An Infrastructure for Hardware-Software Co-design of Embedded Real-Time Java Applications

Elias Teodoro Silva Jr1,2; David Andrews3; Carlos Eduardo Pereira1; Flávio Rech Wagner1
1Instituto de Informática, Federal University of Rio Grande do Sul, Brazil
{etsilvajr,cpereira, flavio}@inf.ufrgs.br
2on leave from Telecommunication and Computing Department, Federal Center of Technological Education of Ceará, Brazil
elias@cefetce.br
3Information and Telecommunication Technology Center, University of Kansas, USA
dandrews@ittc.ku.edu

Abstract

The partitioning of applications into hardware and software is an important issue in embedded systems, opening room for high level specifications as well as the exploration of different implementation strategies. This paper presents a software architecture to specify threads in hardware in the context of the Real Time Specification for Java (RTSJ) standard. There is a Java class that encapsulates hardware components, providing an abstraction layer to the application developer. Below this Java class, a wrapper hardware component provides a standard interface between RTSJ-based software components and the hardware that implements the thread behavior. This approach provides a high flexibility in choosing either a hardware or software implementation, allowing to postpone hardware/software partitioning to the very end of system development. The paper includes some quantitative data from an example containing hardware and software threads. While both implementations are compatible with the rest of the application from an interface point-of-view, they lead to very different timing and area results.

1. Introduction

Real-time and embedded systems can be found from microcontrollers in ticket 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. Moreover, for consumer markets of embedded applications, an important goal is the reduction of time-to-market and development costs.

Designing embedded systems often requires satisfying or optimizing a variety of constraints. These constraints include, among others, performance, cost, and energy consumption. The boundary of the hardware/software partition plays an important role in meeting these 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 only 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 behavior from hardware to software and vice-versa, by defining a uniform programming model for both implementations.

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 [1] is the most prominent example of such popularization in the real-time domain. The RTSJ defines an Application Programming Interface (API) for the Java language that allows the creation, execution, and management of real-time threads, whose correction also depends on the fulfillment of timing requirements.

In [2] a middleware for multiprocessor-based embedded systems is proposed to encapsulate hardware implemented services into objects, reducing development time, while simultaneously achieving real-time predictability, better performance, and lower energy consumption. This work extends that one by including the possibility to manage application threads developed either in software or in hardware. The middleware provides enough flexibility such that, during development time, a good exploration of different
implementation strategies can be performed. This paper describes how to implement threads in hardware providing a friendly interface do connect them to an RTSJ implementation. It focuses on the hardware/software implementation of application threads and not on the multiprocessor communication services provided by the middleware. Examples are presented and used to compare hardware and software implementations, highlighting aspects like real-time and scheduling properties and system service access. The platform implements a subset of RTSJ and natively executes Java bytecodes.

The rest of the text is organized as follows. Section 2 discusses related work. Section 3 gives an overview of RTSJ and the adopted platform. Section 4 shows details about the architecture used to connect a hardware thread to an RTSJ API. Section 5 brings an example of utilization, comparing the CPU utilization of hardware and software solutions and analyzing real-time properties. Finally, in Section 6 concluding remarks are drawn.

2. Related Work

When it comes to the hardware implementation of embedded system functions, designers can choose application parts, system services, or both to deploy as hardware components.

A significant amount of work deals with the implementation of operating system services in hardware, particularly task management services [3] [4] [5] [6]. Regarding communication services, we highlight the work carried out by Yau [7], where an ORB (Object Request Broker) is implemented in hardware. In [2], this idea is extended, by encapsulating hardware implementations of operating systems services (task management and also communication) into objects, thus using a framework to reduce development time.

On the other hand, two approaches may be considered to implement an object in hardware. The first one would be the implementation of a general object in hardware. This approach was followed in JavaMen Framework [8].

Another approach would be to restrict the hardware implemented object, defining it as a thread object. This is not a serious restriction, since a hardware implementation usually intends to increase performance, paying a price in area and possibly saving energy. So, for application components, designers usually will want to move a thread from software to hardware, not a generic object. The HybridThreads platform [9] follows this approach. It allows designers to describe hardware threads (in an HDL) and activate them from a Linux application described using the pthreads standard, although it does neither have real-time properties nor is object-oriented.

The contribution of this work is focused on the implementation of application components in hardware, following the object approach and matching real-time requirements.

3. Real-time Java Platform

3.1. RTSJ API

The Real-Time Specification for Java (RTSJ) [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 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 cyclical activation and deadlines.

This work uses an RTSJ-based API, presented in [10]. The implementations of some of the proposed API classes have slight differences in comparison to the RTSJ standard. This is due to constraints in the adopted platform and also for clarity matters. 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.

3.2. JAVA-RT configurable processor

The platform used in this work is the FemtoJava processor [11], a stack-based microcontroller that natively executes Java bytecodes, 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.

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
bytecodes for arbitrary load/store. For real-time applications, a multicycle version of FemtoJava is used. In this processor, all instructions are executed in a fixed number of cycles (3, 4, 7, or 14 cycles), because the microcontroller is cacheless and several instructions are memory bound. After the version published in [11], bytecodes used for object operations were also included, like putfield, getfield, invokevirtual, invokevirtual, and instanceof. In order to support multithread applications, two pseudo-bytecodes, save-context and restore-context, are provided for context switching [12]. An environment called Sashimi is used to generate customized code for the application. It reads the application classes and generates the final code, which includes the HDL description of the processor core (whose ISA contains only instructions used by the application software) and ROM (programs) and RAM (variables) memories. Those generated codes can be used to simulate and/or synthesize the target application. Sashimi is an example of JVM optimization for embedded systems. It provides a powerful and easy-to-use development environment for embedded systems that has been successfully applied to different case studies.

In the development of applications under the FemtoJava-Sashimi platform, applications’ objects must be statically allocated at design time. In other words, all objects in the system are defined a priori, allowing the determination of the total memory necessary to hold them in the RAM. Although such practice incurs into higher memory usage, it is a suitable practice in real-time development, as it avoids the use of the garbage collector, whose overhead is hard to be managed in real-time applications.

4. Architecture

In order to implement an object in hardware in the context of RTSJ, two approaches were considered: implementing a general object or restricting the hardware implemented object, defining it as a thread object. The second one was chosen due to its simplicity. A hardware component to interface to a hardware-implemented RealtimeThread is less complex than one to interface a generic object, which would be able to invoke unlimited methods with unlimited parameters.

Figure 1 shows a HwTI that encapsulates a HwTB and both are implemented in hardware. The first one is the interface between the processor and the hardware object, called HwTI – Hardware Thread Interface. The other hardware component implements the behavior of the thread and is called HwTB – Hardware Thread Behavior, and should be provided by the developer.

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

In 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 FemtoJava processor. This alternative is better described in [10]. A hardware implementation is an HDL code that 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.

![Hardware thread simplified architecture](image)

4.1. Hardware architecture

The component HwTI defines communication operations between the hardware thread and the rest of the system. It has an interface to the processor bus and another to the hardware thread. The processor side offers address-mapped registers, accessible by the software. The other side, which connects to HwTB, offers the signals shown in Table 1.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>intrfc2thrd_value</td>
<td>Interface returns the value read from the memory to the thread</td>
</tr>
<tr>
<td>intrfc2thrd_function</td>
<td>Interface tells to the thread the function it should execute</td>
</tr>
<tr>
<td>intrfc2thrd_rdy2recv</td>
<td>Interface tells it is ready to receive the next opcode</td>
</tr>
<tr>
<td>thrd2intrfc_address</td>
<td>Hardware Thread tells the address it wants to read or write in the memory</td>
</tr>
<tr>
<td>thrd2intrfc_value</td>
<td>Hardware Thread tells the data it wants to write in the memory, or the id of the method to call</td>
</tr>
<tr>
<td>thrd2intrfc_function</td>
<td>Hardware Thread tells the function it wants to be executed by the processor</td>
</tr>
<tr>
<td>thrd2intrfc_opcode</td>
<td>Hardware Thread tells the operation it is asking for</td>
</tr>
</tbody>
</table>

The HwTI accesses RAM through the local bus of the processor. When a hardware thread requires a memory operation, the HwTI retains the address (and data, for a write operation) and waits for the next RAM access window, provided by the local bus. This wait can last for
14 clock cycles (execution time of the longest instruction).

4.2. Software architecture

Some opcodes are defined such that the hardware thread can invoke services provided by the HwTI. There are also some functions that are services performed by the processor or other hardware and made available to the thread through the HwTI. Other functions are implemented by the hardware thread and activated by the processor through the HwTI. Table 2 shows a list of functions and opcodes and their descriptions.

Using GETDATA and PUTDATA opcodes, a thread asks HwTI to access the RAM, performing read and write operations.

RTSJ methods, as waitForNextPeriod() and waitForEvent(), can be invoked using the CALL opcode. Application methods (implemented in software) can also be invoked from the hardware component.

Table 2: Functions and opcodes for communication with the hardware thread.

<table>
<thead>
<tr>
<th>Opcode (Thread to Interface)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GETDATA</td>
<td>The hardware component asks for a position in the memory</td>
</tr>
<tr>
<td>PUTDATA</td>
<td>The hardware component asks to write a data in the memory</td>
</tr>
<tr>
<td>CALL</td>
<td>The hardware component asks to execute some service (function) in the processor</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Functions (Interface to Thread)</th>
</tr>
</thead>
<tbody>
<tr>
<td>START</td>
</tr>
<tr>
<td>RESET</td>
</tr>
<tr>
<td>CONTINUE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Functions (Thread to Interface)</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAIT_FOR_NEXT_PERIOD</td>
</tr>
<tr>
<td>WAIT_FOR_EVENT</td>
</tr>
<tr>
<td>RUN_METHOD</td>
</tr>
<tr>
<td>EXIT</td>
</tr>
</tbody>
</table>

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

As a RealtimeThread child, HwRealtimeThread class implements a mainTask() method, equivalent to run() in a standard Java thread. This method runs when the scheduler activates the thread and encapsulates the communication protocol to HwTI.

The mainTask() method follows these steps:

1. Set Attribute Pointer Register – This register should contain the address where the application object attributes started in the memory. The hardware component uses this value to access object attributes in RAM.
2. Send a RESET command to the hardware component.
3. Send a START command to the hardware component.
4. Wait for a CALL command, coming from HwTI. It permits the hardware component to synchronize with the processor, if necessary.

If the thread (HwRealtimeThread) is preempted, the hardware component can continue its execution. If the hardware component makes a CALL, it will block until mainTask() returns the answer, i.e., when the scheduler gives the processor back to that thread.

The mainTask() method is implemented in the HwRealtimeThread class as a final method and may not be overridden.

If the thread is a periodic task, it will CALL WAIT_FOR_NEXT_PERIOD (see Table 2) after execution. When mainTask() receives this CALL, it will run this method from RealtimeThread, making the scheduler to allocate another task to the processor and scheduling the hardware thread to its next activation time.

A hardware thread can also be sporadic or non-periodic. In this case, it will invoke waitForEvent() to suspend execution. A thread can always finish using an EXIT function. The mainTask() method will invoke RealtimeThread.finish(), telling the scheduler to no longer activate that thread.

The proof-of-concept version offered a limited number of methods, defined as opcodes METHOD_01 and METHOD_02. Each hardware thread was able to invoke only two methods implemented in software. A slight modification is in progress, using one opcode, called RUN_METHOD. When this opcode is requested, HwTI captures the number of the method, provided by the thread through the signal thrd2intrfc_value.

Figure 2 shows an FSM for an implementation of a periodic hardware thread. When initialized, the hardware component (HwTB) must wait for a START function, from the processor. After START, the thread reads the RAM to get the initial address of the attributes of the HwRealtimeThread object. From this point, the hardware thread can get and put data in the memory, using GETDATA and PUTDATA opcodes. Since the
example in Figure 2 is a periodic thread, it should ask the processor to execute the `waitForNextPeriod()` method. In order to wait for the next execution time, provided by the scheduler, the hardware thread waits for a CONTINUE function.

Although a tool to automatically convert a Java description of the task into an HDL description would be interesting, it is not in the scope of this paper. Other works deal with the generation of hardware from Java language [13][14]. For this work the hardware threads were hand-coded using the FSM shown in Figure 2.

![Figure 2: Simplified FSM for a hardware code of a periodic thread](image)

5. RTSJ Hardware Thread Application Example

To illustrate the use of a hardware thread, a Finite Impulse Response (FIR) filter was implemented. The main class is shown in Figure 3 and is in charge of the start up of the system. The `initSystem()` method is equivalent to a `main()` method and starts executing after the system initialization. At this point there is no difference between hardware and software threads. In lines 5 and 6, hardware and software threads are constructed the same way, and in lines 14 to 17 they are added to the scheduler and started.

A software description of the FIR is shown in Figure 4. It is a periodic thread that passes one input value through the filter at each execution. The `mainTask()` method (lines 20-39) is the body of the thread. Since it is a periodic thread, this method finishes a cycle with a `waitForNextPeriod()` invocation (line 37), when the thread yields until its next activation time. For test purposes the thread runs three times the number of taps of the filter.
Figure 5 presents the code used to encapsulate the hardware thread. The class extends an `HwRealtimeThread` class, from the RTSJ packet. The `FIR10hw` class, which is the Java part of a hardware real-time thread, is described identically to a software one. It should contain `SchedulingParameters()` and `ReleaseParameters()` objects (lines 11-15), set in the constructor (line 18). For this example, the same time and scheduling properties were used. However, the constructor for the hardware thread contains a number to define which thread it encapsulates. This constant is called `myHWTI` in the code (line 5). It will be used by the system to select the correct hardware component, making it possible to run more than one hardware thread in the same application. Figure 5 also does not contain a `mainTask()` method because it is implemented in the parent class and is final. In lines 23 and 24 are the methods that can be invoked by the hardware code, as mentioned in Section 4.2.

```java
import saito.sashimi.*;
import saito.sashimi.realtime.*;
public class FIR10hw extends HwRealtimeThread {
    // Tell the number of this hardware thread
    private static final int myHWTI = 0;
    // Application Attributes
    public final int NTAPS = 10;
    public int[] coefs;
    public int[] input;
    public int[] output;
    // ReleaseParameters definition
    // Defines period, start-time, deadline and other RTSJ params.
    // …………..
    // SchedulingParameters definition
    // …………..
    public FIR10hw() {
        super(myHWTI, schedParam, periodicParam);
    }
    public void exceptionTask() {}
    protected void initializeStack() {}
    protected void method01() {}
    protected void method02() {}
}
```

Figure 5: Software code to encapsulate hardware thread

### 5.1. Experimental results

For experimental verification, a SystemC simulator of the FemtoJava processor was used. The Sashimi [11] environment was used to generate both the processor and customized code for the application software. Code produced by Sashimi includes the VHDL description of the customized processor core and ROM (programs) and RAM (variables) memories.

Two experiments were performed. The first one demonstrates the execution time reduction provided by the hardware thread implementation. The second experiment shows the regularity (predictability) of the system when the number of threads increase.

For the first experiment, FIR filters with 10, 30 and 100 taps were implemented in Java, for the software version, while the hardware versions were described in SystemC. The threads were set to run periodically (3ms) to process a new input value, applying the new value in the delay line of the filter and so on. Table 3 shows time values for software and hardware execution. A fixed priority scheduler was used, and the processor runs at 50MHz.

As one could expect, the results show performance gains for the hardware implementation. The execution period for the threads, set to 3ms by construction, was strictly respected.

It is important to notice that the time spent to execute a CALL opcode is 12.5µs (625 clock cycles), which is the main component in a hardware thread cost (for a 10-tap filter it is almost 100% of the cost). Since a call to the `waitForNextPeriod()` method is performed each activation time, a hardware thread would not offer advantage if a software implementation did not cost something more than 625 clock cycles.

<table>
<thead>
<tr>
<th>FIR filter (software)</th>
<th>FIR filter (hardware)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 taps</td>
<td>46.7</td>
</tr>
<tr>
<td>30 taps</td>
<td>125.1</td>
</tr>
<tr>
<td>100 taps</td>
<td>399.6</td>
</tr>
</tbody>
</table>

The second experiment was performed putting the hardware implementation of the 100-tap FIR filter and other periodic threads (T1, T2 and T3) implemented in software as concurrent tasks. The clock frequency of the processor was set to 50 MHz and the scheduler used was an EDF. Table 4 shows execution time, deadline, and period for the tasks used in this experiment. The execution time values were extracted from simulation results, while period and deadline were set in the code as release parameters. If a task does not run before its activation time plus the deadline value, the scheduler will run the `exceptionTask()`.

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>FIR (hw)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0</td>
<td>2.0</td>
<td>2.0</td>
<td>2.0</td>
</tr>
<tr>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>135.0</td>
<td>135.0</td>
<td>135.0</td>
<td>39.8</td>
</tr>
</tbody>
</table>

Table 5 shows results extracted from the simulation of the second experiment. The first column presents the number of running tasks, including the hardware one.
The next column (CPU utilization) indicates the total time in which the CPU is running tasks or the scheduler in a period. If no task is able to run, then the CPU is in a condition called idle. So, the CPU utilization is equal to \((\text{Period} - \text{idle})/\text{Period}\). Activation time is the instant of activation of a periodic task. It was collected 10 samples, it means, the first 10 activation times for each task. The deviation, in this case, was obtained using the latencies to activate a thread in each period.

The results showed a very stable and predictable system. As the number of tasks running grows up, the execution time of the scheduler grows too. However, the activation time does not suffer a significant jitter.

<table>
<thead>
<tr>
<th>Number of Threads</th>
<th>CPU utilization</th>
<th>Standard deviation of Activation time (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>FIR (hw)</td>
</tr>
<tr>
<td>2</td>
<td>48%</td>
<td>0.09</td>
</tr>
<tr>
<td>3</td>
<td>71%</td>
<td>0.35</td>
</tr>
<tr>
<td>4</td>
<td>96%</td>
<td>0.42</td>
</tr>
</tbody>
</table>

Finally, there is a synthesizable HDL description of the FemtoJava processor. The synthesis of an HDL version of HwTI and some hardware threads is a future goal of this work.

6. Conclusions and future work

The paper presented how developers can use a Real Time Specification for Java (RTSJ) compatible API to implement a multithread object-oriented real-time embedded application, choosing hardware- or software-implemented threads. A hardware component (provided by the platform) works as a wrapper encapsulating the component that implements the hardware thread behavior (provided by the application developer). This wrapper is encapsulated by a Java class that extends the RealtimeThread class from RTSJ.

An example containing hardware and software threads was implemented as a proof of concept. This approach provides a high flexibility in choosing either a hardware or software implementation, allowing designers to relegate hardware/software partitioning to the very end of system development. It opens room to try different sets of objects to implement the application, abstracting hardware or software implementations.

The application development process is not influenced by architectural choices and does not need to consider if some particular thread must be implemented in hardware or software. Moreover, the developer can reuse all the application code if he/she needs to change some thread implementation.

This work is part of a middleware for multiprocessor embedded systems whose purpose is to provide support to real-time embedded and distributed applications using hardware implemented services and hardware implemented threads as well.

The choice between a hardware and software implementation is currently made at design time. The support to runtime selection is left as future work.

References


