SHAP — Secure Hardware Agent Platform

Martin Zabel, Thomas B. Preußer, Peter Reichel, Rainer G. Spallek

Institute of Computer Engineering, Technische Universität Dresden, Germany
Email: {*zabel,preusser,rgs*}*@ite.inf.tu-dresden.de*, *Peter.Reichel@mailbox.tu-dresden.de*

Abstract— **This paper presents a novel implementation of an embedded Java microarchitecture for secure, real-time, and multi-threaded applications. Together with the support of modern features of object-oriented languages, such as exception handling, automatic garbage collection and interface types, a general-purpose platform is established which also fits for the agent concept. Especially, considering real-time issues, new techniques have been implemented in our Java microarchitecture, such as an integrated stack and thread management for fast context switching, concurrent garbage collection for real-time threads and autonomous control flows through preemptive round-robin scheduling.**

I. Introduction

Object-oriented programming has led to a fast and easy development of complex applications with a short time-to-market. In this domain, Java is very popular as it addresses also portability and security features through the definition of the Java Virtual Machine (JVM). As more and more target systems implement the JVM, the same application can be executed anywhere. Another important feature is the compact Java bytecode leading to small memory requirements and reduced download time. So, it is predestined for the use in resource constrained devices. This enables applications running anywhere at any time as well as the support of agent systems.

As the execution of Java bytecode by interpretation is known to be rather slow, Just-In-Time (JIT) compilation has been used to translate Java bytecode to the host processor's native instruction set. As this requires much memory, the alternative of executing Java bytecode natively has been considered for embedded Java implementations. The challenge is to combine this execution with real-time constraints to support this application domain.

Our implementation of an embedded Java microarchitecture, called SHAP, fills this gap by implementing new techniques to support multi-threaded general-purpose applications in a secure environment under real-time constraints. Especially, all JVM concepts are considered here.

All trademarks are the property of their respective owners.

II. Related Work

Quite some research has already been carried out on the efficient execution of Java bytecode directly on hardware. This main goal is joined with the mapping of the basic JVM concepts on hardware structures to enable the fast operation in embedded systems without the help of an operating system. Several Java processors have been developed [1]–[21], which will be briefly surveyed in this section.

Except for the FemtoJava, all Java processors are initially designed to completely support all Java bytecodes. Thus, we will analyze the support of several JVM concepts and their implementation, particularly with regard to the support of multiple, real-time threads. An overview of the properties and the features of selected Java processors is given in Tab. I. Besides them, there are several other processors, however, with only limited information available, which prohibits their inclusion in the table: the reconfigurable simple Java core (R-Java) [13], the asynchronous Java processor [14], the VLIW Java processor [15], the JA108 Java coprocessor [16], the Jazelle extension of the ARM processor [17], [18], Lightfoot [19] and AMIDAR [20], [21].

Typically, the design of a Java processor follows that of a RISC processor. Pipelined bytecode execution is common, which leads to the well-known conflict between high clock frequencies (throughput) and the execution latency of branch instructions. This problem may be solved with branch prediction, which, however, requires additional chip area and energy and, thus, conflicts with the target application domain in embedded systems.

As Java bytecode applies a stack-based execution model, the stack implementation of a design is a performance-critical issue. So the stack is typically mapped onto a fast internal memory or even a register file. An exception to this rule is, again, FemtoJava, which

TABLE I

PROPERTIES AND FEATURES OF SELECTED JAVA PROCESSORS

	FemtoJava [1]	PicoJava-II [4]	Berekovic [5]	Komodo [6], [22]	JOP [7]	JEM2 (aJ-100) [11]	SHAP (this paper)
Pipeline stages	5	6	?	5	4	?	4
Instruction folding	-	+	+	-	-	?	-
Microcode	-	+	+	+	+	+	+
Software Traps	-	+	?	+	+	+	+
Stack Realization	e	r	r	?	i	r	i
Objects	-	+	+	+	+	+	+
Multiple Threads	-	+	-	(+)[a]	(+)[a]	+	+
Thread HW-Support	-	-	-	+	-	+	+
Scheduler	-	?	-	H	S	M	M
Synchronization	-	+	-	?	(+)[b]	M	M
Real-time Threads	-	?	-	+	+	+	+
Garbage Collector	-	S	S	S	S	S	H
GC for Real-time Threads	-	?	-	+	-	-	+
Dynamic Class Loading	-	?	-	-	-	-	+

H	Hardware	M	Microcode	S	Software
i	Internal Mem.	r	Register	e	External Mem.
+	supported	-	not supported	?	unknown

[a] statically allocated threads

[b] single global monitor

locates the stack in external memory while mirroring the two topmost stack entries into registers.

The mapping of the stack on a register file requires the technique of instruction folding [23]–[26], which assimilates instructions for stack data movement, like the loading of local variables, with the actual computational operation. This yields a typical RISC instruction with direct operand addressing and may accelerate the execution of Java programs as it eliminates the unproductive cycles for data movement. The cost is additional chip area required for the implementation of the read / write ports of the register file. To bound the required chip area, only a limited number of registers is available, and, thus, an automatic stack spill and fill must be implemented (called: stack cache). Instruction folding is also available for stacks mapped onto internal memory but its effect is known to be much smaller, cf. [2].

Besides the direct implementation of the Java bytecodes by state machines, it is also common to rely on the help of microcode (firmware) as well as software traps, which themselves are backed by Java programs. This simplifies the implementation of complex bytecodes such as `invokevirtual` tremendously and has a positive effect on the maintainability. Because this approach requires additional memory, there is no general statement if it requires more or less chip area than an implementation with complex state machines only.

In addition to the Java bytecodes defined by the Java Virtual Machine Specification [27], the Java processors implement special bytecodes for the access to the runtime system and internal data structures, a task that is performed by native methods in software JVMs.

Java is an object-oriented platform storing all data except for primitive data types inside objects located on the Java heap. This is a memory area usually managed by a garbage collector (GC), which frees the memory occupied by unreferenced, i.e. unneeded, objects. The GC can be implemented in either software or hardware and is typically responsible for the memory used by regular (non-real-time) threads.

Multi-threading is another essential JVM concept that allows the parallel execution of multiple tasks. Usually regular threads are distinguished from real-time threads, where real-time means that guarantees about the execution and answer time of a task can be given. Scheduling can either be done in hardware, via microcode or with in software. For a fast thread switch needed for short response times, hardware assistance for context saving and loading is required. Last but not least, controlled access to objects used by multiple threads requires a synchronization mechanism. Such is commonly available on a per object basis. An exception is JOP, which provides only a single global monitor. Furthermore, this Java processor as well as Komodo support only a statically allocated number of threads.

The support for real-time threads by the currently available Java processors is limited. Particularly, there is no automatic garbage collection in the memory areas used by real-time threads. Instead, the approach defined by the Real-Time Specification for JAVA (RTSJ) [28] is used. It defines designated heap areas, typically one per real-time thread, with possibly different properties. Real-time threads then allocate their objects exclusively from their heap area without any garbage collector inference. These heap areas can only be destroyed as a whole as triggered manually by the programmer. Another approach is the Ravenscar-Java profile [29] – a subset of the RTSJ. Here, allocation of objects is only allowed in an initialization phase. On the other hand, some research results are available, which show that automatic garbage collection is possible under specific real-time constraints [22], [30]–[32].

Neither is dynamic class loading at run time standard. Rather, the whole application is pre-linked into a memory image, which enables fast execution but inhibits dynamic class loading afterwards. This is caused by the layout of the memory image, which would require new information to be inserted rather than appended. The linking step by JOP [7] is one example. To provide dynamic class loading, the information must be stored in objects on a per class basis instead. The JEM2 processor [12] takes another approach by managing two parallel Java Virtual Machines (JVMs). Here, each JVM uses a pre-linked memory image. It is not stated clearly whether these images can be replaced at run time.

The aspects of automatic GC for real-time threads and dynamic class loading at run time are covered in particular by our project in addition to providing a general-purpose embedded Java processor for secure, real-time and multi-threaded applications.

III. The SHAP Concept

The agent concept as developed in the field of artificial intelligence, provides a new view on designing complex systems: interaction of autonomous computing nodes in distributed systems to establish intelligent behavior. In spite of the fact that there is no unique definition of an agent [33]–[39], all these definitions share some common properties:

- An agent works autonomous by having its own thread of control.
- An agent is located in an environment and interacts with it by sensing and acting on it.

Further frequently assigned properties are:

- An agent is reactive, if it continuously perceives the environment and responds to changes in a timely fashion.
- A deliberative agent, instead, has an explicit model of the world, and engage in planning and negotiation with other agents.
- Instead of (simple) sensing, the agent may also directly communicate with other agents through specific protocols. This property is also called social ability.
- A pro-active agent takes the initiative by acting on the environment, e.g., sending messages, with preceding requests.
- A mobile agent may travel through a network to obtain information on-the-spot.

After defining agency, software frameworks arise introducing APIs for implementing agents. As these frameworks run on typical operating systems, usage in embedded systems is not applicable. Furthermore, the complexity of such frameworks prohibits the prove of secure parallel execution of agents.

Our aim is to design a hardware platform which directly provides concepts to run agents with the above properties in embedded systems in a secure context. This leads to the main features of our *Secure Hardware Agent Platform*, SHAP for short:

- multi-threading (with guaranteed time slots) and fast thread switching establishing autonomous control flows,
- thread synchronization through monitors,
- exception handling (for secure execution),
- multiple inheritance through interfaces to provide complex class frameworks for deliberative and pro-active agents,
- automatic memory management with garbage collection,
- object serialization, for data transfer on a per object basis in multi-agent systems to implement mobility,
- integrated devices for sensing and acting of agents as well as communication between agents.

Real-time support, to establish reactive agents, is achieved through:

- integrated stack and thread management in hardware,
- integrated automatic and concurrent, i.e. processor-independent, memory management with garbage collection,
- fairly distributed processing time through preemptive round-robin scheduling.

Especially, no underlying operating system is required for the JVM. The maintained low memory footprint makes SHAP suitable for the application in the embedded system domain. Typical use cases for SHAP are:

- Combination of several SHAP cores to form a multi-core system with shared memory and concurrent memory management with automatic garbage collection;
- Interconnecting several SHAP microarchitectures to form a multi-agent-system with data-transfer using Java objects;
- Embedded system platform providing dynamic loading of user applications beside their normal functionality. For example, a prepackaged UMTS / GSM / GPS transmitter-receiver module with user specified data pre- / post-processing without integration of additional hardware;

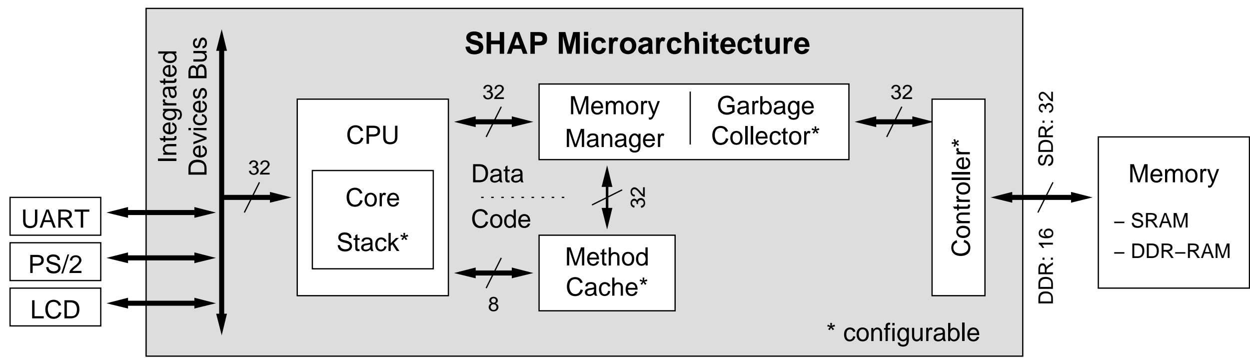

Fig. 1. Schematic of SHAP Microarchitecture

- Monitor processor checking the consistency of system states;
- Adaptive filtering monitor processor for system debugging and error search.

IV. Components

The SHAP concept spans not only the SHAP microarchitecture but also all the software components required to execute Java applications in hardware under real-time constraints. All components of the microarchitecture are required to perform their tasks in constant time to achieve this main goal.

A. Hardware

The design of SHAP follows a strict modular approach. The interfaces of the individual components, as depicted in Fig. 1, are on a high logical level as to achieve a high degree of encapsulation of a component's responsibility and its implementation. The autonomous handling of their responsibilities by the components enables a high degree of parallelism and frees the central computing core from continuous burdens as the management of the stack and the heap.

a) The CPU: consists of the core (fetch unit, decoder, arithmetic / logic unit) as well as the on-chip stack module, and directly executes Java bytecode with the following differences:

- All branch instructions use absolute target instruction offsets from the start of the method. This does not impose a tighter bound on the allowable size of method code blocks than specified in the current JVM specification [27, § 4.10] but it eliminates the need for the `GOTO_W` instruction.
- Extra bytecodes for system interfacing have been introduced taking responsibilities of typically native code of software JVMs.
- Extra bytecodes for interface type coercion.

All Java bytecodes are executed through microcode in a 4-stage pipeline: bytecode fetch, instruction fetch, decode and execute. As the microcode subroutine can be as short as a single instruction, frequently used Java bytecodes are executed within one cycle. All Java bytecodes are either executed in constant time or their execution time is known in advance based on statically available information, such as the size of a method or the number of exception table entries. This property enables the calculation of the execution time of a method and consequently for a complete application as required for the real-time constraint. There is one exception with `invokevirtual` and `invokeinterface`. The called method may be unknown, so that their execution time cannot be included. Special care has to be taken by the programmer in this case.

Before the execution of a Java bytecode, several constraints have to be checked as defined by the JVM Specification [27]. Static constraints, such as the appropriate types of stack operands, are verified at link time by a bytecode verifier. Thus, the core must only perform truly dynamic checks such as testing for null references. The handling of exceptions whether so raised by the system or by user code is fully supported.

Thread scheduling is implemented in microcode and assisted by the multi-context capability of the stack, which is described below. For the scheduler, various techniques can be considered, of which we chose a preemptive round-robin scheduling to distribute the execution time fairly. Blocking accesses to devices on the integrated devices bus are exploited by the scheduler, which suspends the blocking thread's execution for high core utilization in favor of the next in line. Finally, also monitor synchronization is implemented in microcode where monitors are associated with object instances including the instances of class objects for the synchronization of static code blocks.

TABLE II

OPERATIONS PROVIDED BY STACK COMPONENT

Operation	Description
PUSH	Pushes a word onto the top of the stack.
POP	Pops a word from the top of the stack.
RD_VAR	Loads a local variable (application date) from the current stack frame onto the top of the stack.
ST_VAR	Stores the top of the stack into a local variable of the current stack frame.
RD_FRAME	Loads a frame variable (JVM data) from the current stack frame onto the top of the stack.
RD_BW	Use top of stack to index this many positions down into the stack and replace it by the value found there.
ENTER	Establish new method frame with the number of arguments and local variables just pushed.
LEAVE	Destroy current method frame and activate preceding one.
SWITCH	Activate stack of the (possibly new) thread specified by the top of stack.
KILL	Destroy stack of the thread specified by the top of stack.

The rather complex issue of the handling of interface types and the efficient implementation of the interface bytecode is covered in a separate paper. It shall only be pointed out here that appropriate support is provided.

The core also provides an interface to the GC as to enable its scanning of the stack for alive references. All references on the stack are marked with an additional 33rd bit set so that the stack module actually handles 33-bit instead of 32-bit data.

The stack subcomponent provides high-level stack operations executed in constant time to the core. More so, the operations frequently required by the JVM are implemented such that they are shadowed totally by the regular execution of the core pipeline and are guaranteed to never slow the execution by requiring stalls. These operations are, in particular: PUSH, POP, LD_VAR, ST_VAR and LD_FRAME. A short description of their and the other operation's function can be found in Tab. II. For fast access by the core, the two topmost stack values are stored inside the registers TOS (top-of-stack) and NOS (next-on-stack).

Besides these rather standard stack operations, also the method frame management is performed autonomously through the operations ENTER and LEAVE. As an exception, the operations require two cycles in total, but still run in constant time.

Java method invocation is further supported by the operation RD_BW, which allows to copy a value located further down in the stack onto its top. Such an operation is necessary to retrieve the this argument of a method invocation to resolve interface and virtual method calls by its runtime class type.

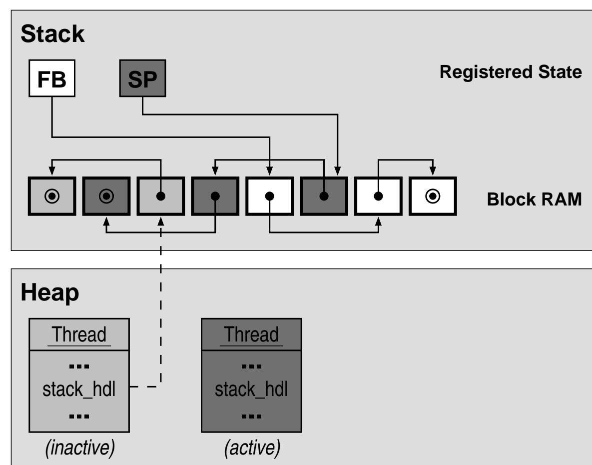

Fig. 2. Exemplary Stack Block Organization

The stack component also takes care of the stack substitution (context saving and loading) due to thread switching. Although thread switching by SWITCH occupies the stack component for 5 cycles, it is still performed in constant time. The destruction of a thread's stack that finishes normally takes 3 cycles, a forceful destruction of a large stack space may take longer.

To provide one separate stack for each thread, the storage space used by the stack module is divided into equally-sized blocks currently holding up to 64 words. These blocks are organized in multiple disjunct singly-linked lists. Each list, except for the list of unused blocks, represents the stack of one thread. Active thread stacks are linked backwards so that the topmost block is the root of the list. The state maintained by the stack module for the basic management of these lists is limited to the index of the first block of the free list FB and to the stack pointer SP identifying the block and the internal offset of the top of stack of the currently active thread. The heads of the stacks of inactive threads hold management information internal to the stack module and are identified by a handle passed to and returned by the SWITCH operation. These handles are managed by the runtime system typically as part of the state of the Thread objects. An exemplary situation is depicted in Fig. 2. The creation of a new thread stack is also achieved through the SWITCH operation by passing it the special handle -1.

1) Implementation: The organization of the stack blocks in linked lists enables a fast dynamic growing and shrinking of the active stack by relinking a block

between the free block list and the list of the active stack. As the occupied stacks are backwards, this only requires fast manipulations at the heads of the lists.

The constant execution time of block relinking can, however, only be guaranteed when the number of blocks to be relinked is bound. In our case, we restrict ourselves to a single block whose relinking can be performed within one cycle. This limits the extend of a method frame, which must be restricted to a block size with all its local variables (including arguments) and remaining operand stack contents, at least, on method exit. The currently available space of 64 words seems, however, to be sufficient for just about all practically relevant applications, cf. [40], [41].

a) The Memory Manager: The memory manager manages the Java heap by allocating objects, performing read and write operations on them and, finally, by freeing memory used by unreferenced objects. This module encapsulates the complete object management. The CPU only acts on references, which identify objects, and offsets into these objects.

All operations of the memory manager are executed in parallel to the CPU and in constant time but may take several cycles. This may lead to extra wait cycles in the pipelined execution of the core but their worst-case number is known in advance. Thus, execution under real-time constraints is still available.

To realize garbage collection under real-time constraints, the GC itself must fulfill real-time constraints, i.e. the execution of Java bytecode must still perform in constant time. One simple approach is to use a stop-the-world GC, which is implemented as another real-time thread, which performs a complete heap scan within its assigned time slot. The worst-case execution time of such a GC can be calculated in dependence of the heap size. Although this yields a fixed bound, it would typically be prohibitively large requiring a very long scheduling period. The resulting guaranteed response time of the system would be unacceptable for many applications.

A solution to this problem, is an incremental GC. We took this approach but, in contrast to others, we implemented the GC directly in hardware. This enables parallel execution of the GC to all Java threads. Additionally, the integration into the memory manager minimizes the path to the memory for fastest possible access.

b) The Method Cache: which caches the currently executed Java method, which is regularly stored on the heap (inside the class objects). This is required only for the von-Neumann architecture of the prototyping board.

c) An Integrated Memory Controller: which provides a direct interface to external memory, like SRAM oder DDR-SDRAM, and, thus, does not incur additional latencies due to external protocols.

d) The Integrated Devices Bus: connects the SHAP core with its secondary components. Many of these implement the communication with the outside world. Others realize a secondary interface to internal components as the statistics port to the memory manager. The bus is mastered and arbitrated exclusively by the CPU. All connected devices are slaves.

The devices bus supports full 32-bit wide data and addresses. While the upper part of the address selects the targeted device, the lower bits may be evaluated by the device to distinguish several ports or commands. Every device further supplies the core with two status signals as applicable. It signals `ready` when it is able to receive data from the core, and it asserts `available` when data is available for reading from the selected port.

The CPU does not handle all bus devices directly. It rather interfaces to a single set of address, data and status lines. The activation of a device to drive these lines is performed through the device selection based on the supplied address.

The currently implemented range of devices includes a serial interface (RS232), a PS/2 keyboard controller, an LCD controller as well as the statistics interface to the memory manager. Due to the simple bus interface, the addition of further devices is straightforward.

B. Software

The currently available software components are:

- An implementation of the “Connected Limited Device Configuration” (CLDC) API [42] — a subset of the standard Java API especially designed for embedded devices.
- The ShapLinker, which pre-processes and links the input class files into a SHAP file ready for execution on the SHAP microarchitecture. The SHAP file is not a flat memory image, it rather contains a designated section with the information for the construction of a separate runtime class object for each class. This lays the foundation for dynamic class loading. The linker may also be ported to the SHAP microarchitecture to provide an integrated system.
- An assembler for the microcode used internally by the core implementation.

V. Prototype

The SHAP microarchitecture is configurable to fit for the specific application needs: size of the internal stack, multi-threading is optional, size of the method cache, garbage collection is optional, memory controller is selectable, different (integrated) bus devices.

Currently, prototyping of the SHAP microarchitecture is done on a SPARTAN-3 Starter Kit Board. The actual configuration is:

- 8 KByte stack, up to 32 threads,
- 2 KByte method cache,
- memory manager with GC,
- memory controller for external 1 MByte SRAM,
- bus devices: UART, LCD, PS/2, memory statistics unit
- clock frequency of 50 Mhz,

and has a resource usage on the Spartan3 XC3S1000 of:

Slices	2433	31%
Block RAMs	10	41%
18×18 multiplier:	3	12%
User I/O pins	95	54%

VI. Conclusion

This paper presented a novel implementation of an embedded Java microarchitecture for secure, real-time, and multi-threaded applications, thus fitting for operation in multi-agent systems. Due to its additional support for modern features of object-oriented languages, such as exception handling, automatic garbage collection and interfaces, it also establishes a general-purpose platform built without an underlying operating system.

New techniques have been implemented for specific real-time issues, such as an integrated stack and thread management for fast context switching, concurrent GC for real-time threads and autonomous control flows through preemptive round-robin scheduling. Open issues are the further improvement of the memory management as well as the integration of dynamic class loading.

References

[1] S. A. Ito, L. Carro, and R. P. Jacobi, "Making Java work for microcontroller applications," *IEEE Des. Test*, vol. 18, no. 5, pp. 100–110, 2001.

[2] J. M. O'Connor and M. Tremblay, "picoJava-I: The Java virtual machine in hardware," *IEEE Micro*, vol. 17, no. 2, pp. 45–53, 1997.

[3] H. McGhan and M. O'Connor, "PicoJava: A direct execution engine for Java bytecode," *Computer*, vol. 31, no. 10, pp. 22–30, 1998.

[4] "picoJava(tm)-II data sheet," SUN Microsystems, 1999.

[5] M. Berekovic, H. Kloos, and P. Pirsch, "Hardware realization of a Java virtual machine for high performance multimedia applications," *J. VLSI Signal Process. Syst.*, vol. 22, no. 1, pp. 31–43, 1999.

[6] S. Uhrig, "Der Komodo mikrocontroller," Institute of Computer Science, University of Augsburg, Tech. Rep. 2003-03, July 2003.

[7] M. Schoeberl, "JOP: A Java Optimized Processor for embedded real-time systems," Ph.D. dissertation, Vienna University of Technology, 2005.

[8] F. Gruian, P. Andersson, K. Kuchcinski, and M. Schoeberl, "Automatic generation of application-specific systems based on a micro-programmed Java core," in *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*. New York, NY, USA: ACM Press, 2005, pp. 879–884.

[9] M. Schoeberl and R. Pedersen, "WCET analysis for a Java processor," in *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*. New York, NY, USA: ACM Press, 2006, pp. 202–211.

[10] M. Schoeberl, "A time predictable Java processor," in *DATE '06: Proceedings of the conference on Design, automation and test in Europe*. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2006, pp. 800–805.

[11] D. S. Hardin, "Real-time objects on the bare metal: An efficient hardware realization of the Java virtual machine," in *ISORC '01: Proceedings of the Fourth International Symposium on Object-Oriented Real-Time Distributed Computing*. Washington, DC, USA: IEEE Computer Society, 2001, p. 53.

[12] "aJ-100 real-time low power Java processor," aJile Systems, Inc., Dec. 2001.

[13] S. Kimura, H. Kida, K. Takagi, T. Abematsu, and K. Watanabe, "An application specific Java processor with reconfigurabilities," in *ASP-DAC '00: Proceedings of the 2000 conference on Asia South Pacific design automation*. New York, NY, USA: ACM Press, 2000, pp. 25–26.

[14] C.-P. Yu, C.-S. Choy, H. Min, C.-F. Chan, and K.-P. Pun, "A low power asynchronous Java processor for contactless smart card," in *ASP-DAC '04: Proceedings of the 2004 conference on Asia South Pacific design automation*. Piscataway, NJ, USA: IEEE Press, 2004, pp. 553–554.

[15] A. C. S. Beck and L. Carro, "A VLIW low power Java processor for embedded applications," in *SBCCI '04: Proceedings of the 17th symposium on Integrated circuits and system design*. New York, NY, USA: ACM Press, 2004, pp. 157–162.

[16] "JA108 — multimedia application processor," http://www.nazomi.com/jChip.asp?tab=products&sub=jChip.

[17] C. Porthouse, "High performance Java on embedded devices," ARM Limited, Oct. 2005, http://www.arm.com/documentation/White_Papers/index.html.

[18] ——, "Jazelle for execution environments," ARM Limited, May 2005, http://www.arm.com/documentation/White_Papers/index.html.

[19] "Lightfoot 32-bit Java processor core," dct – digital communication technologies, Aug. 2001, http://www.dctl.com/lightfoot.html.

[20] S. Gatzka and C. Hochberger, "The AMIDAR class of reconfigurable processors," *J. Supercomput.*, vol. 32, no. 2, pp. 163–181, 2005.

[21] ——, "Hardware based online profiling in AMIDAR processors," in *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 3*. Washington, DC, USA: IEEE Computer Society, 2005, p. 144.2.

[22] M. Pfeffer, T. Ungerer, S. Fuhrmann, J. Kreuzinger, and U. Brinkschulte, "Real-time garbage collection for a multi-threaded Java microcontroller," *Real-Time Syst.*, vol. 26, no. 1, pp. 89–106, 2004.

[23] I. Sideris, G. Economakos, and K. Pekmestzi, "A cache based stack folding technique for high performance Java processors," in *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*. New York, NY, USA: ACM Press, 2006, pp. 48–57.

[24] R. Radhakrishnan, R. Bhargava, and L. K. John, "Improving Java performance using hardware translation," in *ICS '01: Proceedings of the 15th international conference on Supercomputing*. New York, NY, USA: ACM Press, 2001, pp. 427–439.

[25] R. Radhakrishnan, D. Talla, and L. K. John, "Allowing for ILP in an embedded Java processor," in *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*. New York, NY, USA: ACM Press, 2000, pp. 294–305.

[26] M. W. El-Kharashi, F. Elguibaly, and K. F. Li, "Adapting Tomasulo's algorithm for bytecode folding based Java processors," *SIGARCH Comput. Archit. News*, vol. 29, no. 5, pp. 1–8, 2001.

[27] T. Lindholm and F. Yellin, *The Java(TM) Virtual Machine Specification*, 2nd ed. Addison-Wesley Professional, Apr. 1999.

[28] "JSR-100, real-time specification for java," SUN Microsystems, http://java.sun.com/javase/technologies/realtime.jsp.

[29] J. Kwon, A. Wellings, and S. King, "Ravenscar-Java: a high integrity profile for real-time Java," in *JGI '02: Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*. New York, NY, USA: ACM Press, 2002, pp. 131–140.

[30] D. Detlefs, C. Flood, S. Heller, and T. Printezis, "Garbage-first garbage collection," in *ISMM '04: Proceedings of the 4th international symposium on Memory management*. New York, NY, USA: ACM Press, 2004, pp. 37–48.

[31] D. F. Bacon, P. Cheng, and V. T. Rajan, "A real-time garbage collector with low overhead and consistent utilization," in *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM Press, 2003, pp. 285–298.

[32] ——, "The metronome: A simpler approach to garbage collection in real-time systems." in *OTM Workshops*, 2003, pp. 466–478.

[33] H. S. Nwana and D. T. Ndumu, "An introduction to agent technology," in *Software Agents and Soft Computing Towards Enhancing Machine Intelligence*, ser. LNAI, vol. 1198. Berlin Heidelberg: Springer-Verlag, 1997, pp. 3–26.

[34] S. N. K. Watt, "Artificial societies and psychological agents," in *Software Agents and Soft Computing Towards Enhancing Machine Intelligence*, ser. LNAI, vol. 1198. Berlin Heidelberg: Springer-Verlag, 1997, pp. 27–41.

[35] S. Franklin and A. Graesser, "Is it an agent, or just a program?: A taxonomy for autonomous agents," in *ECAI '96: Proceedings of the Workshop on Intelligent Agents III, Agent Theories, Architectures, and Languages*. London, UK: Springer-Verlag, 1997, pp. 21–35.

[36] M. Wooldridge, "Agents as a rorschach test: A response to franklin and graesser," in *ECAI '96: Proceedings of the Workshop on Intelligent Agents III, Agent Theories, Architectures, and Languages*, ser. LNAI, vol. 1193. London, UK: Springer-Verlag, 1997, pp. 47–48.

[37] T. Menzies, A. Pearce, C. Heinze, and S. Goss, ""what is an agent and why should i care?"," in *Formal Approaches to Agent-Based Systems*, ser. LNAI, vol. 2699. Berlin Heidelberg: Springer-Verlag, 2002, pp. 1–14.

[38] M. Wooldridge, "Intelligent agents: The key concepts," in *Proceedings of the 9th ECCAI-ACAI/EASSS 2001, AEMAS 2001, HoloMAS 2001 on Multi-Agent-Systems and Applications II-Selected Revised Papers*, ser. LNAI, vol. 2322. London, UK: Springer-Verlag, 2002, pp. 3–43.

[39] N. R. Jennings, K. Sycara, and M. Wooldridge, "A roadmap of agent research and development," *Autonomous Agents and Multi-Agent Systems*, vol. 1, no. 1, pp. 7–38, 1998.

[40] M. W. El-Kharashi, F. ElGuibaly, and K. F. Li, "A quantitative study for Java microprocessor architectural requirements. Part I: Instruction set design," *Microprocessors and Microsystems*, vol. 24, no. 5, pp. 225–236, Sept. 2000.

[41] ——, "A quantitative study for Java microprocessor architectural requirements. Part II: high-level language support," *Microprocessors and Microsystems*, vol. 24, no. 5, pp. 237–250, Sept. 2000.

[42] *Connected Limited Device Configuration Specification, Version 1.1*, Sun Microsystems, Mar. 2003.