in Figure 1, in the two cases of 16KB and 256KB WSS. As it can be seen, in the cases of small WSS, clustered policies succeed in keeping the workload inflation at contained and negligible amounts, as compared to global strategies. However, as the WSS increases, the differences among the cache-related overheads of clustered and global scheduling tend to vanish. Working very close to the L2 cache boundary, it is very likely that each task has to reload a large part of its WSS when it is released, independently from the processor it starts running on.

Now, let us observe how the various scheduler configurations impact the performance as observable from within the applications themselves, i.e., let us focus on the normalised laxity figures. The average figures obtained under EDF and RM based policies are basically equivalent when the system load is low. This is due to the fact that, when a task is favoured over another by the scheduler, then the former will finish earlier and the latter will finish later, but when averaging figures this difference is lost. The situation is different reaching the top edge with respect to utilisation (i.e., $U = 0.8$) and number of tasks. Both EDF and RM policies exhibit negative average laxity (tardiness) with 192 tasks, but only RM policies see this behaviour with also 92 and 144 tasks. It is furthermore evident that clustered policies come out to be better from this viewpoint, as slight negative values appear with 192 tasks only; moreover, C-EDF outperforms C-RM with higher average laxity figures.

In addition, let us look at the whole set of obtained laxity values, under various scheduling policies, looking at their cumulative distribution functions (CDF), first for the whole task set, then focusing on a few specific ones. Figure 3 shows the obtained curves when the laxity of all the tasks is considered. It is evident that (top sub-figure) global and clustered strategies (both for EDF and RM, first and second curves) differ in a negligible way from each other, and that they tend to lead to generally higher laxity values (better performance) than partitioned ones (last two curves). Also, no difference can be appreciated between EDF and RM from this viewpoint. However, zooming the figure in the critical area near a null laxity (when the tasks are close to miss their deadlines), the differences between the three policy types are more evident, with still global policies better than the others. However, partitioned EDF from this

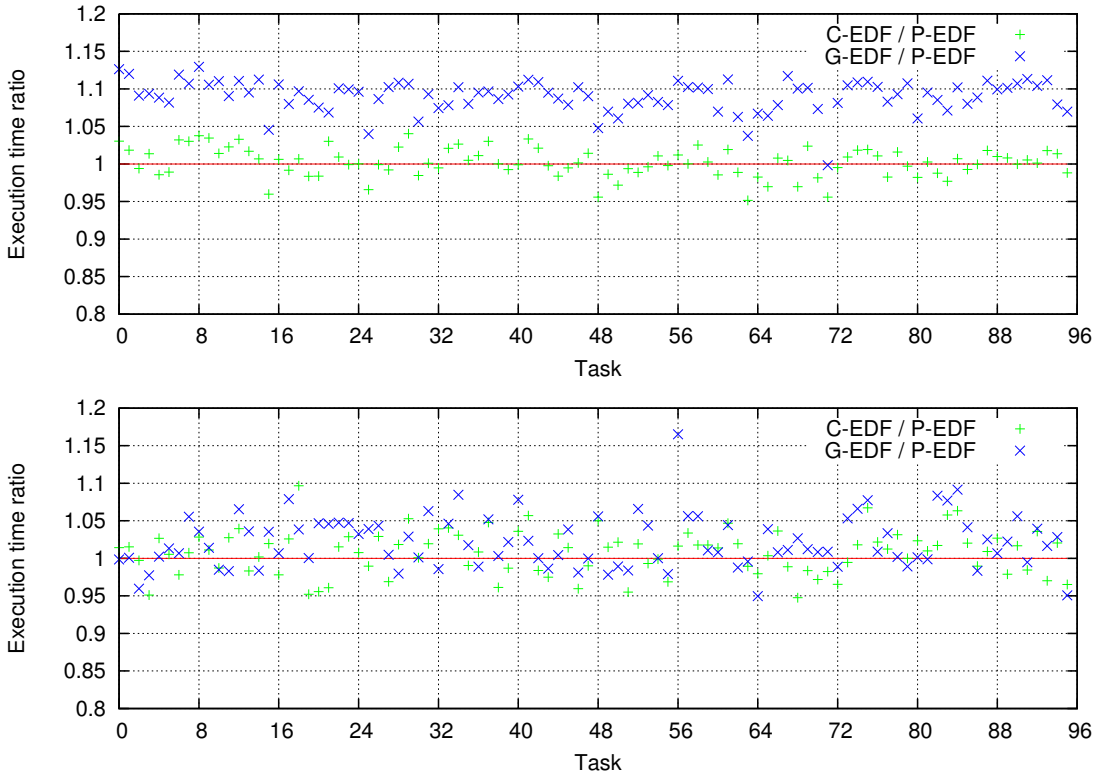
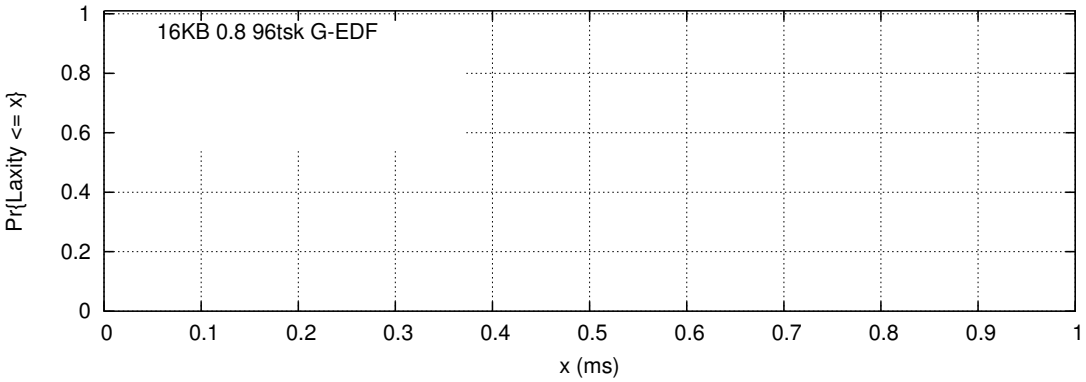


Figure 2: Execution times obtained for the various tasks (averaged over the task activations) under C-EDF (plus signs) and G-EDF (multiply signs) relative to the figures obtained under P-EDF, in the cases of 16KB (top) and 256KB (bottom) WSS.

viewpoint performs much better than partitioned RM, and it achieves a similar performance to both clustered policies. This difference is due to the particular task set and the way it has been partitioned (nearly optimally – see Section 5.4) across CPUs. Also, it can be observed that there is a certain amount of jobs missing their deadlines. However, these are below 1% of the total jobs that have been executed, with the exception of P-RM that stands slightly above this value.

Now let us focus on the tasks with minimum period (there were 2 such tasks in the experiment with a $10ms$ period), which have the most critical timing requirements, and the ones with maximum periods (there was 1 single task with a $98ms$ period), which most often are preempted by lower period tasks. Figure 4 shows the obtained CDF curves when only considering the normalised laxity of these two subsets of tasks. It can be seen that global and clustered policies perform better than the partitioned ones, with RM-based policies providing an exceptionally good performance to the minimum-period tasks (Figure 4 top) at the expense of the maximum-period ones (Figure 4 bottom). These are served very badly by G-RM, while G-EDF achieves a borderline acceptable performance with nearly 18% of deadline misses. However, the reduced overheads of clustered policies (see also Section 6.4) show up clearly in this case, where both C-EDF and C-RM manage to perform quite well.


16KB 0.8 96tsk G-EDF
Pr{Laxity <= x}
x (ms)
0
0.2
0.4
0.6
0.8
1
0.1
0.3
0.5
0.7
0.9

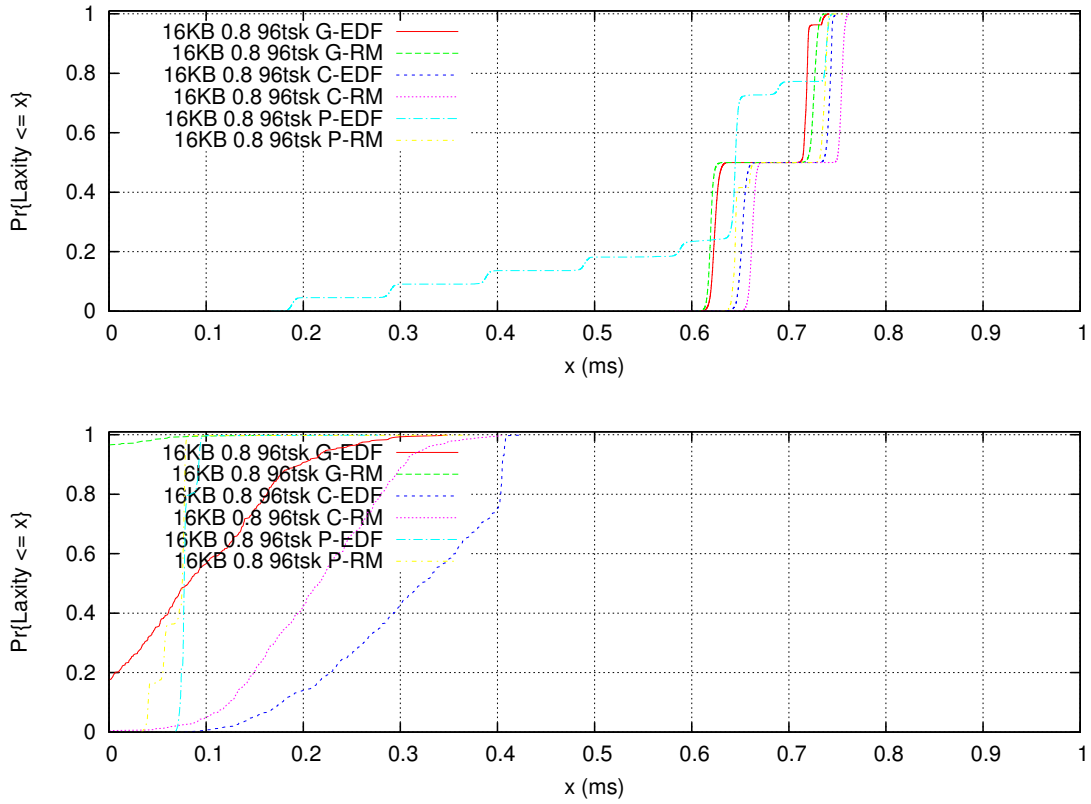


Figure 4: CDF of the normalised laxity for the minimum-period (top) and maximum-period (bottom) tasks, under various scheduling policies, with U=0.8 and 96 tasks.

From the 16KB WSS tables, it might seem that clustering policies lead to higher cache miss rates than global scheduling, which is counter-intuitive. However, the differences are really small and sometimes within the variability range attested by the measured standard deviation. Also, for the 256KB WSS cases, partitioned policies seem to experience fewer cache misses than the others, but this is true for L1D cache miss rates only. Since the WSS exceeds the L1D border, little can be said about the L1D behaviour. Instead, partitioning policies lead to the lowest L2 cache miss rates as expected.

6.4. Scheduling Overheads Comparison

In this paragraph, the scheduling overheads obtained under various policies are compared, using the metrics described in Section 5.

In Table 6, the overheads obtained in various configurations with a WSS of 16KB are reported. All numbers are expressed in clock cycles.

EDF based policies consistently present lower number of cycles than RM ones for enqueue operations, while, on the contrary, this is true for the partitioned case only for dequeue operations. The difference in performance is due to the completely different data structures used in the two cases. Concerning the *pull* operations, in the global

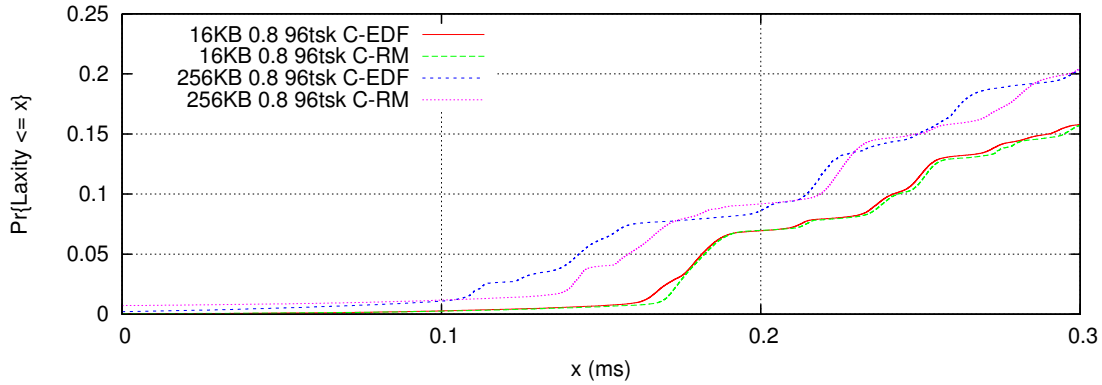


Figure 5: CDF of the obtained normalised laxity for clustered EDF and RM policies with WSS of 16KB and 256KB, with U=0.8 and 96 tasks.

and clustered scheduling cases the overheads of EDF are about twice or three times the ones of RM, whilst in the partitioned cases they are both null. Note that the small numbers shown as overheads of *pull* operations in the partitioned case are due to the few migrations at the beginning of the experiment needed to deploy the tasks over the assigned cores. Looking at the *push* operations, the overheads of G-EDF and G-RM are comparable, while C-EDF exhibits less overhead numbers than C-RM. These kind of results were expected, since in [34] only *push* operations were addressed, making them efficient.

7. Conclusions

In this paper, an experimental comparison of various multi-processor scheduling algorithms has been performed by running synthetic workloads of real tasks on a Linux system. The performance of the various solutions has been evaluated under diverse metrics and under multiple combinations of CPU utilisation and number of tasks.

The experimental results lead to some interesting considerations. It appears clear that global and clustered algorithms are a viable solution for multi-core platforms with a high degree of cache sharing, mainly because the overhead caused by migrations is not more costly than a “traditional” context switch. Partitioned scheduling requires a potentially costly operation of allocation of tasks to cores, and it may not manage to make use of the whole computing power available on a system. On the other hand, global scheduling has the ability to perform automatic load balancing dynamically at run-time, achieving better normalised laxity figures. However, with a large number of cores, true global scheduling may lead to a growth in overheads that needs to be kept under control, whilst clustered scheduling approaches may perform better thanks to their reduced overheads, especially when the tasks have been properly/optimally partitioned across the clusters. Indeed, at reduced sizes of the tasks WSS, clustered algorithms manage to keep applications data in the cache shared among the cores in a cluster (if present), thus achieving very little migration costs, as compared to what happens with global strategies. However, the difference tends to vanish when the WSS grows.

U	Algo.	L1D Miss Rate	L2 Miss Rate	L1D Miss Rate	L2 Miss Rate
		WSS 16KB		WSS 256KB	
0.6	G-EDF	0.100 ± 0.02	0.085 ± 0.05	4.405 ± 0.04	0.163 ± 0.07
	G-RM	0.099 ± 0.03	0.091 ± 0.05	4.392 ± 0.04	0.171 ± 0.07
0.7	G-EDF	0.101 ± 0.03	0.069 ± 0.04	4.421 ± 0.05	0.154 ± 0.06
	G-RM	0.098 ± 0.03	0.072 ± 0.04	4.352 ± 0.04	0.157 ± 0.06
0.8	G-EDF	0.119 ± 0.05	0.056 ± 0.03	4.808 ± 0.07	0.169 ± 0.07
	G-RM	0.103 ± 0.03	0.050 ± 0.02	4.344 ± 0.05	0.156 ± 0.05

U	Algo.	L1D Miss Rate	L2 Miss Rate	L1D Miss Rate	L2 Miss Rate
		WSS 16KB		WSS 256KB	
0.6	C-EDF	0.107 ± 0.03	0.101 ± 0.06	4.475 ± 0.04	0.173 ± 0.07
	C-RM	0.107 ± 0.03	0.096 ± 0.05	4.467 ± 0.04	0.178 ± 0.07
0.7	C-EDF	0.104 ± 0.02	0.083 ± 0.05	4.479 ± 0.04	0.159 ± 0.06
	C-RM	0.102 ± 0.02	0.079 ± 0.05	4.472 ± 0.04	0.165 ± 0.06
0.8	C-EDF	0.108 ± 0.03	0.064 ± 0.04	4.493 ± 0.05	0.153 ± 0.06
	C-RM	0.105 ± 0.02	0.061 ± 0.03	4.464 ± 0.04	0.163 ± 0.05

U	Algo.	L1D Miss Rate	L2 Miss Rate	L1D Miss Rate	L2 Miss Rate
		WSS 16KB		WSS 256KB	
0.6	P-EDF	0.095 ± 0.02	0.060 ± 0.03	4.504 ± 0.03	0.125 ± 0.03
	P-RM	0.093 ± 0.02	0.059 ± 0.03	4.505 ± 0.03	0.127 ± 0.03
0.7	P-EDF	0.090 ± 0.01	0.048 ± 0.02	4.529 ± 0.03	0.109 ± 0.02
	P-RM	0.088 ± 0.01	0.048 ± 0.02	4.506 ± 0.03	0.116 ± 0.03
0.8	P-EDF	0.087 ± 0.01	0.038 ± 0.02	4.787 ± 0.03	0.104 ± 0.02
	P-RM	0.088 ± 0.01	0.040 ± 0.02	4.502 ± 0.03	0.119 ± 0.03

Table 5: Cache related behaviour of global (top sub-table), clustered (middle sub-table) and partitioned (bottom sub-table) EDF and RM policies for various configurations (values are averages of all the runs for each configuration) and WSS.

It is important to compare our results with the ones in [23], where the authors concluded that "G-EDF is never a suitable policy for hard RT systems". This may seem in conflict with our conclusions about the viability of global scheduling for RT systems. First, the reader will notice that the former conclusions concern hard real-time systems, where a formal assessment on the obtainable performance (and existence of deadline misses) can only be obtained via proper schedulability analysis techniques. On the other hand, our conclusions derive from observations of the actual behaviour of the tasks as scheduled on a real OS over a certain time horizon. In this regard, it is useful to observe that all of the task sets used in the experiments shown in our paper have been found as non-schedulable according to one [36] of the tests for G-EDF which is known to be among the best known [37]. Second, it was not possible to set-up an experiment exactly identical to the one(s) in [23], due to the unavailability of the whole used data set (task WCETs, periods and clusters/partitions). Third, the analysis in [23] relies on the weighted schedulability, computed over task-sets with heterogeneous overall utilisation, obtained as a weighted average of the percentages of schedulable task-sets, obtained using the utilisation itself as weight. This metric was not meaningful in our context, because we were not interested in schedulability as assessed by a conservative analysis technique, but rather in the soft real-time performance as observed from the running system. On a related note, all of the task sets we considered were not schedulable by G-EDF, as stated above. Finally, another difficulty in comparing the two approaches is due to the differences in the actual task body and in the underlying physical platform, which cause differences in the cache behaviour of the tasks when deployed on the various cores under the various configurations.

As a further remark, in both cases the conclusions cannot be considered absolutely generic, because they derive

U	#	Enqueue mean time		Dequeue mean time		Push mean time		Pull mean time	
		G-EDF	G-RM	G-EDF	G-RM	G-EDF	G-RM	G-EDF	G-RM
	96	4099 ± 104.1	4867 ± 72.3	4672 ± 370.0	4518 ± 32.6	13444 ± 952.6	13895 ± 192.8	13689 ± 3772	4905 ± 124.6
	144	4147 ± 148.4	5298 ± 377.4	5158 ± 266.6	4719 ± 114.0	12637 ± 603.0	12310 ± 1018	30775 ± 4081	9143 ± 954.0
0.6	192	4078 ± 185.1	5304 ± 248.7	5803 ± 57.2	4740 ± 135.8	11938 ± 320.9	10705 ± 913.9	54308 ± 1236	15379 ± 1063
		G-EDF	G-RM	G-EDF	G-RM	G-EDF	G-RM	G-EDF	G-RM
	96	3977 ± 297.0	4810 ± 146.2	4304 ± 22.2	4183 ± 68.3	11836 ± 832.9	11743 ± 425.0	11299 ± 1398	6635 ± 859.9
	144	3840 ± 56.2	5006 ± 356.1	5288 ± 39.3	4309 ± 56.1	9674 ± 495.2	9144 ± 460.1	35216 ± 3847	13480 ± 534.9
0.7	192	3740 ± 132.8	4985 ± 220.0	5833 ± 333.4	4239 ± 83.0	8469 ± 497.5	7610 ± 478.0	64636 ± 13444	23708 ± 2175
		G-EDF	G-RM	G-EDF	G-RM	G-EDF	G-RM	G-EDF	G-RM
	96	3516 ± 108.6	4295 ± 174.1	4338 ± 221.8	3796 ± 16.8	7056 ± 299.3	7250 ± 175.5	15072 ± 2540	13278 ± 131.7
	144	3367 ± 23.0	4391 ± 49.1	5317 ± 66.9	3896 ± 75.9	5428 ± 390.6	5917 ± 362.3	42396 ± 5308	24977 ± 2114
0.8	192	3420 ± 136.3	4531 ± 80.2	6172 ± 266.2	3965 ± 34.8	5830 ± 514.9	5872 ± 131.8	102436 ± 14741	40633 ± 2124

U	#	Enqueue mean time		Dequeue mean time		Push mean time		Pull mean time	
		C-EDF	C-RM	C-EDF	C-RM	C-EDF	C-RM	C-EDF	C-RM
	96	3857 ± 207.8	4961 ± 86.2	4531 ± 34.4	4552 ± 59.7	7388 ± 295.0	8840 ± 593.0	15726 ± 1143	11367 ± 1316
	144	3742 ± 98.7	5006 ± 268.1	4920 ± 75.0	4523 ± 53.6	6154 ± 195.5	7442 ± 88.4	33045 ± 2679	19974 ± 2637
0.6	192	3864 ± 164.6	5056 ± 178.3	5465 ± 147.2	4533 ± 97.6	5271 ± 408.8	6631 ± 160.8	71980 ± 12418	31313 ± 6240
		C-EDF	C-RM	C-EDF	C-RM	C-EDF	C-RM	C-EDF	C-RM
	96	3519 ± 125.7	4568 ± 192.2	4388 ± 108.2	4292 ± 23.1	6074 ± 683.1	7356 ± 152.9	17161 ± 2258	15319 ± 1646
	144	3592 ± 164.7	4487 ± 246.4	5102 ± 219.6	4185 ± 62.4	4837 ± 278.9	6712 ± 357.4	44885 ± 2780	24738 ± 1861
0.7	192	3658 ± 97.2	4925 ± 210.3	5576 ± 172.9	4377 ± 48.7	4022 ± 345.6	6144 ± 657.8	81507 ± 14115	40518 ± 2069
		C-EDF	C-RM	C-EDF	C-RM	C-EDF	C-RM	C-EDF	C-RM
	96	3295 ± 34.1	4326 ± 99.5	4348 ± 127.6	3874 ± 70.8	4139 ± 217.4	6108 ± 149.2	20888 ± 1987	18963 ± 105.9
	144	3310 ± 155.6	4375 ± 309.3	5001 ± 171.6	3981 ± 121.8	3066 ± 319.7	5667 ± 236.2	50546 ± 5266	33437 ± 2427
0.8	192	3384 ± 30.2	4616 ± 105.0	5563 ± 108.3	4095 ± 81.2	2382 ± 152.2	4821 ± 138.2	90440 ± 7631	52870 ± 3211

U	#	Enqueue mean time		Dequeue mean time		Push mean time		Pull mean time	
		P-EDF	P-RM	P-EDF	P-RM	P-EDF	P-RM	P-EDF	P-RM
	96	3159 ± 225.0	4347 ± 213.3	4089 ± 57.9	5309 ± 85.7	0.0 ± 0.0	0.0 ± 0.0	131.0 ± 2.5	168.6 ± 6.2
	144	2956 ± 172.8	4106 ± 174.2	4094 ± 42.9	5397 ± 142.5	0.0 ± 0.0	0.0 ± 0.0	132.1 ± 3.2	124.2 ± 34.6
0.6	192	2751 ± 145.6	3997 ± 259.5	4014 ± 82.1	5375 ± 81.9	0.0 ± 0.0	0.0 ± 0.0	130.2 ± 7.9	121.1 ± 14.4
		P-EDF	P-RM	P-EDF	P-RM	P-EDF	P-RM	P-EDF	P-RM
	96	3471 ± 530.6	4025 ± 375.5	3990 ± 25.8	5251 ± 159.3	0.0 ± 0.0	0.0 ± 0.0	127.5 ± 4.5	151.1 ± 12.0
	144	2541 ± 171.3	3898 ± 516.0	3962 ± 157.9	5372 ± 128.4	0.0 ± 0.0	0.0 ± 0.0	126.2 ± 9.6	124.8 ± 1.2
0.7	192	2363 ± 121.5	3375 ± 152.5	3879 ± 37.9	5352 ± 106.1	0.0 ± 0.0	0.0 ± 0.0	119.0 ± 3.7	103.6 ± 10.2
		P-EDF	P-RM	P-EDF	P-RM	P-EDF	P-RM	P-EDF	P-RM
	96	2754 ± 143.4	3884 ± 140.8	3847 ± 84.3	5087 ± 63.3	0.0 ± 0.0	0.0 ± 0.0	126.9 ± 5.2	96.8 ± 8.4
	144	2530 ± 29.6	3471 ± 143.4	3886 ± 44.9	5231 ± 115.4	0.0 ± 0.0	0.0 ± 0.0	115.9 ± 2.0	91.4 ± 11.1
0.8	192	2156 ± 112.6	3796 ± 536.2	3598 ± 43.5	4995 ± 88.0	0.0 ± 0.0	0.0 ± 0.0	109.1 ± 0.9	82.5 ± 2.7

Table 6: Scheduling and migration related function durations (on average, in clock cycles) for global (top sub-table), clustered (middle sub-table) and partitioned (bottom sub-table) `EDF` and `RM` policies, in the case of WSS=16KB.

from the use of a limited number of randomly generated task sets.

In view of complete transparency, we have to state it clear that it was not possible (for practical reasons) to run the task sets till the hyper-period (even though the experiments duration was tuned so as to allow for a number of activations of each task between 600 and 6000, depending on their periods). Therefore, the reported measurements are referred to a limited observation horizon over each task-set schedule, and cannot be considered to be completely exhaustive of each and every possible foreseeable behaviour under the considered scheduling algorithms. Still, we believe the results reported in this paper are very useful for soft real-time systems, where the main focus in on how applications may behave most of the times, and the actual worst-case that may show up once in a while may merely lead to temporary degradations of the provided service that can be easily tolerated by the system (and final users).

The implementation of G-EDF used in this paper presents acceptable overhead figures in terms of durations of the

scheduling functions, being comparable with the corresponding ones present in the standard fixed priority scheduler of the Linux kernel. The performed evaluation highlights that such overheads grow anyway with the number of cores over which one is globally scheduling. *Clustered* policies lead to most of the benefits of global ones, and they can actually perform better thanks to the reduced overhead figures, posing the foundations for scalability of the technique to a high number of cores. The cost to pay is again the need for allocating tasks to clusters. However, such problem comes in a much more reduced form than the one of allocating to cores.

8. Future Work

There are various parameters that have not been considered during our practical evaluation, yet.

In clustered and partitioning strategies, we applied an off-line partitioning algorithm based on Linear Programming for partitioning tasks among cores (or clusters). However, in an open system, such an off-line optimisation phase might not be feasible, and one might want to keep in consideration a far simpler and quicker heuristic for this activity (e.g., first-fit, worst-fit). We would expect clustered scheduling to perform probably worse than shown in this paper, under such conditions, however this needs to be investigated experimentally.

While creating dynamically real-time tasks, it might be interesting to investigate on whether it may be worth or not to change dynamically the scheduler configuration depending on the particular workload conditions, e.g., to adapt how tasks are partitioned across the available cores.

Furthermore, on big multi-core machines under dynamically varying real-time workloads, it may be interesting to enrich the investigation with energy-awareness. For example, energy-aware schedulers (such as [38]) may be added to the schedulers under comparison, and the energy consumption may be added to the evaluation metrics. Also, the possibility for the various processors/cores to run at different frequencies may bring into the problem issues typical of global scheduling on heterogeneous systems.

Finally, it may be worth to perform some experiments considering a realistic case-study, for example audio/video trans-coding or others. The particular pattern of access to the memory of a workload composed of real applications, along with the existence of other non real-time tasks running on the OS as it may be required in a real application set-up, would surely impact the gathered results concerning the cache-related behaviour of the considered schedulers. On big multi-core systems, particularly interesting workloads may be those ones consisting of DAGs of computations with end-to-end deadlines. Therefore, interesting lines of expansion of this work may comprise application workloads of such kind.

References

[1] E. von Weizsaecker, K. Hargroves, M. Smith, C. Desha, P. Stasinopoulos, Factor Five – Transforming the Global Economy through 80% Improvements in Resource Productivity, Earthscan, 2009.

[2] P. Valente, G. Lipari, An upper bound to the lateness of soft real-time tasks scheduled by EDF on multiprocessors, in: RTSS, IEEE Computer Society, 2005, pp. 311–320.

[3] U. C. Devi, J. H. Anderson, Improved conditions for bounded tardiness under EPDF Pfair multiprocessor scheduling, J. Comput. Syst. Sci. 75 (2009) 388–420.

[4] U. Devi, J. Anderson, Tardiness bounds under global EDF scheduling on a multiprocessor, Real-Time Systems 38 (2008) 133–189. 10.1007/s11241-007-9042-1.

[5] T. P. Baker, A comparison of global and partitioned EDF schedulability tests for multiprocessors, in: Proceeding of the International Conference on Real-Time and Network Systems, Poitiers, France.

[6] M. Bertogna, S. Baruah, Tests for global EDF schedulability analysis, Journal of Systems Architecture 57 (2011) 487 – 497. Special Issue on Multiprocessor Real-time Scheduling.

[7] A. Masrur, S. Chakraborty, G. Faerber, Constant-time admission control for partitioned EDF, in: Real-Time Systems (ECRTS), 2010 22nd Euromicro Conference on, pp. 34 –43.

[8] G. Buttazzo, Rate Monotonic vs. EDF: Judgment Day, Real-Time Systems 29 (2005).

[9] D. Hardy, I. Puaut, Estimation of cache related migration delays for multi-core processors with shared instruction caches, in: Proc. of the 17^{th} International Conference on Real-Time and Network Systems (RTNS 2009), Paris, France, pp. 45–54.

[10] D. Chandra, F. Guo, S. Kim, Y. Solihin, Predicting inter-thread cache contention on a chip multi-processor architecture, in: High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on, pp. 340 – 351.

[11] J. Yan, W. Zhang, WCET analysis for multi-core processors with shared l2 instruction caches, in: Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium, IEEE Computer Society, Washington, DC, USA, 2008, pp. 80–89.

[12] J. C. Mogul, A. Borg, The effect of context switches on cache performance, ACM SIGPLAN Notices 26 (1991) 75–84.

[13] J. Starner, L. Asplund, Measuring the cache interference cost in preemptive real-time systems, ACM SIGPLAN Notices 39 (2004) 146–154.

[14] F. M. David, J. C. Carlyle, R. H. Campbell, Context switch overheads for linux on arm platforms, in: Proc. of the 2007 Workshop on Experimental Computer Science, San Diego, USA.

[15] C. Li, C. Ding, K. Shen, Quantifying the cost of context switch, in: Proceedings of the 2007 workshop on Experimental computer science, ExpCS '07, ACM, New York, NY, USA, 2007.

[16] D. Tsafrir, The context-switch overhead inflicted by hardware interrupts, in: Proc. of the 2007 Workshop on Experimental Computer Science, San Diego, USA.

[17] X. Tang, K. Li, G. Liao, K. Fang, F. Wu, A stochastic scheduling algorithm for precedence constrained tasks on grid, Future Generation Computer Systems 27 (2011) 1083 – 1091.

[18] G. L. Stavrinides, H. D. Karatza, Scheduling multiple task graphs in heterogeneous distributed real-time systems by exploiting schedule holes with bin packing techniques, Simulation Modelling Practice and Theory 19 (2011) 540 – 552. Modeling and Performance Analysis of Networking and Collaborative Systems.

[19] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, J. H. Anderson, LITMUS-RT: A testbed for empirically comparing real-time multiprocessor schedulers, in: Proceedings of the 27th IEEE International Real-Time Systems Symposium, IEEE Computer Society, Washington, DC, USA, 2006, pp. 111–126.

[20] B. Brandenburg, J. Calandrino, J. Anderson, On the scalability of real-time scheduling algorithms on multicore platforms: A case study, in: Real-Time Systems Symposium, 2008, pp. 157–169.

[21] B. B. Brandenburg, J. H. Anderson, On the implementation of global real-time schedulers, in: Proceedings of the 30th IEEE Real-Time Systems Symposium, RTSS '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 214–224.

[22] A. Bastoni, B. B. Brandenburg, J. H. Anderson, Cache-related preemption and migration delays: Empirical approximation and impact on schedulability, in: Proc. 6^{th} International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2010), Brussels, Belgium, pp. 33–44.

[23] A. Bastoni, B. Brandenburg, J. Anderson, An empirical comparison of global, partitioned, and clustered multiprocessor edf schedulers, in: Real-Time Systems Symposium (RTSS), 2010 IEEE 31st, pp. 14–24.

[24] A. Bastoni, B. B. Brandenburg, and J. H. Anderson, Is semi-partitioned scheduling practical?, in: Proc. 23rd Euromicro Conference on

Real-Time Systems (ECRTS 2011), Porto, Portugal, pp. 125–135.

[25] B. Andersson, S. Baruah, J. Jansson, Static-priority scheduling on multiprocessors, in: Proceedings of the IEEE Real-Time Systems Symposium, IEEE Computer Society Press, 2001, pp. 193–202.

[26] B. Andersson, Static-Priority Scheduling on Multiprocessors, Ph.D. thesis, Department of Computer Engineering, Chalmers University, 2003.

[27] T. P. Baker, An analysis of fixed-priority schedulability on a multiprocessor, Real-Time Systems: The International Journal of Time-Critical Computing 32 (2006) 49–71.

[28] M. Bertogna, M. Cirinei, Response-time analysis for globally scheduled symmetric multiprocessor platforms, in: Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International, pp. 149–160.

[29] IEEE, Information Technology - Portable Operating System Interface - Part 1: System Application Program Interface Amendment: Additional Realtime Extensions., 2004.

[30] D. Faggioli, F. Checconi, M. Trimarchi, C. Scordino, An EDF scheduling class for the Linux kernel, in: Proceedings of the 11th Real-Time Linux Workshop, Dresden, Germany.

[31] C. W. Mercer, S. Savage, H. Tokuda, Processor Capacity Reserves for Multimedia Operating Systems, Technical Report CMU-CS-93-157, Carnegie Mellon University, Pittsburg, 1993.

[32] G. Buttazzo, G. Lipari, L. Abeni, M. Caccamo, Soft Real-Time Systems Predictability vs. Efficiency, number 10.1007/0-387-28147-9-3 in Series in Computer Science, Springer, 2005.

[33] L. Abeni, G. Buttazzo, Integrating multimedia applications in hard real-time systems, in: Proc. IEEE Real-Time Systems Symposium, Madrid, Spain.

[34] J. Lelli, G. Lipari, D. Faggioli, T. Cucinotta, An efficient and scalable implementation of global EDF in linux, in: Proceedings of the 7th Annual Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT 2011), Porto, Portugal.

[35] P. Emberson, R. Stafford, R. I. Davis, Techniques for the synthesis of multiprocessor tasksets, in: Proceedings of the 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010), Brussels, Belgium, pp. 6–11.

[36] M. Bertogna, M. Cirinei, Response-time analysis for globally scheduled symmetric multiprocessor platforms, in: Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International, pp. 149–160.

[37] M. Bertogna, S. Baruah, Tests for global EDF schedulability analysis, Journal of Systems Architecture 57 (2010) 487–497.

[38] D. Zhu, N. AbouGhazaleh, D. Mosse';, R. Melhem, Power aware scheduling for and/or graphs in multi-processor real-time systems, Parallel Processing, International Conference on 0 (2002) 593.