A Virtual Platform for Multiprocessor Real-Time Embedded Systems

Elias T. Silva Jr[2] Daniel Barcelos[1] Flávio R. Wagner[1] Carlos E. Pereira[1]

[1]PPGC – Instituto de Informática, UFRGS
Av. Bento Gonçalves, 9500 – Bloco IV
91501-970 Porto Alegre - Brazil
+55 51 3308-6161
{danielb, flavio}@inf.ufrgs.br
cpereira@ece.ufrgs.br

[2]Telecommunication and Computer Dept., CEFET-CE
Av. Treze de Maio, 2081
60040-531 Fortaleza - Brazil
+55 85 3307-3607
elias@cefetce.br

ABSTRACT

This paper presents a virtual platform for the development and test of application software, low-level software, and hardware components for an MPSoC (Multiprocessor System-on-Chip) platform, where components are interconnected by a network-on-chip (NoC). The environment is aimed at the development of multithread real-time embedded applications in Java language. Communication and task management services are provided that are able to deal with real-time restrictions, following the RTSJ standard. SystemC models are used to describe processors and network connections that have equivalent descriptions in VHDL. Performance and power / energy evaluation are made possible, helping to shorten cycles for embedded system development, integration, and test.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems – *Real-time and embedded systems*.

General Terms

Design, Performance, Experimentation.

Keywords

Embedded applications, Real-time systems, Energy efficiency, Network on Chip

1. Introduction

Embedded and real-time systems can be found from microcontrollers in coffee machines up to networked processors in automotive and avionic control systems. This market is growing strongly because of the increasing number of application areas. As those systems grow in complexity, it becomes more difficult for programmers and architects to build, configure, and maintain them with respect to correctness, resource optimization, and system stability. In particular, real-time systems depend not only on the logical results of computation, but also on the timeliness of those results [1].

In the competitive market of embedded applications, an important goal is the reduction of time-to-market, demanding that teams for software development, integration, and test obtain short development cycles, increase product quality, and reduce costs. To go around these requirements, designers need tools that mitigate existing limitations of the software development process, such as the availability of the physical hardware.

On the other hand, over the last years, Java 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 [2] is the most prominent example of such popularization in the real-time domain. RTSJ defines an Application Programming Interface (API) for the Java language that allows the creation, execution, and management of real-time threads.

Programming model and task control management will be major challenges for MPSoC (multiprocessor systems-on-chip) designs in the coming years. The work presented in this paper is part of an effort to enlarge design space exploration when developing distributed applications in an MPSoC using a homogeneous ISA (Instruction Set Architecture) and abstracting both interfaces between HW-SW component implementations and network communication. A virtual platform for the development and validation of real-time embedded MPSoCs, called SIMPLE (Simple Multiprocessor Platform Environment), is discussed. Processing elements (PEs) are Java processors and implement RTSJ, thus supporting multithread and real-time applications. The communication infrastructure is based on a packet switching network-on-chip (NoC) that uses a mesh topology to connect PEs.

This work combines simple processors in an MPSoC under a layer of software facilities that provides real-time scheduling and communication services. Moreover, hardware and software integration is made easy by a high abstraction level interface. The platform can be used to evaluate software and hardware behavior and performance as well as power and energy consumption.

A case study in automatic control was chosen to demonstrate communication and synchronization functionalities.

This work is part of a larger project aiming at providing a flexible and reusable middleware to raise the abstraction level for the development of real-time embedded applications upon an MPSoC platform. Object-oriented programming facilitates the reuse of software components, making possible the exploration of classical features of distributed systems, such as remote method invocation and task migration. This paper focuses on the structure layer of the middleware, which provides services for task management and communication.

The remaining of this paper is organized as follows. Section 2 discusses related work. Section 3 gives a brief overview of the hardware platform, including processor and network connection. Section 4 describes the software platform provided to application developers, with task management and communication services. Section 5 details the simulator and the resources it offers to the development team. Section 6 shows, as example of utilization of the virtual platform, the development of a control application. Experimental results and also implementation characteristics are presented. Finally, Section 7 draws main conclusions and discusses future work.

2. Related work

Platform-based design is a reality, as can be seen for instance in the OMAP platform from Texas Instruments [3] or Nexperia from NXP [4]. Several companies are introducing their own development frameworks. The tools allow designers to evaluate different platform configurations, making a fine tuning. For example, the designer can turn on/off the use of caches, special instructions (e.g. DSP instructions), co-processors, and reconfigurable areas.

CoWare recently introduced its Virtual Platform tool [5]. With that, development teams can rapidly deliver platform simulators based on already developed component models. New models can be built and added to the component library too.

In the academic domain, Saint-Jean et al. present a framework [6] to develop both hardware and software for an MPSoC platform. In that work, the hardware architecture is based on RISC processors, interconnected by a network-on-chip, where each core has it own memory.

This work, in turn, uses real-time Java processors as main processing elements. This allows system software designers to program directly in the Java language, using high abstraction level APIs. The real-time guarantees of the system make it a good solution for the development of hard and software real-time applications. Moreover, energy evaluation is possible at the instruction level, for the processor, and at the architectural-level for the network.

3. Hardware Platform

3.1 Configurable processor

For this work a configurable Java processor [7] is used, which implements an execution engine for Java in hardware, through a stack machine that is a subset of the specification of the Java Virtual Machine (JVM). There are different processor organizations, such as multi-cycle, pipeline, and VLIW [8]. 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.

The initially proposed instruction set was expanded [9] to include the bytecodes `putfield`, `getfield`, `invokevirtual`, `invokespecial`, and `instanceof`, opening space for non-static objects. In order to support multithread applications, two pseudo-bytecodes, `save_ctx` and `restore_ctx`, were created to provide context switching [10].

A compiler that follows the JVM specification is used. An environment [7] generates customized code for both the application software and the processor and allows the synthesis of an ASIP (application-specific integrated processor). The code produced includes the VHDL description of the customized processor core (whose ISA contains only instructions used by the application software) and ROM (programs) and RAM (variables) memories and can be used to simulate and/or synthesize the target application. All unreferenced methods and attributes are eliminated, as well as the unused JVM instructions, thus automatically customizing the final hardware and software code.

3.2 Communication infrastructure

The NoC SoCIN [11] is used to connect the processors. SoCIN was proposed to be scalable and 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 smaller 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, such that it uses small buffers in the routers, thus 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 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.

The NoC SoCIN can support other devices connected to the routers, besides processors. In spite of that, this work considered only processors connected trough the network, using homogeneous ISA (Instruction Set Architecture) and private memory. Other research efforts have been conducted to use heterogeneous processors and shared memory, but they will not be explored in this paper.

An ongoing work is implementing priorities in the SoCIN routers. For simplicity reasons the original routers did not include any QoS resource.

4. Software Platform

4.1 Multithread real-time development

The Real-Time Specification for Java (RTSJ) [2] 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 the `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 [12] that supports the specification of concurrent tasks and allows the specification of timing constraints, implementing a subset of the RTSJ standard. The implementations of some of the API classes have slight differences in comparison to the RTSJ standard. This is due to constraints in the adopted platform and also for better clarity. An example of such differences appears in the `RealtimeThread` class. It uses two abstract methods that have to be implemented in the derived subclasses: `mainTask()` and `exceptionTask()`. They represent, respectively, the task body – equivalent to the `run()` method from a normal Java thread – and the exception handling code required to deal with deadline misses.

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.

The RTSJ-API supports both static and dynamic scheduling algorithms. More specifically, it provides a dispatching mechanism that is able to work with any fixed-priority algorithm, like RM (Rate Monotonic) and Deadline Monotonic (DM) [13]. Additionally it includes support for dynamic scheduling with the EDF (Earliest Deadline First) algorithm [14]. Currently, four scheduling algorithms are available: EDF, RM, Fixed Priority (software and hardware implementations), and Time-Triggered.

The EDF scheduler, provided by RTSJ-API, was extended to include DVS (Dynamic Voltage Scheduling) algorithms, such as cycle-conserving [15]. This opens space for energy explorations at development time. From the designer's point-of-view it is enough to use a scheduler that is able to manage DVS resources. The scheduler can manage the local processor frequency to the lowest value able to match the deadlines of the threads added to the scheduler.

4.2 Communication API

The communication API (COM-API) encapsulates transport and datalink layers, providing an interface to the application layer [16].

The communication system was proposed to provide 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.

Figure 1 shows the overall platform architecture, which includes the COM-API. 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.

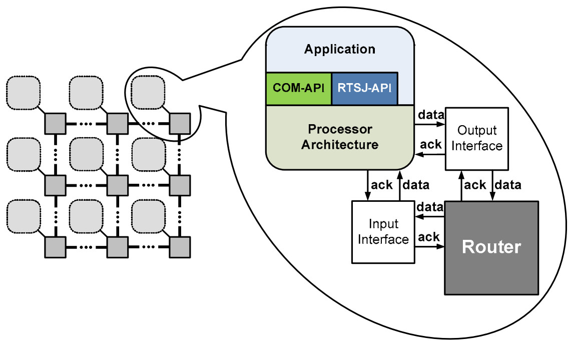

Figure 1: General Platform Architecture

5. Implementation

A synthesizable RTL model exists for each platform component, in order to guarantee a path to silicon for any system designed with the framework. However, this paper emphasizes a SystemC virtual platform, which can be used to evaluate application software and low-level software and to add hardware components to the platform as well.

5.1 Simulator

The simulator has been developed in SystemC. The tool can instantiate an arbitrary number of processors over the NoC. In this work, processors are supposed to be homogeneous, even though the simulator supports heterogeneous cores. The simulator also supports different description levels – the processor timing behavior was described at register-transfer level (RTL), and for the NoC a transaction level modeling (TLM) was used.

New hardware components can be attached to the system. These modules can be connected to a network router, composing a new tile of the network, or to the processor bus [17], where the local memory and the resource devices are arranged. The devices currently available are timers, I/O ports, a real-time clock, and the network interface. Transactors are used to connect components from different description levels. Transactors are timeless interfaces responsible for the translation of the components' inputs and outputs.

A UART module was also implemented, which allows the processor to communicate with a desktop computer, for example. The simulator uses a virtual COM port, so that the simulation behavior is exactly the same of the prototyped platform and it is possible to monitor the processors' messages using softwares like minicom or HyperTerminal.

Another simulator feature is power and energy evaluation. The energy consumption behavior is independent from the timing one and can be modeled at different abstraction levels too. For the processors, energy estimation was extracted by the Synopsys Power Compiler tool based on the power consumption data of the instruction-set. With that, it is possible to have an instruction-accurate energy model instead of a cycle-accurate one as in the timing description. This allows faster simulations, without a significant loss of precision.

The dynamic power consumption of the network routers and links is calculated with help of the Orion library [18] (the same used in the Xpipes NoC [19]). It implements an energy estimator for the arbiter, the crossbar, and the buffers inside the routers. Buffers are usually responsible for 90% of the energy consumption in the router. The consumption of the links is also taken in account. Regarding them, the energy spent by a data phit to be transferred between two routers is defined similarly to [20].

All energy estimations were tuned to correspond to consumption results of devices integrated in the TSMC 0.18 µm technology [21].

6. Case study

To illustrate the platform features, a crane controller has been implemented. A high level description of the crane as well as its control system are given in [22]. The main component of the system, a controller implemented using a floating point package, was split in three threads. Each thread performs a matrix multiplication and other operations. Since it is a control application, there is not too much parallelism to explore. However, the example is useful to explore synchronism in the communication among the threads. The correct behavior of this real-time application emerges from a right computation result in the correct time. So, the activation time of the threads and communication latencies should be kept under control.

Figure 2 shows the threads that implement the control algorithm and the communication as well. They are periodic threads and should exchange messages at each execution cycle. Thread scheduling and message passing are managed by services implemented in Java (RTSJ-API & COM-API), and the application is described in Java too. For exploration purposes, each thread runs in a different processor.

There are two kinds of data dependencies in this application. The first one occurs for the same execution period, which implies that no parallelism is allowed. This is the case in the communication between Mul_Bx and the two other threads. The other data dependency occurs when a thread uses a data calculated by other ones at the prior execution period. This is the way how Mul_Y depends on Mul_Aq, such that a concurrent execution is possible.

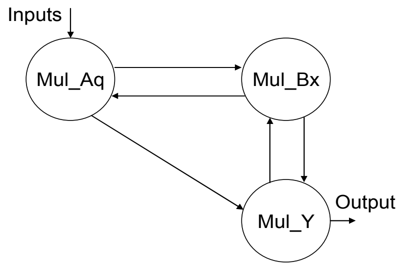

Figure 2: Task set association for a crane controller

Figure 3 shows the execution time-line for the system. In the left side it is possible to see which processor (PE) is running the thread indicated as a box, over the t-axis. The time interval filled up includes the costs to schedule the thread and to receive and send messages.

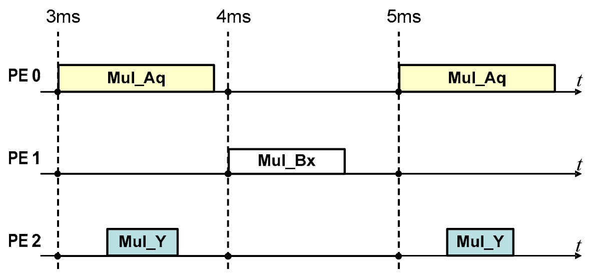

Figure 3: Time-line for crane threads

For this experiment the processors run at 100MHz, and an EDF scheduler was used without DVS. The COM-API uses messages of 500 bytes and packets of 49 bytes. Table 1 shows the execution time for the threads and the system overhead due to the task scheduler and to the communication service. Scheduler overhead depends on the number of threads running, and communication costs depend on the size of the messages. These numbers can be used in the design to evaluate the impact of communication in the total latency. In this particular case, the time spent in the computation of the threads is not too much larger than the time in communication because an aggressive thread-granularity was forced.

Table 2 shows latency values to send and receive messages. Those values do not take in account the latency in the NoC, just the processing time in the PE that sends and/or receives a message. The latencies are high because the protocol stack is implemented in software. This throughput can be optimized using a hardware-implemented communication service [23] or a processor with higher performance [8].

Table 1: Execution times

	Scheduler	Thread	Communication
PE 0	151 μs	938 μs	148 μs
PE 1	150 μs	519 μs	228 μs
PE 2	145 μs	244 μs	205 μs

Table 2: Latencies to send and receive a message

Message Length (bytes)	Sending (μs)	Receiving (μs)
01	48	37
02	51	46
03	53	52
04	56	59
10	109	126
15	160	187
20	174	226
50	252	422
490	3121	3887

To evaluate the memory consumption of the provided services, Table 3 shows code memory (ROM) and variables memory (RAM) sizes. The RTSJ column gives the cost to manage real-time threads and includes classes like scheduler, time representation, event handling, and so on. The communication column brings two different cases. The Pack49-Msg500 column shows data from the case study presented here, which uses more RAM space than a case where the packet size is reduced to 7 bytes and messages to 20 bytes. In both cases, the code size does not change.

To evaluate energy issues the application was run for 50 ms. Only dynamic energy was considered in this experiment because static energy is not relevant for the given technology (0.18 μm).

Table 3: Memory usage

	RTSJ	Communication	
		Pack7-Msg20	*Pack49-Msg500*
ROM	4491	4493	4493
RAM	177	913	6345

Table 4 shows the total energy consumption in each core. It reflects the energy consumed in the processor for all the instructions executed from start until the simulation stops (50 ms). Figure 4 shows the PEs connected in the routers. The energy consumption in the routers and in the links are shown. The energy below the routers indicates the total value for the router, including arbiter, crossbar, and the buffers, although the simulator provides each value separately.

The energy consumption in the network is not significant, if compared to the energy in the processors, because the application does not produce a large traffic.

Table 4: Energy consumption for each core

	PE 0	PE 1	PE 2
Total energy (μJ)	325.64	287.27	267.45

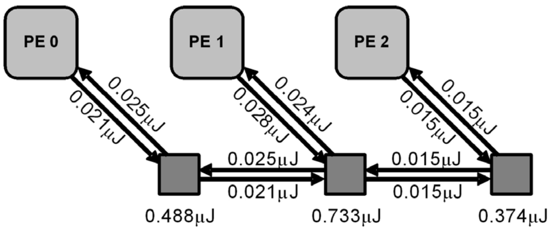

Figure 4: Energy consumption for the NoC

The simulator allows the extraction of energy values for each method in the Java code. Table 5 shows the energy consumed in the processor exclusively when running communication processes. These numbers can be used in the design to evaluate the impact of communication in the total energy cost, as seen in Table 4.

Table 5: Energy consumption in communication

	PE 0	PE 1	PE 2
Send Msg (μJ)	0.749	0.754	0.419
Receive Msg (μJ)	1.140	0.940	0.894

Figure 5 shows the energy consumption in the processing element PE 0 along the first nine milliseconds of execution. The task Mul_Aq starts to be scheduled at 3 ms. The system initialization occurs from 0 to 1 ms. The shadowed regions represent the task execution time. Since the task computation takes approximately 1 ms and its period is equal to 2 ms, the PE remains idle for another 1 ms between two task schedulings.

The curve inclination represents the power dissipated in the core. Note that during task computation more energy is spent in the core, with regard to the period where the PE is idle. The energy spent in each task execution is approximately 8 μJ, while when the processor is idle the energy spent in almost the same interval is about 3 μJ.

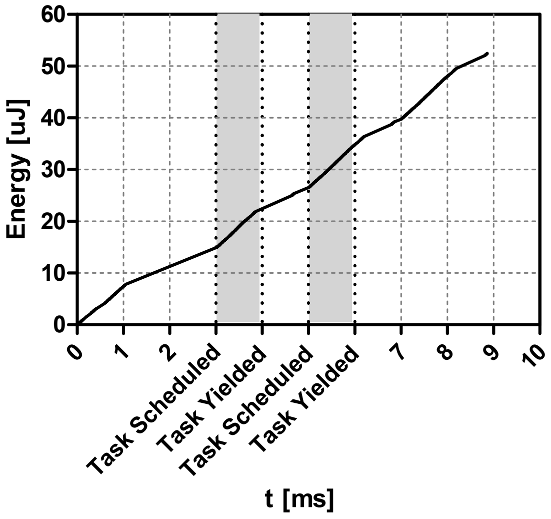

Figure 5: Energy consumption over time

The simulation time for this case, with three cores running at 100 MHz, was 16.8 seconds for each 1 ms of execution in the real system, when running in a Pentium4 2.67GHz with 1Gbyte of RAM.

7. Conclusions and future work

This paper described a framework to accelerate development and test of application software, low-level software, and hardware components for an MPSoC platform. The environment offers support for multithread real-time embedded object-oriented applications and allows design space exploration, evaluating performance and power costs of the application under development.

This platform emulates the real hardware and can help to shorten cycles for embedded system development, integration, and test.

A real-time application is used to demonstrate capabilities and results on latencies and energy consumption.

This work is the structure layer of a larger project that provides a flexible and reusable middleware to raise the abstraction level to develop real-time applications under an MPSoC (Multiprocessor System-on-Chip) platform.

Acknowledgements

The work in this paper is partially sponsored by Brazilian funding agency CNPq.

References

[1] Stankovic, J.A. “Misconceptions about Real-Time Computing: A Serious Problem for Next-Generation Systems”. IEEE Computer, Oct. 1988, v.21, n.10, p.10-19.

[2] Bollella, G. et al. “The Real-Time Specification for Java”, http://www.rtj.org/rtsj-V1.0.pdf, 2001.

[3] OMAP Technology – Texas Instruments, http://www.omap.com/

[4] NXP Semiconductors, www.nxp.com/

[5] CoWare corp., http://www.coware.com/

[6] Saint-Jean, N. et al. “HS-Scale: a Hardware-Software Scalable MP-SOC Architecture for Embedded Systems”, in Proceedings of VLSI 2007. IEEE Computer Society, pp. 21-28, 2007.

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

[8] Beck Filho, A.C.S. and Carro, L. “Low Power Java Processor for Embedded Applications”, In: IFIP VLSI-SoC’2003, Darmstadt, 2003, pp. 239-244.

[9] Wehrmeister, M.A., et al. “Optimizing the Generation of Object-Oriented Real-Time Embedded Applications Based on the Real-Time Specification for Java”, in Proceedings of DATE 2006, IEEE Computer Society. Munich, Germany, pp. 806-811, 2006.

[10] Rosa Jr. L.S., et al. “Scheduling Policy Costs on a Java Microcontroller”, in Proceedings of the JTRES, 2003, pp. 520-533.

[11] Zeferino, C.A.; Susin, A.A. “SoCIN: A Parametric and Scalable Network-on-chip”, in Proceedings of SBCCI 2003, IEEE Computer Society. p. 169-174, 2003.

[12] Wehrmeister, M.A., Becker, L.B., Pereira, C.E. “Optimizing Real-Time Embedded Systems Development Using a RTSJ-based API”, in Proceedings of JTRES 2004. Springer LNCS, pp. 292-302, 2004.

[13] Leung, J.Y.T. and J. Whitehead. “On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks”. Performance Evaluation, v.2, n.4, p. 237-250, 1992.

[14] Liu, C.L. and Layand, J.W. “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment”. Journal of the Association for Computer Machinery, v. 20, n.1, p.46-61, 1973.

[15] Pillai, P. and Shin, K.G. “Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems”, in Proceedings of 18th ACM Symp. on Operating Systems Principles 2001, pp. 89-102, 2001.

[16] Silva Jr., E.T. et al. “Java Framework for Distributed Real-Time Embedded Systems”, in Proceedings of ISORC 2006, Gyeongju, Korea, pp. 85-92, 2006.

[17] Silva Jr, E.T.; Andrews, D.; Pereira, C.E.; F.R. Wagner. “An Infrastructure for Hardware-Software Co-design of Embedded Real-Time Java Applications”, in Proceedings of ISORC 2008. IEEE Computer Society. Orlando, USA, pp. 273-280, 2008.

[18] Wang, H. “Orion: A Power-performance Simulator for Interconnection Networks”, in Proceedings of ACM MICRO. Istanbul, Turkey, pp. 294-305, 2002.

[19] Jalabert, A. et al. “XpipesCompiler: A Tool for Instantiating Application Specific Networks on Chip”, in Proceedings of DATE 2004. IEEE Computer Society. Paris, France, pp. 884-889, 2004.

[20] Ye, T.; DeMicheli, G.; Benini, L. “Analysis of Power Consumption on Switch Fabrics in Network Routers” , in Proceedings of DAC 2002, pp 524-529, 2002.

[21] Taiwan Semiconductor Manufacturer Company: http://www.tsmc.com.tw.

[22] Moser, E. and Nebel; W. “Case Study: System Model of Crane and Embedded Control”. ”, in Proceedings of DATE 1999. IEEE Computer Society. Munich, Germany, pp 721-723, 1999.

[23] 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.