Predictability and efficiency in contemporary Hard RTOS for multiprocessor systems

Andre Nógueira
Large-Scale Informatics Systems Laboratory
Informatics Department, Faculty of Sciences
Lisbon, Portugal
anogueira@lasige.di.fc.ul.pt

Mário Calha
Large-Scale Informatics Systems Laboratory
Informatics Department, Faculty of Sciences
Lisbon, Portugal
mjc@di.fc.ul.pt

***Abstract*—Hard real-time operating systems have been mostly designed for uniprocessors. Driven by the high computational demand of real-time systems, these operating systems have been re-designed for multiprocessors. Researchers have been identifying issues that affect the predictability of multiprocessor hard real-time systems. These sources of unpredictability make it difficult to accurately predict safe and tight worst-case execution time bounds of real-time tasks. As a result, the performance of multiprocessor hard real-time systems is compromised.**

In this paper, we present a middleware architecture that aims to provide predictability and high efficiency for multiprocessor hard real-time systems. The middleware supports hybrid hard real-time tasks, which are adaptive hard real-time services that intend to ensure predictability and performance. We also present a prototype of this architecture. Moreover, we identify and discuss issues that affect the predictability and performance of multiprocessor hard real-time systems.

I. Introduction

In hard real-time systems not meeting timing constraints leads to system failure. These systems are used when it is imperative that an event is processed to within a strict deadline. Such strong guarantees are required in systems for which not reacting in a certain interval of time would affect the system and cause great loss. Examples of hard real-time systems include traffic control, industrial control, robotics, avionics, etc.

Hard real-time operating systems have been designed for uniprocessors. Real-time applications with high computational demands are emerging and the processing requirements of these applications exceed the capacity of a single processor [1]. Multiprocessors provide more computational power than uniprocessors, meeting the processing requirements of high-performance real-time systems. Researchers have been identifying issues that need to be addressed in order to ensure predictability on multiprocessors. Modern multiprocessors have brought challenges for the following areas: architecture, scheduling, cache memory, execution, etc [2], [4], [5], [6], [3]. These issues make it difficult to accurately predict safe and tight worst-case execution time bounds of real-time tasks. The execution time of each real-time task is overestimated to ensure that the task meets its deadline. Consequently, the performance of the whole system is affected, increasing the overall cost.

In this paper, we present a middleware architecture that aims to provide predictability and high efficiency for multiprocessor hard real-time systems. This architecture follows a hybrid approach, offering support for different types of tasks such as hybrid hard real-time tasks. This type of task offers a adaptive hard real-time service that ensures performance and predictability, combining a demanding algorithm, which requires significant computational power, with a basic algorithm that meets the basic needs.

This paper is organized as follows: In Section II, we describe the execution time properties of a real-time task. In Section III, we present and discuss issues that compromise the predictability and performance of multiprocessor hard real-time systems. In Section IV, we present a middleware architecture that aims to provide predictability and high efficiency for multiprocessor hard real-time systems. We also present a prototype of the architecture. In Section V, we evaluate the prototype. Finally, in Section VI we present the conclusions.

II. Properties of a real-time task

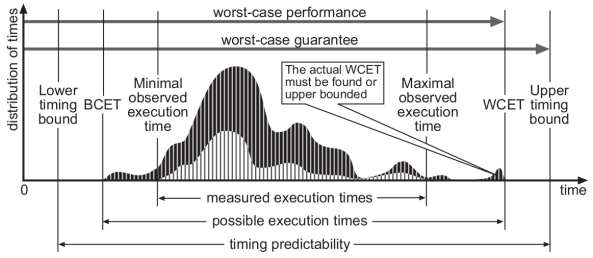

Figure 1. Execution time properties of a real-time task [7].

Figure 1 presents several properties of the execution time of a real-time task [7]. The execution times are represented in two curves. The lower curve represents a subset of measured executions, and the darker curve represents the times of all executions. The minimal and maximal observed execution times define an interval of end-to-end execution time for a subset of the possible executions. The best case (BCET) and worst case (WCET) set, respectively, the minimum and maximum possible execution times under all admissible system and environment states. The Lower Timing Bound (LTB) and

1533-2306/11 $26.00 © 2011 IEEE
DOI 10.1109/RTCSA.2011.56

the Upper Timing Bound (UTB) ensure that the system meets its timing requirements, bounding, respectively, the BCET and the WCET.

When a real-time task is executed several times, a major subset of all execution times occurs between the minimal and maximal observed bounds. These bounds are not safe for hard real-time systems because some executions still happen outside of them. The execution time variation between the BCET and WCET depends on the input data, the hardware and the code executing on the hardware. End-to-end measurements of program execution times under a limited amount of test cases are risky with regards to the reliability of having observed the worst case. As BCET and WCET bounds are not usually known, two other values, LTB and UTB, have to be derived through analysis. WCET analysis aims to find a safe UTB on the execution time.

There are two main approaches for WCET analysis: *static analysis*, which involves modelling of the target architecture; and *measurement-based techniques*, which requires execution time measurements taken with an assumed worst-case input. Static analysis may produce estimates for the WCET that are very pessimistic, because the presence of architectural features such as instruction caches, branch prediction and pipelining. In case of measurement-based techniques, the prediction of WCET bound may not be correct, because the execution time depends on target architecture state and may not be possible to enumerate all possible inputs for a task and measure each execution time. As a result, the WCET bound may have to be overestimated, because the execution time may exceed the WCET observed. By overestimating the execution time of the real-time tasks, the performance of the whole system is affected, because the computer resources will not be used efficiently.

III. Sources of unpredictability

In this section, we present and discuss issues that make it difficult to accurately predict a safe and tight WCET bound, thereby compromising the predictability and performance of multiprocessor hard real-time systems. We discuss in detail the following issues: multiprocessor architecture systems, pipeline optimization and cache memory. Moreover, we also discuss how multiprocessor scheduling algorithms affect the performance of hard real-time systems.

A. Multiprocessor architecture systems

A multiprocessor system is a system where more than one processor is working simultaneously. There are two general classes of shared memory multiprocessors [8]: Uniform Memory Access (UMA) architectures – also known as Symmetric MultiProcessors (SMP); and Non-Uniform Memory Access (NUMA). Also, a cache coherence mechanism can maintain coherency between the various copies of the same data in a NUMA – this architecture is know as cc-NUMA.

Comparing both classes of shared memory, NUMA scales better than UMA, however, the allocation of tasks is a major issue [10]. The latency and bandwidth limitations induced by the interconnection network may result in additional overhead when the scheduler tries to access remote scheduling information. In addition, the memory latency is a well-known issue in cc-NUMA, which affects the execution time of the tasks [11]. UMA architecture has a major disadvantage of not scaling well when the processors communicate using a shared communication infrastructure [2]. When several processors attempt to gain control of the bus, the time necessary to transfer data from memory to the cache is not fixed. The delay introduced depends on the time necessary to acquire control of the bus.

B. Pipeline optimization

Modern processors used in multiprocessors can resort to features, such as pipeline, out-of-order execution, and branch prediction, that permit an improvement in the overall execution speed [8]. The presence of such features enhances the performance of modern processors, however, it makes it difficult to accurately predict the WCET. The behaviour of the pipeline is difficult to model when a preemptive scheduling algorithm is used [12]. For instance, the overlapping execution of instructions in the pipeline results in an increasing execution time. Also, out-of-order execution has a severe impact on worst-case execution time analysis due to an event called *timing anomaly* [13]. In some cases a cache miss can result in a shorter execution time than a cache hit. As well, erroneous branch prediction results in an additional number of processor clock cycles, because the pipeline has to be flushed of the speculative instructions [14].

C. Cache

Cache memories are used for speeding up memory access, improving the overall performance of computer systems. However, caches are a source of unpredictability for hard real-time system. Cache interference is a phenomena which increases the memory traffic and the amount of communication [15]. Cache locking and cache partitioning are used to minimize the cache interference [16], [17].

There have been many research efforts on cache modelling for WCET analysis [7], [18]. Unfortunately, the existing techniques for uniprocessors are not applicable for multiprocessors due to the possible inter-task interferences in shared resources such as L2 caches [4], [5]. The inter-task cache conflicts in multiprocessors with shared cache memories can lead to timing anomalies. Thus, the WCET analysis of each task running on each processor cannot be performed independently, which can significantly increase the complexity of the timing analysis.

D. Scheduling

There have been two approaches for scheduling tasks in multiprocessors [3]: *global*, where each task is put in a queue of runnable tasks, which is shared by all processors, and the scheduler selects the task with the highest priority from this queue; and *partitioning*, where each task is assigned to a single processor and the processors are scheduled independently.

When the global scheduling approach is used with optimal uniprocessor scheduling algorithms, such as the Rate-monotonic (RM) and Earliest deadline first (EDF) algorithms, it may result in arbitrarily low processor utilization in multiprocessor systems. The global-scheduling algorithms based on the concept of *proportionate fairness* (Pfair) are well-known methods for optimally scheduling periodic, sporadic, and rate-based task systems on multiprocessors [20], [21]. However, the frequency of context switching and migration in Pfair-scheduled systems has led to some questions concerning the practicality of Pfair scheduling. On the other hand, the partitioning approach reduces a multiprocessor scheduling problem to a set of uniprocessor ones. However, finding an optimal assignment of tasks among a set of processors is an NP-hard problem. There are several assignment schemes that apply variants of heuristic bin-packing algorithms [22], [23]. Several polynomial-time heuristics such as First Fit and Best Fit have been proposed for solving this problem, although several complex heuristics may introduce unacceptable run-time overhead. Also, the total utilization of processors is not optimal, affecting the performance of the whole system.

IV. Wormhole model in hard real-time context

In this Section, we present a middleware architecture, inspired in the Wormhole model [24], that aims to provide predictability and high efficiency for multiprocessor hard real-time systems. First, we describe the Wormhole model. Next, we present the middleware architecture and explain in detail each component. Then, we present a prototype implementation of the architecture.

A. Wormhole model

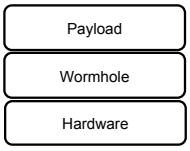

Figure 2. Wormhole model

Figure 2 presents the Wormhole model [24], [25], which was proposed for distributed systems, and consists in making some stronger properties happen in a well-defined and safe way, while preserving the canonical model's weak abstractions. The Wormhole model divides the whole system in two subsystems: wormhole and payload. The wormhole subsystem is synchronous and runs simple and robust algorithms. The payload subsystem may be asynchronous, runs complex and intelligent algorithms, which might not be deterministic and the computation time may vary.

In the Wormhole model, each application is divided in two subtasks: payload and wormhole. The payload subtask runs in the payload subsystem, executes complex algorithms, and may use sophisticated resources that meet the optimal needs of the application. The wormhole subtask runs in the wormhole subsystem, executes simple algorithms, and uses the appropriate resources that meet the basic needs. When the application initiates, the payload subtask starts by sending a promise to the Timely Timing Failure Detector (TTFD). TTFD is a service responsible for confirming if the payload subtask meets its promise. The promise is a time instance when the payload subtask expects to finish its execution, and it is calculated based on the conditions of the payload subsystem. TTFD launches the wormhole subtask when the payload subtask has not met its promise. Otherwise, the wormhole subtask is not launched. The performance and the timeliness requirements of the applications are ensured, respectively, by the payload and wormhole subtasks.

B. Middleware Architecture

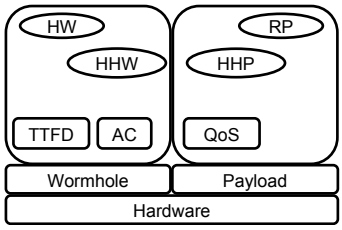

Figure 3. A middleware architecture for multiprocessor hard real-time systems.

Figure 3 illustrates the various components of our middleware architecture. This middleware supports the following types of tasks. The regular task (RP) runs in the Payload, without any real-time requirement. The hard real-time task (HW) runs in the Wormhole, having hard real-time requirements. The hybrid hard real-time task provides a hard real-time service with two modes: basic and optimal. The hybrid task is implemented using two subtasks: HHW, which runs in the Wormhole, executes a basic algorithm that meets the basic needs of the hybrid task; and HHP, which runs in the Payload, executes a demanding algorithm that requires significant computational power, ensuring the optimal needs of the hybrid task.

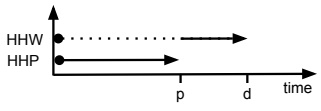

Figure 4. The behaviour of a hybrid hard real-time task.

Figure 4 illustrates the behaviour of a hybrid hard real-time task. HHP begins by making a promise (p) to TTFD. p is the time instant when HHP expects to finish its execution. TTFD waits until p has expired and starts by verifying if HHP has finished its execution. HHP is responsible for meeting the deadline (d) of the hybrid task when its execution finishes before p. When p is reached and HHP is still processing, HHW is launched by the TTFD and meets d.

Figure 5 presents the execution time of a hybrid hard real-time task. We assume p is a time instant between the minimal

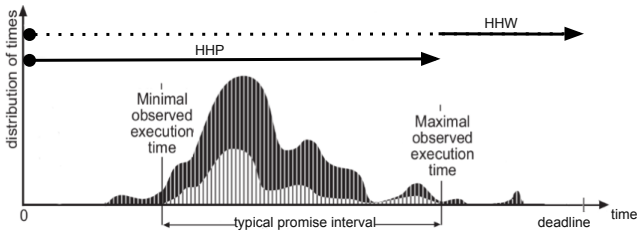

Figure 5. Execution time of a hybrid hard real-time task.

(*minoet(HHP)*) and the maximal observed (*maxoet(HHP)*) execution times of HHP (1).

$$minoet(HHP) \leq p \leq maxoet(HHP) \quad (1)$$

HHP subtask predicts p based on d of the hybrid task and other criteria. These criteria may be based on the Payload conditions or on the needs of the hybrid task. For instance, when the computer resources, where the Payload is running, are overloaded, HHP adapts p in order to guarantee that HHW meets d, ensuring the basic mode of the hybrid task. Otherwise, HHP changes p, ensuring that it meets d and guarantees the optimal mode.

We define the worst-case computation time of HHW (*C(HHW)*) as follows:

$$C(HHW) \leq d - p \quad (2)$$

Eq.(2) guarantees that when HHP fails its p, HHW finishes its execution before d. As HHW runs in the Wormhole, which is a more predictable environment than the Payload, and executes basic algorithms that meet the basic needs of the hybrid task, the prediction of a WCET bound for HHW must be safe not tight.

The Wormhole, where HW and HHW tasks run, is a minimal hard real-time system where all the services available have a time-bounded response. The Wormhole runs in dedicated processors where mechanisms and algorithms that provide more predictability are applied. Cache locking or cache partitioning may be used for improving execution time predictability for HW and HHW tasks. Inside the Wormhole runs also two main services: Timely Timing Failure Detector (TTFD), which is a service responsible for receiving the promises made by the HHP subtasks, and for launching the corresponding HHW subtasks when their promises are not met; and Admission Control (AC), which is a service responsible for controlling the admission of HW and HHW tasks. AC uses data from the tasks, such as execution time and deadline, and from the Wormhole, such as context switching latency overhead, AC ensures that each task meets its timeliness requirements.

The Payload, where RP and HHP tasks run, may be a COTS RTOS which offers soft real-time requirements. The Payload runs in processors where the Wormhole is not executing, providing a variety of complex services and sophisticate resources. The Quality of Service (QoS) is a service responsible for keeping the high efficiency of the computer resources where the Payload is running. QoS monitors the Payload activities and acquires data from the Wormhole, for increasing the performance of HHP tasks, measuring for each one how many times the task has not met its p. QoS changes the task properties, such as priority and memory used, and adapts p of the task, offering conditions to the task to finish its execution before p.

C. Prototype implementation

Figure 6 presents a prototype implementation of our middleware architecture in a dual-core multiprocessor system (Table I).

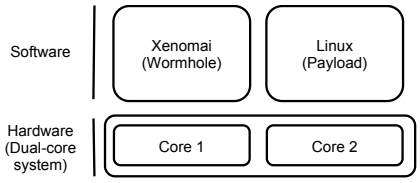

Figure 6. A prototype of our middleware architecture.

We chose a real-time development framework called Xenomai [26], which offers hard real-time properties and multiprocessor support (SMP-capable), for performing as a Wormhole. All the services available in Xenomai, such as task and alarm services, were designed to have deterministic response time. The Xenomai scheduler supports fixed priority-based FIFO and round-robin policies and task processor affinity. In addition, Xenomai is integrated seamlessly into Linux, enabling communication between Linux and Xenomai in a time-bounded fashion. Thus, we chose a Linux environment for performing as a Payload.

Table I
PROCESSOR FEATURES

CPU	Intel Core 2 Duo -T6400
Speed	2.00GHz
Cache L1	Instruction: 32KB, Data: 32KB
Cache L2	2MB

The Linux kernel offers multiprocessor support (SMP-capable) and soft real-time scheduling [27]. The Linux scheduler arranges processes within a constant amount of time (O(1)), giving to the real-time task priority over any other task on the system. The real-time tasks are scheduled with FIFO or round-robin polices. Additionally, the Linux kernel is a fully preemptive kernel. The scheduler is capable of rescheduling a task while it is in the kernel.

In our prototype, we isolated Xenomai in Core 1 and Linux in Core 2, changing the interrupts and CPU affinity of Linux. We implemented the HW and HHW tasks using, respectively, the task and alarm services available in Xenomai. The alarm services are watchdog timers that permit the alarms to be launched after a specific initial delay. We implemented HHP task promise using the initial delay. The HHP tasks were implemented as real-time tasks of Linux, and RP tasks as Linux tasks.

Our prototype was developed to demonstrate the availability of hybrid hard real-time tasks, so TTFD, AC and QoS services were not yet implemented. A technical report [28] describes in detail the necessary steps to implement our prototype.

V. Evaluation

In this section, we evaluate our prototype and assumptions by running experimental tests. We start first by describing in detail the tests, and then we present and discuss the results.

A. Experimental tests

We tested our prototype by developing two experimental tests that aimed to verify that a hybrid hard real-time task guaranteed predictability and improved system performance.

In the first test, we created a hybrid hard real-time task by developing two subtasks: HHW, which ran in Xenomai, simulated the execution of a basic algorithm (Algorithm 1); and HHP, which ran in Linux, simulated the execution of a demanding algorithm (Algorithm 2). Each algorithm simulated processing data using a 8KB data buffer, which fitted in cache L1, for processing data. Algorithm 2 iterated the data in buffer more times than Algorithm 1 in order to simulate a demanding algorithm.

Algorithm 1 Basic algorithm used by the HHW subtask

```
1: maxElems ← (1024 * 8);
2: buffer ← buffer[maxElems];
3: loops ← 200;
4: elems ← maxElems;
5: x ← 0;
6: y ← 0;
7:
8: Begin.
9: while x < loops do
10:   while y < elems do
11:     buffer[y] ← buffer[(elems − 1) − y] * 999 + 1;
12:     y ← y + 1;
13:   end while
14:   y ← 0;
15:   x ← x + 1;
16: end while
17: End.
```

Next, we measured the end-to-end execution time of both subtasks, running each subtask 100000 times. Table II shows the minimal and maximal observed execution times of HHW and HHP. These measurements are not safe for hard real-time systems, although these tests were designed to evaluate the existence of hybrid hard real-time tasks in the proposed prototype.

Then, we analysed the distribution of execution times of the HHP to define the promise (*p*). Table III shows that 98% of all executions finished before 29.41 ms.

Thus, we assumed that HHP announced to HHW a *p* of 29.41 ms. Only in 2% of all executions, HHW had to run, offering the basic needs of the hybrid task. In the remaining executions, HHP ensured the optimal needs. We defined *p* based on the distribution of execution times, although other criteria might be used to define *p*. Finally, we defined that the hybrid task was periodic and its deadline was equal to its period (60 ms).

Algorithm 2 Complex algorithm used by the HHP subtask

```
1: maxElems ← (1024 * 8);
2: buffer ← buffer[maxElems];
3: loops ← 545;
4: elems ← maxElems;
5: x ← 0;
6: y ← 0;
7:
8: Begin.
9: while x < loops do
10:   while y < elems do
11:     buffer[y] ← buffer[(elems − 1) − y] * 999 + 1;
12:     buffer[(elems − 1) − y] ← 1023 * x * buffer[y];
13:     y ← y + 1;
14:   end while
15:   y ← 0;
16:   x ← x + 1;
17: end while
18: End.
```

Table II
The minimal and maximal observed execution times of HHW and HHP.

	Minimal (ms)	Maximal (ms)
HHW	6.13	9.07
HHP	28.79	53.25

Table III
Execution times distribution of HHP.

C(HHP) (ms)	[28.79, 29.41]	]29.41, 53.25]
Number of executions	98642	1358

In the second test, we repeated the first test, adding a workload generator called *stress* [29], which imposes a configurable amount of CPU, memory and disk stress into Linux, as a RP task. Using the workload generator, we aimed to overload the computer resources where the Linux was running.

B. Results

Table IV presents the results of the tests. We measured, for each test, how many times each subtask met *d* of the hybrid task.

Table IV
Number of times each subtask had to meet *d*

	Test 1	Test 2
HHP	98602	90642
HHW	1398	9358

In the first test, the HHP subtask finished most times its

execution before p, meeting d and providing the optimal needs of the hybrid task. As HHP ran in Linux as a real-time task and the computer resources where Linux was running were not overloaded, the HHW subtask ran and ensured d a few times, guaranteeing the basic needs of the hybrid task.

In the second test, we added a RP task in Linux, which was responsible for imposing kernel stress, overloading the computer resources where the Linux was running, such as pipelining and cache memory. Consequently, HHW had to run and met d more times, offering the basic needs of the hybrid task. The execution time of HHP was affected because, for instance, when HHP started its execution the data buffer might not be allocated in cache L1, and before starting the processing, the buffer copied from cache L2 or from main memory, increasing the execution time of the task.

VI. Conclusions

The predictability of hard real-time systems is compromised by modern multiprocessors. The hardware engineers are continuously improving the throughput of multiprocessors, for instance, changing the architecture or increasing the number of resources available, such as the number of processors. As a consequence, WCET analysis of multiprocessors may not accurately predict safe and tight WCET bounds of real-time tasks. Thus, the problem of ensuring predictability on multiprocessors while offering high efficiency of hardware resources is becoming a greater challenge.

We presented a middleware architecture that aims to provide predictability and high efficiency for multiprocessor hard real-time systems. Our middleware supports hybrid hard real-time tasks, which are adaptive hard real-time services that intend to ensure predictability and performance. Another benefit is that the prediction of a WCET bound for a hybrid real-time task must be safe and not tight, reducing the required effort of the WCET analysis. We have also presented and evaluated a prototype of our architecture.

As future work, we will modify Xenomai in order to make it a minimal hard real-time system, and apply a cache technique that minimizes the cache interference, improving the predictability.

References

[1] K.-J. Lin, Y. Wang, T.-H. Chien, and Y.-J. Yeh, "Designing multimedia applications on real-time systems with smp architecture," pp. 17–, 2002. [Online]. Available: http://portal.acm.org/citation.cfm?id=824463.824791

[2] J. Rosen, A. Andrei, P. Eles, and Z. Peng, "Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip," pp. 49–60, 2007.

[3] J. C. S. Funk, "A categorization of real-time multiprocessor scheduling problems and algorithms," in *Handbook on Scheduling Algorithms, Methods, and Models*. Chapman Hall/CRC, Boca, 2004.

[4] J. Yan and W. Zhang, *WCET Analysis for Multi-Core Processors with Shared L2 Instruction Caches*. Washington, DC, USA: IEEE Computer Society, 2008.

[5] N. Guan, M. Stigge, W. Yi, and G. Yu, "Cache-aware scheduling and analysis for multicores," in *EMSOFT '09: Proceedings of the seventh ACM international conference on Embedded software*. New York, NY, USA: ACM, 2009, pp. 245–254.

[6] C. Berg, J. Engblom, and R. Wilhelm, "Requirements for and design of a processor with predictable timing," 2004.

[7] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 1–53, 2008.

[8] A. S. Tanenbaum, "Structured computer organization (5th edition)," 2005.

[9] N. B. Lakshminarayana, J. Lee, and H. Kim, "Age based scheduling for asymmetric multiprocessors," New York, NY, USA, pp. 1–12, 2009.

[10] H. Zhou, K. Schwan, and I. Akyildiz, "Performance effects of information sharing in a distributed multiprocessor real-time scheduler," pp. 46 –55, dec 1992.

[11] T. N. Nguyen, Z. Li, J. Huang, G. Jin, and D. Kim, "Performance evaluation of memory allocation schemes on cc-numa multiprocessors."

[12] N. Zhang, Z. Burns, and M. Nicholson, "Pipelined processors and worst case execution times," *Real-Time Systems*, vol. 5, pp. 319–343, 1993.

[13] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker, "A definition and classification of timing anomalies," in *6th Intl Workshop on Worst-Case Execution Time (WCET) Analysis*, 2006.

[14] A. Colin and I. Puaut, "Worst case execution time analysis for a processor with branch prediction," *Real-Time Syst.*, vol. 18, no. 2/3, pp. 249–274, 2000.

[15] O. Temam, C. Fricker, and W. Jalby, "Cache interference phenomena," pp. 261–271, 1994. [Online]. Available: http://doi.acm.org/10.1145/183018.183047

[16] X. Vera, B. Lisper, and J. Xue, "Data cache locking for higher program predictability," in *SIGMETRICS '03: Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM, 2003, pp. 272–282.

[17] A. Arnaud and I. Puaut, "Dynamic instruction cache locking in hard real-time systems," May 2006.

[18] P. Puschner and A. Burns, "Guest editorial: A review of worst-case execution-time analysis," *Real-Time Systems*, vol. 18, pp. 115–128, 2000, 10.1023/A:1008119029962. [Online]. Available: http://dx.doi.org/10.1023/A:1008119029962

[19] G. C. Buttazzo, "Rate monotonic vs. edf: judgment day," *Real-Time Syst.*, vol. 29, no. 1, pp. 5–26, 2005.

[20] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. Varvel, "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 15, pp. 600–625, 1994.

[21] A. Srinivasan, P. Holman, J. H. Anderson, and S. Baruah, "The case for fair multiprocessor scheduling," in *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2003, p. 114.1.

[22] D.-I. Oh and T. Bakker, "Utilization bounds for n-processor rate monotone scheduling with static processor assignment," *Real-Time Systems*, vol. 15, pp. 183–192, 1998, 10.1023/A:1008098013753. [Online]. Available: http://dx.doi.org/10.1023/A:1008098013753

[23] J. M. López, J. L. Díaz, and D. F. García, "Utilization bounds for edf scheduling on real-time multiprocessor systems," *Real-Time Systems*, vol. 28, pp. 39–68, 2004, 10.1023/B:TIME.0000033378.56741.14. [Online]. Available: http://dx.doi.org/10.1023/B:TIME.0000033378.56741.14

[24] P. Verissimo, "Travelling through wormholes: a new look at distributed systems models," *SIGACT News*, vol. 37, no. 1, pp. 66–81, 2006. [Online]. Available: http://www.navigators.di.fc.ul.pt/docs/abstracts/verissimo06travelling.html

[25] L. Marques, A. Casimiro, and M. Calha, "Design and development of a proof-of-concept platooning application using the hidenets architecture," in *Proceedings of the 2009 IEEE/IFIP Conference on Dependable Systems and Networks*, Estoril, Lisboa, Portugal, Jun. 2009, pp. 223–228.

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

[27] J. Aas, "Understanding the linux 2.6.8.1 cpu scheduler," 2 2005. [Online]. Available: http://josh.trancesoftware.com/linux/linux_cpu_scheduler.pdf

[28] A. Nogueira, "Installation of xenomai in multicore environment," 2011. [Online]. Available: http://dl.dropbox.com/u/8908271/xenomai-multicore-conf.pdf

[29] A. Waterland, "stress," http://weather.ou.edu/~apw/projects/stress//.