Some Results on Experimental Evaluation of Real-Time Scheduling.

Gabriel A. Wainer

Universidad de Buenos Aires.
Facultad de Ciencias Exactas y Naturales.
Departamento de Computacion.
Pabellón I. Ciudad Universitaria.
Buenos Aires. Argentina.
gabrielw@dc.uba.ar.

ABSTRACT: The development of Hard Real-Time systems presents several difficulties to the developers, as they must synchronize carefully the execution of the tasks in the system to produce predictable responses. In this work we present the results of a project devoted to test theoretical results that can simplify the development of real-time software. We have made many changes to the MINIX operating system (mainly to the kernel), and obtained a new version serving as framework to test real-time scheduling algorithms. To show our proposal, we present the results of running real-time workloads with two different schedulers. We also present performance results comparing these algorithms with non real-time schedulers. The results obtained allowed us to analyze the proposal of new scheduling solutions.

1. INTRODUCTION

In these days the number of Hard Real-Time systems is growing rapidly. These systems must control events occurring within the real world, and every task in the system must respond to these events with a constrained response time.

Conventional multitasking operating systems does not provide enough support to build real-time software. Usually, they allow concurrent programming, task synchronization and communication, resource sharing and further services, but do not incorporate primitives to define timing constraints. Due to these reasons, many real-time designers still develop "ad-hoc" solutions to solve each problem.

In this work we present the results of a project devoted to provide programming facilities to develop hard real-time software. To avoid developing software from scratch, we resolved to extend the services provided by MINIX operating system [Tan87]. We selected MINIX motivated by several considerations, mainly our academic purposes, the availability of the hardware and software, and the previous experience in the subject [Wai92].

We want the operating system to ensure predictable behavior of the time critical tasks, entitling the programmer to define timing constraints for the tasks, and letting the operating system to run them in a timely fashion. We also want to provide a framework to test real-time scheduling algorithms, allowing to examine them empirically (instead of using simulations), and allowing the proposal of new solutions.

We devote the rest of this work to expose the sketch of our project, the changes we made, and some experimental results obtained.

2. REAL-TIME SCHEDULING

Real-time scheduling theory relates with fulfilling the task's timing constraints in a real-time system. To avoid unpredictable behavior, we must schedule the distribution of the system resources as well. Other topics to regard include the precedence restrictions among tasks, and the criticality of every task.

The aim of a real-time scheduler is to decide whether there is a schedule to meet the timing restrictions of a task set. In that circumstance, we will say that such collection is *schedulable*.

Most of the real-time schedulers use priority-driven algorithms, both static and dynamic. The scheduler should be *preemptive* as well as a non preemptive scheduler could lead to run a low priority task while a high priority one is waiting.

We can recognize two classes of tasks: periodic and aperiodic (sporadic). The periodic tasks must run regularly, and within a fixed interval. The aperiodic tasks run occasionally, and run only once when we invoke them.

Considering our goals, we decided to implement scheduling algorithms for centralized systems. We selected two traditional ones: the *Rate-Monotonic (RM)* and the *Earliest-Deadline-First (EDF)* [Liu73]. These algorithms only contemplate timing restrictions of the time-critical tasks. Both of them are capable of guaranteeing predictable execution of a task set supposing the processor load is beneath a certain bound. This fact eases the work involved in the development process, reducing the costs.

The RM algorithm is used to schedule periodic independent tasks, giving fixed priorities to every task (inverse to its execution period). The EDF is a dynamic priorities' algorithm. The tasks with earlier deadlines run before those with later deadlines. Both algorithms are preemptive.

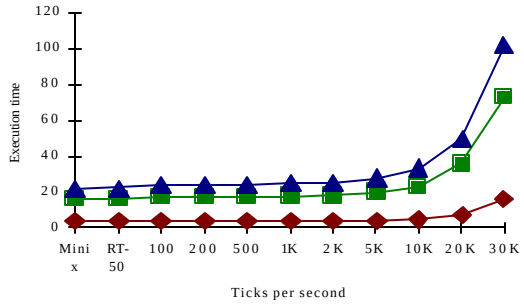
3. KERNEL MODIFICATION

MINIX is a multitasking operating system designed with academic goals, using a Round-Robin scheduler. We have modified this scheduler to allow the execution of real-time *instances* (the minimum useful compute unit of a real-time task, with a constrained compute time). Periodic tasks start one instance per period (task deadline is the period), and aperiodic only have one instance, with a deadline to meet.

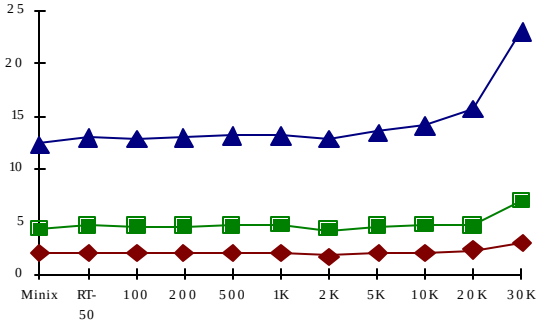
As a first step, we included services provided by many real-time operating systems to compare its performance with conventional real-time schedulers.

To increase timing precision, we incorporated a collection of system calls to modify the activation rate of the timer. We also included a new service (**clock)** to measure the time with a precision of one tick.

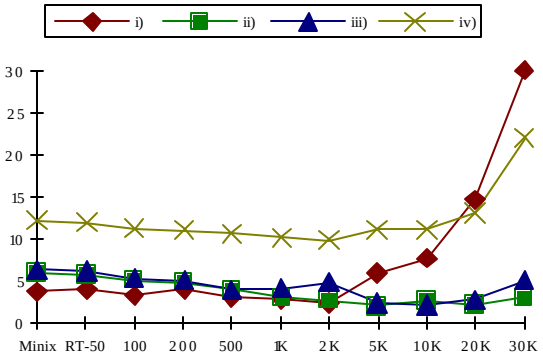
As we know, the greater number of interrupts increases overhead, but, in this case, the behavior of the system was stable up to 10000 ticks per second (meaning we are able to run periodic tasks with a precision of 100 microseconds instead of the original 20 milliseconds). At 30000 ticks per second the performance degrades seriously, and at 40000 ticks the system becomes useless. Besides, we could see that the I/O-bound tasks' response time is approximately the same even with higher clock rates, as these tasks are rarely affected by timeouts.



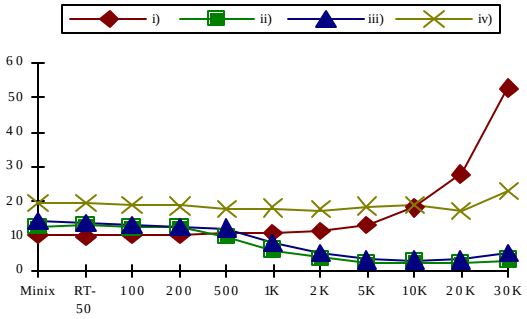
(a)



(b)



(c)



(d)

Figure 1. Overloads changing the clock grain. (a) CPU-bound task sets; (b) I/O-bound task sets; (c), (d) Task mix. i) CPU-bound; ii)-iii) Interactive; iv) mixed

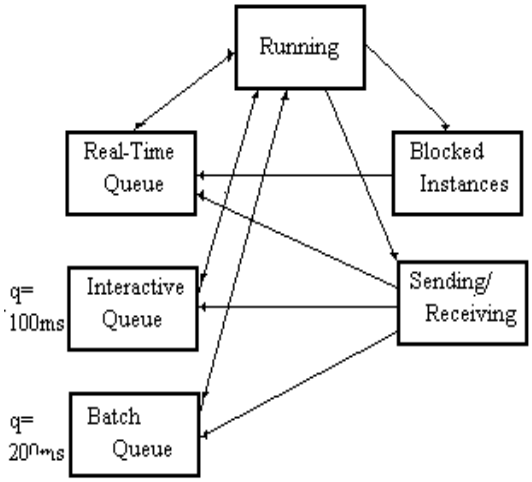


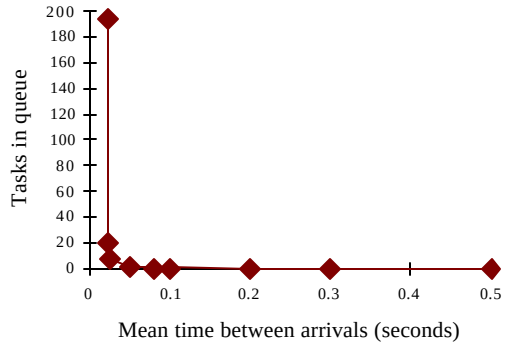
Figure 2. New scheduler structure.

Then, we changed the structure of the task scheduler to suit our task model. We have used a preemptive multiqueue scheduler with three ready queues.

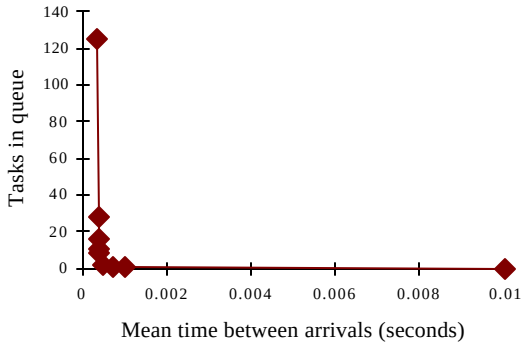
The first queue holds real-time instances ordered using real-time scheduling algorithms. Periodic instances must start within fixed intervals. To do so, the timer driver manages the blocked instances queue, used to store periodic instances waiting for their next period. When a new task period starts, the timer driver removes the Task Control Block from this queue and includes it in the real-time ready queue.

We took special care in the management of the blocked instances queue, trying to lower its overhead. As an initial step, we decided to cut down the activation rate of the timer driver. Instead of running the scheduling routine in every tick, it is enough to consider the least common divisor of the task's periods. Surprisingly, the results of the benchmark tests did not improve (moreover, due to implementation problems, we got worst behavior of the real-time tasks).

We made statistic studies to analyze this behavior. We started studying the queue's mean service time with a large number of tasks (about 500). We ran dissimilar workloads, and arrived to an average service time of 22 milliseconds. Next, we adopted a more realistic approach, and examined typical MINIX workloads (about 40 tasks). In this case, the mean service time of the queue was of 350 microseconds. Then, we modeled the queue using a M/M/1 model, and we concluded that, in average, there are among two and five tasks in the queue, adding only a mean overhead of 100 microseconds to the timer interrupt.



(a)



(b)

Figure 3. Average tasks in the blocked instances queue. (a) Considering large number of tasks; (b) MINIX workloads

As a result, we decided to keep the activation rate unchanged. Now, we are using more efficient data structures and algorithms for instance activation, and we will study the new service time, repeating the previous analysis.

After changing the scheduler operation, we built libraries to exploit the new services of the operating system. The primitives include periods or deadlines of the tasks, and worst execution times. Using this information, we implemented guarantee routines, using Theorem 2 in [Sha90], and Theorem 4 in [Liu73]. We decided to add these new services, instead of changing the old calls, keeping the original MINIX system calls. To have further information about these changes see [Wai95].

4. SOME EMPIRICAL RESULTS

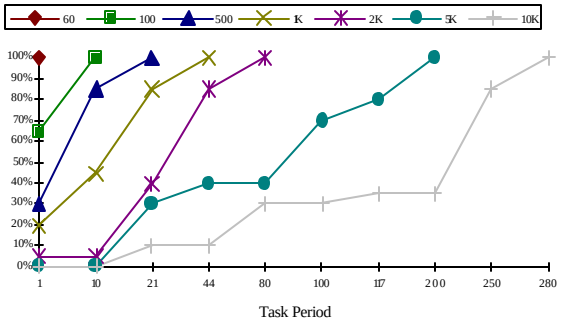
We have built an environment to test scheduling algorithms, and our idea is to use this framework to analyze the theoretic results in an experimental fashion. We paid special attention to implementation issues, performance and overhead of the algorithms.

Our main aim was to examine the reliability of the scheduling algorithms to run predictably a given task set. With this purpose, we studied the *guarantee ratio* of each task set. The guarantee ratio measures the relationship between scheduled instances and deadline missing.

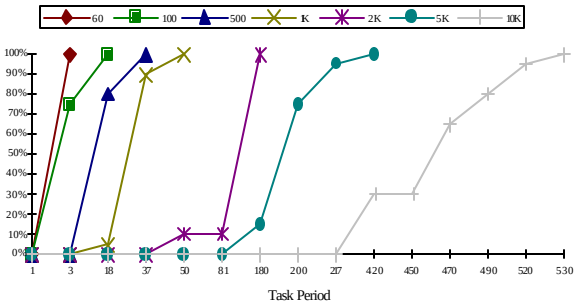
Even a task set satisfying the theoretic bounds should run predictably (100% guarantee ratio) we run tasks not respecting the theory. We want to study the resulting problems in detail, to allow the proposal of new solutions. Following, we present some results obtained (the results presented are worst cases for several workloads).

4.1. Real-time scheduling using a time-sharing scheduler.

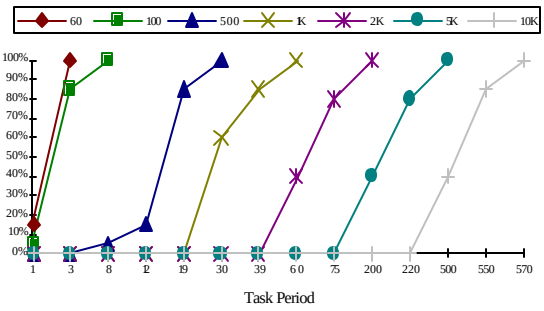
To begin, we examined the performance of the original time-sharing algorithm when scheduling real-time tasks. We compared the



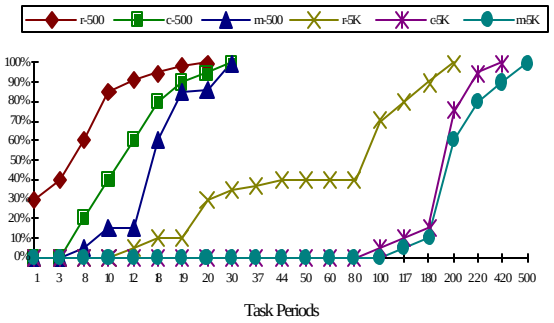
(a)



(b)



(c)



(d)

*Figure 4. Guarantee ratio (5 periodic tasks). Different clock grains (curves). (a) RT schedulers; (b) MINIX scheduler + **clock**; (c)MINIX scheduler; (d) combination (r: real-time; m: MINIX; c: Minix + **clock**).*

results with those obtained using the new **clock** system call. At last, we contrasted both results with a real-time scheduling algorithm (in this case, the RM). We built different task sets, some of them consisting of tasks with the same period and execution times, and others with variable periods.

Then, we tested the guarantee ratio of the scheduling algorithms. Using different task sets, we studied the influence of the schedulers, and also examined the behavior with changes in the clock granularity.

First, we considered periodic instances with a worst execution time of 25 ms. (same period for every task). We also studied the influence of changing the task periods.

We can observe that the comportment of the guarantee function is sigmoid. This happens because, as we have a fixed task set, once a task looses a deadline, will lose all subsequent regularly. With small periods, the guarantee ratio keeps close to 0%, because we have an overloaded system. When we relax the task period, the percentage increases quickly due reduction of the overload. Once a task meets a deadline, will meet it regularly, reducing the cascaded deadline missing.

We can clearly recognize that the guarantee ratio we got using the real-time scheduler is a great deal better than using Round-robin scheduling. Moreover, the results we obtained using the **clock** system call are better than those got without using it, due to the increased precision of the task timing.

Then, we tested a similar task set, with worst execution time of 120 ms. The results for round-robin scheduling are much worse, than the previous case, since MINIX has a 100 millisecond quantum. Hence, the scheduler preempts the running in each period, making more difficult to meet its deadline.

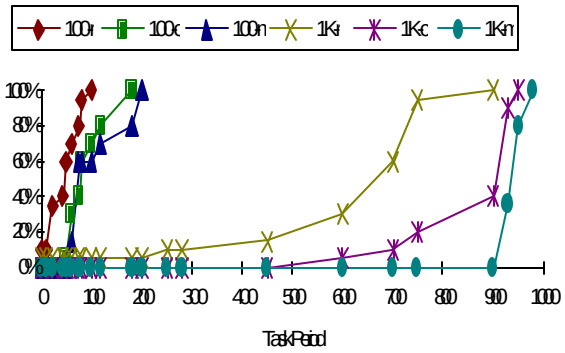


Figure 5. G.R. of 5 tasks (worst execution time: 120ms). m: Minix scheduler; c: Minix using clock() service; r: real-time scheduler.

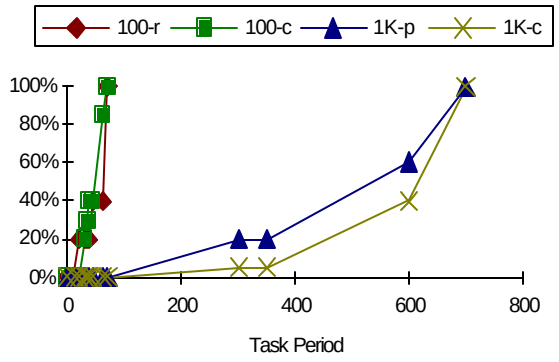
Later, we increased the number of tasks in the set. In this case, we only run task sets using the **clock** call and the real-time scheduler (the results we got running simpler task sets using the MINIX scheduler without the **clock** call showed to be enough bad).

The differences of the guarantee ratio between both examples are not so wide like the previous cases. This happens because the system is almost continuously overloaded due to the number of

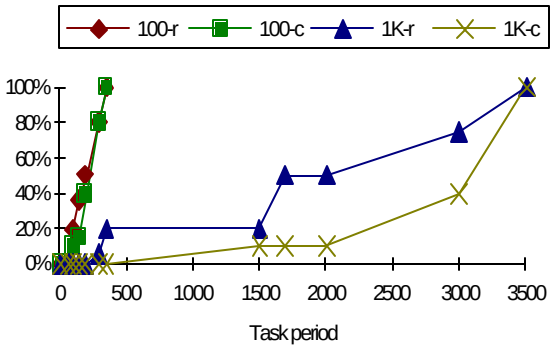
tasks and their execution times. Therefore, the real-time algorithm runs predictably only when the bounds are respected.

The RM algorithm obtained better performance even with overloading. This happens because it runs first the tasks with higher priority, making them to lose fewer deadlines. Again, the situation was worse for MINIX scheduler running instances with longer execution times. We can recognize the influence of the scheduler time-out.

Subsequently, we started more realistic benchmarks consisting of tasks with variable periods. We can see the results in figure 7 (The number in the x-axis of the graphic is proportionate to the tasks' period in the collection).



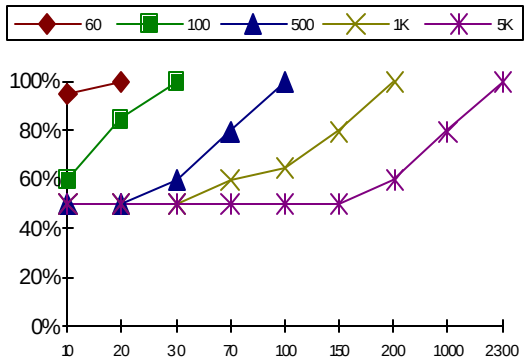
(a)



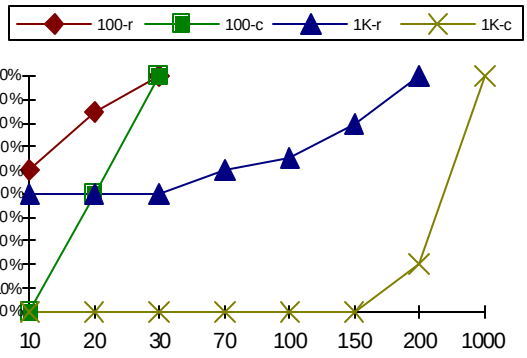
(b)

*Figure 6. 15 tasks. Real-time scheduler (r) and Minix scheduler with **clock** service (c). Worst execution time: (a) 25 ms; (b) 120 ms.*

The differences were so large that we considered not necessary to run overloaded task sets. As the variable periods of the tasks difficult their predictable execution under the round-robin scheduler, the guarantee ratio is clearly better with the real-time scheduler. In the previous cases, we got more reduced differences, because every task had the same period. In this situation, the real time scheduler behaves as a cyclic executive (as the round-robin scheduler does).



(a)



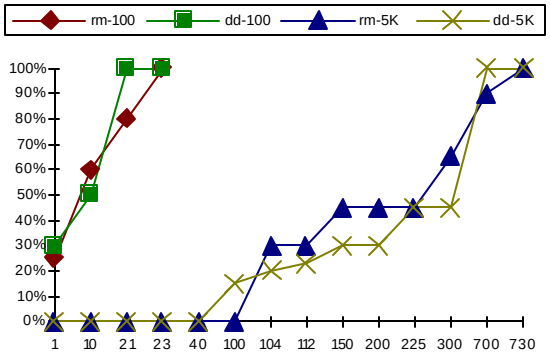
(b)

*Figure 7. Variable periods. (a) real-time scheduler with different clock grains; (b) comparing real-time (r) and Minix + **clock** (c).*

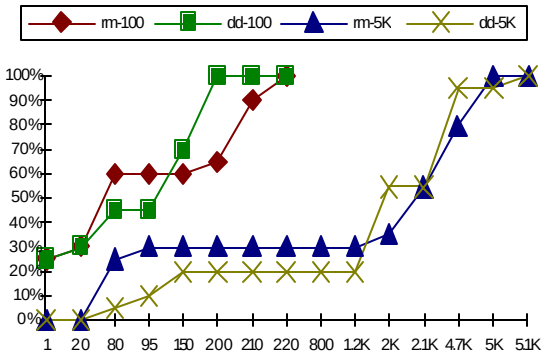
4.2. Comparing two real-time scheduling algorithms

A second idea was to implement other scheduling algorithms, and to relate their performance. We started implementing the EDF, and compared it with the RM approach. We can observe those results in figures 8, and 9. We have repeated the tests we made in the previous section, contemplating the influence of having tasks with equal or distinct periods. We have regarded overloaded systems, increasing the processing times of every task, or increasing the number of tasks in the system as well. In this way, we are able to examine the influence of overloads.

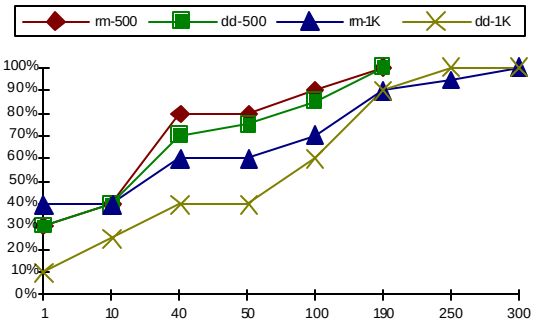
In the foremost case, we considered tasks with equal periods (figure 8). As the RM runs first the most crucial tasks in a stable fashion, the algorithm acts better than the EDF when there is an overload in the system. As the EDF algorithm has a more relaxed bound of processor utilization, it performs better when the load diminishes. The algorithm yields to better processor load, increasing the guarantee ratio. The values for both algorithms are ultimately alike (100%), because when we have longer tasks' periods we are below the conceptual bounds.



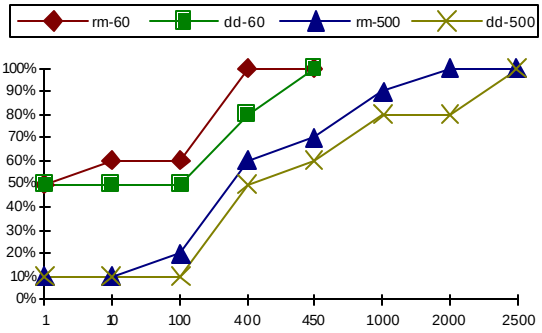
(a)



(b)

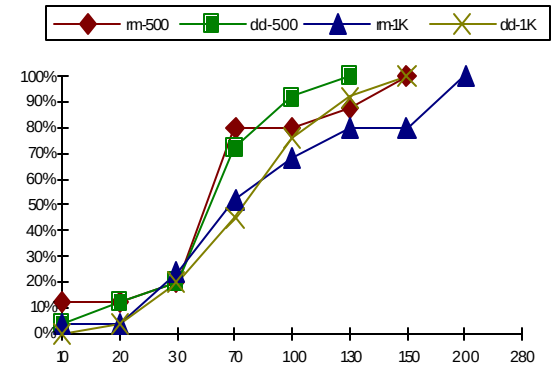


(c)

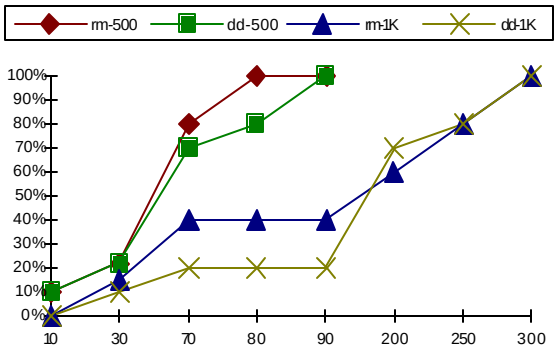


(d)

Figure 8. Comparing RM (rm) and EDF (dd). Same period. Worst execution time: 5 tasks(a) 25 ms (b) 120 ms 15 tasks (c) 25 ms (d) 120 ms



(a)



(b)

Figure 9. Comparing RM and EDF. 5 tasks with variable periods. Worst execution case: (a) 25 ms. (b) 120 ms.

The same occurs when the task periods are variable, as we can see in figure 9 (again, the numbers we can see in the x-axis are proportional to the task's periods).

The situation changes when we increase the number of tasks and each one make extensive use of the processor. In this case, the overload is always high. In this circumstance, the RM acts regularly better. This algorithm permits to satisfy a larger number of deadlines since it schedules the tasks with shorter period earlier, allowing to some of them (the most critical) to meet their deadlines.

4.3. Testing sporadic tasks

Our last task sets considered combinations of periodic and sporadic tasks. As the scheduling algorithm only guarantees the execution of periodic tasks, we want to study in detail the system behavior when we include aperiodic tasks.

We ran different task sets with different combinations of periodic and aperiodic tasks. Likewise, we have contemplated the relationship among task periods and sporadic deadlines.

We run periodic tasks with equal period, varying the sporadic deadlines (we can observe the results in figure 10, 11 and 12). Studying the figures, we can distinguish two cases.

Case 1. Task period equal or larger that the sporadic deadline.

a) Periodic tasks: both algorithms behave with similar shapes. When the overload is too high, the RM has a slightly better guarantee ratio. When we run a large number of tasks with variable periods, the distinction is still greater. We have a system overload up to the moment the task's periods approach the ideal limits. We can see approximately the same results we saw in section 4.2.

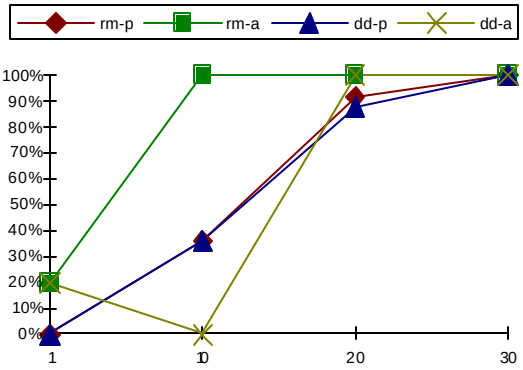
b) Sporadic tasks: the guarantee ratio grows constantly for the RM. Using the earliest deadline first, the overloads make the sporadic tasks to perform irregularly.

Case 2. Sporadic deadlines shorter than periodic frequency

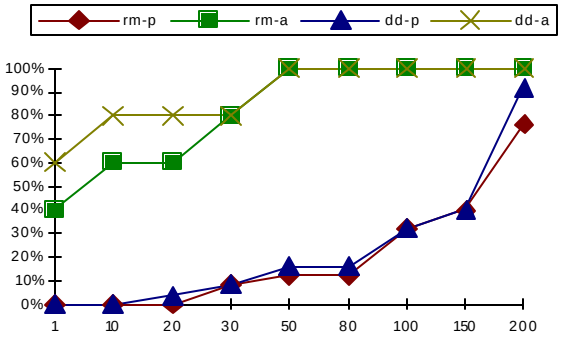
a) Periodic tasks: the earliest deadline first has worse comportment when there are overloads. When we cut down the overload, the behavior is better for this algorithm. The sporadic tasks with

brief deadlines will insert first in the ready queue, delaying the execution of the periodic tasks.

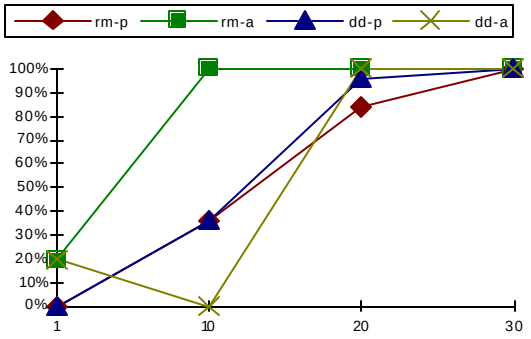
b) Sporadic tasks: the overload makes the earliest deadline first to act unpredictably. When the overload reduces, we have a 100% of guarantee ratio for these tasks. The proportion is better for the periodic tasks as well.



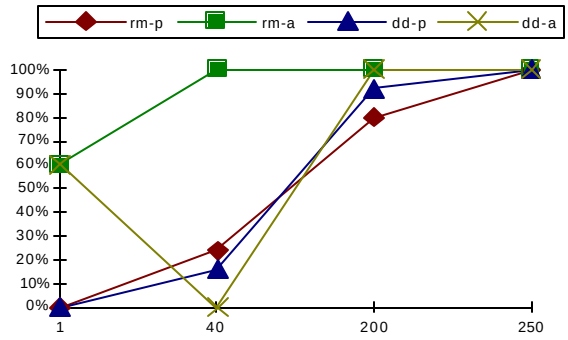
(a)



(b)

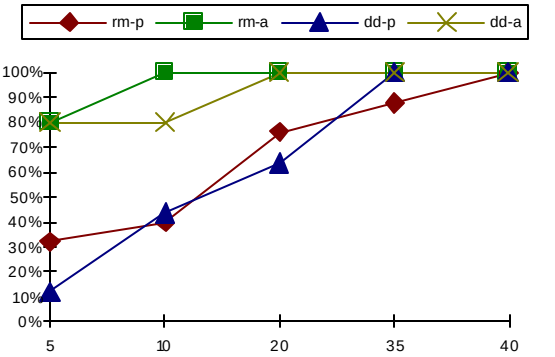


(c)

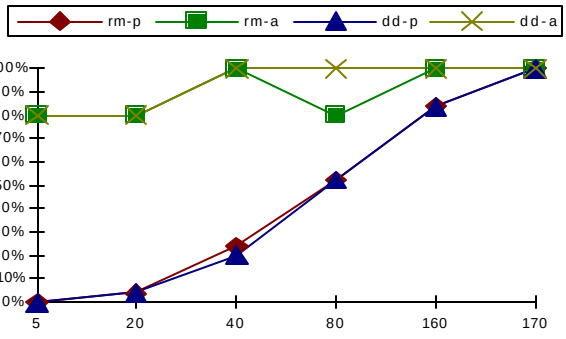


(d)

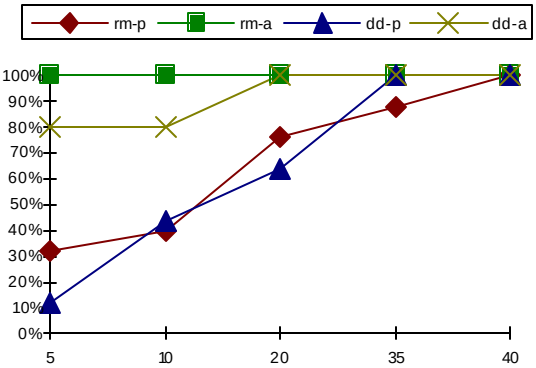
Figure 10. Mix of periodic and sporadic. Equal periods. Period equal to sporadic deadline: (a)100 ticks/sec, (b) 1000 ticks/sec. Deadline smaller than period: (c) 100 ticks/sec; (d) 1000 ticks/sec.

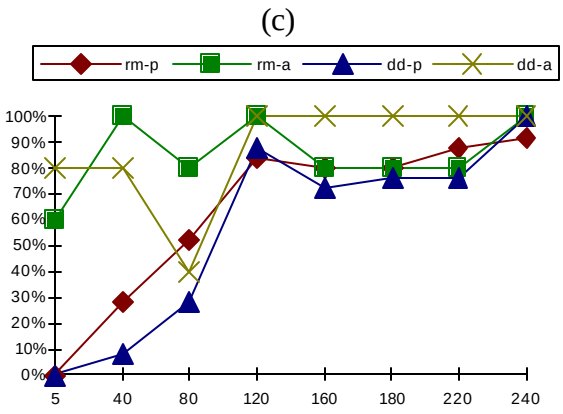


(a)



(b)

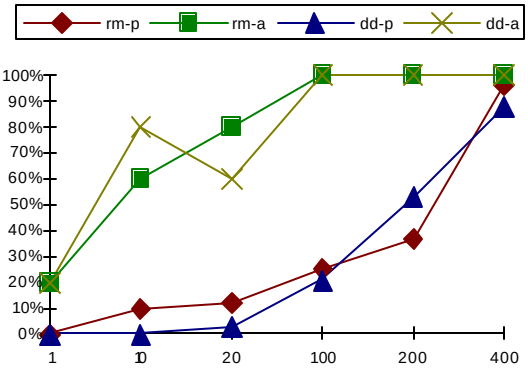




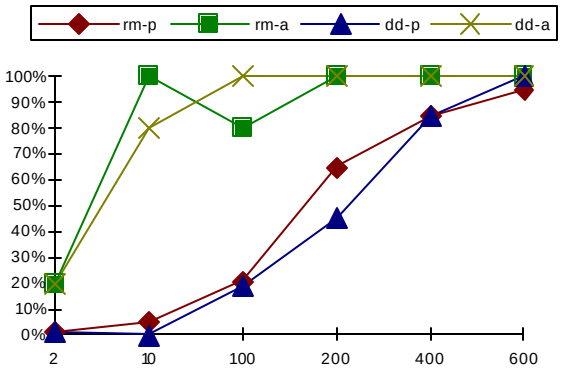
(c)

(d)

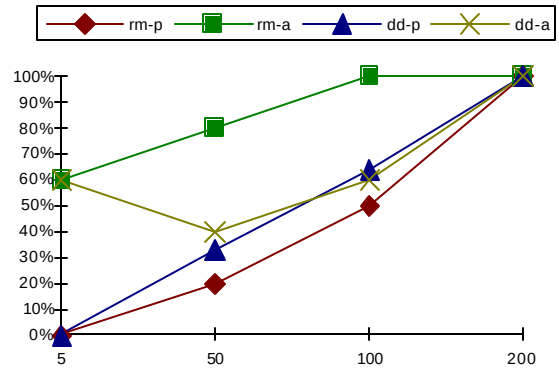
Figure 11. Mix of periodic and sporadic tasks. Variable periods. Period equal to sporadic deadline: (a) 100 ticks/sec, (b) 1K ticks/sec. Sporadic deadline smaller than period (c) 100 ticks/sec; (d) 1K ticks/sec.



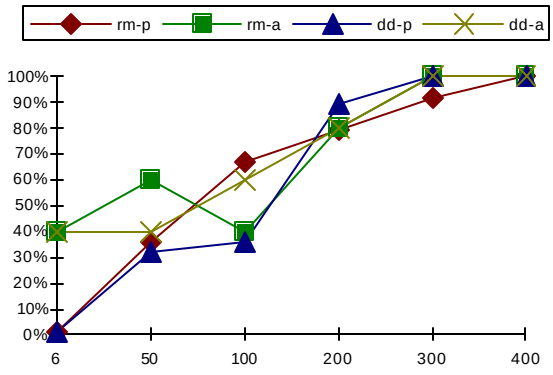
(a)



(b)



(c)



(d)

Figure 12. Mix of periodic and sporadic tasks. 1000 ticks/second. Fixed period (a) Period equal to sporadic deadline (b) Sporadic deadline smaller than period. Variable periods (c) Period equal to sporadic deadline. (d) Sporadic deadline smaller than period.

To finish with this section, we must say that the results always satisfy the ideal bounds. After finishing each benchmark, we inspected the results using the corresponding formulas. In those circumstances the benchmarks showed predictable execution, we were respecting the conceptual limits.

5. PRESENT WORK. CONCLUSION

We have just tested the properties of two classic scheduling algorithms. Our next step will be to experience with different schedulers. Our last goal is trying to find new solutions, and examine them empirically.

We are capable to get good response time with a round-robin scheduler in cases when the activation frequency for every task is approximately equivalent. This is not a good solution indeed we were careful while developing the system. If we must alter a task period, the results are not reliable. The algorithm act even less predictably when it runs tasks with different periods.

We also could see that the real-time scheduling algorithms we implemented have some problems. The RM theory rest in the notion that the criticality of a task depends on its period (which is not always true). The algorithm has low time-loading as well.

The EDF algorithm is dynamic, and allows to run aperiodic tasks more safely. It also has higher time-loading. Its main drawback is its unstable behavior in circumstances of overloading (instead, the RM executes the most critical tasks in a stable fashion).

Although these inconveniences, these algorithms are much better than conventional time-sharing schedulers. We got better guarantee ratio, development times and small overhead. These facts make advisable to use them in any operating system having preemptive priorities as a real-time executive devoted to schedule real-time tasks.

At present, we are testing the implementation of new solutions, such as those presented in [Sha90] and [Liu91], between others. We will implement solutions for sporadic servers, and a heuristic mixed algorithm, that use different schedulers depending on the actual workload conditions [Wai95b]. We will relate these solutions with other algorithms, such as period transformation and imprecise computation models. We have provided facilities letting the user to use multiple version methods, that we will compare with another known solutions.

With this work we tried a first proposal to solve real-time scheduling problems. To do so, we built a framework to develop real-time software, running at predictable times in dynamic environments. We can easily substitute this environment to examine different real-time scheduling algorithms.

We provided new services, permitting the programmers to define tasks with time restrictions, and leaving the scheduler to run them at the needed times. We avoided the programmer intervention with timing issues. In this way, he can focus in solving the application problem, reducing the difficulties related with completing the timing restrictions. In this way, we are able to improve productivity, security and costs during the development cycle. Finally, we could test different task sets, showing advantages and drawbacks of using different real- time schedulers.

6. BIBLIOGRAPHY

[Hin88] HINNANT, D. "Accurate Unix Benchmarking". *IEEE Micro*, October 1988.

[Liu73] LIU, C.; LAYLAND, J. "Scheduling algorithms for multiprogramming in a Hard Real Time System Environtment". *Journal of the ACM*, Vol. 20, No. 1, 1973, pp 46-61.

[Liu91] LIU, W.S. et al. "Algorithms for scheduling imprecise computations". *IEEE Computer*. May 1991.

[Sha90] SHA, L.; GOODENOUGH, J. "Real-Time Scheduling Theory and Ada". *IEEE Computer*, April 1990. pp 53-62.

[Tan87] TANNENBAUM, A. "A Unix clone with source code for Operating Systems courses". *Operating Systems Review*, vol. 21, January 1987.

[Wai92] Wainer, G. A survey of the results of using Minix as a tool for teaching in Operating Systems Courses. *Proceedings of the XII International Conference of the SCCC*. Editorial de la USACH. October 1992.

[Wai95a] WAINER, G. "Implementing real-time scheduling in a time-sharing operating system". *Proceedings of AARTC'95*. Mayo de 1995.

[Wai95b] WAINER, G. "Una estrategia de planificación para mejorar utilización de recursos en planificadores para tiempo real estables". Internal Report of the Computer Sciences Department. FCEN-UBA.

[Wei89] WEIDERMAN, N. "Hartstone: synthetic benchmark requirements for Hard Real-Time applications". *Technical Report CMU/SEI-89-TR-23*. Carnegie Mellon University. Software Engineering Institute.