REAL-TIME JAVA FOR EMBEDDED DEVICES: THE JAVAMEN PROJECT*

A. Borg, N. Audsley, A. Wellings
The University of York, UK

ABSTRACT: Hardware Java-specific processors have been shown to provide the performance benefits over their software counterparts that make Java a feasible environment for executing even the most computationally expensive systems. In most cases, the core of these processors is a simple stack machine on which stack operations and logic and arithmetic operations are carried out. More complex bytecodes are implemented either in microcode through a sequence of stack and memory operations or in Java and therefore through a set of bytecodes. This paper investigates the state-of-the-art in Java processors and identifies two areas of improvement for specialising these processors for real-time applications. This is achieved through a combination of the implementation of real-time Java components in hardware and by using application-specific characteristics expressed at the Java level to drive a co-design strategy. An implementation of these propositions will provide a flexible Ravenscar-compliant virtual machine that provides better performance while still guaranteeing real-time requirements.

1. INTRODUCTION

The Java programming model has become established in mainstream software development as a platform for general-purpose applications. The abstraction provided by the virtual machine realises the attractive Write-Once-Run-Anywhere (WORA) concept. Initially created for developing software for consumer devices, it achieved its success in the rapid expansion of the Internet for which the WORA concept found an ideal application. Today, Java is penetrating into more niche markets, from large enterprise applications to small embedded devices such as mobile phones. In particular, the Micro Edition of the Java 2 Platform (J2ME) provides an application development environment that specifically addresses the needs of embedded devices such as personal digital assistants and set-top boxes.[1]

Despite this new focus on embedded devices, J2ME and the Java superset do not address the application domain of real-time applications. Indeed, until recently, "Java" and "real-time" were considered an oxymoron as Java lacked the infrastructure to enable development of real-time systems. The specifications of both Language (1) and Virtual Machine (2) were never designed with timeliness as a key issue. The language therefore fails to allow more advanced temporal requirements of threads to be expressed and virtual machine implementations may behave unpredictably in this context. For example, whereas a basic thread priority can be specified, it is not required to be observed by a virtual machine and there is no guarantee that the highest priority thread will preempt lower priority threads. In order to address this shortcoming, two competing specifications have been introduced: the "Real-Time Core Extensions" from the J-Consortium (3) and the ``Real-Time Specification for Java" (RTSJ) (4) from Sun Microsystems. The former has not achieved the exposure of the RTSJ, though attempts are now being made to combine the two standards for a safety-critical Java (5).

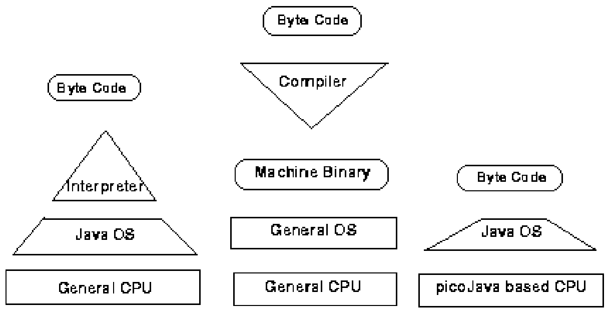

Figure 1: Three alternatives for executing Java code (take from (6))

The Java Virtual Machine specification describes an abstract stack machine that executes byte-code, the intermediate code of the Java language. The implementation of this abstraction could be handled either as a middleware application on top of stock hardware and operating systems or by a direct hardware implementation. The tradeoff between these two approaches is one of cost and speed; hardware implementations are fast but could be expensive. As with any programming language, Java code could be compiled down to native code unique to a particular platform, thereby deriving a performance benefit at the cost of portability and a significantly larger code size. This latter overhead occurs because the runtime semantics must be embedded in application code. These three approaches are depicted in Figure (1) as described in (6) for the picoJava processor. The majority of current deployments of Java virtual machine implementations are of the first type. Even devices that provide J2ME implementations generally execute a software JVM with no Java-specific hardware support.

The JavaMen research project is being carried out at the University of York to capture the requirements of a virtual machine implementation with hardware support

* This work is funded by the Next Wave Technologies Program (UK Government DTI) in conjunction with Sun Microsystems.

[1] The J2ME combines a set of configurations, profiles and optional packages that can be combined together to provide this environment. Broadly speaking, this gives a subset of the full Java standard edition with cut-down functionality in the virtual machine and a smaller set of core and application libraries.

that is compatible with a high-integrity subset of the RTSJ, the Ravenscar Java profile (7). Briefly, the two goals of the JavaMen project are to:

- Design and implement a minimal real-time Java virtual machine to run as an IP-core.
- Provide distribution functionality through which an application may perform a remote invocation to offload part of its computation.

This paper focuses on the first goal by identifying the research niches that are being addressed. The aim here is to motivate the proposed solutions currently under implementation. As will be shown in the next section, some areas of this field have already been extensively researched. While adopting whenever possible positive results from existing research in our own implementation, this work is unique in that it optimises for the real-time characteristics of the application while supporting a flexible co-design strategy for application-specific environments.

The rest of this paper is set out as follows: Section 2 gives a brief overview of the state-of-the-art in Java processor research. Section 3 identifies the research niches that are being addressed in the JavaMen project. A brief description of the RTSJ and the basic principles of hardware/software co-design is provided here in order to set the context of this work. Section 4 introduces the proposed solutions and, finally, Section 5 concludes.

2. HARDWARE SUPPORT FOR REAL-TIME JAVA

In 1996, Romer *et. al.* argued that there existed so much possible optimisation of standard Java runtimes that proposing hardware support was premature (8). This is contentious statement given that software runtimes are emulations of the Java abstract machine and that proposed optimisations are only targeted at better emulation. However, given the poor optimisation of software virtual machines at the time, this statement was probably true. A number of early hardware implementations of the Java virtual machine were developed soon after this time but none reached any significant levels of distribution. As optimisation of software implementations reaches its peak, hardware support is once again beginning to prove necessary in order to overcome the performance barrier of Java interpretation (and just-in-time runtime compilation) on general purpose platforms.

There are two ways of improving bytecode execution using hardware. The first is to use a coprocessor that works in conjunction with a general purpose CPU by either providing a fast bytecode translation to that CPU's native instruction set or by executing simple bytecode itself. Examples of the coprocessor option include Jazelle (9) and JSTAR(10). The second option is to provide a Java-specific processor which therefore is limited but specialised to Java applications. PicoJava (6), aJile (11) and JOP (12,13) are examples of Java processors that adopt this approach, each with slightly difference optimisations. In essence however, the architectures are similar; they mostly implement a stack machine architecture with simpler bytecodes being executed in a small number of cycles and more complex ones executed in microcode or software. PicoJava's optimisations include features such as instruction folding and bytecode prediction, aJile provides simple synchronisation and scheduling primitives in microcode whereas JOP opts for as minimal a model as required to provide predictable real-time guarantees. Other examples of existing Java Processors are Moon (14), Lightfoot (15), LavaCore (16) and FemtoJava (17), each providing different stack-based implementations. LavaCore and FemtoJava allow an analysis of the application to be carried out in order to fine-tune the hardware generated, thereby providing an application-specific hardware solution.

In order to provide the best possible hardware support, the JVM abstraction must be analysed in order to identify the optimisations over a naive stack-machine implementation. In general this investigation is carried out on a set of applications such as the SPECjvm98 (18) benchmark suite with optimisations then being driven by the ``average" application. The literature contains several examples of research into the characterisation of the Java runtime but with the goal generally being to drive the design of software JVMs. In (19,20), the authors use trace information to optimise components of the runtime such as the method cache and the decision strategy of the JIT compiler. Other work applicable to software JVMs can also be used in hardware implementations. Examples include the optimisation of JVM operations such as synchronisation (21) and using common application characteristics such as the average size of objects in making caching decisions.

El-Kharashi *et. al.* carry out a two-part investigation into the architectural requirements of a Java microprocessors (22, 23). The goal of the work is to direct the instruction set design of a hardware JVM by discovering the common cases of bytecode execution. Their approach is to to insert probes into the main interpretation loop of a software JVM in order to calculate information such as the frequency of execution and time cost of each bytecode when executing a given test-bench application. Other information collected includes the access patterns and frequency of conversion for different data types, the number of local variables in the method frame, the average number of operands and the average instruction length. This information can be used to drive design decisions in a hardware JVM. For example, since this work suggests that the number of local variables is usually between 16 and 32, this motivates the mapping of this number of local variables to hardware registers. Crucially, this work reports that multithreading is at the heart of the JVM but make no recommendations on how to take advantage of this fact.

There are similarities between research work targeted at optimising Java processors and that aimed at software JVMs; for example, optimisation of a common trait in software that leads to caching of particular data can also be transferred to the hardware domain by using dedicated registers. In any case, the use of tools to gather information about a general application behaviour could be use to drive the final design of an application-specific hardware solution. For example, LavaCore (16) provides a tool that can be used to identify the subset of used bytecodes, thereby eliminating the hardware structures or microcode of those bytecodes that are not used. This application-specific approach could be taken to another level to reconfigure the hardware. For example, if cache sizes and bus widths could be configured, then these could be optimised for a given application.

The JVM abstraction defines the execution of three broad types of instructions:

- Simple memory loads and stores (for example through loading constant-pool data) and arithmetic and logic manipulations of the stack,
- Control flow operations such as jumps and invocations,
- Higher-level operations such as object instantiation and monitor control.

A simple stack machine can implement many of the first and second types of instructions through standard hardware components such as multiplexors, adders, shifters, etc. Hardware support for the third type of operations is rare due to the complexity of these instructions and is often argued to be unnecessary due to the infrequent use of these bytecodes. The notable exception is jHISC (24) which provides hardware support for object-oriented bytecodes such as **putfield** and object invocation. Achieving this requires hardware components that make the core "object-aware", that is by storing object information in hardware registers. In most Java processor implementations, the complexity of the third type of instructions is addressed through an implementation on top of the simple stack machine abstraction. There are two ways that this is achieved: the first is to specify in the microcode control store the sequence of stack instructions required to implement this bytecode; the second is to trap on these bytecodes and call a Java method which implements this functionality. The latter option requires a set of extended bytecodes to be defined that allow access to hardware elements from the application level. The JOP processor (12,13) is an example of a Java processor that makes use of both approaches. This allows for a range of these bytecodes to be moved between the microcode instruction store residing in on-chip memory and a standard method for which the implementing bytecodes are stored in main memory.

The full functionality of the JVM is not expressed solely by the semantics of bytecodes. For example, class-loading semantics are not expressed as part of the bytecode set. Crucially for real-time systems, multithreading semantics are undefined at the bytecode level: there are no bytecodes for starting a thread, yielding execution, etc. In software VMs, these operations are captured instead by the Java runtime which intrusively modifies the state of the virtual machine's stack and registers. In most Java processors, multithreading support is absent at the hardware level. Instead, the Java bytecode set is extended to allow Java-level access to hardware structures such as the stack. Alternatively, an extended bytecode set is defined to provide the semantics of these higher level operations and which therefore can be handled by a set of microcode instructions without any vertical switching between hardware and software levels. Note that both these approaches are similar to those commonly used in addressing the third type of bytecodes above. Again, direct hardware support for these types of operations is rare.

The provision of better hardware support for real-time Java applications is the first area that JavaMen addressing. Two components of the RTSJ are to be considered: real-time scheduling and dispatching and the unique memory model of the RTSJ. The majority of research into hardware schedulers has been related to the field of packet-switching in networks (25) though some hardware schedulers for real-time systems have been developed. For example in (26), a hardware scheduler that can be configured for three types of scheduling policies is described. The scheduler here is defined in a separate block to the microprocessor and interacts with the main processor through a shared bus. Albeit an attractive solution for porting purposes, this arrangement is subject to unpredictable bus contention and is not as efficient as a scheduler with direct access to the CPU registers. In a Java-specific environment, the Komodo Java microcontroller (27) provides on-chip registers to save the context of up to four threads, thereby providing very fast context switching. The aJile processor (11) also claims to provide multithreading but this is achieved through extended bytecodes and associated microcode instructions rather than through any direct hardware support. Hardware support for the RTSJ's novel scoped memory model is thus far an unexplored area of research. In (28) the authors show how the semantics of this memory model can be integrated with that of a hardware processor. However, it is only assumed that the hardware traps to a software level on the relevant bytecodes and there is no support either in microcode or directly in hardware to implement the model's semantics.

The second area of research JavaMen is addressing is the potential of a co-design solution for Ravenscar applications. The goal here is to provide a development environment were the functionality of the application can easily traverse the hardware/software boundary. Two components will be considered here: JVM-specific functionality (in particular the scheduler and support for memory management) as well as application-specific functions for which equivalent hardware components may exist. A more detailed description of these two research areas is provided in Section 4. Next, brief introductions to the RTSJ and the fundamentals of application-specific codesign is provided as a background for this description.

3. THE RTSJ AND CO-DESIGN TECHNOLOGY

The first step at achieving the first goal of the JavaMen project is to identify a suitable development environment for real-time Java applications. The developed core and runtime would then need to implement the functionality that can be expressed in this environment. Expressing real-time requirements will be possible using a subset of the RTSJ APIs: the Ravenscar Java Profile (7). As the RTSJ provides a wide range of features suitable for both hard and soft real-time systems, Ravenscar defines this subset in order to eliminate those features unsuitable for high-integrity systems. A second API specific to JavaMen will be used to express co-design functionality. A brief description of the scheduling and memory-management features of the RTSJ and its Ravenscar subset are described next. This is followed by a brief introduction into application-specific co-design principles.

3.1 The RTSJ

The RTSJ describes ``seven enhanced areas," that bring real-time functionality to the Java platform. The three areas relevant to this work and which feature in Ravenscar are:

- the definition of real-time thread scheduling and dispatching mechanisms,
- appropriate synchronisation and resource sharing mechanisms and
- a new memory model that allows the application to avoid the penalties of garbage collection.

As the heart of the RTSJ is the Schedulable object, the unit of execution that the real-time scheduler understands. This added abstraction over standard threads allows other classes such as event handlers to form part of the scheduling and dispatching decision. Schedulable objects implement the interface Schedulable which, as an extension of the standard Runnable interface, defines the abstract method *run()*. The information supplied by the methods defined in Schedulable is used by the scheduler to create a suitable context to execute *run()*. In Ravenscar these are strictly priority parameters as the scheduler must be a fixed priority scheduler. The RTSJ defines two types of Schedulable objects: AsyncEventHandler and RealtimeThread. NoHeapRealtimeThread extends RealtimeThread and implements the same methods but has special memory characteristics as described below. For the purposes of this work, only NoHeapRealtimeThread instances are being considered as they provide the timing guarantees needed but eliminating use of the garbage collector.

The RTSJ provides three types of memory areas: scoped memory, immortal memory and heap memory which are encapsulated in the ScopedMemory, ImmortalMemory and HeapMemory classes respectively. HeapMemory provides application-level access to the standard Java heap in which objects are allocated by java.lang.Thread threads. Every application contains one instance of ImmortalMemory in which residing objects live for the duration of the application. Every ScopedMemory instance is associated with a physical memory area called the *backing store* that can be allocated and freed dynamically. ScopedMemory is an abstract class with two subclasses: VTMemory and LTMemory that describe memory areas with variable and linear allocation times respectively. NoHeapRealtimeThreads can only access objects in scoped and immortal memory.

Every RTSJ thread executes in some memory area at any time and this memory area is called the *current execution context* of that thread. For most purposes, it can be assumed that the initial execution context of a thread is immortal memory, a memory region in which allocated object live for the duration of the application. When a thread invokes the *new* operator to create a new object, that object is created in the current execution context of that thread. For example, if the current execution context of a thread is the immortal memory region, then any new objects created are allocated from the immortal area. By changing their current execution context, threads can create objects in either this immortal region or scoped memory regions. Changing execution context is done using the methods of the ScopedMemory class namely the methods *enter()* and *executeInArea()*. Allocation and deallocation of scoped memory regions is defined by the semantics of scoped memory as defined in the RTSJ. The reader is referred to (4) for a full description of the RTSJ memory model and the semantics of the methods in the memory classes. For the purposes of this paper this brief introduction suffices for describing the motivation of the approach taken in this project.

3.2 Application-Specific Hardware/Software Co-Design

Designing embedded systems often requires satisfying or optimising for a variety of constraints. Amongst others, these constraints include cost, power consumption and performance. The boundary of the hardware/software partition is often decided at the early stages of development and it is often hard to move this boundary at later stages. The motivation behind hardware/software co-design is to allow a unified development environment of these two parts of the system that delays the fixing of these partitions until a better understanding of the system emerges at later stages of development. Moving the boundary as further properties emerge is also simplified by the supporting tools. The co-design field has several niche research areas that range from finding suitable unified abstractions for specifying systems that can then be transparently mapped to either hardware or software to algorithms that automatically search the solution space of partitions for a particular partition that best satisfies a set of user-specified requirements. The term "application-specific" as used in this context is broadly one of granularity in the co-design search space. If the exact characteristics of the software that is to be executed are unknown, then the co-design solution will be one that will be optimised for a broad application domain. In this case, the co-design solution could therefore prove sub-optimal for a particular application. At the finest granularity, if only one application is to be executed on the hardware then a full characterisation of that application may be identified and the co-design solution could therefore be optimised.

The JavaMen processor implementation will provide co-design support at two levels: Firstly, a configurable fixed-priority hardware scheduler will be provided with the scheduling logic and task structures specified as required in Ravenscar. In particular, the initialisation phase of Ravenscar requires the specification of the number of executing tasks. This information can be used to generate the exact number of thread context structures either in software or in hardware at the synthesis stage. Alternatively, a tradeoff can be provided with a subset of these structures available directly in hardware registers, another subset in on-chip memory and the remainder in off-chip memory. Secondly, an RMI-type abstraction will be developed to allow an application to use available hardware directly. Available hardware components will be wrapped in object-like semantics for which equivalent software implementations could be available. The availability of this hardware will therefore be unknown at development time and the decision on whether to provide this hardware or use the software implementation can be delayed until deployment. At runtime, the application will have access to this functionality transparently, at times using the hardware component if it is available and at others making use of the software implementation.

4. TOWARDS AN APPLICATION-SPECIFIC DEVELOPMENT ENVIRONMENT FOR THE RTSJ

Rather than implement a processor core from scratch, we have opted to extend an existing open-source processor, the Java Optimised Processor (JOP) (12,13), for this research. JOP's hardware core is a simple 32-bit stack machine with each bytecode being mapped to one or more microcode instructions. The core is a minimalistic 4-stage pipelined machine and, apart from a simple bytecode cache, contains no complex functionality such as bytecode folding or branch prediction. This is ideal for the purposes of this project as it allows for a clear evaluation of the proposed solutions without the interference of these features.

4.1 Real-Time Scheduling and Memory Management in JOP

JOP provides no support for multithreading within the hardware core or at microcode level. The current release provides a simple scheduler and allows multithreading through a Java-level implementation that changes the stack of the virtual machine though a set of extended bytecodes. For example, native method *getSP()* is converted to the special bytecode **jopsys_getsp** that returns the value of the current stack pointer. A similar bytecode, **jopsys_setsp** changes the value of the stack pointer. Using these bytecodes, the context of a thread can be loaded and unloaded.

Two alternative implementations of the scheduler are proposed. The first is that similar to aJile's (11) approach where threading specific bytecodes are introduced. The entire scheduler can therefore be implemented in microcode and vertical switching between hardware and software is avoided. The close mapping between bytecode and microcode instructions is the primary motivation for JOP currently avoiding this approach. However, since the bytecode for scheduler operations is held in off-chip memory, scheduling operations are subject to the latency of the bytecode cache. Placing scheduling logic in the microcode store consumes scarce on-chip ROM memory but avoids these latencies.

The second implementation is a pure hardware scheduler that is closely integrated in the core. Some of the parameters of threads as defined in the RTSJ can be mapped automatically to hardware registers. This means that the static nature of Ravenscar (as necessary to provide real-time guarantees) is also taken advantage of in a single-application environment. To this end, a simple tool is proposed that takes a Ravenscar-compliant program specification for which the thread components are generated. The maximum number of threads depends on the available memory in the given architecture and state information for these threads is stored partially in off-chip memory and partially in on-chip memory. In particular, information pertaining to scheduling decisions (such as state and release parameters) resides on-chip whereas context information (such as the stack) resides off-chip. If it can be guaranteed that no changes are made to priority parameters in the mission phase, then the priority ordering can be implied in the architecture due to the fixed number of real-time threads.

There are two goals of this part of the research: the first is a better understanding of the tradeoffs across the three implementations. The second is to gauge how useful static knowledge of the system's threads as available in a Ravenscar-compliant application can be. For example, knowing a system has exactly *n* periodic threads will allow optimisations in the scheduler that should improve performance. The Komodo (25) JVM showed this to be the case for between 1 and 3 real-time threads. In-keeping with our application-specific approach, these implementations allow an arbitrary but fixed amount of threads to be specified and the scheduler configured accordingly. Comparing the performance and space requirements gives an indication of the scalability of this approach.

A similar process is being used to investigate different solutions of the RTSJ memory model. The Ravenscar-Java profile allows each Schedulable object to have only one scoped memory region with no nested *enter()* invocations. This makes the implementations of this model trivial to implement both in hardware and software. However, current research is aiming to remove this restriction and it is therefore she current JOP release provides no implementation of the scoped memory model. Again, three implementations are proposed: a purely Java solution, a microcode solution and a hardware solution. The latter in particular is a challenge as there is no previous work whatsoever in this area. Despite popular belief that the RTSJ reference checks add a significant overhead to the runtime cost of an application, it was shown in (29) that this overhead is minimal. If this is true then it would be expected that little improvement would be evident across implementations. The algorithms that maintain the scope-tree structure are not trivial and require a restructuring of the object representation in JOP.

4.2 An Object Abstraction of Hardware Components for a Co-Design Environment

Some Java processors such as aJile allow access to external hardware through user-defined extended bytecodes. This allows Java applications to access this hardware but the interface is still ad-hoc and defined at a low level. Hardware components are similar to software objects in that they encapsulate state and function. The central idea in this part of the project is to allow hardware components to be wrapped as objects and therefore be accessible from Java in a transparent manner.

A framework is being developed that allows the specification of a Java interface that defines the functionality of a hardware component. The main challenge here is to define the semantics of this encapsulation. We are using a similar technique to that used in Remote Method Invocation (RMI) where an interface is specified that describes a set of methods including get/set-style methods that change state. An equivalent software version of this functionality is also provided. A special object implements one or more of these interfaces and maintains references to the Java implementation of these methods as well as the handoff logic to the hardware implementation. This latter code holds the low-level code similar to that used in aJile's approach. Note that the semantics of software and hardware invocation are not identical. This is because invocation of hardware inside the runtime can be done in parallel. To take advantage of this parallelism, control should be returned to the processor in order to execute other tasks. This means that some invocations will be asynchronous and therefore the timing characteristics of the application change. Appropriate schedulabilty analysis to capture this behaviour is already being investigated at York in the form of the Limited Parallel Model (30) and will be used in this work.

5 CONCLUSION

This paper has introduced the JavaMen project and the goals that we hope to achieved. The design and implementation of the three Reverencer schedulers and scoped memory implementations in JOP is currently underway and initial results should be available at year-end. An API framework for expressing co-design decisions is also being developed and the similarity of RMI and parallel execution is being exploited in order to provide a unified semantics of these two types of execution. We are hoping to have a complete co-design development framework for developing Ravenscar applications on a JavaMen processor by the middle of the next year.

References

1. Gosling J., Joy B., Steele G., and Bracha G., *Java Language Specification, Second Edition: The Java Series*, 2000, Addison-Wesley Longman Publishing Co. Inc., Boston, MA, USA

2. Lindholm T., and Yellin F., *Java Virtual Machine Specification, Second Edition*, 1999, Addison-Wesley Longman Publishing Co. Inc., Boston, MA, USA

3. The J-Consortium, *International J-Consortium Specification - High Integrity Profile Revision 0.2*, February, 2001, Available at: www.opengroup.org

4. Bollella G., and Gosling J., *The Real-Time Specification for Java*, 2000 Computer, 33.6, 47-54, Note: Latest version available at: www.rtj.org

5. Child J., *Real-Time Java Takes Aim at Embedded Control*, 2004, Note: Available at: www.rtcmagazine.com/home/article.php?id = 100111

6. *picoJava Microprocessor Core Overview*, Web Publication available at: www.sun.com/microelectronics/picoJava/overview.html

7. Kwon J., Wellings A., and King S., *Ravenscar-Java: A High Integrity Profile for Real-Time Java*, 2002, Proceedings of the 2002 joint ACM-ISCOPE Conference on Java Grande, 131-140, Seattle, Washington, USA, ACM Press

8. Romer T. H., Lee D., Voelker G. M., Wolman A., Wong W. A., Baer and Brian J-L., Bershad N., and Levy H. M., *The Structure and Performance of Interpreters*, 1996, ASPLOS-VII: Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, 150-159, Cambridge, Massachusetts, United States, ACM Press

9. Information and white papers on Jazelle available at: www.arm.com/products/solutions/Jazelle.html

10. Information on JStar available at: www.nazomi.com

11. Hardin D., *Real-Time Objects on the Bare Metal: An Efficient Hardware Realization of the Java Virtual Machine*, 2001, ISORC '01: Proceedings of the Fourth International Symposium on Object-Oriented Real-Time Distributed Computing, Washington, DC, USA, IEEE Computer Society

12. Schoeberl M. *Design and Implementation of an Efficient Stack Machine*, 2005
Proceedings of the 12th IEEE Reconfigurable Architecture Workshop (RAW2005), Denver, Colorado, USA, IEEE,

13. Schoeberl M. *JOP: A Java Optimized Processor*, 2003, On the Move to Meaningfull Internet Systems 2003: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2003) LNCS 2889, 346-359, Catania, Italy, Springer

14. Information on Moon available at: www.vulcanasic.com

15. Product specification available at: www.xilinx.com/products/logicore/alliance/digital_comm_tech/dct_lightfoot_32bit_processor.pdf

16. Product specification available at: www.xilinx.com/products/logicore/alliance/dsi/dsi_java_proc.pdf

17. Ito S. A., Carro L., and JacobiR. P., *Making Java Work for Microcontroller Applications*, 2001, IEEE Design and Test, 18.5, 100-110, IEEE Computer Society Press, Los Alamitos, CA, USA

18. Information available at: www.spec.org/osg/jvm98/

19. Radhakrishnan R., Bhargava .,and John L. K., *Improving Java Performance using Hardware Translation*, 2001, ICS '01: Proceedings of the 15th international conference on Supercomputing, Sorrento, Italy 427-439, ACM Press, New York, NY, USA

20. Radhakrishnan R., N. Vijaykrishnan, John L. K., Sivasubramaniam A., Rubio J., and Sabarinathan J., *Java Runtime Systems: Characterization and Architectural Implications*, 2001, IEEE Transactions on Computers, 50.2, 131-146, IEEE Computer Society, Washington, DC, USA

21. Yang B-S., Moon S-M., and Ebcioglu K., *Lightweight Monitors for the Java Virtual Machine*, 2005, Software Practice and Experience, 35.3, 281-299, John Wiley & Sons, Inc., New York, NY, USA

22. El-Kharashi M. W., Elguibaly F., and Li K. F., *A Quantitative Study for Java Microprocessor Architectural Requirements. Part I: Instruction Set Design*, 2000, Microprocessors and Microsystems, 24.5, 225-236

23. El-Kharashi M. W., Elguibaly F., and Li K. F., *A Quantitative Study for Java Microprocessor Architectural Requirements. Part II: High-level Language Support*, 2000, Microprocessors and Microsystems, 24.5, 237-250

24. Hau G. K. W., Fong A., and Lun M. P., *Support of Java API for the jHISC System*, 2003, SIGARCH Computer Architecture News, 31.4, 12-17, ACM Press, New York, NY, USA

25. Moon S. W., Shin K. G., and Rexford J., *Scalable Hardware Priority Queue Architectures for High-Speed Packet Switches*, 2000, IEEE Transactions on Computers, 49.11, 1215--1227, IEEE Computer Society, Washington, DC, USA

26. Kuacharoen P., Shalan M., and Mooney V.J., *A Configurable Hardware Scheduler for Real-Time Systems*, 2003, Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms, Las Vegas, Nevada, US, 95-101, CSREA Press

27. Kreuzinger J., Schulz A., Pfeffer M., Ungerer T., Brinkschulte U., and Krakowski C., *Real-Time Scheduling on Multithreaded Processors*, 2000, Proceedings of the Seventh International Conference on Real-Time Systems and Applications RTCSA'00), Washington, DC, USA, IEEE Computer Society

28. Higuera T., Issarny V., Banare M., and Parain F., *Memory Management for Real-Time Java: An Efficient Solution using Hardware Support*, 2004, Real-Time Systems Journal, 26.1, 63-87, Kluwer Academic Publishers, Norwell, MA, USA

29. Niessner A., Benowitz E., *RTSJ Memory Areas and Their Affects*, 2003, On the Move to Meaningful Internet Systems: OTM 2003 Workshops, LNCS 2889, 508-519, publisher = "Springer"

30. Bletsas K., *Extending The Limited Parallel Model*, 2005, Technical Report at the University of York, Dept. of Computer Science, YCS391 (2005)