Hardware support in a middleware for distributed and real-time embedded applications

Elias T. Silva Jr[1,2], Flávio R. Wagner[1], Edison P. Freitas[1], Leonardo Kunz[1], Carlos E. Pereira[1]

[1] PPGC – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil
[2] Departamento de Telemática, CEFET-CE, Fortaleza, Brazil
e-mail: (etsilvajr,flavio,epfreitas,lkunz)@inf.ufrgs.br, cpereira@ece.ufrgs.br

ABSTRACT

One of the main challenges in the development of tools and methodologies for a multiprocessor real-time embedded system is to reuse already developed software, but at the same time obtaining low memory footprint, low energy consumption, and minimal area, obviously addressing the real-time constraints. This work aims to face these problems at the middleware level. We show that adaptations in the platform architecture, for instance exploring hardware implementations of middleware services, such as task scheduling and communication, can drive better gains in application requirements like energy and performance, which are essential for embedded applications. This approach is coupled with a high flexibility in choosing either a hardware or a software implementation, because services are encapsulated into objects and the application development and the design space exploration at middleware level can be performed independently from each other, in a fully transparent way. Furthermore, the use of the object-oriented approach reduces time-to-market and development costs.

Index Terms: Embedded Applications, Middleware, Real-time Systems, MPSoCs, Energy Efficiency.

1. INTRODUCTION

MPRE (Multi-Processor Real-time Embedded) systems are becoming largely used in several application domains nowadays. A particular group of MPRE systems is called DRE (Distributed Real-time Embedded), and their typical application is remote sensing and telemetry, or even automotive and avionic control systems. DREs are often associated to distributed systems connected by a network that is outside the chips implementing the processors. However, the principles of networked systems, found in a DRE system, can be reused when the network is intra-chip. Even in this case, an MPSoC (Multi-Processor System-on-Chip) platform can make good use of technologies developed in the DRE context. MPSoC platforms are frequently used in domains that require real-time properties.

For those consumer markets of embedded applications, important goals are the reduction of time-to-market and development costs. To fulfill these and other requirements, great efforts are being done in the research of adequate technological support. Particularly, the middleware support has been investigated only in the context of DRE applications. Nevertheless, also in the context of MPSoCs a middleware could be a solution to raise the level of abstraction, helping to achieve shorter development times.

The challenges related with the development of a middleware in a real-time and embedded system are: (1) to reuse already developed software, (2) to address real-time constraints, and (3) to present a small memory footprint and (4) a low power consumption. The challenge in developing an MPRE system that addresses all those features comes from the fact that the hardware components of MPRE systems are generally elementary devices, with limited CPU capacity and low memory.

A middleware can solve these problems by managing run-time adaptability and leaving the programmer unaware of them. The middleware resides just below the application software and, hence, it is often best suited to monitor various application-specific runtime data. Since applications mainly drive communication and other costs, a middleware-oriented approach can provide the greatest benefit, besides providing services in a higher abstraction layer.

Another way to use the middleware is as a framework to encapsulate hardware devices into objects, thus reducing development time, while simultaneously achieving real-time predictability, better performance, and lower energy consumption. When designing dedicated applications, this approach will allow the exploration of hardware- or software-implemented services.

This paper presents hardware implementations of task management and communication services. Since they are transparently encapsulated into the middleware, the designer can easily explore alternative hardware and software services, looking for the middleware configuration that best matches the application requirements, without affecting the application development.

The middleware operation implies communication, so it requires a communication support in order to provide the distribution of the tasks to be performed by the system. The first step in the development of a middleware is thus to define a communication infrastructure that makes possible the data exchange among system components. In this work, this support is provided by a transport layer communication API that provides real-time guarantees [1]. Since the middleware is being developed in Java and focuses on real-time applications, an API that implements RTSJ (Real Time Specification for Java) is required [2].

The remaining of the paper is organized as follows. Section 2 gives an overview about related work in the area. In Section 3, the development platform is presented. The hardware implementations under the middleware are presented in Section 4, highlighting communication resources. In Section 5, experimental results present area, time, and energy measures in the use of the hardware services, providing comparisons to their counterparts in software. Finally, in Section 6 concluding remarks are drawn.

2. RELATED WORK

Although significant amount of work has been done in middleware for embedded and/or real-time systems, they are not optimized for the MPSoC context and only a few have addressed energy-efficiency as their main foci. We highlight the work carried out by Yau et alii [3], where an ORB (Object Request Broker) is implemented in hardware to achieve high performance. However, the focus of their work is context-sensitivity and reconfigurability for mobile and ad-hoc networks, and no design exploration was performed.

On the other hand, many works have proposed the implementation of operating system services in hardware, particularly task management services. Burleson et al. present the Spring scheduling co-processor [4], which was built to improve task and resource management services of an operating system. FastHard [5] is a multitasking stand-alone real-time kernel in hardware for single processor systems. The custom hardware, implemented in an FPGA, is used to execute the functionality of the priority scheduler. In [6], the implementation of a parameterized scheduling algorithm in an external FPGA is presented. In this proposal, the scheduling algorithm may be changed during run-time without reprogramming the FPGA. This is achieved by implementing in a single circuit several algorithms, which in fact share the same hardware components. The proper scheduler discipline is chosen through manipulation of some parameters. In [7], the authors describe the hardware design of a priority scheduler module developed as part of a multithreaded RTOS (real-time operating system) kernel. They extend the multithreaded programming model to abstract the FPGA components, which are attached to the CPU bus. The work presented in [8] describes the Real-Time Task Manager (RTM). It supports in hardware a few of the common RTOS operations that represent performance bottlenecks, like task scheduling, time management, and event management. The goal of the RTM is to increase the performance obtained by the RTOS.

These previous approaches focused on the implementation of specific services (task management) of real-time operating systems in order to take advantage of the parallel nature of the hardware implementation. Thus, the known overhead introduced into the system by these services, when implemented in software, is significantly reduced. In our work, the purpose is to extend this idea, by encapsulating hardware implementations of operating systems services (task management and also communication) into objects, thus using a framework to reduce development time. This will allow the exploration of hardware- or software-implemented services of the middleware, when designing dedicated applications.

There are few works describing extensions to the RTSJ to achieve distribution. An initial framework [9] integrating the RTSJ with RMI proposes a three-level approach. Level 0 – no guarantee of timely delivery, level 1 – real-time remote object, and level 2 – distributed thread model. Borg and Wellings [10] explore facilities that must be provided by a real-time RMI (RT-RMI), focusing on the integration level 1, where the notion of a real-time remote object is introduced and supported by a real-time RMI that provides timely invocation guarantees. Their work differs from that presented in this paper since they assume a real-time network and consider the real-time aspects at a higher level, focusing on the remote invocation of threads. Our work, in turn, considers facilities at a lower abstraction level, providing a unicast/broadcast mechanism to exchange messages meeting time restrictions. Moreover, our development is focused on embedded platforms with restricted performance and tight memory resources, while RT-RMI does not consider these restrictions.

3. DEVELOPMENT PLATFORM

A. JAVA-RT Configurable Processor

The development platform used in this work is the FemtoJava processor [11], a stack-based microcontroller that natively executes Java byte-codes, whose major characteristics are a reduced and configurable instruction set, Harvard architecture, and small size. It implements an execution engine for Java in hardware, through a stack machine that is compatible with the specification of the Java Virtual Machine (JVM). A compiler that follows the JVM specification is used and allows the synthesis of an ASIP (application-specific integrated processor) version of FemtoJava. For real-time applications, a multi-cycle version of FemtoJava is used. The supported instructions, a subset of the JVM bytecodes, are basic integer arithmetic and bitwise operations, conditional and unconditional jumps, load/store instructions, stack operations, and two extra byte-codes for arbitrary load/store. Additionally, in [12], the instruction set of FemtoJava was expanded do support RTSJ, with the inclusion of bytecodes put-field, getfield, invokevirtual, invokespecial, and instanceof. In this processor, all instructions are executed in 3, 4, 7, or 14 cycles, because the micro-controller is cacheless and several instructions are memory bound. In order to support multithread applications, two pseudo-bytecodes, save-ctx and restore-ctx, were created to provide context switching [13].

B. Design and Simulation Tools

The Sashimi environment [11] is used to generate customized code for the application. The code includes the VHDL description of the processor core and ROM (programs) and RAM (variables) memories and can be used to simulate and synthesize the application. Sashimi, as an example of JVM optimization for embedded systems, eliminates all un-referenced methods and attributes, automatically customizing the final code.

The Sashimi environment has been extended to incorporate an API [2] that supports the object-oriented specification of concurrent tasks and allows the specification of timing constraints, implementing the RTSJ standard. These facilities increase the code abstraction level and optimize the development of real-time embedded systems. The intent is to minimize architecture-dependent characteristics within the scheduling algorithms, thus making the framework as general as possible.

The RTSJ-API uses the concept of schedulable objects, which are instances of classes that implement the Schedulable interface, for instance the RealtimeThread. It also uses a set of classes to store parameters that represent a particular resource demand from one or more schedulable objects. The ReleaseParameters class (superclass of AperiodicParameters and PeriodicParameters), for example, includes several useful parameters for the specification of real-time requirements. Moreover, the API supports the expression of the following elements: absolute and relative time values, timers, periodic and aperiodic tasks, and scheduling policies. The term 'task' derives from the scheduling literature, representing a schedulable element within the system context. It is also a synonym for schedulable object.

4. HARDWARE IMPLEMENTATIONS

Since our middleware focuses on embedded and real-time applications, hardware implementations of some components can be useful to fulfill deadlines and reduce energy consumption.

Application developers want to use a friendly interface when choosing among hardware- or software-implemented resources. This means that applications should be developed without caring about the physical implementation of middleware and platform resources. This transparency is provided by the middleware, encapsulating communication and scheduling resources.

Figure 1 shows the overall platform architecture. A middleware encapsulates communication facilities (APICOM) and manages RTSJ resources. The APICOM works together with the RTSJ-API, using the FemtoJava features to provide communication via a network interface.

The implementation of RTSJ includes a hardware implementation of the real-time scheduler. Likewise, some communication services, when implemented in hardware, are encapsulated by the API-COM block.

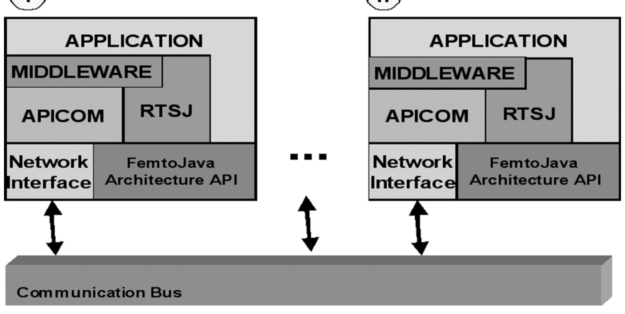

Figure 1. General Platform Architecture.

A. RT-Scheduler

When implemented in software, the scheduler object consists of an additional runtime process (or task) that is in charge of allocating the CPU for those application-processes that are ready to execute, exactly as in any RTOS. Application developers should choose the most suitable scheduling algorithm at design time. Later on, this algorithm is synthesized with the scheduler process into the embedded target.

The hardware scheduler object has the same responsibilities. However, its hardware component contains additional tables that store task descriptors sent by the FemtoJava processor, as well as operators to manipulate those tables. A class called HardwareScheduler, which interacts with the real hardware and performs context switching and dispatching, encapsulates the hardware. Context switching and dispatching imply a minimum cost when compared to the scheduling computation, especially when using complex scheduling algorithms.

By moving the scheduling algorithm from software to hardware, this operating system function no longer competes with the application tasks for the processor. Now, the scheduling function has its own dedicated hardware unit, which is able to run more complex scheduling algorithms and to provide a really non-intrusive task scheduling, thus enhancing the tasks' temporal predictability.

The architecture for the hardware scheduler is shown in Figure 2. The main components of this hardware scheduler are General Register, Scheduler, SyncEvent, and AsyncEvent. Each sub-system is an autonomous machine with its own datapath and control part. They contain a set of registers and a control logic that, considering its inputs and register contents, generates output signals and changes register values. Details of each sub-system and their external connections, as well as on the encapsulation of the hardware scheduler into objects, are shown in [14].

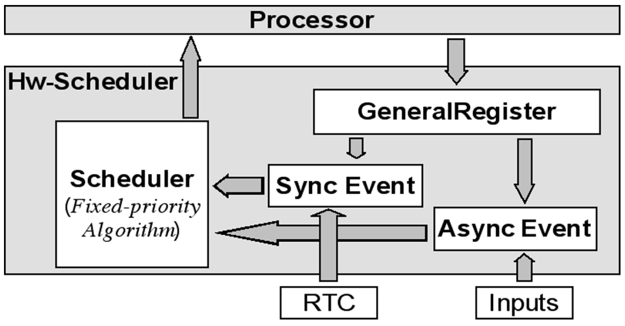

Figure 2. Architecture for the hardware scheduler.

B. Communication Support

In order to provide real-time communication facilities, a communication API was developed for the real-time FemtoJava processor, providing an interface in the transport layer [1].

The communication system provides message exchange among applications running in different FemtoJava 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, in a client-server mode, or even to run in a publish-subscribe mode.

A general description of the services that are provided by the communication API is given in Table 1.

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

When sending a message, the HwTransport class reads the Message object passed as a parameter and delivers it to the hardware. Likewise, when receiving a message, the class fills a Message object passed by the application with data read from the hardware. These operations are transparent to the programmer, since they are encapsulated in the HwTransport class, which has the same interface provided by Transport Class, which implements all operations in software.

Table 1. Services provided by the communication API

Service	Description
Establish connection	Applications can request and wait for connections. The API provides a code that identifies the connection and is used to send and/or receive messages.
Exchange message	Applications exchange information by sending and receiving messages, which are sequences of up to 20 bytes.
Establish a logic ocal address	Applications can set their own addresses, which will be used to identify stations.
Message broadcast	Messages can be sent directly to a specific host, through a predefined connection, or broadcasted in the network. This option is made by calling different primitives of the API when sending a message. A host needs to perform a subscription in order to receive broadcast messages.

Figure 3 shows the general architecture of the hardware that implements communication. It has been described in VHDL in order to be synthesized in an FPGA, together with the FemtoJava processor. The Network Interface is the block that put packets in the physical layer. The current implementation uses a bus that is synchronous and applies a bit-dominance protocol (CSMA/AMP – Carrier Sense Multiple Access with Arbitration on Message Priority) [15]. In this protocol there is no collision and the highest priority packets always gain access to the bus. This strategy was chosen to meet real-time constraints.

The OP_READER block receives and interprets commands from the processor and dispatches commands and data to blocks OUTPUT_MESSAGE_STORAGE or CONNECTION_ MANAGER.

The OUTPUT_MESSAGE_STORAGE block is in charge of storing messages that should be sent. As soon as it receives a complete message, it starts interaction with the FRAG block, which will fill the necessary number of packets and deliver them to the Network Interface. After sending each packet, the FRAG block waits for an acknowledgment from the network before sending another packet.

The TIMEOUT block monitors the FRAG operation, looking at the RTC (Real-Time Clock) evolution. If the time allowed to send the message is finished, an exception is communicated to the processor, thus providing the application with a way to recover the control when the message fails trying to get access to the bus.

PACK_SOLVER is able to identify the type of packet arriving from the physical layer. If it is a data packet, it is sent to the DEFRAG block, otherwise it is a control packet and must be sent to the CONNECTION_MANAGER.

The DEFRAG block receives packets from the Network Interface. When a message is ready, it is sent to the INPUT_MESSAGE_STORAGE block, which

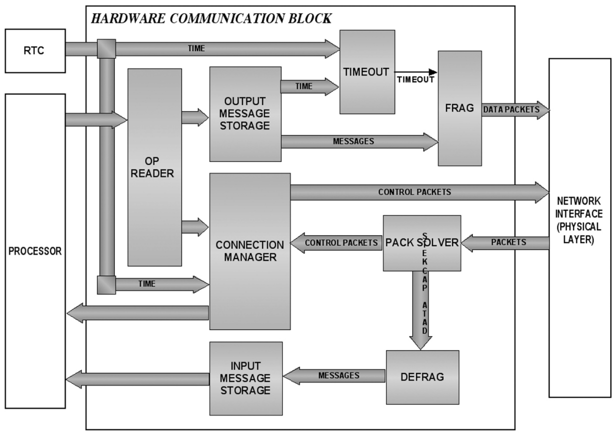

Figure 3. General architecture for the communication service implemented in hardware.

signalizes the message to the processor. Afterwards, INPUT_ MESSAGE_STORAGE delivers the messages to the processor in a pre-defined sequence of bytes.

The CONNECTION_MANAGER block is responsible for opening and closing connections. It also interacts with the TIMEOUT block in order to ensure predictability of tasks when trying to establish connections.

5. EXPERIMENTAL RESULTS

Experiments have been performed to evaluate the hardware implementations when encapsulated in Java objects. The VHDL models were compiled using Altera tools - Quartus II v.5.1. An evaluation in terms of performance, energy, and FPGA area is shown. Java code running on the FemtoJava processor was simulated using a cycle-accurate power simulator [16]. The clock rate of the system (processors and hardware-implemented services) was 20 MHz.

A. RT-Scheduler

A benchmark of 8 synthetic tasks was used to evaluate the scheduler. Details about the task set can be found in [14].

Table 2 shows the cost to run the scheduling algorithm in the software and hardware versions, for several executions. The cost is evaluated in milliseconds. For the hardware scheduler the execution time does not depend on the number of tasks, since the times to communicate with an I/O device and to switch task contexts are fixed. Thus, increasing the number of tasks does not affect the performance of the hardware scheduler. The software scheduler, however, scans a table of added tasks for verifying which task is ready to run, such that its performance is affected by the number of tasks and by the position of the selected task in the table. The software scheduler also has a high cost because it is developed using a high-level language (Java) and uses the object-oriented paradigm.

The energy consumed when running the scheduler was measured by capturing the total gate capacitance switching, which is proportional to the dynamic power. Table 3 shows the energy consumed by the processor when running the scheduler. The cost running the application was not taken into account. All experiments were performed using the hardware scheduler for 8 tasks. When a hardware scheduler is used, around 99% of the total energy is due to the processor. As shown in the table, the energy consumed when using a hardware scheduler varies from 17% to 7% of the energy consumed when a software scheduler is used.

Table 2. Time consumed by the Scheduler.

Num. Of Tasks	Execution Time (ms)			
	Hardware Scheduler	Software Scheduler		
		Avg	Min	Max
2	0.0575	0.3348	0.2285	0.3639
4	0.0575	0.5142	0.3728	0.5984
8	0.0575	0.7779	0.5853	0.9780

Table 3. Gate capacitances switching when Scheduler runs.

Num. Of Tasks	Switching of gate capacitances			
	Hardware Scheduler	Software Scheduler		
		Avg	Min	Max
2	2.02E+06	16.4E+06	13.8E+06	18.9E+06
4	2.02E+06	18.3E+06	13.8E+06	20.7E+06
8	2.02E+06	27.4E+06	20.4E+06	33.8E+06

The area for the hardware scheduler heavily depends on the number of tasks to be managed. An area of 3305 logic cells is required for 4 tasks, and 15181 cells are needed for 8 tasks. The hardware scheduler has a relatively large area, since the FemtoJava processor costs only nearly 3500 logic cells (but we are using a very simple multi-cycle version of the microcontroller, and the hardware scheduler would not represent a too large area overhead in case of more complex processors). This area overhead is the price to be paid for a lower time overhead, real-time predictability, and lower system energy cost. Actually, around 70% of the hardware scheduler area is due to the SyncEvent block, which is not exactly the scheduler, but an event detector block, which was designed for maximum parallelism.

The ratio between the power consumption of the software scheduler and the total CPU power and the performance of the software/hardware RT-schedulers are shown in [14].

B. Communication Support

The communication service was evaluated using a producer-consumer benchmark that sends 20 messages whose lengths vary from 1 up to 20 bytes. Time spent sending and receiving messages is shown in Figure 4, with the x-axis indicating the length of messages. An evident aspect in Figure 4 is the step seen when the length of the message increases from 7 to 8 bytes or from 14 to 15 bytes. This happens because the API needs to use one more packet to send the message. In this example, the packet can carry on 7 bytes. This cost is related to the fragmentation/re-assembly procedure.

Using the hardware implementation under APICOM, latencies to send and receive messages are largely reduced, as can be seen in Figure 5. For messages with 7 bytes, for instance, the transmission latencies are reduced from 0.201 ms, in the case of the

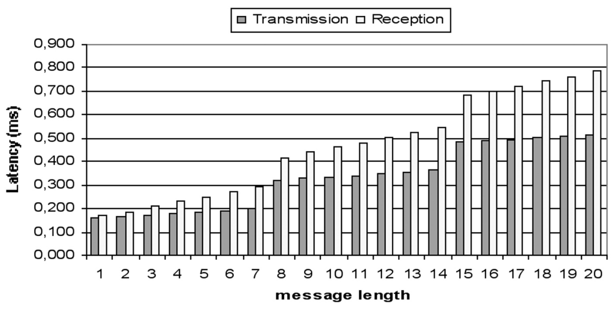

Figure 4. Communication latencies (sw implementation).

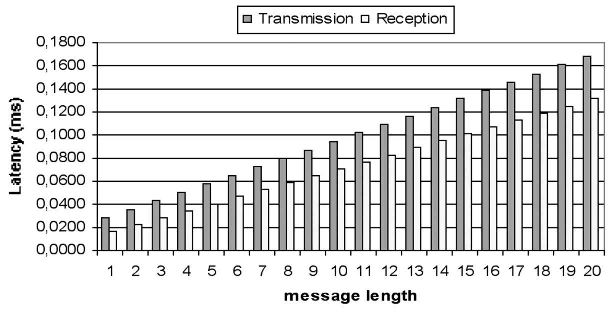

Figure 5. Communication latencies (hw implementation).

software implementation (see Figure 4), to only 0.073 ms. The cost shown is due to the operations performed by the class HwTransport. The time spent by the hardware to build packets and to deliver them to the physical layer of the network is negligible. Actually, the hardware uses 6 clock cycles to deliver a full message, which means less than 1 μs considering its operating frequency.

A noticeable aspect is that the latency grows up in a linear way, when messages are sent through a hardware device. This occurs because the processor only delivers the message contents to the hardware device, which manages fragmentation operations in a few clock cycles. It is also interesting to notice that, as opposed to the software implementation, the transmission cost is now larger than the reception one. This happens because, in a send process, the API needs to deliver information about the message to the hardware block, while during reception the hardware signalizes to the processor only when a message is ready, and the processor just reads the message. The ratio between the software communication service time and the total CPU time is presented in [1].

The energy is strongly related to the time spent by Java classes processing messages. Thus, the energy is proportional to latencies and its curve presents a shape similar to Figure 4 and Figure 5. Figure 6 shows a comparison of the values of the energy costs involved sending and receiving a message using both hardware and software-implemented services, for different message lengths. The energy spent in the software version was divided by the energy spent when

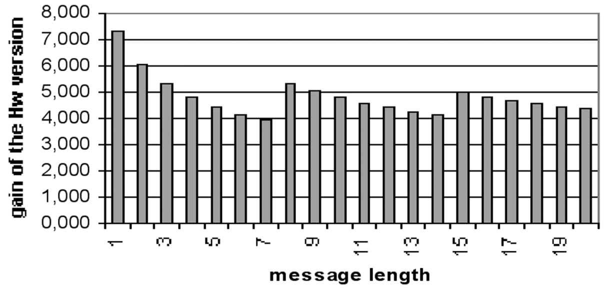

Figure 6. Switching capacitances in the communication (software / hardware comparison)

using the hardware version, for each length of message. The results indicate a gain of 7 times in the better case (messages of 1 byte) and of almost 4 times for messages of 7 bytes.

Table 4 shows the area, in logic cells, of each block of the communication hardware. The extra hardware cost cannot be neglected if compared to the FemtoJava processor cost.

Table 4. Logic cells used by communication hardware.

Sub-system name	*Logic Cells*
OP_READER	47
OUTPUT_MESSAGE_STORAGE	857
FRAG	399
TIMEOUT	43
DEFRAG	855
INPUT_MESSAGE_STORAGE	357
CONNECTION_MANAGER	202
PACK_SOLVER	89
Total	2849

6. CONCLUSIONS

Middleware approaches that propose adaptability usually supply the application with information to adapt itself or to provide software adaptation in the middleware services. This paper shows that adaptations in the platform architecture, by using hardware services, can provide gains in energy consumption and performance.

The paper proposes a design space exploration approach in developing middleware-based object-oriented real-time embedded applications. The developer can choose hardware or software-implemented services, which include a real-time thread scheduler and inter-processor communication facilities. Latencies and costs in energy and area, for both implementations, were evaluated. While hardware services present smaller execution times, smaller energy consumption, and higher real-time predictability, they in turn occupy a large area and thus also imply larger power consumption.

This approach is coupled with a high flexibility in choosing either a hardware or software implementation in order to meet application requirements, because the services are encapsulated into objects and the application development and the design space exploration at middleware level can be performed independently from each other, in a fully transparent way. Furthermore, the use of the object-oriented approach reduces time-to-market and development costs.

The paper focused on design space exploration at design time. The authors are currently working on mechanisms to provide run-time adaptability in an embedded environment with dynamic load, optimizing metrics like energy or performance according to application requirements.

ACKNOWLEDGEMENTS

Thanks to the Brazilian funding agency CNPq, which is the project sponsor. The authors also thank to the other SEEP-project researchers for the valuable discussions.

REFERENCES

[1] Silva Jr., E.T., Freitas, E.P., Wagner, F.R., Carvalho, F.C., and Pereira, C.E. "Java Framework for Distributed Real-Time Embedded Systems", in *Proceedings of 9th IEEE ISORC*, 2006, pp. 85-92.
[2] 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, Proceedings Springer LNCS*, 2004, pp. 292-302.
[3] Yau, S.S. et al. "Reconfigurable Context-Sensitive Middleware for Pervasive Computing", *IEEE Pervasive Computing*, vol.1, no.3, Jul/Sep-2002, pp. 33-40.
[4] Burleson, W. et al. "The Spring Scheduling Co-Processor: A Scheduling Accelerator", *IEEE Transactions on VLSI Systems*, vol.1, no.7, Mar-1999, pp. 38-48.
[5] Lindh, L. "Fasthard - a Fast Time Deterministic Hardware based Real-time Kernel", in *Proceedings of IV Euromicro Workshop on Real-Time Systems*, 1992, pp. 21-25.
[6] Kuacharoen, P., Shalan, M., and Mooney, V. "A Configurable Hardware Scheduler for Real-time Systems", in *Proceedings of International Conference on Engineering of Reconfigurable Systems and Algorithms - ERSA*, 2003, pp. 96-101.
[7] Agron, J., Andrews, D., Finley, M., Komp, E., and Peck, W. "FPGA Implementation of a Priority Scheduler Module", in *Proceedings of the 25th IEEE RTSS*, *WIP*, 2004.
[8] Kohout, P., Ganesh, B., and Jacob, B. "Hardware Support for Real-time Operating Systems", in *Proceedings of 1st IEEE/ACM/IFIP International Conference on HW/SW Codesign and System Synthesis*, 2003, pp. 45–51.
[9] Wellings, A., Clark, R., Jensen, D., and Wells, D.A. "Framework for Integrating the Real-Time Specification for Java and Java's Remote Method Invocation", in *Proceedings of the 5th IEEE ISORC*, 2002, pp. 13-22.
[10] Borg, A. and Wellings, A. "A Real-Time RMI Framework for the RTSJ", in *Proceedings of the 15th Euromicro Conference on Real-time Systems*, 2003, pp. 238-246.
[11] Ito, S.A., Carro, L., and Jacobi, R.P. "Making Java Work for Microcontroller Applications", *IEEE Design & Test of Computers*, vol.18, no.5, Sep/Oct-2001, pp. 100-110.

[12] 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, pp. 806-811.

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

[14] Silva Jr., E.T., Carro, L., Wagner, F.R., and Pereira, C.E. "Development of Multithread Real-Time Applications Using a Hardware Scheduler", in *Proceedings of 13th IFIP VLSI-SoC*, 2005, pp. 311-316

[15] Wolf, W.H.. *Computer as Components: Principles of Embedded Computing System Design*. Morgan Kaufmann Publishers, San Francisco: 2000.

[16] Beck Filho, A.C.S., Mattos, J., Wagner, F.R., and Carro, L.. "CACO-PS: A General-Purpose Cycle-Accurate Configurable Power Simulator", in *Proceedings of the 16th SBCCI*, 2003, pp. 349-354