Optimizing the Generation of Object-Oriented Real-Time Embedded Applications Based on the Real-Time Specification for Java

Marco Aurélio Wehrmeister[1], Carlos Eduardo Pereira[1,2], Leandro Buss Becker[3]

[1]Computer Science Institute, UFRGS, Brazil
[2]Electrial Engineering Department, UFRGS, Brazil
[3]Automation and Control Systems Department, UFSC, Brazil
{mawehrmeister@inf.ufrgs.br, cpereira@ece.ufrgs.br, lbecker@das.ufsc.br}

Abstract

The object-oriented paradigm has become popular over the last years due to its characteristics that help managing the complexity in computer systems design. This feature also attracted the embedded systems community, as today's embedded systems need to cope with several complex functionalities as well as timing, power, and area restrictions. Such scenario has promoted the use of the Java language and its real-time extension (RTSJ) for embedded real-time systems design. Nevertheless, the RTSJ was not primarily designed to be used within the embedded domain. This paper presents an approach to optimize the use of the RTSJ for the development of embedded real-time systems. Firstly, it describes how to design real-time embedded applications using an API based on RTSJ. Secondly, it shows how the generated code is optimized to cope with the tight resources available, without interfering in the mandatory timing predictability of the generated system. Finally it discusses an approach to synthesize the applications on top of affordable FPGAs. The approach used to synthesize the embedded real-time system ensures a bounded timing behavior of the object-oriented aspects of the application, like the polymorphism mechanism and read/write access to object's data fields.

1. Introduction

The interest in the use of object-orientation and Java in the embedded real-time systems (ERTS) domain has been increasing over the last years [13][14]. An important step was related to the definition of the Real-Time Specification for Java (RTSJ) [19], which specifies both an API to be used in development of real-time applications and also the expected behavior of a RTSJ-compatible Java Virtual Machine (JVM). Example of available RTSJ partially compliant virtual machines are found in [9][15][16]. However, such implementations are not primarily designed to be used within the embedded domain.

While object-orientation definitely increases the abstraction level, it also adds additional overhead. This overhead must be small on ERTS that have tight computational resources (e.g. memory and processing power). To allow the use of the object-oriented (OO) paradigm by means of the Java language in ERTS design, this paper describes the use of a dedicated Java processor, named FemtoJava [1], which has versions from 8 to 32 bits with different organizations (e.g. multicycle, pipeline, VLIW). The ERTS generation and synthesis (Java processor + application code) is made by a synthesis tool named SASHIMI (introduced in [1]), which originally allowed only the synthesis of static methods and attributes. In the current work this tool is extended to support the synthesis of OO code, adding capability to synthesize instance fields and methods, to make static object allocation, and to provide a predictable polymorphism mechanism to the method calls. The adopted approach in the ERTS synthesis ensures a bounded execution time of OO aspects of the application, such as the polymorphism mechanism and the object's instance field access (even for the inherited fields).

The remainder of this paper is organized as follows. Section 2 describes related work dealing with the use of Java in the embedded systems market. The proposed synthesis tool is described in section 3. Section 4 presents our synthesis method and the modifications performed in the original synthesis tool. Some actual data related to the generated embedded system are presented in section 6. Finally, conclusions and future work are presented in section 7.

2. Related Work

According to the Java specification any application developed using the Java platform must be executed in a JVM [6]. A JVM can be implemented as software, which is a layer between the Java application and the real machine, or implemented as hw that executes Java bytecodes natively. Both approaches have pros and cons, and depending on the application requirements and constraints one choice is better than the other. The default JVM spec. can not be used in the real-time domain, specially due to undeterministic timing behavior introduced by garbage collector (GC) and also by inheritance mechanisms. In order to solve such problems, the RTSJ defines how a RTSJ-compatible JVM must behave.

3-9810801-0-6/DATE06 © 2006 EDAA

JamaicaVM [8][22] is an example of JVM implemented as software layer and implements partially the RTSJ specification. JamaicaVM has two ways to execute embedded real-time applications: using a virtual machine layer or executing applications compiled into machine native code that incorporates also the JamaicaVM code. The JamaicaVM GC has a deterministic worst case execution time (WCET), it executes on each object allocation, and frees up to 32 bytes per execution. Another example of RSTJ-based JVM is jRate [9] which extends the GNU Compile for Java (GCJ) execution environment. This means that application code is compiled into target machine native code and therefore there is no JVM layer to execute the real-time Java application. In jRate the real-time threads implementation simply relies upon the underlying real-time operation system scheduler (which usually are preemptive and adopts a fixed priority mechanism). jRate implements only two kinds of memory, both from RTSJ: ScopedMemory and ImmortalMemory.

Both Jamaica and jRate lead to an ERTS with higher memory usage. This is due to the fact that all method codes of application classes must be included into the application binaries, even the unused ones. Also all information present on class constant pool [6] of all application classes must be included into these binaries. Clearly, this issue can be optimized by removing the unused methods and the constant pool. Moreover, OO mechanisms such as object's field access and polymorphism require allocation of additional memory.

On the other hand, the Java Optimized Processor (JOP) [10] is an example of a JVM implemented as hardware. JOP follows the Connected, Limited Device Configuration (CLDC) profile for the J2ME specification [11] and can be implemented using low-cost FPGAs. The GC is not implemented. aJile [12] is another example of JVM implemented in hardware, which also follows de CLDC profile for the J2ME. It implements a simple mark-and-sweep GC as a thread that uses Java synchronization mechanism. Applications can be compiled using a standard Java compiler, however before deploying it on the aJile processor the bytecodes must be adapted using the JEMBuilder. This builder generates RAM and ROM memory images and eliminates unused methods, fields, and constants, optimizing the footprint of the embedded real-time application. aJile has most of the functionality specified in the RTSJ. However, according to [17], compliance to RTSJ "will require business initiatives to pursue". Both processors lack support to the RTSJ and also, for some applications, there is waste of FPGA area as it contains unused instructions.

3. The Synthesis Process

According to our approach, the embedded system design flow starts with the development of a Java application. This application can be implemented and tested into the host development environment. After the functional validation, the classfiles (which are generated by standard Java compiler) are used as input to the SASHIMI synthesis tool, which analyses them and makes additional optimizations into the code in order to generate an optimized and small footprint application code. The SASHIMI synthesis tool also generates a customized Femto-Java that implements only the instructions used by application. In other words, it analyses the application code and customizes the processor control unit in order to support only the used instructions, reducing the processor FPGA size. The synthesis flow can be observed in Fig. 1.

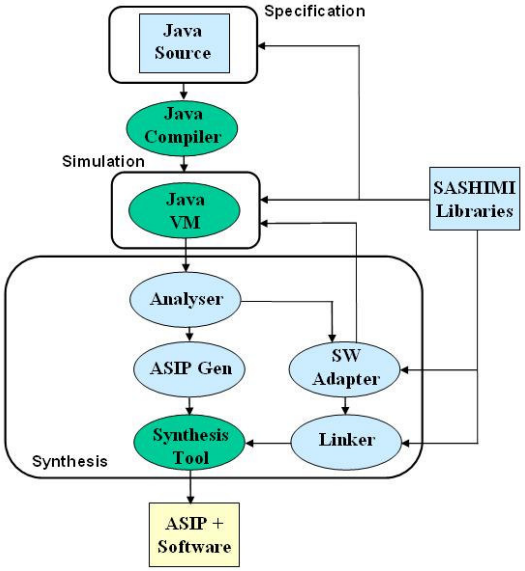

Fig. 1 Synthesis Process Flow

To cope with the synthesis process, designers must develop their Java application following a set of rules. For instance, only integer numbers and APIs provided by the environment are allowed. Moreover, in the original version of the tool designers could use only static methods and attributes, since there was no support for object allocation. This problem is now solved in the present work, as detailed in the next section.

Additionally our approach allows concurrent real-time programming by means of an operating system layer in charge of CPU scheduling. In this model each thread has its own stack, which must be initialized individually. Also, there are different scheduling algorithms which are supported by the environment. An evaluation of the impact of these algorithms in terms of footprint, power consumption and real-time performance are described in [5]. A drawback from the original environment is the lack a high-level API, enforcing designers to write programs "polluted" with low-level system calls to interact with the scheduler. Additionally, there was no mechanism to express clearly the tasks timing constraints. These issues are addressed by our RTSJ-based API. This API provides

classes to express concurrent real-time threads, release and schedule parameters, schedulers, timers, and also absolute and relative time values. Interested readers can refer [2] to obtain details on our RTSJ-based API.

4. Synthesizing Object-Oriented Code

As mentioned in the previous sections, the original version from our synthesis environment provided no support for object creation. Therefore, some adaptations are needed in order to provide full support for the proposed API in our FemtoJava platform. The following subsections describe how the code is synthesized.

4.1. Object allocation

One of the main motivations for working with Java is the related facilities for working with objects. Nevertheless, such observation contrasts with the restrictions from the previous version of our synthesis tool that had no support for objects, allowing designers to work only with static classes. Therefore this work presents an extension from the synthesis tool to support the synthesis of objects.

The adopted scheme followed the proposal from Pushner and Wellings in [18], which defines that all objects in the system must be defined a priori, avoiding overheads related to dynamic memory allocation and management. Such practice avoids the use of the GC, which is the main source of indeterminism in Java applications. As a result, in our approach all application objects are allocated statically at synthesis time. For each application object the SASHIMI tool analyses the class hierarchy in order to identify all fields that an instance of this class must contain. After all, the size and the structure of the object are known at synthesis time. The object allocation follows the model presented in Fig. 2.

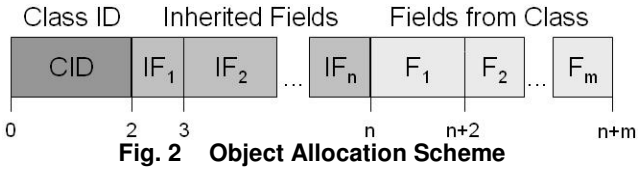

Fig. 2 Object Allocation Scheme

The first field of the allocated object is used to hold the class unique identifier. Each application class has its own number, which is used by the polymorphism mechanism and also by the class related opcodes (e.g. "instanceof" and "checkcast") in order to distinguish instances of different classes. The memory for all inherited fields are allocated after the class identifier field. The highest class fields (i.e. the fields of the most abstract class) are allocated first, followed by its subclasses fields. Finally, the class fields are allocated after the inherited fields. The order of field allocation obeyed by the class hierarchy is as follows: the fields from the topmost class (in hierarchy) are allocated first, followed by the fields from its child class, and so on.

An example of this statically allocation of objects is shown in Fig. 3 containing a fragment of the VHDL-compatible file that was generated using our tool and which represents the FemtoJava RAM memory. In this figure the allocated object represents an instance of *RTTask1Class* class which extends the *RealtimeThread* class, thereby represents a real-time task in the generated embedded system. The class unique identifier can be observed at line 63h. The following nine fields (lines 64h to 6Ch) are the instance fields inherited from the *RealtimeThread* class, and the remaining fields are instance fields from the class.

```
...
63 : 0b;-- ----> ID for RTTask1Class
64 : 00;-- m_relParam[LReleaseParameters]
65 : 00;-- m_schedParam[LSchedulingParameters]
66 : 00;-- m_isStarted [Z.1]
67 : 00;-- m_isFinalized [Z.1]
68 : 00;-- m_isRunning [Z.1]
69 : 00;-- m_isBlocked [Z.1]
6a : 00;-- m_BaseStackPointer [I.1]
6b : 00;-- m_StackPointer [I.1]
6c : 00;-- m_ResumeTime [LAbsoluteTime;.1]
6d : 00;-- m_aux [I.1]
6e : 00;-- m_Loop [I.1]
6f : 00;-- m_SchedCount [I.1]
...
```

Fig. 3 Object statically allocated into memory

Another required adaptation is changing the bytecodes of the Java class files in order to allow objects static allocation. Therefore the reference for the allocated object (RAM memory address of the first attribute in the object) is inserted into the bytecode in substitution of the "new" opcode. All bytecode occurrences related to the Java command "new" are replaced by this fixed memory address. It is important to emphasize that the object is never released from memory, once there is no GC running. For example, in *RTTask1Class* object allocation the instruction "new" is changed to "sipush 63h" (push object reference in the top of the stack).

To allow accessing the object attributes, the "getfield" and "putfield" opcodes [6] also need to be supported in the synthesis. The proposed allocation approach eliminates the need of a class constant pool [6] because the class structure (i.e. fields from class and its superclasses) is analyzed at synthesis time, so it is removed from the synthesized classes. As consequence, the semantic of "getfield and "putfield" opcodes must be modified. The reference to the constant pool was replaced by the memory offset of the field related to the initial address of the object. In summary, to access an object field its memory address should be calculated based on the object initial address and the respective field offset. For example, to access the value of *RTTask1Class*'s m_Loop field that is stored on the 6Eh memory address (see Fig. 3), the "getfield ConstatPoolReferece" instruction is changed to "getfield 0Bh". Therefore when this instruction is executed, the value 63h (reference of the *RTTask1Class* object) is popped from top of stack and be added to 0Bh offset in

order to obtain the RAM address to access m_Loop field value (i.e access the RAM address 6Eh). The "putfield" instruction is changed similarly to the "getfield" one, however the "putfield" writes a value into the calculated RAM address instead of reading the memory address.

The main advantage of this approach is that an object field can be accessed using only two instructions (i.e. object reference push instruction and access field instruction) in contrast to execute a lot of instructions to access constant pool information to know where field's data are stored in memory. A disadvantage is that application class hierarchy must be evaluated at synthesis time, forbidding the instantiation of classes that were not previously defined.

4.2. Predictable Polymorphism Mechanism

Polymorphism is another desirable characteristic from object-orientation that was also not supported in the original version of the SASHIMI synthesis tool. Our goal here is to define a solution to make the polymorphic method invocation predictable. Therefore our mechanism is neither implemented using tables to store the class hierarchy nor overrides method tables as specified in [6].

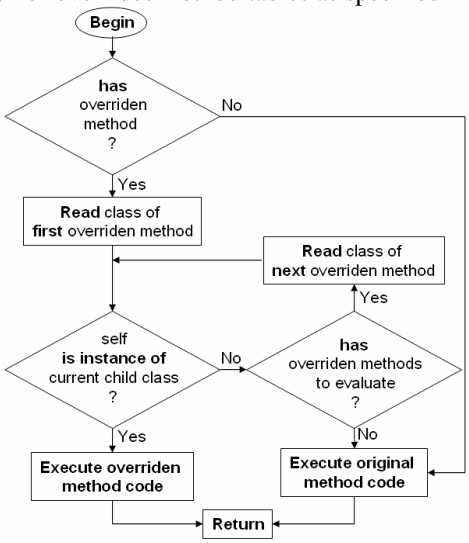

Fig. 4 Algorithm for Polymorphic Method Invocation

The proposed mechanism uses the class hierarchy information extracted from application code that was collected during analysis phase. For each method that is overridden, an extra piece of code is inserted before the users code. This code represents the polymorphism mechanism algorithm shown in Fig. 4.

This algorithm induces the conclusion that the proposed polymorphism mechanism introduces indeterminism into ERTS. Particularly, if the application classes are loaded dynamically, the above conclusion is true because this algorithm is highly dependent on the input (i.e. using dynamic load of classes, the class hierarchy level is evaluated at runtime what introduces indeterminism in polymorphism mechanism). On the other hand, if class hierarchy is known at synthesis time, this algorithm can be unrolled and transformed on an algorithm that follows the single path paradigm [20].

According to Pushner and Burns [20], in the single path paradigm the behavior of algorithms is fully temporally predictable because they execute on the single possible execution path, independently from the input data. Therefore, knowing the class hierarchy the algorithm presented in Fig. 4 can be unrolled into several nested "if" instructions, one for each class that overwrites methods from its superior classes (in hierarchy). The use of single path paradigm fulfills the two identified central facts to determine the WCET of a program: (i) the possible sequence of actions and (ii) the time needed for each action of this sequence [21]. In summary, the WCET for polymorphic method invocation can be determined at synthesis time, using the following formula:

$$f_{WCET}(meth.call) = t_1 + \sum_{i=1}^{n} t_{2_i} + \sum_{i=1}^{n-1} t_{3_i} \quad (1)$$

Where:

- n is the number of subclasses that overwrite the original method
- t_1 is the time spend by "has overridden method" condition plus "Read class of first overridden method" command;
- t_{2i} is the time needed by "Self if instance of current child class" condition;
- t_{3i} is the time needed by "has overridden methods to evaluate" condition plus "Read class of next overridden method".

Other issue supported by the polymorphism mechanism is the "instanceof" and "checkcast" instructions. The first one is used to test if an object is an instance of a determined class, the later evaluates an object typecast. As in our approach the constant pool was removed from classfiles, the semantics of these instructions had to be changed.

The "instanceof" instruction had its reference to the constant pool replaced by the class identifier relative to the class that is originally referenced by the constant pool entry used by this instruction. So, the object class identifier is compared with this unique identifier and, if the object is an instance of the tested class, a "true" value is pushed into the stack. The "checkcast" instruction follows the same approach. Its reference to constant pool is replaced by a class identifier. So, this instruction compares the class identifier field of the object which is on the top of stack with the class identifier of "checkcast" instruction. If they are equal, the stack remains unchanged; otherwise an exception is generated.

The proposed modification in these two instructions allows some inheritance evaluation. It avoids spending time to evaluate application class hierarchy at runtime, in

order to discover the type of a class instance. Evaluation of class instances can be made using only two instructions (e.g. push object reference instruction and "instanceof" or "checkcast" instruction). However, this approach has a drawback: the type of a class instance comparison always will be "the class of objectX is equal to classY ?" opposite to original comparison "objectX belongs to classY or its superclasses ?". This limitation is imposed by the constant pool removal.

4.3. Threads Stack Initialization

In our scheme each real-time thread has its own stack (i.e. the global stack is divided in parts, one for each thread). This thread stack must be initialized with all method frames[1] needed to execute a real-time thread (thread frame). In the previous version of SASHIMI tool this initialization was done by the programmer. This means that at each compilation, different code addresses needed to be pushed in the thread stack. This could lead to initialization errors and a lot of extra-work to fix this address code changes. To solve this problem, an automatic thread stack initialization routine was proposed.

The code for the thread frame allocation in the global stack is inserted automatically by the synthesis tool during synthesis of the ERTS. The maximum number of tasks allowed in the embedded system generated by synthesis tool is direct related with the amount of available RAM memory. The method frames are constructed according the FemtoJava frame organization [1] and is popped from stack by the method return instructions (e.g. "return", "ireturn", "areturn", etc.). The thread stack is composed by three method frames. Firstly, the method frame of the scheduling algorithm method is pushed. Afterwards the *mainTask()* method frame is pushed on the stack. The last pushed frame is a frame that is used only for the first task context switch, which means that this frame is consumed on the first task activation.

5. Results on ERTS Generation

To illustrate the proposed ERTS synthesis method, this section presents results from two case studies. The first one is a wheelchair movement control, on each users can control the whellschair's velocity and direction using a joystick. This system has four periodic tasks with different periods: one to read the joystick sensor; other to read movement sensors (speed and angle); a control task; and a movement mode controller task. The second case study is a crane control system [7], which is a known benchmark for ERTS. It moves a load through its track to a desired position. The implemented system has also four tasks: one to read the angle sensor; other to read the position sensor; a control task; and a diagnosis task.

After having generated the RTSJ-based Java code for each application, this code is then compiled and linked with the RTSJ-based API using a standard Java compiler. The resulting class files are used as input to the synthesis tool and are analyzed in order to generate the VHDL files that represent the customized FemtoJava processor. In the final step the resulting VHDL files are synthesized with Altera Quartus II 4.0 Web Edition Full using the Cyclone EP1C20F400C6 FPGA device. The synthesized ERTS characteristics are shown in Table 1.

Table 1: Size of the generated systems

	Wheelchair System	**Crane System**
Compiled code		
Application size	17536 bytes	31258 bytes
Used API classes size / %	25848 bytes / 60%	29470 bytes / 48%
Total size (application+API)	43384 byte	60728 bytes
Synthesized code		
Application size (ROM)	2557 bytes	6105 bytes
Used API classes size (ROM) / %	2866 bytes / 53%	3084 bytes / 33%
Total ROM size (aplic.+API)	5423 bytes	9189 bytes
RAM size	636 bytes	1066 bytes
Application size (ROM+RAM)	6059 bytes	10255 bytes
Code size reduction / %	37961 bytes / 87,5%	51539 bytes / 84,9%
Total of diffetent used instructions	67 instructions	77 instructions
FemtoJava Size	6011 logic cells	6081 logic cells

The compiled code represents the size of all classfiles generated by the standard Java compiler. After synthesis, the total application code size was reduced from approximately 43,4 kb to 5,4 kb (87,5%) in the wheelchair system and from 60,8 kb to 9,2 kb (84,9%) in the crane system. This reduction is even higher if we consider the size of a RTSJ-compatible JVM that must be deployed together with the application code. For example, JamaicaVM needs a minimum of 128 kb [22]. Considering this information the reduction of total application size is, respectively, 96,9% and 95,2%.

Considering the additional code related to the RTSJ-based API, which main benefit is to increase the code readability, in the wheelchair experiment it represents 53% of total application code, and 33% in the crane experiment. This difference is due to the more complex control mechanisms in the crane experiment. This increased code shows the reduction of designer work that is provided by high-level languages. Similarly, the total number of FemtoJava instructions is higher in the crane experiment. As stated before, the size of the control unit of the Java processor is directly proportional to the total number of instructions used in the application. Therefore the total number of logic cells needed in the crane experiment is higher than in the wheelchair one.

As stated before, the approach used to synthesize the

[1] A method frame contains all information (local variables and return context) for current executing method

OO aspects allows determining, at design time, the overhead needed to access the instance fields and to invoke the overridden methods. To exemplify, let us assume that a method of an abstract class is overridden by two subclasses. Applying formula (1) to the algorithm components, t_1 always requires four instructions that need 20 cycles to execute and t_2 has two instructions that need 9 cycles, while the t_3 component requires 2 instructions that execute in 12 cycles. The WCET of this method invocation is 50 cycles. If the FemtoJava runs at 5 Mhz, the WCET is 10 µs. Object type evaluation is even simpler to analyze. As stated before our approach is unable to evaluate if the class of an object is equal to compared class. So, it is needed only three instructions ("aload", "instanceof" and "ifeq") to do this evaluation that needs only 16 cycles. On other words, the WCET is 3.2 µs considering the FemtoJava running at 5 Mhz.

6. Conclusion and Future Work

This paper presented the use of OO techniques in a Java processor used in the ERTS domain. All the synthesized OO constructions can be time-bounded, therefore it is possible to know in advance the runtime and memory overhead. The statically objects allocation avoids using the GC unpredictable behavior. Also the simplified object representation allows a predictable access to object fields, avoiding the indeterminist behavior to access object fields that are inherited from ancestor classes. The polymorphism mechanism has a predictable execution time due to follow the single path paradigm. Its implementation is done just as binary code instead of using the constant pool to solve code address. Therefore the sum of individual instruction time is the time needed to invoke overloaded methods. Nevertheless, designer must cope with a small increase of methods code size.

The obtained results present the impact of our synthesis approach to optimize the raise of abstraction level in the synthesized application size. The gains on the code optimization can be up to 97% in comparison to existing RTSJ-compatible JVM. Moreover, it is also demonstrated that a higher level of abstraction is reached with the OO code, although this enhancement incurs into extra code which is added in the final application code. In some applications, like the wheelchair, the extra code is higher than the application code, what proofs the importance of hiding implementation details. The proposed approach also optimizes RAM memory usage avoiding the use of the constant pool to resolve class hierarchy at synthesis time. However, this approach leads to a small increase of overridden methods code to allow the use of polymorphism.

As future work, the authors intend to optimize memory usage by supporting the additional memory areas defined in the RTSJ. Additionally we aim to develop an additional GC with bounded execution time (it must have a known worst case execution time). Also other RTSJ constructions should to be supported, as for example AsyncEventHandler and Timed.

7. References

[1] S. Ito, L. Carro, and R.P. Jacobi, "Making Java Work for Microcontroller Applications", IEEE Design & Test of Computers, Vol. 18, No. 5, 2001, pp. 100-110.

[2] M.A. Wehrmeister, L.B. Becker, and C.E. Pereira, "Optimizing Real-Time Embedded Systems Development Using a RTSJ-based API", Lecture Notes in Computer Science, Vol.3292, 2004, pp.292

[3] M.A. Wehrmeister, F.H. Ataide, F.C. Carvalho, and C.E. Pereira, "Assessing the Use of RT-Java in Automotive Time-Triggered Applications", IFIP International Federation for Information Processing, Vol. 184, 2005, pp.223

[4] E.T. Silva Jr., M.A. Wehrmeister, C.E. Pereira, L. Carro, F.R. Wagner, and L.B. Becker, "Design Exploration in Hw/Sw Co-design of Real-Time Object-oriented Embedded Systems: the Scheduler Object", Proc. of Workshop on Object-Oriented Real-Time Dependable Systems, IEEE Computer, 2005, pp. 378-388

[5] L.B. Becker, M.A. Wehrmeister, L. Carro, F. Wagner, and C.E. Pereira. "Assessing the Impact of Traditional Real-Time Scheduling Algorithms on Top of Embedded Applications", Proc. of 29th IFAC/IFIP Workshop on Real-Time Programming. Istanbul, Turkey, 2004.

[6] T. Lindholm, and F. Yellin, "The Java Virtual Machine Specification", Addison-Wesley, 1997.

[7] E. Moser, and W. Nebel, "Case Study: System Model of Crane and Embedded Control", Proc. of DATE, 1999.

[8] F. Siebert, and A. Walter, "Deterministic Execution of Java's Primitive Bytecode Operations", Proc. of Java Virtual Machine Research And Technology Symposium, 2001.

[9] A. Corsaro, D.C. Schmidt, "The Design and Performance of the jRate Real-Time Java Implementation", Proc. of International Symposium On Distributed Objects And Applications, 2002.

[10] M. Schoeberl, "Design Rationale of a Processor Architecture for Predictable Real-Time Execution of Java Programs", Proc. of International Conference On Real-Time And Embedded Computing Systems And Applications, 2004.

[11] Sun Microsystems, "Java 2 Micro Edition Platform Specification", http://java.sun.com/j2me/1.4.2/docs/api/

[12] D. Hardin, "aJile Systems: Low-Power Direct-Execution Java Microprocessors for Real-Time and Networked Embedded Applications". White paper available in http://www.ajile.com/downloads/aJile-white-paper.pdf

[13] A. Burns, and A. Wellings, "Real-Time Systems and Programming Languages (3rd Edi-tion). Ada 95, Real-Time Java, and Real-Time Posix", Addison-Wesley, 2001.

[14] A. Wellings, "Concurrent and Real-Time Programming in Java". Wiley, 2004.

[15] OVM Team, "OMJF Project", http://www.ovmj.org, 2005.

[16] Timesys, "JTime VM", http://www.timesys.com, 2002.

[17] aJile Systems, "Frequently Asked Questions", http://forum.ajile.com/faqhtml/cache/129.html

[18] P. Puschner, and A. Wellings, "A Profile for High Integrity Real-Time Java Programs", Proc. of ISORC, 2001.

[19] G. Bollella, J. Gosling, B. Brosgol, "The Real-Time Specification for Java", http://www.rtj.org/rtsj-V1.0.pdf

[20] P. Puschner, and A. Burns, "Writing Temporally Predictable Code", Proc of IEEE International Workshop on Object-Oriented Real-time Dependable Systems, 2002. pp 85-91.

[21] P. Puschner, "Is Worst-Case Execution-Time Analysis a Non-Problem? – Towards New Software and Hardware Architectures", Proc of 2nd Euromicro International Workshop on WCET Analysis, 2002.

[22] AICAS, "JamaicaVM – User documentation", http://www.aicas.com/jamaica/doc/html/index.html