Middleware for MPSoC Real-Time Embedded Applications: Task Migration and Allocation Services

Elias Teodoro Silva Jr[1,2]; Carlos Eduardo Pereira[1,3]; Flávio Rech Wagner[1]

[1]*Institute of Informatics, Federal University of Rio Grande do Sul, Brazil*
{etsilvajr, flavio}@inf.ufrgs.br
[2]*Federal Center of Technological Education of Ceará, Brazil*
elias@cefetce.br
[3]*Electrical Engineering Department, Federal University of Rio Grande do Sul, Brazil*
cpereira@ece.ufrgs.br

ABSTRACT

The use of MPSoCs (multiprocessor systems-on-chip) is a clear tendency for embedded systems in the current days, especially for consumer markets. Applications are growing in complexity, and multiprocessor platforms can provide performance and flexibility. On the other hand, developers are looking for platforms that help to cope with conflicting design demands, such as low energy consumption, reduced area, timing requirements, and tight time-to-market. This paper proposes to move up the abstraction level to deal with this challenge, by offering a middleware to encapsulate platform details and preserving real-time properties. Task migration and allocation services are emphasized in this paper, and initial results in task migration are presented and evaluated.

1. INTRODUCTION

Real-time embedded systems are expanding and growing in complexity, imposing multiprocessing resources to face high performance and low-energy requirements. MPSoC (Multiprocessor System on Chip) is becoming a widely adopted design style, to achieve tight time-to-market design goals, provide flexibility and programmability, and maximize design reuse. The use of a multiprocessor platform brings with it the well known challenges from parallel and distributed systems [2], related to concurrency. Sometimes, these processors may have a fixed ISA (Instruction Set Architecture); sometimes a mix of processor types is used, like RISC+DSP, for example. Additionally, embedded systems impose restrictions to the solution, like limitations in CPU performance, memory, and power consumption. Therefore, solutions that come from the distributed systems context should be customized to be used for embedded applications.

Developing applications for embedded multiprocessor architectures requires a higher level programming model to reduce software development cost and overall design time [3]. Such a model reduces the amount of architecture details that need to be handled by application software designers and then speeds up the design process. The use of a higher level programming model will also allow concurrent software/hardware design, thus reducing the overall design time.

On the other hand, improving the performance of the overall system requires going through low level programming, exposing architectural properties to the application level.

Since applications are partitioned in processes and processors, a middleware could be used to provide a high level interface, hiding distribution aspects [4]. As a consequence, system resources and application components can be easily reused, saving time for a better application development. In a typical DRE (Distributed Real-time Embedded) system, a middleware usually integrates reusable software components and decreases the cycle-time and effort required to develop high-quality real-time and embedded applications and services [5]. The middleware support has not been investigated in the context of MPSoC applications, but only for DRE systems. Nevertheless, in the context of MPSoCs a middleware could also become an interesting approach to raise the abstraction level, helping to achieve shorter development times. Moreover, energy consumption is a key issue for embedded systems and a high abstraction level development tools should also take such issues into account.

This paper describes a middleware to deal with distributed applications in an MPSoC using a homogeneous ISA (Instruction Set Architecture) and abstracting interfaces between HW-SW implementations and network communication as well. It also includes energy management services, which work transparently, integrated with high level services.

Dynamic task allocation and migration has been shown to be a promising technique to ensure an adequate load balancing among processing units in an MPSoC [6][7], allowing the minimization of some metrics, such as execution time or power consumption. This work proposes

to combine those allocation and migration services with energy management in a transparent way.

The remaining of the paper is organized as follows. Section 2 gives an overview about the development platform. The proposed middleware is presented in Section 3. Task allocation and migration services are described in Section 4. In Section 5, experimental results are presented. Finally, in Section 6 concluding remarks are drawn.

2. Hardware platform

Figure 1 depicts the overall platform architecture, which includes the network and the processor.

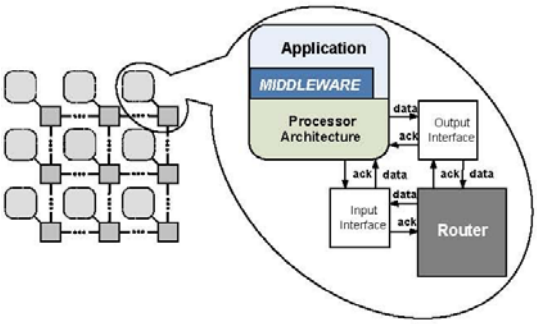

Figure 1: General Platform Architecture

2.1 Configurable processor

Over the last years, Java has gained popularity as a suitable programming language for embedded and real-time systems development. The definition of the Real-Time Specification for Java (RTSJ) standard [1] is the most prominent example of such popularization in the real-time domain.

For this work, a customizable Java processor called FemtoJava [8] is used, which implements an execution engine for Java in hardware, through a stack machine that is compatible with the specification of the Java Virtual Machine (JVM). Different processor organizations are supported, such as multi-cycle, pipeline, and VLIW [9]. For the multi-cycle processor, used for the experiments in this work, all instructions are executed in 3, 4, 7, or 14 cycles, because the microcontroller is cacheless and several instructions are memory bound.

A compiler that follows the JVM specification is used. An environment called Sashimi [8] generates both customized code for the application software and the processor description and allows the synthesis of an ASIP (application-specific integrated processor). The generated code includes the VHDL description of the customized processor core (whose ISA contains only instructions used by the application software), as well as ROM (programs) and RAM (variables) memories and can be used to simulate and/or synthesize the target application. Sashimi eliminates all unreferenced methods and attributes, as well as the unused JVM instructions, automatically customizing and optimizing the final hardware and software code.

2.2 Communication infrastructure

Networks-on-chip [10] have been proposed in recent years as a scalable, high-bandwidth, and energy-efficient communication infrastructure for MPSoCs containing a large number of cores. In this work, the network-on-chip (NoC) SoCIN [11] is used to interconnect the processors inside the MPSoC. SoCIN is based on a flexible router, called RaSoC.

Communication is based on message passing. Messages are sent in packets, which are composed by flits. A flit (flow control unit) is the smallest unit over which the flow control is performed. A flit also coincides with the physical channel word (or phit – physical unit).

SoCIN utilizes wormhole packet switching, so it uses small buffers in the routers, saving size and energy. The routing is XY, which is deadlock free. Each router has 5 bi-directional ports with input buffer size of 4 phits. The phit size is 4 bytes.

The router description provides parameters to perform fine adjustment in the NoC properties, aiming at matching application requirements as well as possible. The cost-performance trade-offs can be explored by changing NoC parameters.

SoCIN can support other devices connected to the routers, besides processors. In spite of that, this work considered only processors connected through the network, using homogeneous ISA and private memory. Other research efforts in our research group have been conducted to use heterogeneous processors and shared memory, but they will not be discussed in this paper.

3. Outline of the MPSoC Middleware

This section presents the middleware proposed to fulfill the requirements of a real-time and embedded system with energy restrictions. An MPSoC is assumed as the target hardware platform.

Within the context of this work, the middleware aims at promoting software and hardware reuse and includes mechanisms that help to express real-time requirements and constraints. Those properties should be fulfilled having in mind limitations in physical resources like energy, memory, and processor performance.

The proposed architecture allows a flexible and broad design space exploration, by acting upon issues like

hardware or software implementation of services and objects and locality of objects in the network.

Figure 2 shows the proposed architecture, which is organized in two abstraction levels: structure and service levels. The structure level offers the more elementary resources of the middleware, namely network communication and multithread management. Using classical definitions, this level could be defined as an RTOS. However, to offer flexibility and enhance overall efficiency, RTOS-like capabilities are included in the middleware. The service level offers a higher abstraction and uses resources implemented at the structure level. It offers basic services, if one considers the complexity of a general purpose distributed system. However, these services are sufficient to support multiprocessor embedded application design, allowing the exploration of different arrangements in the allocation of tasks either at design time or at execution time.

It is important to highlight the monitoring and DVS (Dynamic Voltage Scaling) services, at the structure level. Those services are not part of the original RTSJ standard, but they were defined in the middleware to support some facilities at the service level.

This paper discusses the task migration and task allocation services in detail. A more generic view will be given for the other services.

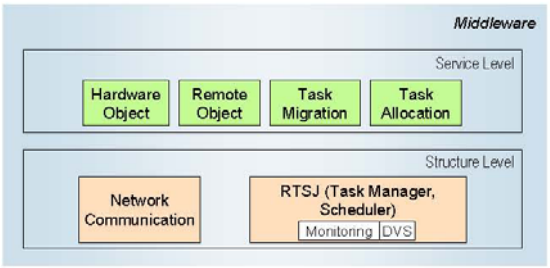

Figure 2: Middleware architecture

3.1 Real-time multithread management

In the context of this work, a thread is a synonym for a schedulable object and is also called task.

The Real-Time Specification for Java (RTSJ) standard [1] defines a set of interfaces and behavioral specifications to allow the development of real-time applications using the Java programming language. Among its major features are: scheduling properties suitable for real-time applications with provisions for periodic and sporadic tasks and support for deadlines and CPU time budgets.

RTSJ allows the use of schedulable objects, which are instances of classes that implement the so called Schedulable interface, such as `RealtimeThread`. It also specifies a set of classes to store parameters that represent a particular resource demand from one or more schedulable objects. For example, the `ReleaseParameters` class (superclass from `AperiodicParameters` and `PeriodicParameters`) includes several useful parameters for the specification of real-time requirements, such as cyclic activation and deadlines. Moreover, it supports the expression of the following elements: absolute and relative time values, timers, periodic and aperiodic tasks, and scheduling policies.

Along with the Java processor there is an API [13] that supports the specification of concurrent tasks and allows the specification of timing constraints, implementing a subset of the RTSJ standard.

The scheduling structure consists of an additional process that is in charge of allocating the CPU for those application-processes that are ready to execute, exactly like in an RTOS. Application developers should choose the most suitable scheduling algorithm at design time. Therefore, a customized scheduler is synthesized with the whole application into the embedded target system.

Currently, four scheduling algorithms are available: EDF, RM, Fixed Priority (software and hardware implementations), and Time-Triggered.

3.1.1 Additional functions to RTSJ

The so called function 'monitoring' aims at measuring resources of the local processor, like available memory and processor utilization. This function is offered to the task allocation service to help its decision when adding a new thread to a processor.

A DVS (Dynamic Voltage Scaling) functionality is added to the schedulers and allows the application to act upon the hardware for energy reduction purposes in a transparent way. The use of DVS algorithms, like the cycle-conserving one [15], opens space for energy reduction at execution time. By using a DVS capability, the scheduler can manage the local processor frequency to the lowest value able to match the deadlines of the threads added to the scheduler. From the designer's point-of-view, it is enough to use a scheduler that is able to manage DVS resources.

3.2 Network communication

The communication API (COM-API) encapsulates transport and datalink layers, providing an interface to the application layer [2].

The communication system provides support to message exchange among applications running in different

processors. The API allows applications to establish a communication channel through the network, which can be used to send and receive messages. The service allows the assignment of different priorities to messages and can run in a multithread environment. From the application point-of-view, the system is able to open and close connections as well as to send and receive messages, being accessed by different threads simultaneously.

The COM-API works together with the RTSJ-API, using processor features to provide communication via a network interface. RTSJ-API provides schedulable objects (for real-time threads) and relative time objects.

In order to offer a larger design space to be explored in the development of application-specific systems, a hardware implementation of the communication service was also developed [18]. It is encapsulated in a class called `HwTransport` and can be used in the same way as the software implementation (called `Transport`). The Java processor interacts with this communication block implemented in hardware as with any other I/O device.

The differences when using hardware and software implementations are transparent to the developer, since they are encapsulated in different classes that implement the same interface.

3.3 Locality abstraction

An important demand for design space exploration in an MPSoC system is to allow the allocation of threads everywhere in the network, making this locality transparent to the application until run-time. This property requires an abstract locality mechanism in order to allow access to other objects even when their location is unknown at development time. Moreover, this mechanism should be integrated with the RTSJ-API in order to offer temporal guarantees for message delivery.

A simplified mechanism for remote method invocation was proposed and implemented based on RMI from standard Java [17]. A conceptual modification was introduced in this mechanism using time bounds for its operations using RTSJ objects. A specific class to encapsulate real-time properties was added (`RealTimeParameters`) both in the client and in the server sides. The thread (`ConnectionHandler`) that deals with connections on the server side is another component to bring predictability to RMI. This thread has real-time properties following RTSJ rules. This means that it will be scheduled according to its real-time properties, like period and deadline. The RTSJ API allows the developer to choose among different scheduling policies, as already mentioned.

Similarly, a maximum execution time is defined for the `ConnectionHandler` thread at development time, using an asynchronous event mechanism, as defined by RTSJ. Thus, the communication operations will not violate the time reserved for the other application threads or tasks.

3.4 Hardware-object implementation

The boundary of the hardware/software partition plays an important role in meeting design constraints. This boundary is often decided upon at the early stages of development, leading to premature and inadequate design decisions. Moreover, it is hard to move this boundary at later stages. Better design decisions could be made at later stages in the development, when a better understanding of impacts of alternative hardware and software implementations emerges. This is only possible if the design process includes tools that simplify the movement of components' deployment from hardware to software and vice-versa, by defining a uniform programming model for both implementations.

Within the context of this work, a real-time thread can be implemented in two different ways. A software implementation is a Java code executed by the processor, as described in [13]. A hardware implementation executes autonomously, although controlled by the processor. A hardware thread has its own Finite State Machine (FSM) and can run in parallel with the processor.

A hardware component (HwTI – Hardware Thread Interface) is defined as an interface between the processor and the hardware thread. Another hardware component must implement the thread behavior and is called Hardware Thread Behavior (HwTB). HwTI is part of the platform, available to developers, while HwTB is part of the application and must be implemented by developers using a hardware description language. The proposed architecture is introduced in [14], where it is better described.

The communication between the application and the HwTB component is managed in software, by an RTSJ compatible class.

From the software point-of-view, the hardware thread is encapsulated by an object that extends the `RealtimeThread` class from RTSJ. So, the hardware thread will be controlled similarly to other threads implemented in software, by reusing schedulers already available in the RTSJ implementation.

3.5 Energy management

An important demand for MPSoC platforms is energy management, since most of them are powered by batteries. Low energy means a smaller battery, lower weight, lower cost, and so on.

Within the context of this work, low power and low energy are provided by hardware implemented objects that can be included as services or as application components. To reach flexibility, a DVS/DFS (Dynamic Voltage Scaling / Dynamic Frequency Scaling) functionality is included in the task schedulers and exposed to be selected by application developers. The application can define the scheduler to be used in each processor, thus defining if DVS should be used or not.

4. Task allocation and migration

Task allocation and migration are services related to load balancing, and a homogeneous ISA is required. A task is allocated before it starts running and can be migrated during its execution.

4.1 Task migration

Dynamic task allocation has been shown to be a promising technique to obtain load balancing among processing units in an MPSoC [6] [7], allowing the minimization of some variables, like execution time or power consumption. To reach dynamic allocation, a migration task mechanism is required. Two cases are possible: (1) when a new allocation is required in a set of not-empty processors, some tasks could be moved to optimize the new distribution; (2) when a set of tasks is finished, a new arrangement can be made to optimize the overall processors' utilization.

Task-migration approaches usually adopt shared-memory as the communication model in an SMP (symmetrical multi-processing) environment. This work considers an AMP (asymmetrical multi-processing) model, since processors have dedicated local memory resources. Although processors can have different organizations, like pipelines and multi-cycle ones, they share a common ISA, and, thus, tasks can be assigned to different processors. For the adopted platform (a NoC), message exchange is a natural choice due to its scalability. However, a shared-memory model is also under investigation as a communication strategy, but it is not in the scope of this paper.

To offer task migration as a service in the middleware, all communication operations should be submitted to the communication API. Moreover, these communications are submitted to the discipline of a periodic real-time thread with a pre-established maximum cost. These properties make the task migration service independent from the underlying network and adequate to be used in real-time applications.

In the context of the adopted platform, a task is not built at run-time, but it is defined at development time, together with its accessed objects. Thus, its address space is known *a priori*. The middleware can only move quite independent tasks. Currently, if different tasks share the same data, the application is supposed to take care of data coherence after migration. Some mechanisms to solve this situation are still under investigation, since they introduce an important overhead in the communication.

Figure 3 shows the class diagram of the implemented service. The migration service is activated by another service, called task allocation, which decides which task to move and its destination, based on restrictions, like processor or memory utilization, and on objectives, like load balancing.

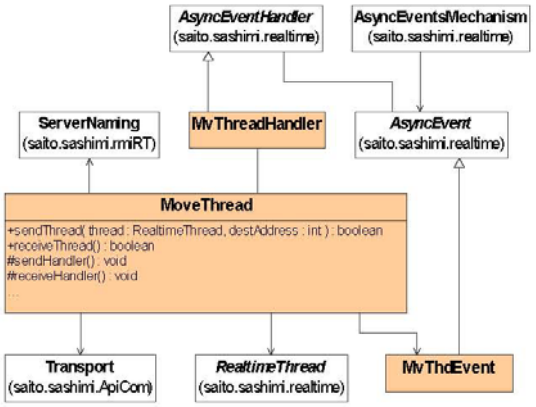

Figure 3: Task migration class diagram

The `MoveThread` class contains the public methods `sendThread()` and `receiveThread()`. They are used to activate task sending and -receiving services. They both return FALSE if the service is not available. When the `sendThread()` method is executed, an event handler (`MvThreadHandler`) is executed and configured to move the task, which is passed as a parameter. The task is sliced into blocks, and the first one is sent. After that, each time an ACK is received (from the receiver), an event is generated and the next block is sent. The event handler follows the asynchronous event management policy, defined by `AsyncEventMechanism`. Although this procedure leads to an increase in the latency of a migration, it ensures a balanced use of the processor, avoiding any interference in other running real-time tasks.

Task migration means to send code (methods) and attributes of the `RealtimeThread` object as well as the

objects referred to in the RealtimeThread. The stack is also sent. Being a stack machine, Java preserves task variables in the stack, such that the task context is replicated when the stack is copied. This property makes context copy easy, avoiding the use of checkpoints. In other words, the memory used by the task is confined to the attributes of its classes and to the stack, which contains method variables.

For the adopted platform, the position of objects (code and attributes) in the memory is defined by a post-compilation tool, which can set appropriate attributes of the RealtimeThread class. The stack position and size are known by the RealtimeThread. The MoveThread class obtains those values at run-time before moving the task.

The migration service should be activated in the destination too, as in the origin of the migration, by invoking the receiveThread() method.

4.2 Task allocation

The task allocation service consists in pointing out nodes for tasks in the network using a distribution function. In [6] and [7] different distribution algorithms were investigated and some solutions were proposed. Those algorithms have been firstly evaluated in a high abstraction level simulator and afterwards implemented as part of the middleware. The role of the middleware is to offer an interface to the service, thus making easy for the application developer the choice of a distribution algorithm.

Each node of the network should have an instance of the monitoring service (middleware structure level), which is able to inform about the availability of resources (memory, processor time).

Figure 4 shows the class diagram for the task allocation service. First of all, the class RealtimeThread, from RTSJ, is extended, thus creating an XtdRealtimeThread class. This new class has the properties the task should inform to the allocation service, as memory and processor utilization. In fact, for a periodic task, it is possible to obtain the processor utilization by referring to RTSJ parameters, since a periodic RealtimeThread knows its worst case execution time (WCET) and the period as well. The utilization is equal to the WCET divided by the period.

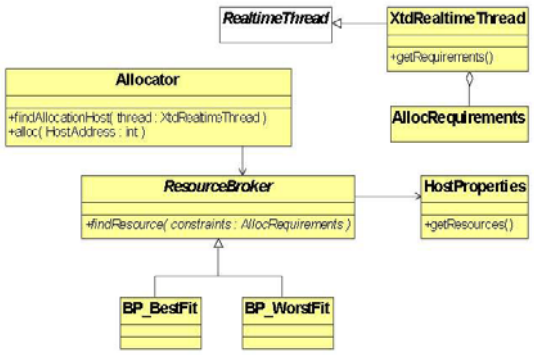

Figure 4: Task allocation class diagram

Task distribution is implemented by the classes Allocator and ResourceBroker. The design pattern Strategy is used to offer abstract access to different allocation algorithms. In the diagram provided in Figure 4, Bin-Packing Best Fit (BP_BestFit) and Bin-Packing Worst Fit (BP_WorstFit) are shown to illustrate possible algorithms, as proposed in [6]. The findResource() method is implemented in the concrete classes to perform a search for a node to allocate a task.

5. EXPERIMENTAL RESULTS

For experimental verification, a SystemC simulator uses an RTL description of the FemtoJava processor. The network is implemented as a TLM (transaction-level) model.

The example presented in this paper is a demonstration of the task migration service. In this example, three synthetic tasks are executed in one processor and one task in another one. This example represents four different applications that do not have communication between them. The tasks are periodic and the migration should not jeopardize their deadlines. After some time, one task (TaskC) migrates from the first processor to the second one.

The AsyncEventMechanism period was chosen such that the task that migrates (TaskC) could do it between two consecutive execution periods. Figure 5 shows the activation times for TaskC, where the x-axis represents time in milliseconds. The first two executions occur at the origin processor, while the remaining ones occur at the destination. The third execution experiences latency due to the migration time. One can see that the task promptly recovers its original period (30 ms), as started in the origin. The activation time of a RealtimeThread is part of its attributes and is copied in the migration process.

Thus, the scheduler in the destination can keep the original behavior of the task.

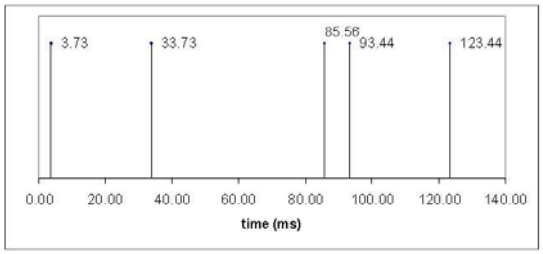

Figure 5: Activation time for a migrated task

The time required to migrate a task can be evaluated from two different points-of-view, as shown in Table 1. The first line shows the computational cost to migrate `TaskC`, it means, the real cost in processing the migration service. The cost grows linearly as the task size increases. The time values depend on the latencies imposed by the communication service, provided by the structure layer. This throughput can be optimized using a hardware-implemented communication service [18] or a processor with higher performance [9].

The migration does not occur in a continuous flow, which could compromise the deadline of other tasks running. On the contrary, the transmission is sliced in blocks controlled by the `AsyncEventMechanism` from RTSJ. It increases total latency observed by the user of the service, as shown in the second line in Table 1. In the origin, it is the time elapsed since the `sendThread()` method is invoked until the service finishes. In the destination, it is the time elapsed since the first block starts to arrive until the `start()` of the `RealtimeThread` in the destination processor. At both sides (origin and destination), the end of the service is transparent to the user, i.e., the methods that activate the service do not retain the flow of the code that invokes them. The total latency grows following the period of the task that implements the `AsyncEventMechanism`.

Table 1: Time measures for task migration

	Origin node	Destination node
Effective cost (ms)	3.27	3.93
Total latency (ms)	53.30	51.42

Using the middleware, developers save development time required to implement capabilities already provided as services. Code provided to implement HW-SW communication, task migration, remote method invocation, and so on can be reused in all projects. Table 2 shows the amount of memory used by some services of the middleware compared with a classical embedded application, an MP3 player. The table shows that the total memory consumed by the middleware is acceptable for real applications.

Table 2: Memory usage

***Middleware* component**	**ROM**	**RAM**
Remote method (server)	2139 Bytes	118 Bytes
Task migration (origin)	2343 Bytes	81 Bytes
COM-API (Pack49-Msg500)	4493 Bytes	6345 Bytes
API-RTSJ + DVS	4849 Bytes	242 Bytes
TOTAL (*middleware*)	13824 Bytes	6786 Bytes
Application	**ROM**	**RAM**
Mp3Player	48548 Bytes	63702 Bytes

6. CONCLUSIONS

Multiprocessor platforms bring new challenges to the development of applications with high quality, matching real-time requirements and keeping a low energy usage. This paper proposes to face this challenge using a middleware to abstract platform details and allowing developers to express real-time requirements.

An MPSoC with homogeneous ISA is considered for task migration and allocation services.

Preliminary results on task migration are presented and evaluated. Results show that this service presents an acceptable cost and offers an adequate abstraction for the application developer. This service runs upon the structure level of the middleware, which is similar to an RTOS.

The next step of this work is to validate and evaluate the task allocation service, based on algorithms previously evaluated in [6].

REFERENCES

[1] Bollella, G.; Gosling, J.; Brosgol, B. The Real-Time Specification for Java. 2001. <http://www.rtj.org/rtsj-V1.0.pdf>.

[2] Martin, G. *Overview of the MPSoC Design Challenge*. In: Design Automation Conference, DAC, 2006, San Francisco, p. 274-279.

[3] Jerraya, A.A.; Bouchhima, A. and Pétrot, F. *Programming models and HW-SW Interfaces Abstraction for Multi-Processor SoC*. In: Design Automation Conference, DAC, 2006, San Francisco, p. 280-285.

[4] Bernstein, P.A. *Middleware: A Model for Distributed System Services*. Communications of the ACM, New York, vol.3, n.2, p.86-97, Feb. 1996.

[5] Schmidt, D.C. *Middleware Techniques and Optimizations for Realtime, Embedded Systems*. In: International Symposium on System Synthesis, 1999, San Jose, CA, p. 12-16.

[6] Wronski, F.; Brião, E.W.; Wagner, F.R. *Evaluating Energy-aware Task Allocation Strategies for MPSoCs*. In: IFIP TC-10 Working Conference on Distributed and Parallel Embedded Systems, DIPES, 2006, Braga, p. 215-224.

[7] Acquaviva, A. et al. *Assessing Task Migration Impact on Embedded Soft Real-Time Streaming Multimedia Applications*. EURASIP Journal on Embedded Systems, New York, v. 2008, n.2, p.1-15, Apr. 2008.

[8] Ito, S.A., Carro, L., Jacobi, R.P. *Making Java Work for Microcontroller Applications*. IEEE Design & Test of Computers, v. 18, n. 5, p. 100-110, Sept/Oct. 2001.

[9] Beck Filho, A.C.S.; Carro, L. *Low Power Java Processor for Embedded Applications*. In: IFIP VLSI-SOC, 2003, Darmstadt, p. 239-244.

[10] Benini, L.; Demicheli, G. *Networks on Chip: A New SoC Paradigm*. IEEE Computer, v.35, n.1, p. 490-504, Jan. 2002.

[11] Zeferino, C.A.; Susin, A.A. *SoCIN: A parametric and scalable network-on-chip*. In: Symposium on Integrated Circuits and Systems Design, SBCCI, 2003. Los Alamitos: IEEE Computer, 2003. p. 169-174.

[12] Silva Jr., E.T.; Freitas, E.P.; Wagner, F.R.; Carvalho, F.C.; Pereira, C.E. *Java Framework for Distributed Real-Time Embedded Systems*, In: 9th IEEE ISORC, Gyeongju, Korea, 2006, p. 85-92.

[13] Wehrmeister, M.A.; Becker, L.B. and Pereira, C.E. *Optimizing Real-Time Embedded Systems Development Using a RTSJ-based API*. In: JTRES 2004, Proceedings Springer LNCS, Cyprus, October 2004, p. 292-302.

[14] Silva Jr., E.T.; Andrews, D.; Pereira, C.E. and Wagner, F.R. *An Infrastructure for Hardware-Software Co-design of Embedded Real-Time Java Applications*. In: 11th IEEE ISORC, Orlando, USA, 2008.

[15] Pillai, P. and Shin, K.G. *Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems*. In Proc. of the 18th ACM Symp. on Operating Systems Principles, 2001, p. 89-102.

[16] Spuri, M. and Butazzo, G. *Efficiente aperiodic service under earliest deadline scheduling*. In: IEEE Real-Time Systems Symposium, RTSS, 1994.

[17] Grosso, W. *Java RMI*. O'Reilly Media, 2001.572 p.

[18] Silva Jr, E.T.; Wagner, F.R.; Freitas, E.P.; Kunz, L. and Pereira, C.E. *Hardware Support in a Middleware for Distributed and Real-Time Embedded Applications*. Journal of Integrated Circuits and Systems, v. 2, n.1, p. 38-44, Mar. 2007.