Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2014, Article ID 945850, 14 pages
http://dx.doi.org/10.1155/2014/945850

Research Article

Real-Time and Real-Fast Performance of General-Purpose and Real-Time Operating Systems in Multithreaded Physical Simulation of Complex Mechanical Systems

Carlos Garre,[1] Domenico Mundo,[1] Marco Gubitosa,[2] and Alessandro Toso[2]

[1] *Universita della Calabria, Viale Pietro Bucci, Cubo 45 C., 87036 Arcavacata di Rende, Italy*
[2] *LMS International, Interleuvenlaan 68, 3001 Leuven, Belgium*

Correspondence should be addressed to Carlos Garre; carlos.garre@gmail.com

Received 27 February 2014; Accepted 16 April 2014; Published 22 May 2014

Academic Editor: Her-Terng Yau

Physical simulation is a valuable tool in many fields of engineering for the tasks of design, prototyping, and testing. General-purpose operating systems (GPOS) are designed for real-fast tasks, such as offline simulation of complex physical models that should finish as soon as possible. Interfacing hardware at a given rate (as in a hardware-in-the-loop test) requires instead maximizing time determinism, for which real-time operating systems (RTOS) are designed. In this paper, real-fast and real-time performance of RTOS and GPOS are compared when simulating models of high complexity with large time steps. This type of applications is usually present in the automotive industry and requires a good trade-off between real-fast and real-time performance. The performance of an RTOS and a GPOS is compared by running a tire model scalable on the number of degrees-of-freedom and parallel threads. The benchmark shows that the GPOS present better performance in real-fast runs but worse in real-time due to nonexplicit task switches and to the latency associated with interprocess communication (IPC) and task switch.

1. Introduction

Real-time operating systems (RTOS) are present in the automotive industry mainly in two scenarios. One is the systems embedded in the vehicle for controlling active systems. These systems are simple compared with a general-purpose computer and in some cases the system is so simple that one could not say that it has a running operating system but rather an ad hoc control logic. In the other side, RTOS are also present during the process of vehicle prototyping and testing, in this case running on powerful computers capable of executing complex physical simulations. One typical scenario for a RTOS is hardware-in-the-loop (HIL) testing, where the electronic control unit (ECU) of some subsystem of the vehicle is tested against computer-controlled conditions, that is, a physical simulation of the vehicle. The ECU typically works at a specific rate, and the computer running the HIL must be able to communicate with the ECU at this rate. The bottleneck to achieve communication at this rate does not come from the hardware, which is typically composed of communication boards specifically designed for this purpose. Instead, the bottleneck is on the capability of the system to attend the inputs coming from the ECU and to give feedback (output) to the ECU at the right time. Figure 1 shows two different scenarios where an external device (the ECU in the case of a HIL test) communicates with a computer running a physical simulation. For each input, a time step of the simulation is run (the green block in Figure 1) to provide an output.

In the figure, T_i represents the time needed to attend an input (interrupt dispatch latency), T_s the time required to compute one time step of the simulation, and T_w the idle time waiting for a new input. If T is the inverse of the working frequency of the external system, then

$$T = T_i + T_s + T_w. \quad (1)$$

Having $T_w < 0$ means that the system is not capable of running at the given rate. For a fixed value of T, this situation may happen when the value of T_i or the value of T_s is too large. Figure 1(a) shows a scenario where a high value of T_i

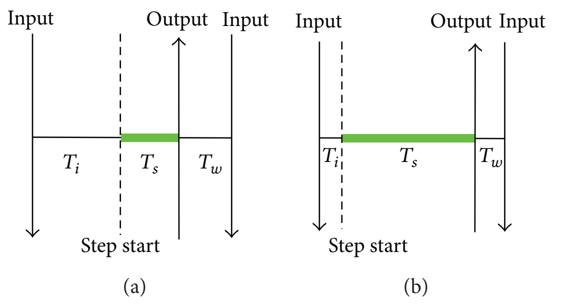


Figure 1: Two different scenarios of physical simulation with an external system in the loop. (a) Short time step and simple models. (b) Larger time step and more complex models.

is more prone to be the cause of a missed deadline. This is the case of a system working at a very high rate and running a simple simulation, that is, a simulation not implying too many bodies nor complex dynamics or constraints. In this case, having a small interrupt dispatch latency (T_i) is critical. For this reason, RTOS are the obvious choice for this type of scenario.

Figure 1(b) shows a different scenario, where the rate is not so high but instead the simulation is very complex; that is, it includes a high number of bodies and/or complex constraints and/or complex physical properties of the bodies (such as nonlinear elasticity). In this case, the system may miss a deadline in the case of a complete fail of the interrupt dispatch (extremely high value of T_i), but the most probable source of missed deadlines would be having a too large value of T_s, which means that the system is not able to compute all the complexity of a simulation step in the given time. The classical scenario for HIL testing in automotive applications has been more similar to that of Figure 1(a) [1], but the complexity of the physical simulation scenarios that modern computers can achieve is growing more and more and nowadays it is possible to run interactive simulations of highly complex systems [2, 3]. When simulating more complex models which are closer to reality, the accuracy of the tests is significantly improving, but the time required to each step (T_s) may be higher, which means that the scenario gets closer to that of Figure 1(b).

RTOS are designed with a focus on maximizing time determinism, by minimizing latencies of basic operations such as task switch (mainly when the switch is from a low priority to a high priority task), interrupt dispatching (T_i), or interprocess communication (IPC) in the case of parallelized applications. On the other side, general-purpose operating systems (GPOS) are designed with a focus on maximizing throughput. The main goal of a GPOS is to be able to accomplish more work in less time, and for this reason it is more suitable for tasks where instead of wanting results at a specific moment we want as many results as possible in less time (such as computing T_s in the case of a complex simulation). These tasks, known as best-effort or real-fast tasks, require a lot of computation power and a lot of resources (such as memory). Unfortunately, the mechanisms used for maximizing throughput, such as fair scheduling [4] or memory paging [5], typically break determinism and thus a GPOS cannot give guarantees on when a specific task will start and finish.

In this paper, we analyze the scalability of a physical simulation running in a GPOS and in a RTOS. The idea is to find the system that better responds to the increasing complexity of the simulation in terms of number of bodies to simulate and number of concurrent threads for parallelization of the solver. Real-time and real-fast performance of both systems are compared. In the case of real-time, the performance of the system is analyzed in terms of hard real-time [6], which means that, instead of measuring the difference between the desired rate and the achieved rate (the value of T_w), we want to measure the binary condition: the system can work at this rate or not ($T_w \geq 0$). This is the condition desired in HIL testing, where losing the synchronization with the ECU may cause a system failure and more in an embedded system where the consequence can be critical with the loss of human lives. We analyze the results for strict hard real-time applications where a single miss can be critical and also for more permissive scenarios where a small number of misses may be acceptable. To test real-fast performance, 60 seconds of simulation was run for each configuration (number of bodies and threads) without a fixed rate ($T_w = T_i = 0$), and the total computation time was measured.

This paper is an extension of the work in [7], where a real-time microbenchmark for choosing a representative case of RTOS was run and a preliminary study on real-time performance of RTOS and GPOS was presented. The present work begins with the description in Section 3 of new aspects to take into account when choosing a RTOS, starting from a summary of the microbenchmark in [7]. A detailed comparison with the results obtained by other authors is also presented in Section 3.1.

In Section 4, the performance of the RTOS (Xenomai) and the GPOS (GNU/Linux) chosen in the microbenchmark is compared in a physical simulation scenario. A simple tire model was designed specifically for this purpose. The model is easily scalable in the number of elements to simulate (masses and springs) and in the level of parallelization of the solver (number of concurrent threads). The real-time and real-fast simulations are run on each system (the RTOS and the GPOS) with different configurations (different number of elements and different levels of concurrency). To benchmark real-time performance, the scenario consists of the physical simulation loop running at a specific rate isolated from the intervention of external interrupts. The lower interrupt dispatch latencies (T_i) of RTOS with respect to GPOS are extensively analyzed in other works on benchmarking [8, 9] and we want to focus instead on the computation of the time step (T_s). For real-fast performance, the same scenario is used but with eliminating the idle time for synchronization (T_w) and thus simulating at the highest possible rate.

The presented results are an extension of the preliminary study in [7] with three main contributions. First, real-fast performance is studied only on this work and was not considered in [7]. Second, the model is much closer to real applications due to the optimization of the parallel solver and to the well-balanced discretization of the geometry. The new parallel solver allows us to simulate up to 57000 masses in hard real-time while the preliminary solution was capable of simulating up to 3149 masses instead. The performance scalability of the new solver is much different, as explained in Section 4.2, and is closer to the scalability of a real parallel solution. The discretization is improved on one side with the balanced distribution of the mass achieved by proper subdivision of the circumference arcs and on the other side by interconnecting the tire spokes without the need of a central mass. Having a central mass connected to all the spokes breaks the balance between the threads because the thread simulating the central mass always presents a much higher computational load. The last contribution is the deep study of the obtained results, including the analytical formulation of the computation time (in Section 4.5) and the detailed study of the scalability with different levels of parallelization (in Section 4.2). Independent results are presented for visualization of hard and soft real-time performance as well as for real-fast performance.

2. Related Work

Benchmarking the performance of complex applications such as physical simulation is not an easy task and there exist many different approaches depending on the nature of the applications and on the aspects of performance which are more relevant in each case. In this paper we establish a clear difference between real-time and real-fast performance, but even when focusing on only one of these aspects the choices are multiple.

Real-time performance can be measured at the lowest level using a microbenchmark [10], which consists in measuring the latency of basic operations such as task switching or interprocess communication (IPC). The basic reference of real-time microbenchmarking is Rhealstone [11], which provides a unique score (the Rhealstone value) of rt-compliance for each system. Rhealstone can be useful for having an overview of the basic real-time performance of a system, but it is difficult to extrapolate its results to systems running complex applications. Synthetic benchmarks [12], on the other side, try to model the behavior of typical applications (usually through stochasthic models) but in a controlled way. Hartstone [13] is the most representative of real-time synthetic benchmarks and is the core of many other implementations [14]. Unfortunately, the results of synthetic benchmarks are still difficult to extrapolate to the performance of final complex applications. Many authors have provided benchmarks for specific applications such as air defense [15], unmanned vehicles control [16], or air traffic collision detection [17], but these benchmarks are focused on measuring the absolute performance of the application rather than on comparing the performance on different systems or configurations. Comparing the performance of a complex application running in different systems is a big challenge, mainly because of the impossibility to isolate the different factors that may affect performance. Porting a complex application to the different systems to be compared usually implies a big effort for adapting the source code to the native API of each system, and a lot of decisions have to be taken on how to apply these modifications in an optimal way.

The main difference between microbenchmarks, synthetic benchmarks, and benchmarking with final applications is on the level of abstraction on which performance is measured. A different classification can be established depending not on the level at which performance is measured, but on the nature of the applications. There exists a lot of work on real-time benchmarking for embedded applications [18–20]. Our work focuses instead on RTOS designed for complex applications running on general-purpose computers. The fast rate at which these systems evolve makes the results obtained in previous works [6, 8, 15–17, 20] probably not applicable to current systems when there is an important difference in time and then in all the technologies implied. Although some authors have compared some of the systems considered in our microbenchmark for selection of one RTOS [8, 20, 21], none of them have compared the four of them (RTAI, Xenomai, Preempt-patched Linux, and Linux), and their results may be outdated as will be discussed in Section 3.1. For a more detailed survey on real-time benchmarking, the reader may refer to [6, 22].

Regarding real-fast performance, there exists previous work on integrating real-time and best-effort tasks on real-time systems [23], on improving best-effort performance of real-time systems [24] and even operating systems specifically designed to optimize the trade-off between best-effort and real-time performance [25]. Although we benchmark separately real-time and real-fast performance, the model we have developed cannot be defined as purely real-time or purely real-fast. Having a large time step and a lot of computations inside each step makes the computation of a step a real-fast task which is bounded by a real-time constraint. Other works on benchmarking assume that a task can have either real-time or best-effort nature, but we have not found any study considering hybrid tasks where both performance criteria are relevant. In [26] the performance of a RTOS and a GPOS achieving some specific tasks (controlling fuel injection for an industrial engine and compiling a Linux kernel) is compared, providing very interesting clues on how to choose between a RTOS or a GPOS depending on the application. However, the application cases presented are clearly biased to real-time (in the case of the fuel injection) or to best-effort (in the case of kernel build) without showing any case of an application where a real trade-off is needed.

3. Choosing a Real-Time Operating System

To compare the performance of two different types of systems, such as RTOS and GPOS, the first step is to select one representative case of each type of system. This was achieved in [7] by designing and running a microbenchmark for choosing a good representative case of RTOS from a test set including RTAI [27], Xenomai [28], and GNU/Linux patched with Ingo Molnar's preemption patch (*RT-Preempt* from now on) [29]. Xenomai was the selected RTOS candidate and GNU/Linux was chosen as the closest example of GPOS, considering that Xenomai is Linux based.

A different approach would be to compare a set of GPOS with a set of RTOS, instead of only one case of each, so that the results can be considered of more general application. Unfortunately, doing this would require porting the whole application to each of the operating systems. Porting applications between OS with different APIs is a heavy task and more in the case of a relatively complex application such as the implemented physical model. Apart from saving work time, which may not be a reason in itself, the fact that each RTOS has its own API and programming semantics means that building a complex application may require doing things in a different way and then adding uncontrolled variables which may affect performance. As an example, periodic tasks are implemented in the native API of RTAI [27], while implementing a periodic task using only POSIX is not straightforward. Of course, periodic tasks could be implemented in RTAI in the POSIX way, but then we should answer the question: What is fairer, comparing the systems when doing things in the best way they can or comparing the systems when doing exactly the same things? The availability of the POSIX skin in Xenomai allowed us to build exactly the same code for both systems (Linux and Xenomai) and then to avoid the need to face this question at this point. Although out of the scope of this work, the answer to this question is still open and could be a good starting point for future work.

In the next subsections we will provide additional clues for more complete criteria when choosing RTOS, not only for benchmarking but also for developing final applications. We start in Section 3.1 with a brief summary of the results obtained in [7] and with a discussion on the reasons why our results differ from other authors, including a new run of the microbenchmark comparing different Linux kernels. We complete the microbenchmark with an additional test for checking the vulnerability of each system to one of the biggest risks in complex real-time applications involving parallel tasks, which is priority inversion, in Section 3.2. In Section 3.3 we talk about other qualitative aspects which cannot be measured or benchmarked but can be determinant when choosing a RTOS. Finally, in Section 3.4 we describe in detail the process of latency measurement used in the microbenchmark, which can be used for other works on benchmarking or software profiling.

3.1. Summary and Discussion of the RTOS Selection. The microbenchmark presented in [7] included tests for measuring average and worst-case latencies for three different atomic operations: switching between tasks of equal priority (*task switch*), preempting a low priority task for execution of a high priority task (*preemption*), and obtaining access to a semaphore lock as a typical example of IPC operation (IPC). A summary of the obtained results is shown in Table 1, focusing only on worst-case latencies for the purpose of hard real-time.

These results clearly present Xenomai as the best candidate. RTAI was expected to have better performance, because Xenomai uses the RTAI kernel with an additional abstraction layer that may add some (slight) overhead and because there exists previous work reporting better performance of RTAI compared to Xenomai [8, 21].

Table 1: Summary of worst-case latencies (in nanoseconds) in the three tests of the microbenchmark for RTOS selection.

	Task switch (ns)	Preemption (ns)	IPC (ns)
RTAI	12060	14855	12986
Xenomai	1422	2922	1282
RT-Preempt	27085	15986	19776
Linux	13200	$>10^6$	12453

There are two main explanations for these unexpected results compared to the work of Barbalace et al. [8]. The first reason is the different nature of the experiments. Barbalace et al. do not run a microbenchmark but a benchmark with a final application with three main sources of latency: interrupt handling, datagram processing, and rescheduling. Regarding interrupt handling, in the case of Xenomai all the interrupts are captured by the ADEOS nanokernel, which dispatches them to the Xenomai nucleus in the case of a real-time interrupt or to Linux in other cases. In RTAI instead, all interrupts are first captured by the RTAI nucleus and only forwarded to ADEOS in the case of a non-RT interrupt (and then ADEOS will forward it to Linux as well). This means that handling RT interrupts on Xenomai requires an additional step (passing through ADEOS) not present on RTAI, which is translated in a slight increase of latency. Since we are not benchmarking interrupt handling, this latency is not affecting our results (but it is affecting the results of Barbalace et al.). Interrupt handling is not considered in our benchmark, because most previous work on real-time benchmarking focuses mainly on this aspect. Our main contribution on this work is on studying the other sources of latency implied in physical simulation applications, so we isolated them from the influence of interrupt handling issues. Regarding datagram processing, we implemented a simple UDP protocol to send the simulation states for graphical render of the simulation on an external computer, but this was done only for the demonstrator (see Figure 3) and all UDP communication was removed for the purpose of the benchmark. Summarizing, the only source of latency in common with the work of Barbalace et al. is rescheduling. Not only the software side of their experiment but also the hardware is different. Their setup is basically an embedded system instead of a general-purpose computer. Indeed, they had to do their own port of RTAI to their platform, which means that they were not using an official RTAI version. In our opinion, this makes a big difference for two reasons. On one side, the fact of doing their own port means that they are not really benchmarking RTAI, but their own implementation adapted to their specific hardware. The infinite versatility of open-source turns into a drawback when trying to establish a comparison between different open-source-based systems. We believe that a fair comparison is possible only when official stable versions can be compared without any modification.

The second cause of the different results we obtained compared with the work of Barbalace et al. is the significant

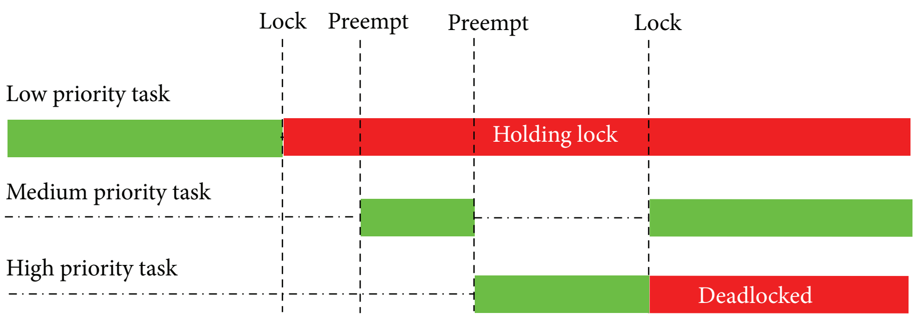


Figure 2: Scenario presenting a deadlock caused by priority inversion.

Table 2: Comparison of the worst-case latencies (in nanoseconds) of the microbenchmark on the Linux kernel used in Xenomai (3.5.7) and on the Linux kernel used in RTAI (2.6.38).

	Task switch (ns)	Preemption (ns)	IPC (ns)
3.5.7	13200	$>10^6$	12453
2.6.38	125435	$>10^6$	147888

difference on versions. Although they do not specify the versions used for each RTOS and for the Linux kernel, even assuming the latest available at the time of publication, they would have used RTAI 3.5 and Xenomai 2.4. From those versions to the versions we have used, the rates of update have been very different for RTAI and Xenomai. The latest available (at the date of this work) RTAI version (3.9.1) supports Linux kernel up to 2.6.38.8, while Xenomai 2.6.2.1 supports Linux kernel up to 3.5.7. We believe that the significant difference (7 years) on the kernels used is one of the main reasons of the different results. To confirm this, we have compared the results of the microbenchmark running on two different kernels: the one used by Xenomai (3.5.7) and the one used by RTAI (2.6.38). The results of this comparison are presented in Table 2.

Regarding the hardware, the processor used by Barbalace et al. (MPC 7455) is from year 2001, while ours (i7-3930K) is from year 2011. This big difference (10 years) in the hardware makes it imprudent to compare the obtained results. Apart from numbers, a study of the changelog of both projects shows that most of the updates of RTAI were basically for supporting more architectures and for improving tools such as RTAI-Lab, which are great but are not used in our benchmark. Instead, the changelog of Xenomai shows an important number of improvements directly related with real-time performance, mainly in version 2.5, when a massive rework of some core components was done to improve performance.

There is another work comparing Xenomai and RTAI real-time performance from the year 2007 [21] presenting very interesting results. In this case, Xenomai performs better than RTAI when working in kernel-space, while RTAI performs better in three of the four experiments in user-space (considering always only the worst-case). The versions used were RTAI 3.5 and Xenomai 2.3.3. The changelog of Xenomai from that version shows a clear effort on improving user-space performance. While there are references only to small kernel-space improvements, there were significant improvements on performance of user-space latency for all architectures in version 2.5.5.1 and full support for real-time IPC in user-space (which was previously only in kernel-space) from version 2.5.

The work in [20] is from a more recent date, comparing Xenomai with RT-Preempt. Unfortunately, RTAI is not included in the benchmark and the nature of the experiments is very different from ours, but Xenomai shows a much better performance compared with RT-Preempt.

3.2. Priority Inversion. Our microbenchmark was based on the Rhealstone benchmark, which includes a test for measuring latency due to priority inversion deadlocks. Instead of measuring how much these deadlocks break determinism, we consider a binary condition for the purposes of hard real-time: the system is vulnerable or not to priority inversion deadlocks. Figure 2 shows an example scenario of priority inversion, which is the one reproduced on this test of the benchmark.

Priority inversion happens when a high priority task cannot start execution due to a lock being held by a low priority task. If a third task, of medium priority, causes starvation of the low priority task, the high priority task will be deadlocked until the medium priority task yields execution. A typical solution to priority inversion is implementing priority inheritance [30]. None of the systems were vulnerable to priority inversion when using semaphores, and even Linux has priority inheritance implemented since kernel version 2.6.18 [31]. The same scenario should also be tested with other IPC mechanisms if they were to be used in a specific application. For example, we found Linux vulnerable to priority inversion when using spin locks instead of semaphores.

3.3. Other Considerations for RTOS Selection. Beside quantitative aspects, such as worst-case latencies, other aspects should be considered when choosing a RTOS, especially for this type of applications where the complexity of the software can be very high. One of the most important qualitative aspects is portability, mainly when following the typical approach of developing under a GPOS for later cross compilation. We started writing the microbenchmark using standard POSIX system calls to run it in Linux. This application was ported almost immediately to the RT-Preempt installation. We only found a different behavior when using the *clock_nanosleep* system call with the *request* argument set to zero. This was used for yielding execution between tasks, so we changed it with the more natural *sched_yield* system call and RT-Preempt was able to run the (POSIX) microbenchmark without problems. We followed the same strategy with Xenomai, using the POSIX skin, and the portability was almost immediate as well. Apart from adding some lines to the makefile (for the POSIX skin) we found some problems with the use of *sem_t* type for implementing semaphores, so we used *pthread_mutex* instead. Summarizing, only minor changes were needed to have a working POSIX application ported from Linux to both RT-Preempt and Xenomai.

In the case of RTAI we had two possibilities. We could use the RTAI API to build a kernel module with the hardest possible real-time or we could follow the (currently more popular) approach of using RTAI LXRT to have hard real-time even in user-space with some POSIX compatibility [32]. We followed the second approach to have the best possible portability, but even in this case the effort of porting the POSIX application to RTAI LXRT was much higher than in the case of the other systems. Porting required not only doing a systematic translation of the system calls to those of the LXRT API, but also changing some semantics of the code.

Considering that porting even a simple application such as the microbenchmark was not straightforward, porting the much more complex physical simulation application would mean a big effort and a too heterogeneous scenario for fair comparison, as mentioned before in Section 3.

3.4. Nonintrusive Time Measuring. Measuring latency implies taking at least two samples of system time. When measuring latency of very simple operations, the operation of sampling itself could be adding a significant latency. It is also important to take the time samples from the most reliable timing source, which implies using timers with the highest possible resolution and nonsensitive to external events. To measure time, we took into account the following.

(1) The system call used for measuring time was *clock_gettime* with *clk_id* set to CLOCK_MONOTONIC. CLOCK_REALTIME presents the same resolution, but it is dependent on system time and thus is not strictly monotonically increasing [33].

(2) The time samples were stored in a static array. Dynamic memory allocation may break determinism [34] and disk and console I/O was completely avoided until the end of the tests.

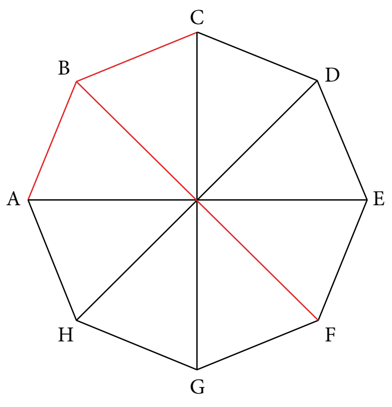
(3) No operations with the time samples were done during the tests. The subtraction of time samples for computing elapsed times was done in a postprocess, as well as statistical computations and finding the worst-case values.

Despite all these precautions, the latency added by the time measuring had to be measured to ensure that it was not significantly affecting the results. For this, we run a test consisting of running a loop with and without time measures for each iteration, comparing then the total time required to finish the loop in both cases. The average overhead of time measuring can then be obtained by subtraction of the average times required to run each iteration of the loop with and without time measuring. The average overhead was in the order of less than 100 nanoseconds, which is negligible compared to the scale of the measures obtained in the tests of the microbenchmark (in the order of 10^3 and 10^4 nanoseconds). Only in the case of the IPC overhead test is the overhead of time measuring significant with respect to the latency of semaphore shuffle in Xenomai, but this value is so far from the latencies obtained for the other systems;magnitude is not affected.

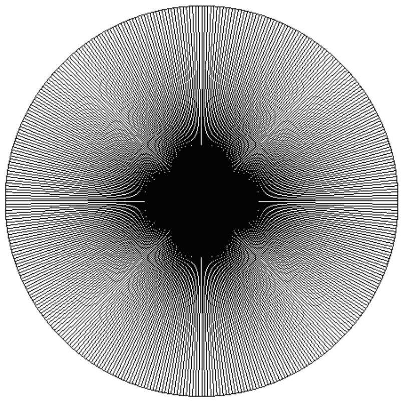
4. Comparing Real-Time and General-Purpose Operating Systems

The performance of a physical simulation application is compared when running in the chosen RTOS (Xenomai) and in the GPOS (GNU/Linux). Both real-time and real-fast performance are considered in separate benchmarks. The test application consists in the simulation of a physical model specifically designed for benchmarking purposes, presenting the typical characteristics of many real applications: multiple bodies affected by forces, contact formulated with constraints, numerical integration of positions and velocities, (linear) elastic forces, and a multithreaded parallel solver. The model consists of a vehicle tire starting in free fall and then deforming due to collision with a floor plane. In the case of the real-time benchmark, the simulation runs at a fixed rate of 100 Hz (time step of 10 ms). For the real-fast benchmark, the time step is also of 10 ms, but the steps are computed as fast as possible instead of being synchronized with the real-time rate.

The hardware used for the benchmarks is an Intel i7-3930K (6 HT cores at 3.2 GHz) with 32 GB of RAM (DDR3 at 1866 MHz) and an ASUS P9X79-LE/C/SI motherboard. The programming language is C, using GCC 4.4.3 compiler with options *-pipe -O2 -D_REENTRANT*. Although a demonstrator was built for visualization of the simulation (see Figure 3(c)) in a second computer, the actual benchmark was run without any graphical output or network communication. The chosen hardware is representative of a typical modern workstation for high performance physical simulation, but care was taken to minimize the impact of the choice of the hardware on the results. Regarding the CPU, the number of available cores is critical for the performance of a multithreaded solution and more in the case of a simple algorithm as the one presented, not designed for massive parallelization. For this reason, the tests were done only for a number of threads multiple of the number of cores available in our setup. In this way, instead of considering a quantitative number of threads (1, 6, 12, and 24 in our case),



(a)


(b)

(c)

Figure 3: Example of different resolutions of the tire model. (a) 8 nodes and 12 springs, showing the detail of the springs connected to node B. (b) 400 nodes and 600 springs. (c) Textured render for the demonstrator.

we consider qualitative differences in the number of threads independently from the number of cores available:

(1) one thread only (monoprocessor),

(2) one thread per physical core (maximum parallelism without considering hyperthreading),

(3) one thread per virtual core (maximum parallelism considering hyperthreading),

(4) two threads per virtual core (more than one thread per core).

The influence of the amount of available memory is avoided thanks to the use of the *mlockall* system call in all the tests. This avoids memory swapping and thus the only benefit of having more memory is to be able to simulate more elements, but the performance for each configuration is the same regardless of the amount of memory. The latency associated with the technology of the memory may increase or decrease the time measures, but the average offset would be the same regardless of the operating system used.

4.1. Physical Model. The physical model of a vehicle tire has been chosen because of its scalability. The tire surface can be discretized with any number of nodes. Each node is a point mass linked through a spring and damper to all adjacent nodes. Each node is considered to be adjacent to other two nodes in the same quadrant and also to a specular node in the opposite quadrant of the circumference. As an example, Figure 3(a) shows in red the three spring-dampers linked to node B, coupling it with nodes A, C, and F. The minimum discretization of the surface starts with 4 nodes (one on each edge of the vertical and horizontal axis) and can grow in steps of 4 by subdivision of the four quadrants of the circumference. Figure 3 shows an example of two tire models using different resolutions (number of surface masses and spring-dampers) and the final render used in the demonstrator.

The parameterization of the model and the methods used for solving each time step must be chosen to ensure that the computational cost is the same for each step except for the operations requiring the use of operating system services, such as task switching or interprocess communication. The mass-spring system is solved using an explicit Euler integrator, because it guarantees that all the time steps will be solved in a fixed number of iterations. A low stiffness had to be used to guarantee stability of the simulation with the explicit solver. Gravity was used as the only force source, and the floor was simulated with a simple geometric constraint. Since collision detection is checked only against the floor level, all time steps require the same amount of computations to detect collisions. Using more constraints and/or a more complex collision detection scheme such as bounding volume hierarchies [35] would imply significantly different costs when computing each time step. Mass-spring-damper systems are a typical approach for dynamics simulation of vehicle tires [36], although there exist other widely used approaches [37]. Although mass-spring-damper systems are only one of many simulation approaches, the characteristics of the tire model make it a good representative case of the main components usually found in most physical simulation scenarios.

4.2. Parallel Solver. Solving one time step for the tire model with the explicit integration scheme implies first computing the forces of all springs and then integrating positions and velocities of all nodes. Both operations can be parallelized by division of the elements to simulate in a number of groups and launching one thread for each group of elements. In a first attempt, we divided both the spring forces and the integration of nodes, but we found that the synchronization between threads when waiting to have other forces computed before integrating nodes was causing an important overhead. We found it more efficient to split only the integration of the nodes. In this way, all the threads compute all the forces affecting their associated nodes, although they have each force computed twice (by different threads) and then each thread integrates only a subset of the nodes. The detailed algorithm is shown in Algorithm 1.

```
For each thread:
    Assign $\lfloor n/i \rfloor$ nodes
    For each simulation Step:
    (1) **Read node positions and velocities of adjacent nodes from other threads**
    (2) Compute spring forces
    (3) Integrate positions and velocities of assigned nodes
    (4) **Write positions and velocities of assigned nodes**
```

Algorithm 1: Algorithm for parallelization of the tire model, where n is the total number of nodes and i the number of threads.

The steps with boldface text require access to mutual exclusion regions shared by all threads. The mutual exclusion regions consist of one binary semaphore controlling access to each of the nodes (position and velocity). For each step, each thread must access $3\lfloor n/i \rfloor$ adjacent nodes and must write down the new values for $\lfloor n/i \rfloor$ nodes. This algorithm is not optimal but presents the typical characteristics of a parallel solver for physical simulation, dividing the elements to simulate in a number of threads and synchronizing threads using IPC mechanisms (semaphores in this case). All threads are created with the same priority of the parent process, avoiding forced preemption inside the solver. The parent process waits in a *join* to all threads before starting the compilation of benchmark data (number of missed deadlines for real-time benchmarking and elapsed time for real-fast benchmarking).

To understand how the efficiency of this algorithm scales with respect to the number of threads, Figure 4 shows four example scenarios with different numbers of threads per CPU. Each image represents the main blocks executed by one CPU (not thread) for one time step. There are three types of basic blocks.

Red Blocks. Read/write operations with mutual exclusion (potential locks), which appear only when reading nodes position and velocities from other threads and when writing self nodes positions and velocities to be later read by the other threads: red blocks correspond to the steps with boldface text in the description of the algorithm in Algorithm 1.

Green Blocks. Computation of spring forces and integration of nodes positions and velocities, which are never preempted (considering a real-time scheduler, such as FIFO): the sum of red and green blocks for each time step represents the total computation time T_s.

Blue Blocks. Idle time for synchronization with the real-time rate (T_w).

Figure 4(a) shows a nonparallel solver, with only one thread running in one CPU. In this case, there is only one large green block since there is no communication between threads.

Figure 4(b) presents an example of a configuration without full parallelization, where there is less than one thread per CPU (i.e., some CPUs are idle, but there are at least two CPUs with one thread each). In this case, most of the time is still spent in the green block, since each thread must compute a lot of nodes.

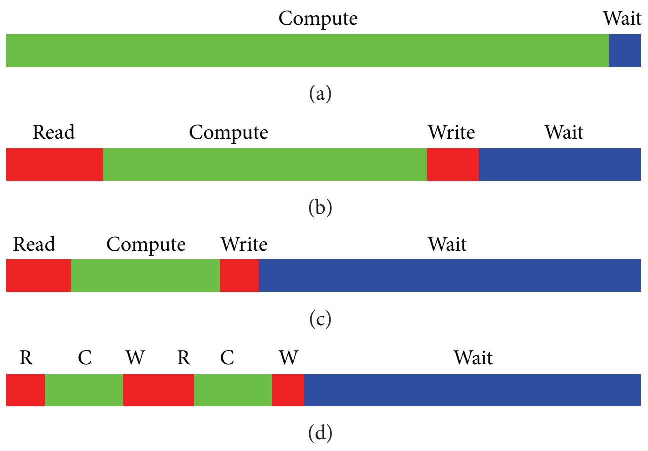


Figure 4: Blocks executed by one CPU in four different configurations: (a) one thread in one CPU (monoprocessor); (b) less than one thread per CPU; (c) one thread on each CPU; and (d) two threads on each CPU.

In the case of having one thread per CPU, Figure 4(c), each CPU computes fewer nodes than in the previous case and then the green block is smaller. The size of the red blocks should decrease in the same scale but, since there are more threads, it is more probable to find a lock closed when trying to access a node. This means that the size of the red blocks may decrease, but there are no guarantees on their size because of potential locks. The net result is that there is an advantage on reducing the size of the green block and a probable (but not guaranteed) reduction of the size of red blocks. Then we have a bigger blue block, which translates in a reduced probability of missing deadlines.

Figure 4(d) shows a configuration with two threads per CPU. In this case, green blocks are even smaller, and the size of the red blocks may (or may not) decrease. The problem is that now we have twice the number of red blocks and that we cannot give any guarantees on the size of red blocks. The net result will depend on the relative size of green and red blocks. In the image, the big size of red blocks shows a clear disadvantage. In other cases (depending on the number of nodes, CPUs, and threads) we could have a more promising situation. It is important to remark that, in any case, the size of red blocks cannot be estimated a priori, due to potential waits in the semaphores. This could also translate in different sequences, as for example having the *red* block of one thread starting before the green block of another thread and finishing after that green block. Summarizing, the size of red blocks is quite unpredictable, but they cannot be avoided because only thanks to them can we reduce the size of green blocks.

4.3. Real-Fast Performance. When real-time is not needed, as in the case of offline simulation without interface with external systems, the tire model can be simulated without timing constraints, trying to compute the whole simulation time as fast as possible. This means that there is no idle time for synchronization ($T_w = 0$) and the blue blocks in Figure 4 are completely removed. This is an example of a real-fast task [26], for which GPOS are specifically designed and optimized.

One of the main differences between a GPOS and a RTOS is the task scheduler. Real-time applications usually trust in real-time scheduling policies such as FIFO (First In, First Out). FIFO scheduling gives the programmer control on when and which task switches happen, unless a task switch is forced due to preemption of a high priority task or due to any condition requiring the task to wait for some resource (such as a semaphore lock). On the other side, GPOS use scheduling policies designed to optimize best-effort (real-fast) performance of all the applications that could be concurrently running on the system and do not trust in the programmer of a single application for control of task switches. In the case of GNU/Linux, the Completely Fair Scheduler (CFS) [4] is designed to provide equal CPU power to all concurrent tasks. Although both systems (Xenomai and Linux) allow the use of both the FIFO and the CFS scheduler, a fair comparison between RTOS and GPOS assumes that each system is using the scheduler for which it is designed, that is, the FIFO scheduler for Xenomai and the CFS scheduler for Linux. For each system, the benchmark is then run using its native scheduler, but results are also presented for CFS scheduling in Xenomai, demonstrating that the results are not only influenced by the choice of the scheduling policy.

Figures 5 and 6 show the computation time of the tire model being run as a real-fast task in the GPOS (Linux) and the RTOS (Xenomai), respectively. The simulation was run with four different parallelization levels: single process (1 thread), one thread per physical core (6 threads), one thread per virtual core (12 threads), and multithreading inside the cores (24 threads). The y-axis shows the total computation time in seconds, while the x-axis shows the number of masses in the model. The difference between the green and red curves represents the advantage of using hyperthreading. For systems not supporting hyperthreading, the correct reference for maximum parallelism would be that the red curve and the green curve should be ignored. As explained in Section 4.2, the black curve (multithreading inside the cores) depends on the number of threads running on each core and could easily go above the blue curve (single process) when the number of threads is excessive and most of the time is spent in accessing mutex locks.

The results of the real-fast execution show, as expected, a better performance of the GPOS when the only timing constraint is to finish as soon as possible. There are two main reasons for the better performance of the GPOS. The first is the overhead added by the RTOS nucleus. For Xenomai, the Linux kernel is seen as a user application running on top of the Xenomai nucleus. This means that the RTOS implements

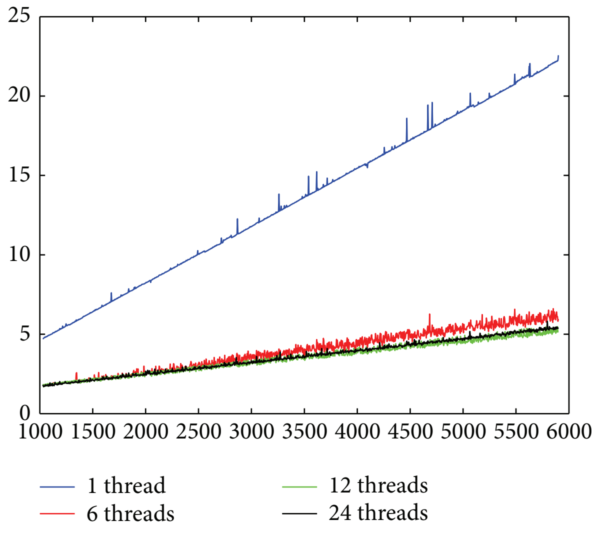


FIGURE 5: Computation time of a real-fast simulation of the tire model using Linux with CFS scheduler. The x-axis shows the number of nodes being simulated, while the y-axis shows the total computation time in seconds. The simulation time was of 60 seconds for all configurations.

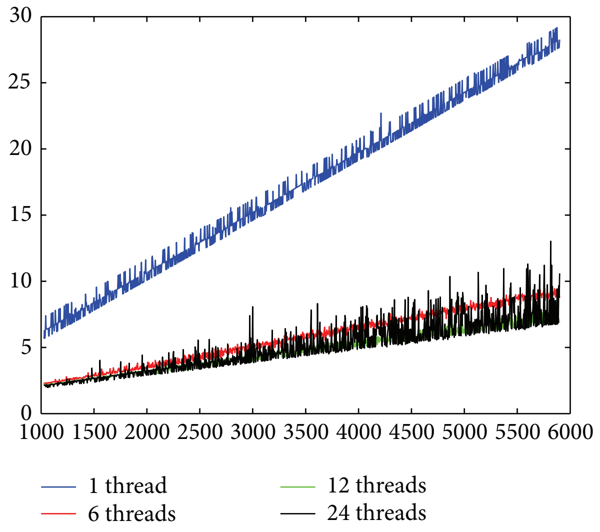


FIGURE 6: Computation time of a real-fast simulation of the tire model using Xenomai with FIFO scheduler. The x-axis shows the number of nodes being simulated, while the y-axis shows the total computation time in seconds. The simulation time was of 60 seconds for all configurations.

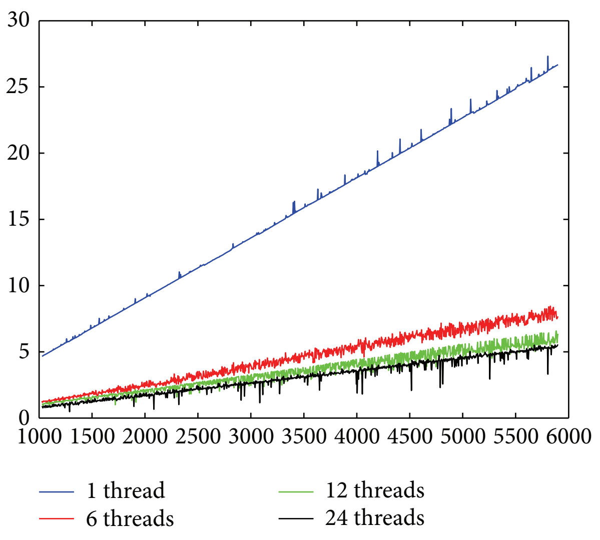


Figure 7: Computation time of a real-fast simulation of the tire model using Xenomai with CFS scheduler. The x-axis shows the number of nodes being simulated, while the y-axis shows the total computation time in seconds. The simulation time was of 60 seconds for all configurations.

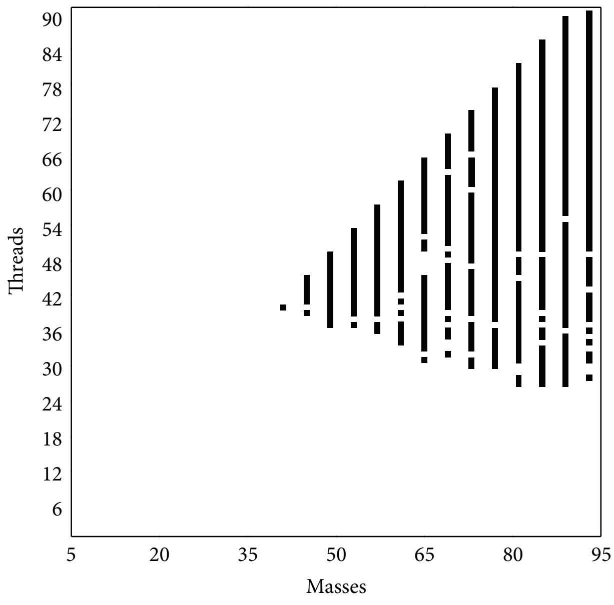


Figure 8: Missed deadlines in Linux when using a high number of threads with few masses and a short time step. The x-axis shows the number of elements to simulate and the y-axis the number of concurrent threads. Each black point represents a configuration for which at least one time step was not computed in real-time.

an additional software layer for all user applications. The second reason is the use of fair scheduling. The algorithms behind the CFS scheduler tend to give an optimal distribution of CPU time among all the running processes. The stochastic nature of these models means that there may not be an advantage for short computations, but each test implied computing 6000 steps, and on each step up to 5900 node positions and velocities were integrated and up to 8850 spring-damper forces were computed.

Figure 7 shows the computation time for a real-fast run of the tire model in the RTOS using the CFS scheduler instead. In this case, the superior performance of the GPOS is due only to the overhead added by the Xenomai nucleus. It demonstrates that, although many current RTOS such as Xenomai are designed for general-purpose computers, GPOS are still a better choice when our applications do not include real-time tasks.

The FIFO scheduler does not have any compensation mechanism for threads that have been waiting for semaphore locks and then the total computation time is very dependent on the particular sequence of locks of each run. The CFS scheduler instead tries to compensate for big wait times rewarding the starving threads with more CPU time. The net effect can be observed in the high amount of oscillations in Figure 6 compared with Figures 5 and 7.

4.4. Real-Time Performance. Real-time performance can be measured in different ways depending on how strict the timing of our applications is. For some soft real-time applications, the deviation between the desired and the obtained rate can be measured as a quantitative value in nanoseconds or in any other time units. However, important concerns regarding these quantitative measures have been discussed [6] and we instead measure real-time performance as a binary condition for each time step: deadline was met or not.

As explained in Section 4.2, due to the overhead added by IPC and task switches there is no advantage in making the number of threads (i) grow above some threshold. To find an empirical threshold for the convenient maximum number of threads for our solver, we run a pretest of the benchmark using a much smaller time step (0.5 ms) to obtain missed deadlines even with few nodes. Figure 8 shows the result of the pretest for Linux. The graph shows on the x-axis the number of masses in the simulation (n) and on the y-axis the number of threads used in the solver (i). A black point in the graph means that, for that combination of masses and threads, the system was not able to meet all the deadlines (at least one deadline was missed).

The first missed deadlines appear when using 40 threads with only 41 nodes, and, when having more nodes, it seems that having more than 24 threads deadlines are missed in most cases. 24 threads are then chosen as the maximum number of threads for all the tests, which means a maximum of 2 threads per virtual core (or 4 threads per physical core). This threshold represents a point of trade-off between reducing computation time (size of green blocks on Figure 4(d)) and having more IPC overhead (more red blocks on Figure 4(d)). Since this threshold is of stochastic nature, it can be estimated only by obtaining empirical data from running simulations and it depends on the algorithm chosen for parallelization of the solver. It is outside of the scope of this paper to discuss the best choice of the algorithm and the scalability of different algorithms with respect to the number of threads.

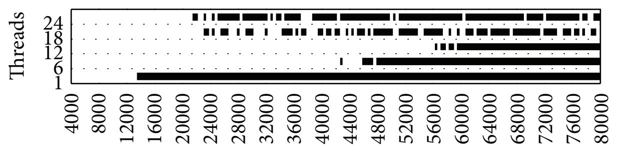


Figure 9: Hard real-time performance of Xenomai using FIFO scheduler. The x-axis shows the number of elements to simulate and the y-axis the number of concurrent threads. Each black point represents a configuration for which at least one time step was not computed in real-time.

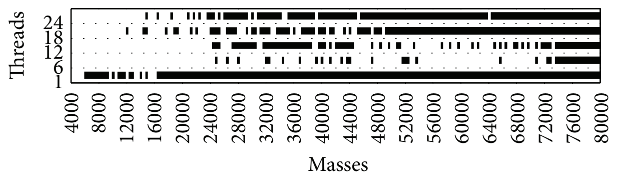


Figure 10: Hard real-time performance of GNU/Linux using CFS scheduler. The x-axis shows the number of elements to simulate and the y-axis the number of concurrent threads. Each black point represents a configuration for which at least one time step was not computed in real-time.

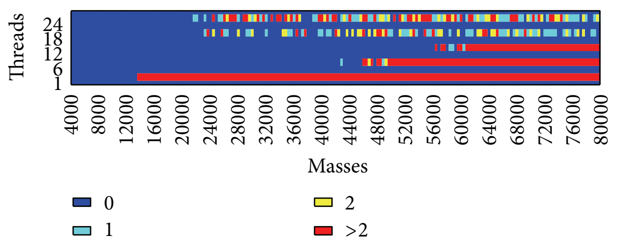


Figure 11: Soft real-time performance of Xenomai using FIFO scheduler. The x-axis shows the number of elements to simulate and the y-axis the number of concurrent threads. The legend shows the number of missed deadlines.

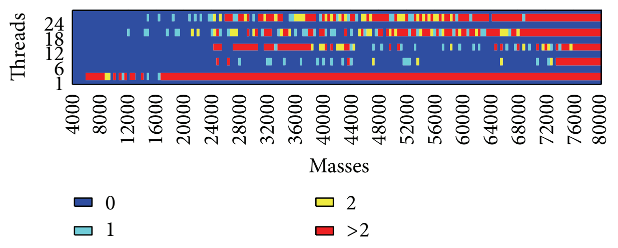


Figure 12: Soft real-time performance of GNU/Linux using CFS scheduler. The x-axis shows the number of elements to simulate and the y-axis the number of concurrent threads. The legend shows the number of missed deadlines.

As in the case of real-fast benchmarking, the real-time benchmark is run using the native scheduler for each system: CFS for the GPOS and FIFO for the RTOS. Results are also presented for Linux running with FIFO scheduler, proving that the choice of the scheduler is not the main factor affecting performance.

The hard real-time performance of each system measured with the benchmark is shown in Figures 9 and 10. The graph is similar to that of Figure 8, showing the number of simulated elements in the x-axis and the number of threads in the y-axis. The y-axis shows only results for a number of threads which are significant with respect to the number of available cores, allowing then easy extrapolation of these results to other setups.

In Figure 9 it is observed that Xenomai achieves hard real-time in simulations with up to 14000 nodes regardless of the number of threads used. The best performance is achieved with maximum parallelism (one thread per virtual core, which is 12 threads in our case). In this case, hard real-time is guaranteed for up to 57000 nodes. In the case of Linux (Figure 10), real-time is lost with only 6000 nodes when running on a single CPU and with 24000 for maximum parallelism (one thread per virtual core).

The results show that the RTOS provides better guarantees of hard real-time than the GPOS during the simulation of a time step of a complex simulation. To isolate the time required for computation of the time step (T_s), the interrupt dispatch latency (T_i) was not taken into account. Considering that other works demonstrate that the interrupt dispatch latency is lower for a RTOS than for a GPOS (mainly in the worst-case) [8, 9], the combination of all latencies (T_i and T_s) presents globally better hard real-time performance for the RTOS even in the case of complex simulations with large time steps (scenario 2 of Figure 1).

Figures 11 and 12 show the same data, but, instead of a binary plot of configurations presenting at least one missed deadline (hard real-time performance), the number of missed deadlines for each configuration is coded with colors. The figures demonstrate that the RTOS performance is very predictable when using a maximum of one thread per core. When the number of nodes to simulate exceeds the capability of the system, the curve turns from blue to red with only a tight transition phase (a few teal and yellow dots). In the case of Linux, there is no clear separation between blue (real-time capable) and red (non-real-time) regions, but instead an unpredictable behavior telling us that in some configurations it may be even better than the RTOS, but there are no guarantees for it. When having more than one thread per core (18 and 24), the performance is quite unpredictable in both systems due to the high amount of potential locks (unexpected size of red blocks in Figure 4(d)), but Xenomai presents fewer red dots even for the more complex configurations.

The use of different scheduling policies for each system could be a determinant factor that may be biasing the results toward one of the candidates if the algorithm used for parallelization was better suited for one specific policy. Figure 13 shows the performance of Linux using a FIFO scheduling policy, demonstrating that obtaining RT performance is not just a matter of using a specific scheduling policy on any system. Comparing the performance of Linux using the CFS or the FIFO scheduler, with FIFO policy the behavior seems to be more predictable, but the performance is significantly

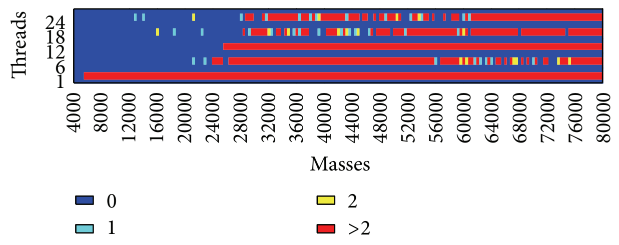


Figure 13: Soft real-time performance of GNU/Linux using FIFO scheduler. The x-axis shows the number of elements to simulate and the y-axis the number of concurrent threads. The legend shows the number of missed deadlines.

worse with respect to Xenomai and even with respect to Linux using the CFS scheduler.

4.5. Discussion of the Results. An explanation for the better real-time performance of the RTOS can be found in the latencies obtained in Table 1. The algorithm used for the parallelization of the solver (see Section 4.2) makes the total time required for computing the time step (T_s) depend on

(i) the total number of elements (n),

(ii) the number of elements assigned to each thread ($e = \lfloor n/i \rfloor$),

(iii) the latencies associated with semaphore lock (T_{ipc}), with task switch (T_{switch}), and with the computation of a spring force (T_{spring}) and a mass position and velocity (T_{mass}),

(iv) the average number of task switches (N_{switch}) not due to semaphore lock and of semaphore locks (N_{ipc}).

The following equation shows the general expression of the total latency (T_s) for each thread of the solver:

$$T_s = N_{\text{switch}} T_{\text{switch}} + 4eN_{\text{ipc}} T_{\text{ipc}} + (2e+1)\, T_{\text{spring}} + eT_{\text{mass}}. \qquad (2)$$

In the case of the FIFO scheduler, N_{switch} is zero considering that the code of the thread does not include any explicit yield, while for the CFS scheduler a time slice could finish during the computation of the time step and thus nonexplicit task switches may happen (the time slice of the CFS scheduler was dynamically changing between 2.25 ms and 18 ms). This implies that the term $N_{\text{switch}} T_{\text{switch}}$ is zero for the RTOS but not for the GPOS. Even having an equal value for N_{switch}, the task switch latency (T_{switch}) is also bigger for the GPOS, as reported in Table 1. The other terms of (2) present similar values for the RTOS and the GPOS (around 80 ns for T_{mass} and 400 ns for T_{spring}) except for the term T_{ipc}, which is also known to be bigger for the GPOS, as reported in Table 1. Apart from the threads created by the benchmark application, there were other processes running on the system, some of which with more than 1% CPU usage. Many processes were related with the graphical interface (*Xorg*, *gnome-terminal*) but others such as *init* or *bash* were needed in any case. It is possible to run the benchmark after stopping the GUI, but we found it anachronistic for a benchmark oriented to modern general-purpose computers where most applications use GUI interfaces [38, 39].

Summarizing, the existence of nonexplicit task switches in the CFS scheduler and the bigger (worst-case) latencies for task switch and for IPC makes the GPOS require more time to compute one time step in the worst-case.

While a single worst-case latency can ruin real-time performance, real-fast performance is much less sensitive to outliers. The stochastic nature of fair scheduling algorithms makes real-fast performance dependent mainly on the average value of T_s and more when there are a large number of statistical samples (a large number of time simulated steps). Hard real-time performance depends instead on (single) outlier values of T_s among all the simulated steps. In some soft real-time applications, the standard deviation of T_s may be the value of interest. Summarizing, a system is real-time capable if it presents a small enough (not as small as possible) average T_s and an upper bound for T_s which is below the time step value. On the other side, a real-fast performance system is better as long as the average value of T_s is lower.

5. Conclusions

In this paper, real-time and real-fast performance of real-time and general-purpose operating systems were compared when running parallelized physical simulations with high computational cost. The physical simulation is run with a large time step, which means that the computation of one single step is closer to the nature of real-fast tasks. If the model must be computed at a fixed rate, the computation time of each step is bounded and the real-fast computation of each step is constrained by real-time requirements. This makes the model lay between the real-fast and real-time frontiers and thus the choice between running the simulation on a real-time or on a general-purpose operating system is not clear.

The first part of this work presented clues on how to choose a RTOS for benchmarking or other applications, starting from the results obtained in previous work by the authors. The second part consisted in comparing the performance of a RTOS (Xenomai) with that of a GPOS (GNU/Linux). A vehicle tire model based on a mass-spring-damper system and solved in a multithreaded fashion was used as application case for benchmarking. The model was configurable in the number of bodies (masses, springs, and dampers) to simulate and on the number of threads of the parallel solver, allowing the measuring of performance for many different configurations. To measure real-fast performance, 60 seconds of simulation time was run for each configuration on each system, and the total elapsed time was measured. The GPOS demonstrated superior real-fast performance compared with the RTOS. For real-time performance, the number of missed deadlines was measured for each configuration, and the RTOS showed a better performance compared to the GPOS for both hard and soft real-time.

Different scheduling policies were tested for each operating system, demonstrating on one side that using a scheduler oriented to real-time applications does not guarantee a better real-time performance in a general-purpose system and on the other side that RTOS designed for general-purpose computers are less performant for real-fast tasks even if the same scheduling policies of a GPOS are used.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

The research leading to these results has received funding from the People Programme (Marie Curie Actions) of the European Union's Seventh Framework Programme FP7/2007 –2013 under REA Grant Agreement no. 285808, INTERACTIVE: Innovative Concept Modelling Techniques for Multi-Attribute Optimization of Active Vehicles (http://www.fp7interactive.eu/). Furthermore, the authors kindly acknowledge IWT Vlaanderen for the support of the ongoing research project "Model Driven Physical Systems Operation-MODRIO," which is part of the ITEA2 project 11004 "MODRIO" (in turn, supported by the European Commission).

References

[1] H. K. Fathy, Z. S. Filipi, J. Hagena, and J. L. Stein, "Review of hardware-in-the-loop simulation and its prospects in the automotive area," in *Modeling and Simulation for Military Applications*, vol. 6628 of *Proceedings of SPIE*, April 2006.

[2] C. Garre and M. A. Otaduy, "Haptic rendering of objects with rigid and deformable parts," *Computers and Graphics*, vol. 34, no. 6, pp. 689–697, 2010.

[3] W. C. Prescott, G. Heirman, J. De Cuyper et al., "Using high-fidelity multibody vehicle models in real-time simulations," in *Proceedings of the SAE World Congress*, pp. 5–22, 2012.

[4] C. S. Pabla, "Completely fair scheduler," *Linux Journal*, no. 184, 2009.

[5] M. Gorman, *Understanding the Linux Virtual Memory Manager*, Prentice Hall, 2004.

[6] W. A. Halang, R. Gumzej, M. Colnarič, and M. Družovec, "Measuring the performance of real-time systems," *Real-Time Systems*, vol. 18, no. 1, pp. 59–68, 2000.

[7] C. Garre, D. Mundo, M. Gubitosa, and A. Toso, "Performance comparison of real-time and general-purpose operating systems in parallel physical simulation with high computational cost," SAE Technical Paper, 2014.

[8] A. Barbalace, A. Luchetta, G. Manduchi, M. Moro, A. Soppelsa, and C. Taliercio, "Performance comparison of VxWorks, Linux, RTAI, and Xenomai in a hard real-time application," *IEEE Transactions on Nuclear Science*, vol. 55, no. 1, pp. 435–439, 2008.

[9] P. Regnier, G. Lima, and L. Barreto, "Evaluation of interrupt handling timeliness in real-time linux operating systems," *SIGOPS Operating Systems Review*, vol. 42, no. 6, pp. 52–63, 2008.

[10] B. Bershad, R. P. Draves, and A. Forin, "Using microbenchmarks to evaluate system performance," in *Proceedings of the 3rd Workshop on Workstation Operating Systems*, pp. 148–153, 1992.

[11] A. C. Heursch, E. Horstkotte, and H. Rzehak, "Preemption concepts, rhealstone benchmark and scheduler analysis of linux 2.4," in *Proceedings of the Real-Time and Embedded Computing Conference*, 2001.

[12] H. J. Curnow and B. A. Wichmann, "A synthetic benchmark," *Computer Journal*, vol. 19, no. 1, pp. 43–49, 1976.

[13] N. Weiderman, "Hartstone: synthetic benchmark requirements for hard real-time applications," in *Proceedings of the Working Group on Ada Performance Issues*, pp. 126–136, 1990.

[14] B. G. Ujvary and N. I. Kameno, "Implementation of the hartstone distributed benchmark for hard realtime distributed systems: results and conclusions," in *Proceedings of the Joint Workshop on Parallel and Distributed Real-Time Systems*, pp. 98–103, 1997.

[15] L. R. Welch and B. A. Shirazi, "Dynamic real-time benchmark for assessment of QoS and resource management technology," in *Proceedings of the 5th IEEE Real-Time Technology and Applications Symposium (RTAS '99)*, pp. 36–45, June 1999.

[16] F. Nemer, H. Cassé, P. Sainrat, J. P. Bahsoun, and M. De Michiel, "Papabench: a free real-time benchmark," in *Proceedings of the Workshop on Worst-Case Execution Time*, 2006.

[17] T. Kalibera, J. Hagelberg, F. Pizlo, A. Plsek, B. Titzer, and J. Vitek, "CDx: a family of real-time java benchmarks," in *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES '09)*, pp. 41–50, September 2009.

[18] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: a free, commercially representative embedded benchmark suite," in *Proceedings of the Workload Characterization IEEE International Workshop*, pp. 3–14, 2001.

[19] S.-L. Tan and B. T. Nguyen, "Survey and performance evaluation of real-time operating systems (rtos) for small microcontrollers," *IEEE Micro*, vol. 99, no. 1, 2009.

[20] J. H. Brown and B. Martin, "How fast is fast enough? choosing between xenomai and linux for realtime applications," in *Proceedings of the Real Time Linux Workshops*, pp. 25–27, 2010.

[21] M. Piątek, "Real-time application interface and xenomai modified gnu/linux real-time operating systems dedicated to control," in *Proceedings of the 6th Conference on Computer Methods and Systems*, pp. 179–184, 2007.

[22] R. Gumzej, A. Halang, and W. Springer, *Real Time Systems Quality of Service. Introducing Quality of Service Considerations in the Life Cycle of Real-Time Systems*, Springer, 2010.

[23] Y. Zhang and A. Sivasubramaniam, "Scheduling best-effort and real-time pipelined applications on time-shared clusters," in *Proceedings of the 13th Annual Symposium on Parallel Algorithms and Architectures (SPAA '01)*, pp. 209–219, July 2001.

[24] S. Banachowski, T. Bisson, and S. A. Brandt, "Integrating best-effort scheduling into a real-time system," in *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS '04)*, pp. 139–150, December 2004.

[25] M. Dellinger, P. Garyali, and B. Ravindran, "ChronOS Linux: a best-effort real-time multiprocessor Linux kernel," in *Proceedings of the 48th ACM/EDAC/IEEE Design Automation Conference (DAC '11)*, pp. 474–479, June 2011.

[26] P. E. McKenney, "Real time vs. real fast: how to choose?" in *Proceedings of the Linux Symposium*, pp. 57–65, 2008.

[27] P. Mantegazza, E. L. Dozio, and S. Papacharalambous, "Rtai: real time application interface," *Linux Journal*, no. 72es, 2000.

[28] P. Gerum, "Xenomai-implementing a rtos emulation framework on gnu/linux," White Paper, Xenomai, 2004.

[29] S.-T. Dietrich and D. Walker, "The evolution of real-time linux," in *Proceedings of the 7th RTL Workshop*, 2005.

[30] D. Locke, L. Sha, R. Rajikumar, J. Lehoczky, and G. Burns, "Priority inversion and its control: an experimental investigation," *ACM SIGAda Ada Letters*, vol. 8, pp. 39–42, 1988.

[31] D. Calleja, "Linux 2.6.18," 2007, http://kernelnewbies.org/Linux_2_6_18.

[32] E. Bianchi and L. Dozio, "Some experiences in fast hard realtime control in user space with rtai-lxrt," in *Proceedings of the Real Time Linux Workshop*, 2000.

[33] R. Love, *Linux System Programming: Talking Directly to the Kernel and C Library*, O'Reilly Media, 2013.

[34] I. Puaut, "Real-time performance of dynamic memory allocation algorithms," in *Proceedings of the 14th Euromicro Conference on Real-Time Systems*, pp. 41–49, 2002.

[35] M. C. Lin and S. Gottschalk, "Collision detection between geometric models: a survey," in *Proceedings of the IMA Conference on Mathematics of Surfaces*, pp. 37–56, 1998.

[36] A. Gallrein and M. Bäcker, "CDTire: a tire model for comfort and durability applications," *Vehicle System Dynamics*, vol. 45, no. 1, pp. 69–77, 2007.

[37] H. B. Pacejka and E. Bakker, "Magic formula tyre model," *Vehicle System Dynamics*, vol. 21, no. 1, pp. 1–18, 1992.

[38] J. De Cuyper, M. Furmann, D. Kading, and M. Gubitosa, "Vehicle dynamics with LMS virtual. lab motion," *Vehicle System Dynamics*, vol. 45, no. 1, pp. 199–206, 2007.

[39] R. Capitani, G. Masi, A. Meneghin, and D. Rosti, "Handling analysis of a two-wheeled vehicle using MSC. ADAMS/motorcycle," *Vehicle System Dynamics*, vol. 44, no. 1, pp. 698–707, 2006.