Design and Test of the RT-NKE Task Scheduling Algorithm for Multicore Architectures

Renato Severo, Celso Maciel da Costa, Adriane Parraga, Debora Motta
State University of Rio Grande do Sul - Brazil
{renatosevero, celsocostars, adriane.parraga, debora.motta}@gmail.com

Ivan Muller
Electrical Engineering Dept.
Federal University of Rio Grande do Sul, Brazil
ivan.muller.poa@gmail.com

Fabian Vargas
Electrical Engineering Dept.
Catholic University – PUCRS, Brazil
vargas@computer.org

Abstract — This paper presents the design and testing of a partitioned task-scheduling algorithm for the RT-NKE real-time operating system (RTOS). In this implementation, there is a scheduling queue for each core. The rate-monotonic (RM) scheduler already used in the single-core version of RT-NKE runs in each queue. The implementation of the algorithm was validated by porting the RT-NKE code to a BeagleBone board. To guarantee the quality of the test process, unitary tests were created using the GTEST library. As results of the work, we ported RT-NKE to the BeagleBone board and implemented the real-time task scheduling algorithm. Since RT-NKE only runs on single-core architectures, an environment was created that simulates the execution of the algorithm in a multi-core platform with the processor alternating the execution of the tasks in the different scheduling queues.

Keywords— real-time operating system, partitioning-based scheduling task, unitary test, multicore architecture, embedded application, RTOS RT-NKE.

I. INTRODUCTION

The computing scene has changed from predominantly personal computers and servers to different platforms with special requirements distinct from those demanded previously [1]. With the low cost and miniaturization of components, these days, embedded systems are present in a wide variety of application areas such as automobiles, airplanes, smartphones, telecommunications equipment [2]. With broad requirements for new features and timing issues, the use of single-core architectures has become unfeasible [3] for reasons of energy consumption and heat dissipation. With higher density microchips, that is, the ability to place a larger number of logical cells in the same area of silicon, the need for processing power is significant. Moreover, this has motivated the evolution of single-core processors to processor architectures with dozens of cores. One example is the Parallella board [4], which has up to 64 cores. According to [5], this board is in the top ten most commonly used. However, this increase in processing capacity via increasing the number of cores has given rise to some challenges that impact how an application is programmed as well as how the operating system manages the resources of this multi-core architecture.

In the context of real-time operating systems, multi-core architectures need new paradigms for architecture and organization yet to be reviewed. One of the fundamental software components that is dramatically impacted by multicore architectures is the scheduler, which implements the task scheduling algorithm. The scheduling algorithms define the ordering of tasks in the ready queue. The scheduling algorithms define the ordering of tasks in the ready queue. They are central components in real-time embedded systems, since the ordering must guarantee the execution of multiple tasks considering temporal constraints associated with each one of the tasks. In the context of multi-core scheduling algorithms, there are three primary types: global scheduling, partitioned, and semi-partitioned scheduling [6].

This article presents the development of a multicore partitioned scheduling algorithm. In the implementation, there is a queue of tasks for each core and there is no migration, i.e., a blocked task should always be placed in the queue of the core in which it was being executed. In the sequence, this work also describes the test environment that was developed to simulate the execution of the partitioned algorithm in a single-core platform, alternating the execution of the tasks in different queues. A contribution of this paper was the implementation of a partitioned scheduling algorithm for multicore embedded real time systems. In addition, another contribution is the development of a software testing environment to aid in the correct implementation of the algorithm. Preliminary concepts are presented in Section II.

Financial Support from FAPERGS - Process Number 16/2551-0000524-9

978-1-5386-1472-3/18/$31.00 ©2018 IEEE

The RT-NKE scheduling, the implementation of the RM and EDF algorithms, the design and implementation of the Task Scheduling Algorithm for Multicore Architecture, the methodology of the tests and the test effort to validate the proposed approach for multicore platform are presented in Sections III to VI, respectively. Then, final considerations are presented at Section VII.

II. PRELIMINAIRES

Real-time scheduling algorithms for multicore architectures can be classified into global, partitioned, and semi-partitioned [7], [8], [9].

In global scheduling there is only one queue where tasks ready to be executed are placed according to their priorities. In this kind of scheduling, the usage of greater number of cores is better since if there is a task available to be executed, then no cores of the server will be idled. In the multi-core partitioned scheduling there is a queue of ready-to-run tasks for each available core in the system. In each queue, a single-core scheduling algorithm can be executed, such as Rate Monotonic (RM), for example. In this approach, the use of cores is not optimal [10], [11] that is, it may be an idle core. It may happen if a task is ready to be executed in the queue of another core, and there is no task migration between the cores. The semi-partitioned algorithms have a merged approach, that is, there is an algorithm for assigning the static training cores, before the execution of the system. During execution, there is also an algorithm that can migrate tasks between the systems cores, avoiding cores to be idled.

RT-NKE is a real-time nanokernel [12], [13] that was developed for critical applications whose time is of fundamental importance. Developed in C language, RT-NKE has support for ARM architectures. It runs in processor supervisory mode, so you can access any available register. It provides the user with some functionality that allows the creation of tasks that have real-time requirements. Among the existing functionalities, we can cite the following: Creation and deletion of tasks, primitives for time management and, also, primitives for the synchronization of tasks. Currently the following scheduling algorithms are available: Round Robin, Rate Monotonic and Earliest Deadline First.

Among the main features of RT-NKE, we can highlight the task creation model, which allows to write a function in the C language and transform it into a periodic task, with the tasks creation primitive. Such same primitive has computational time and period parameters associated to each task, used by the RM and EDF scheduling algorithms. Another aspect to point out is the size of the RT-NKE code, whose executable has only 38Kbytes, which allows for easy and fast learning and, consequently, allows to include new features at a minimum effort in a short time. The development of scheduling algorithms for the mono-core version of RT-NKE expose the complexity related to the elaboration of tests of scheduling algorithms. One of the problems that arises is the correct definition of the tests that must be performed. Incorrect planned tests lead to unpredictable results and false interpretations of these results. In addition, scheduling algorithms are triggered by events, usually generated by hardware or software interrupts (system calls). If the scheduling algorithm does not run correctly, the system has a fatal error. To guarantee the quality of the test process, unit tests were created using the GTEST library.

III. RT-NKE TASK SCHEDULING

Each task in RT-NKE is represented by a task descriptor as shown in Fig.1. Tid stores an integer value that identifies the task whose priority is stored in Prio and the current state in the State field. If the value of Time field is other than zero indicates the lock time task. The EP field contains the start address of the task (entry point), SP is the address of the top of the stack, and Stack is a vector of 120 integer elements (four bytes), which is the execution stack of the task. The Period field stores the original value of the task period, the PeriodCount field contains the value of the period, which is decremented at each timer interrupt. The Comp field has the original value of the computational time of the task, the CompCount field (of the running task) has the value decremented at each interrupt of the timer and the Algorithm field defines the algorithm that will be used to select the task. The Proc field identifies the core in which the task is associated. It was added to allow the implementation of the multicore partitioned algorithm.

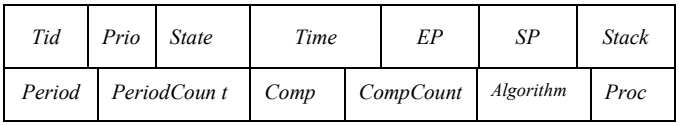

Fig. 1. Task descriptor of NKE.

The RT-NKE supports the scheduling algorithms RR (Round Robin)[14], RM (Rate Monothonic) and EDF (Earliest Deadline First). The system has a Real-Time scheduling queue for real-time tasks and another Round Robin. The Real-Time queue has priority over the Round Robin queue (only runs a Round Robin task if there is no real-time task) and the same algorithm (RM or EDF) must scale all real-time tasks. The Start () system call, called at the beginning of the application's execution, organizes the scheduling queues according to the algorithm, which can take the RR, RM, or EDF values. In the case of the RM and EDF algorithms, the scheduling queue is organized taking into account the priority of each task, stored in the task descriptor. After this organization, the first task is triggered and starts counting the periods (of all the tasks) and the computation time, of the running task.

Two task scheduling routines are Dispatcher and Select. The Dispatcher is called by Select and by the blocking system calls. This routine triggers the execution of the first task of the ready queue. In the second, the order of the tasks is updated and the Dispatcher chooses the first one of the queue to execute. Select is called by Start at the tasks initialization and by Interrupt on the occurrence of a timer interrupt.

The RR scheduling algorithm is implemented by a circular queue. The insert is performed at the end of the queue and the selected task is the first in the queue. Tasks do not have priorities and the selected task executes for a slice of time, decremented with each interruption of the MCU timer. At the end of the time slot, the running task is inserted at the end of the queue and the first one in the queue is triggered.

It is necessary to perform the scheduling of the tasks when a timer interrupt occurs. The UpdateRR function checks which task is running. In the case of Idletask (taskrunning = 0), it continues running until a new task is inserted into the ready queue. Otherwise, the first ready queue task will be selected.

IV. RM AND EDF – ALGORITHMS IMPLEMENTATION

The Monotonic Rate is a preemption scheduling algorithm that assigns fixed priorities to independent periodic tasks. The deadline of each task coincides with its period, the computation time is constant and the switching time is considered null. Priority is assigned according to the period of each task: the shorter the period, the higher the priority [8]. As the algorithm is preemptive, a running task loses the processor by the arrival of a higher priority task in the system.

Tasks are created in the main function of the application and triggered by the system call Start, which is responsible for blocking the main task and inserting all the tasks created by the TaskCreate primitive in the ready queue. If the algorithm used is the RM or the Earliest Deadline First (EDF) [8], the ordering of the tasks in the queue according to the priority is performed, which is assigned according to the period, parameter of the task creation operation.

In the RM it is necessary to update the period and computational time counters, which is done by the *UpdateRM* routine. The task should return to the ready queue when it reaches the period, if its *CompCount* field is zero. If a new period occurs and *CompCount* is not zero it means the deadline loss of the task. If a task with a higher priority restarts the processor (occurrence of a new period), the current task is reinserted into ready queue. The period counter update occurs in all system tasks (excluding *idletask* and main), and the computational time counter is updated in the task that is running. If the computational time of a task is finished, it is blocked and will be able to dispute processor again when a new period occurs.

Earliest Deadline First (EDF) is a priority-based, preemptive dynamic scheduling algorithm. The tasks are periodic and independent, have deadline equal to the period and constant computing time. The switching time between tasks is considered null. Task priorities are assigned dynamically, with the priority being the one that is closest to reaching the deadline. Whenever a task is inserted into the ready queue, the algorithm recalculates the priority of each task and performs the preemption if necessary.

The EDF implementation differs from RM only by the field used in the ordering: instead of the Prio field, the PeriodCount field is used, which is the deadline counter. In UpdateEDF, the update of the PeriodCount field of the descriptor of each task is performed first, and then the period occurs. In this case, the task is placed in an auxiliary queue and later inserted into ready_queue. When all of the fields are updated, the tasks with period occurrence and the running task return to the ready queue.

V. DESIGN AND IMPLEMENTATION OF THE TASK SCHEDULING ALGORITHM FOR MULTICORE ARCHITECTURES

As mentioned in the previous section, the static approach of the multicore partitioned algorithm is equivalent to running a single core algorithm. That is, the tasks are assigned statically to a specific core at system startup and remain running in that core until the end of its run time. Each time the task is launched, that is, every time the task period occurs, it is always served by the same core.

The use of this approach can lead to a non-optimal use of the system, because at a given moment there may be tasks waiting in the ready queue of a core while the list of ready tasks of another core may be empty. However, there is no task migration between cores, which minimizes context switching and hence the time spent by the scheduler taking a task out of the current context, fetching another in the ready queue, and restoring the context of this new task

The following function, as we can see in Figure 2, belongs to the nanokernel API and implements the task creation functionality. The task starts executing after using another API call, the start call.

```
void taskcreate(int ID, void (funtion)(),
                unsigned int Period,
                unsigned int Computacional,,
                int algorithm
                )
```

Figure 2: Task create system call

The ID parameter is a value to used to identify the task within the system. The next parameter, (* function) (), is a pointer to the function that contains the task code, that is, this is the function that will be executed when the scheduler selects the task to execute. Period is the launch period of the task, that is, the time interval in which the task should be put back into the ready list. The Monotonic Rate is being used in such a way that the task priority is assigned according to the task period. The computation is performed after all tasks are created by the user and is performed inside the start function. The Computational parameter is the computation time of the task in units of cycles of the core.

For the selection of the core to which the task will be assigned, the algorithm will make the choice alternately, that is, each time the *taskcreate* function is called, the task will be tied to one of the cores and next to another core and so on.

Since in this work a BeagleBone single core was used, the selection of the core ends up representing which ready list the task will be included. To be able to implement the scheduling in a second core, a second ready list was created that represents the existence of a second core. Considering Figure 7, when a timer interrupt occurs, the Select function is called that is responsible for selecting the next task to be executed. Each call of the select function selects the task from one of the ready lists. In this way, the tasks of the two queues are executed alternately.

VI. TEST AND VALIDATION OF THE PROPOSED ALGORITHM

Embedded Systems are traditionally developed using a waterfall model[15]. In this model, the first step is the identification of the system requirements. Another step in this approach is the system development and the final step is the test phase (QA – Quality Assurance). The drawback of this approach is that the tests are executed only at the final step of the project where in some circumstances can be shortened dramatically to meet time-to-market.

With the usage of agile methodologies[15], such as Scrum, XP, the tests strategies have evolved to an approach where the tests are done in an earlier stage in the project. Studies indicate that the later a defect is found in the software, the higher is the project cost[16]. The TDD – test-driven development - is the alternative to an early test in the software development process. In this methodology, the test cases are first created and then, the code with the application logic.

The philosophy of TDD is based on the theory of writing a test prior to the implementation of the functionality that is to

be tested, that is, is written the test that should fail. After that, the functionality code can be implemented until the tests run successfully. Then it is possible to go to the refactoring code phase [16]. Figure 3 presents the flow chart using TDD.

However, there are two discussion to be considered when developing a project in the context of an embedded system. One is the development environment, usually called the host, where code is implemented and compiled. The other one is the device where the software will run, usually called target. Target can have a different architecture from the host, having hardware components not presented in the host, such as several kinds of sensors and voltage sequencing. In addition, this hardware availability is restricted due to costs, since fabrication of each hardware item increases cost to the companies. Therefore, the use of this kind of hardware for unitary tests is quite costly. However, for system test is crucial.

A. *Test Infrastructure*

As mentioned in the previous section, the use of the target device to execute unitary test is costly. An alternative for this issue is the use of unitary test with the creation of mocks. Mocks are software components that simulate a device behavior being tested. If the target has a temperature sensor, for example, a mock can be create to simulate a temperature value; and also different temperatures, including negatives values, what would be hard to achieve in a real environment.

In this work, we used mocks to simulate the register access of the BeagleBone processor. In this way it is possible to use the same code executed in the hardware for the tests, only modifying the hardware access functions implemented to mock. Figure 4 shows part of the proposed mocks. *StartKernel*, for example, is used in both unitary test and RT-NKE normal operation. This function is responsible for the initialization of the timers based on *IRQTimer2*, *InitTimer2*, *InitWdtfunctions*. Also, this function changes the context to the main function. The following files were created for the mocks implementation: *dmTimerCountermock.c*, *salvamock.c* e *wdtmock.c*, based on the same prototype of the *StartKernel*. The difference appears in the compilation time, when the unitary tests are compiled and the following files are added: dmTimerCounter.c mock, salvamock.candwdtmock.c. When the RT-NKE is compiled, the added files are: *dmTimerCounter.c*, *salva.S* e *wdt.c*, which are the real implementation.

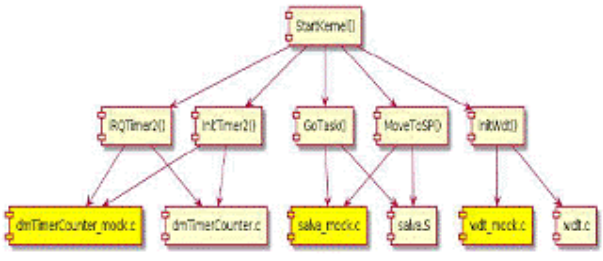

Figure 4: Mocks implementation structure

We used the library GTEST to assist the verification of the returns of function and operating conditions of the system. This library has some *assertions* such as EXPECT EQ(val1, val2), that causes the test to fail if the values of the expressions are not true.

B. *Test Cases*

Test based on mocks help to anticipate problems and enable the software writing without having the real hardware. However, it is not eliminate the need of hardware testing.
In this work, we developed some test case to guarantee system functionality. The first test proposed assuresthe system initialization, the main and idle tasks were created, and also the readylistinitialization. The way of readylist is inserted is the step that distinguish one scheduling algorithm from another. To guarantee the functionality in the list insertion, we created tests to make sure the insertion was correctly.
To test the task selection from the readylist, we created a test case that simulate the scheduler call and select the next task to be executed. Figure 5 shows the result of successful execution of the test case proposed to cover the RT-NKE implementation. We will explaine a test case in detail.

```
[==========] Running 9 tests from 2 test cases.
[----------] Global test environment set-up.
[----------] 8 tests from NanoKernelTest
[ RUN      ] NanoKernelTest.first
[       OK ] NanoKernelTest.first (0 ms)
[ RUN      ] NanoKernelTest.InsertReadyList_equal
[       OK ] NanoKernelTest.InsertReadyList_equal (1 ms)
[ RUN      ] NanoKernelTest.InsertReadyList_inc_seq
[       OK ] NanoKernelTest.InsertReadyList_inc_seq (1 ms)
[ RUN      ] NanoKernelTest.InsertReadyList_dec_seq
[       OK ] NanoKernelTest.InsertReadyList_dec_seq (1 ms)
[ RUN      ] NanoKernelTest.InsertReadyList_neg_seq
[       OK ] NanoKernelTest.InsertReadyList_neg_seq (4 ms)
[ RUN      ] NanoKernelTest.sys_start
[       OK ] NanoKernelTest.sys_start (2 ms)
[ RUN      ] NanoKernelTest.sys_start_2
[       OK ] NanoKernelTest.sys_start_2 (0 ms)
[ RUN      ] NanoKernelTest.sys_start_3
[       OK ] NanoKernelTest.sys_start_3 (0 ms)
[----------] 8 tests from NanoKernelTest (12 ms total)
[----------] 1 test from NanoKernelFunc
[ RUN      ] NanoKernelFunc.first
[       OK ] NanoKernelFunc.first (1 ms)
[----------] 1 test from NanoKernelFunc (1 ms total)
[----------] Global test environment tear-down
[==========] 9 tests from 2 test cases ran. (15 ms total)
[  PASSED  ] 9 tests.
```

Figure 5: Test results using the GTEST library

NanoKernelTest.first, in this test case, was verified the initialization of the RT-NKE. This include ready list verification which must be empty. Another verification is if it was created the basic tasks of the system, such as idletaskemain. NanoKernelTest.insertReadyListequal, in this test case, one of the most important function of the scheduling implementation: the InsertReadyList.This function is responsible to keep a list of tasks accomplished according to the scheduling algorithm criterion. In this test, the tasks priorities were the same.

All the following test cases, NanoKernel-Test insertReadyListincseq, insertReadyListdecseq, insertReadyListnegseq, also test the function InsertReadyList, with the difference that they have different entries. (fiquei na duvida sobre o mesmo conteudo).
For the test case insertReadyListdecseq, fourdecremental priorities tasks were created. Task 1 with priority 4, Task 2 with priority 3, and so on. In this case is expected the following order: Task 4, Tas 3, Task 2 and Task 1. The insertReadyListdecseqhas the same test, but with negative values.

The test case NanoKernelTest.systest the priority task creation algorithm in the system. This simulation is done using the SystackCreatefunction, that is called by the system. We created two tasks with different periods and computation time. One task with period 1 and the other with period 15. After

that, the priority is calculated by the function sysStartThis function verifies if the Task with period 1 is in the top of the done list, because according to the RM rule, tasks with lowest period have highest priority. The others related test cases, NanoKernelTest.sys start2 and NanoKernelTest.sys start3 do the same test, but inverting the task creation order.
We also created a test case to guarantee the functionality of the select function. This function is called when a timer interruption occurs. When this function is called, it is verified first if a task with higher priority is in the list ready to be executed. If positive and also if the running task has lower priority in comparison with the task that is ready by the period, a context is changed for the ready task to be executed.

C. *Implementation on the Beagle Bone Board*

We implemented this work using SBC -Single Board Computer Beagle Bone. Accordingly, to [5], this board is widely used for several reasons, including the vast documentation available. Figure 6 shows a block diagram of BeagleBone.

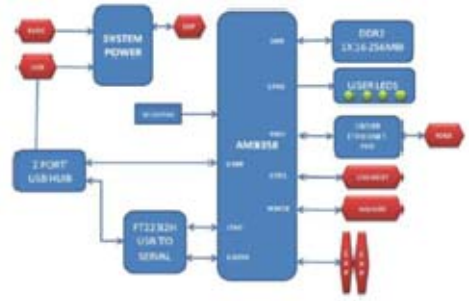

Figure 6: Beaglebone's basic components

The nanokernel is implemented in the Cortex-A8 32 bits processor. Figure 7 shows the initialization and RT-NKE structure, where it is possible to identify several events during system initialization. Also it is possible to identify the 3 main context that RT-NKE is running.

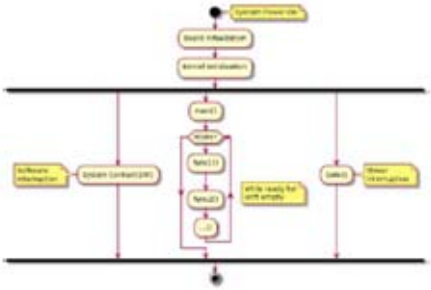

Figure 7: Initializations and RT-NKE structure

After System Power On, when the board is powered up, the following three events, BoardInitialization, KernelInitialization and CallbackRegistration, occur in sequence. In KernelInitialization, or Kernel startup, the internal RT-NKE structures are initialized. The queue of system-ready tasks is initialized, the main and idle tasks are also created. The main task must always be defined by the user application and the idle task is used when the system has no other task to execute. In this step, the RT-NKE also correctly sets the system timers, which will be used to generate the interrupts that trigger the scheduler.

After the initialization of RT-NKE is performed successfully, the execution of the processor is directed to the main function defined by the user. In a standard application, the main function should contain the creation of the tasks the user wants to execute and the call of the function that performs the scheduler initialization. After this call, the scheduler will choose the task with the highest priority and will select it for execution. When a timeout occurs for scheduling purposes, the Select function is called and the scheduler performs the steps shown in Figure 7.

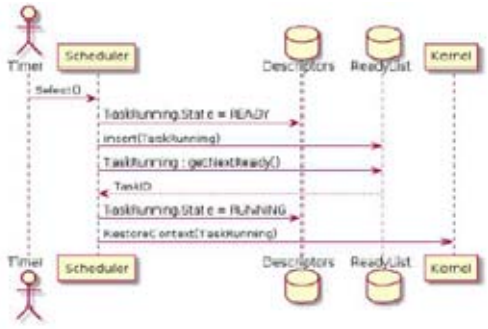

Figure 8: Sequence of events at timer interrupt

Figure 8 shows the sequence of steps that are executed when an interruption timeout occurs, that is, it is time for the scheduler to switch tasks. In the first step the state of the current task is changed to the READY state by changing the information in the Descriptors data structure, which contains the task information. The next step is to insert the current task into the ready list. The task will be included in the list according to its priority. Shortly thereafter, the scheduler selects the next task from the ready list, that is, the next highest priority task. In the sequence, the state of this task is changed to RUNNING, and then the context switch is performed for this task, which is then executed by the processor.

This process is repeated at each time-out interruption, until all tasks, created by the user, are finished. This process is shown by the READY loop shown in figure 8.

VII. CONCLUSIONS

As a result of the work, a task scheduling algorithm was developed for embedded operating system in critical applications based on multicore processors, where real time is a primordial variable. The RTOS kernel, the RT-NKE, used in the implementation is characterized by being compact (small) and therefore fast for execution. This RTOS runs on ARM processors and for the implementation of the Multicore Partitioned algorithm, the RT-NKE was ported to a BeagleBone board.

This work have shown the great difficulty of implementing and testing scheduling algorithms. There is an enormous debugging difficulty and the realization of poorly planned tests generates unpredictable results and lead to misinterpretations about the behavior of the algorithms.

The unitary testing methodology used in the development of the Partitioned algorithm proved to be effective and allowed the detection of implementation errors in a controlled environment. The development time was reduced in relation to the time spent in the implementation of the RM and EDF algorithms, previously performed by the same research group.

REFERÊNCIAS

[1] D. Wentzla, A. Agarwal. “Factored operating systems (fos): the case for a scalable operating system for multicores,” in SIGOPS Operating System Review, 43:76-85, April 2009.

[2] G. Gracioli, Real-time Operating System Support for Multicore applications. PhD thesis, UFSC, Florianópolis, SC, 2014.

[3] W. Sheng, Y. Gao, L. Xi, and X. Zhou, “Schedulability analysis for multicore global scheduling with model checking,” in Proceedings of the 2010 11th International Workshop on Microprocessor Test and Verification, MTV ’10, (Washington, DC, USA), pp. 21–26, IEEE Computer Society, 2010.

[4] A. Olofsson, “Parallella reference manual.” https: //www.parallella.org/docs/parallella manual.pdf, 2014. Acesso: 09-10-16

[5] E. Brown, “Top 10 open source linux and android sbc’s.” www.linux.com/news/ top-10-open-source-linux-and-android-sbcs, 2014 Accessado: 27-11-2016.

[6] J.H. Anderson , J.M. Calandrino , U.C. Devi , “Real-Time Scheduling on Multicore Platforms,” in Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06), 2006.

[7] P. M. B. de Sousa, Real-Time Scheduling on Multi-core: Theory and Practice. PhD thesis, Faculdade de Engenharia Universidade do Porto, Porto, Portugal, 2013.

[8] C. Hiroyuki and Y. Nobuyuki, “Experimental evaluation of global and partitioned semi-fixed-priority scheduling algorithms on multicore systems,” in 5th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, (Calif¨ornia, EUA), IEEE, 2012.

[9] G. Gracioli, A. A. Fro¨hlich, R. Pellizzoni, and S. Fischmeister, “Implementation and evaluation of global and partitioned scheduling in a real-time os,” in Real-Time Syst., vol. 49, pp. 669–714, Nov. 2013.

[10] C. L. Liu and J. W. Layland, “Scheduling algorithms for multiprogramming in a hard-real-time environment,” Journal of the ACM, vol. 20, Jan. 1973.

[11] Stewart, David and Michael Barr. "Rate Monotonic Scheduling,” Embedded Systems Programming”, March 2002, pp. 79-80.

[12] A. C. Bomberger, W. S. Frantz, A. C. Hardy, N. Hardy, C. R. Landau, J. S. Shapiro, “The KeyKOS Nanokernel Architecture”. Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures. Pages 95 – 112. USENIX Association Berkeley, CA, USA, 1992.

[13] Celso M. da Costa, Cassio Brasil, Leonardo L. Silva, Lucas Murliky, João Leonardo Fragoso, Guilherme Debom, Rivalino Matias, Aline Fracalossi, and Margrit Reni Krug. 2016. NKE: an embedded nanokernel for educational purpose. In Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC '16). ACM, New York, NY, USA, 1900-1902. DOI: https://doi.org/10.1145/2851613.2851947

[14] A. S. Tanenbaum, Operating Systems Design and Implementation. Prentice HalL, 3 ed., 2006.

[15] B. Noel and S. Subramaniam, “Accelerating embedded software development via agile techniques.” PwC Technology Institute, London, UK, 2013.

[16] M. J. Karlesky, W. I. Bereza, and C. B. Erickson, “Effective test driven development for embedded software,” in IEEE International Conference on Electro/information Technology, (Michigan, EUA), IEEE, 2006.