Available online at www.sciencedirect.com
ScienceDirect
Microprocessors and Microsystems 30 (2006) 469–479
MICROPROCESSORS AND MICROSYSTEMS
www.elsevier.com/locate/micpro

A Java processor with hardware-support object-oriented instructions

Tan Yiyu, Lo Wan Yiu, Yau Chi Hang, Richard Li, Anthony S. Fong *

Department of Electronic Engineering, City University of Hong Kong, Tat Chee Avenue, Kowloon Tong, Hong Kong

Available online 19 January 2006

Abstract

Java is widely applied from the small embedded devices to enterprise systems nowadays due to its object-oriented features and corresponding advantages of security, robustness, and platform independence. Java programs are compiled into Java Bytecodes, which are executed in the Java virtual machine. Among the current hardware or software solutions to the Java virtual machine, the object-oriented related Bytecodes are implemented by software traps or microcode, and their performance does not match well with the essential requirements of memory-constraint embedded devices, such as real-time operations and low power consumptions. In this paper, a novel Java processor named jHISC is proposed, which mainly targets Java applications in the small embedded devices. In jHISC, 94% of Bytecodes and 83% of the object-oriented related Bytecodes are implemented by hardware directly. Compared with PicoJava II and JOP, jHISC speeds up the overall performance about 30% and 183%, respectively.

Keywords: Java processor; Bytecode; Object-oriented programming; Operand descriptor

1. Introduction

Java was introduced in the mid-1990s to overcome the major weakness of C and C++ [1]. Due to its robustness, security, and portability, it is now widely applied in network applications and small embedded devices, such as PDAs, Pocket PCs, and mobile phones. A study of the ARC Group estimated that the number of J2ME (*Java 2 Platform Micro Edition*) compatible handsets was 421 millions in 2003 and would be increased to 442 millions in 2004 and 1 billion in 2006 [2]. Today J2ME has become the de facto environment for downloadable services and mobile entertainments running on mobile phones and PDAs.

Java applications are compiled to an intermediate representation called Bytecodes instead of particular processor machine codes to ensure their portability. Java Bytecodes are executed in the Java virtual machine and their execution methods are evolving from the simplest interpretation, Just-In-Time (JIT) compilation to the hardware realization [3–6]. The interpretation finds its performance to be significantly affected by the emerging time-consuming loops during software emulation despite its relatively easy implementation and small memory requirement. The JIT compilation introduces additional compilation overheads and needs much more memory in spite of its advantages over the interpretation in eliminating redundant translations and optimizing the generated native instructions. The hardware realization, namely Java processor, is a hardware solution to speed up the execution of Java programs through the implementation of Java virtual machine on silicon. It is also capable of tailoring hardware support for some Java special features, such as security, multithreading, garbage collection, and so on, to deliver much better performance than a general-purpose processor for Java applications. All these led researchers to develop high-performance Java processors for the memory-constraint embedded devices in recent years.

This paper presents a novel Java processor named jHISC, where the object-oriented related Bytecodes are supported in hardware directly. It mainly targets the small embedded devices, such as smart mobile phones, PDAs, Palm PC, etc. In Section 2, the previous work on Java

* Corresponding author. Tel.: +852 2788 7761; fax: +852 2788 7791.
E-mail address: Anthony.Fong@cityu.edu.hk (A.S. Fong).

0141-9331/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.micpro.2005.12.007

processors is summarized. Then some design issues on jHISC architecture, including the operand descriptor format, object representation, instruction set, method manipulation, and system architecture, are described in Section 3. In Section 4, the system performance estimation results and the corresponding analysis are presented. Finally, conclusions are made in Section 5.

2. Related work

Some Java processors have been proposed in recent years [7–23], which execute Java Bytecodes directly or translate them into the native instructions before execution. Because the Java virtual machine is software-based stack architecture, the easiest way to implement a processor which executes Java Bytecodes directly is to substitute the Java virtual machine with a hardware-based stack machine. Among the processors of this type, some are pure Java processors, such as PicoJava I and II from Sun Microsystems, aJ-100 from aJile Systems [7–10]; the others are Java accelerators, which are attached to a host general-purpose processor to execute Java Bytecodes. Java accelerators are either integrated into the core of the host general-purpose processor, such as AU-J2000 from Aurora VLSI [11] and ARM Jazelle [12], or function independently outside the host, such as MOCA-J™ from NanoAmp Solutions [13]. Several techniques based on reconfigurable devices were applied into Java accelerators to enhance their performance. Lattanzi et al. [14] and Yajun et al. [15] proposed schemes to improve the execution of Java applications by dynamically migrating the most heavily used methods on a reconfigurable device. Parnis and Lee [16] built a multi-threaded Java virtual machine on FPGA and enhanced its performance by exploiting the parallelism of FPGA. Kent et al. investigated a software/hardware codesign method to complement the host processor with a FPGA-based Java coprocessor [17,24,25].

Another method to execute Java Bytecodes is to add a hardware unit between the instruction fetch and decoding modules of a general-purpose processor to convert most of the simple Bytecodes into the native instructions at run-time. JA108 is a well-known commercial solution using this method [18], where the multiple stack-based bytecodes can be converted into a register-based native instruction. Radhakrishnan et al. [19] and Schoeberl [22] accelerated the execution of Java applications by the hardware interpretation; Glossner and Vassiliadis [20,21] developed the Delft-Java by translating most of Bytecodes into the Delft-Java instructions.

From the architecture viewpoint, all these Java processors have some limitations for the object-oriented operations, which are shown as follows.

(1) In the Java processors with stack architecture, most of the simple Bytecodes are implemented by hardware directly. However, all operands, such as temporary data, intermediate parameters, and method arguments, are pushed onto or popped frequently from the stack during execution, which makes it difficult to implement the object-oriented instructions by hardware directly due to their complexity and large amounts of data needed to be accessed during execution. Moreover, the data dependency between the successive instructions forbids any techniques of instruction level parallelism.
(2) In the Java processors which execute Java Bytecodes by hardware translation, because the general-purpose processor is not object-oriented architecture, no special hardware resources are provided to handle the object-oriented operations, such as switch of method context, management of local variable frame, and so on. Furthermore, an object-oriented operation is a complicated procedure, for example, an operation to get the value of an instance field needs to obtain the field information, retrieve the object reference, verify the access permission and data type, and fetch the field from the object pointed by the object reference. If there are no special hardware resources to handle these operations, it is complex and difficult to implement the object-oriented instructions by hardware directly.

Because of the above limitations, the current existing Java processors perform the object-oriented operations by software traps or microcode instead. Generally, simple load/store type operations constitute about 50% of all operations and complex object-oriented operations are about 15% of all operations in the Java benchmarks [26,28,29]. Executing a load/store type instruction by hardware takes one to three clock cycles, but executing an object-oriented instruction by software trap may consume more than 100 clock cycles. Hence, the processing of object-oriented operations becomes the performance bottleneck of Java programs [9,27]. In some solutions, the quick variations of some object-oriented related Bytecodes are provided to speed up execution after the related objects are resolved. However, this approach also increases the chip area and power consumption significantly because the quick variations are implemented by microcode, which needed many ROMs or other memory. Moreover, many application programs written by other programming languages are now available, which makes it desirable to have a general-purpose processor with some architecture extensions to support object-oriented programming in hardware directly. To address these problems, we develop jHISC, a Java processor with hardware-support object-oriented instructions.

3. jHISC architecture

jHISC is a 32-bit object-oriented processor based on the High Level Instruction Set Computer (HISC) architecture, which provides a hardware-readable data type called operand descriptor to describe objects [30–32] and supports object-oriented programming in hardware. By letting

hardware know what an object is, the processor also provides various supports to manage system, such as object manipulation, memory management, and so on. Compared with the HISC, the major architectural differences are shown as follows.

1. According to the Java virtual machine specification [33], the operand descriptor is simplified from 128-bit to 32-bit in order to reduce the system complexity.
2. The two operand descriptor tables in the HISC are incorporated into one.

Floating-point operations are not supported in the current version of jHISC.

3.1. Operand descriptor format

Operand descriptors, residing in the operand descriptor table, are used to describe object fields and references. According to the Java virtual machine specification [33], the information about object fields and objects has attribute, access flag, field type, and so on, which are stored in the constant pool. Based on this, we designed the structure of operand descriptor and its uniform format, which includes *Attribute*, *Type Field*, *Static Flag*, *Access Modifier*, *Read-only Flag*, and *Resolved Flag*, is shown in Fig. 1. The details of each item are explained as follows.

- *Address* is a byte offset, which is used to locate the data in the related data spaces.
- *Attribute* defines the attribute of a field or an object.
- *Access Modifier* specifies the access permission. Four access modifiers: public, private, protect, and package, are defined in the current system.
- *Type Field* declares the primitive data types and reference defined in the Java virtual machine, such as byte, int, word, char, reference, and so on. The field values are stored in the data space directly while an address pointer is stored to locate the described object for an object reference.
- *Static Flag* indicates the described object field is a static variable or an instance variable. If the field is an instance variable, its value will be stored in the Instance Data Space (IDS). If it is a static variable, its value will be stored in the Class Data Space (CDS). But when a static field is inherited from a superclass, a direct address pointing to it, not its value, will be stored in the CDS.
- *Read-only Flag* denotes whether the described field can be written or not. When it equals "1", the field can only be read, otherwise, it can be written.
- *Resolved Flag* indicates whether the object reference is resolved or not. If not, the operating system routines will be trapped for the dynamic reference resolution.

There are two kinds of operand descriptors: class operand descriptor and class property descriptor. The class operand descriptor, which consists of *Address, Attribute, Type Field*, and *Resolved Flag*, asserts the resources accessed by a class. The class property descriptor claims the properties owned by a class, and it contains all the items in the uniform format except *Resolved Flag*.

3.2. Object representation model

Object representation is critical in the object-oriented programming system due to its significant impact on the speed of accessing object. A good representation can maximize the efficiency of object operations and minimize the storage overhead, which makes the object model meet the following rules when it is adopted.

(1) The object header is as small as possible and contains sufficient information about the object to reduce the memory accessing overhead.
(2) Given an object reference, system is able to locate the object data quickly.

Based on these rules and the Java virtual machine specification, we defined the object header, which is three-word long and contains seven fields: *ObjectType*, *ArrayType*, *Lock*, *GC Info*, *DSSize*, *Class*, and *ArraySize*. The object header format is shown in Fig. 2 and each field is described as follows.

- *ObjType* defines the object type and is used to distinguish different objects, such as instance, class, method, and array.
- *DSSize* specifies the size of data space, such as CDS, IDS, and Method Code Space (MCS).
- *GCInfo* is used for the garbage collection, which is carried out by the operating system in the current version.
- *Class* links an instance with its affiliated class through a reference pointer. Hence, the affiliated class can be located easily through *Class*.

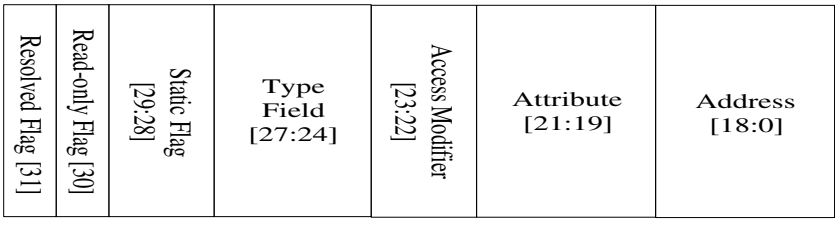

Fig. 1. Operand descriptor format.

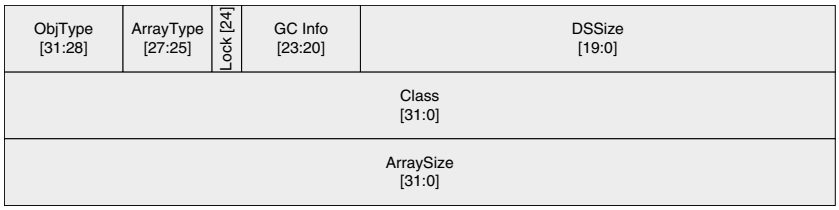

Fig. 2. Object header format.

- *ArraySize* and *ArrayType* exist only when the object is an array. As *ArraySize* denotes the number of elements in an array, *ArrayType* specifies the type of element.
- *Lock* is used to coordinate multi-threaded access to an object. Only one thread at a time can own an object.

In jHISC, three kinds of object contexts, namely instance context, class context, and method context, are mapped to the hardware architecture. Except the object header, an instance context contains Instance Header (IH) and Instance Data Space; a class context also consists of Class Header (CH), Operand Descriptor Table (ODT), and Class Data Space; and a method context includes Method Header (MH), Method Code Space, and Local Variable Frame (LVF). In addition, when applied to represent an array, the instance context also contains array data area, which locates under the instance header. The different object context structures and their relations are shown in Fig. 3.

Each object has a unique object context and a reference always points to the base address of object header after the object is resolved. In an object context, all components are stored continuously and each has a constant address offset to the object header, thus allowing the access of some components in parallel to reduce the access overhead. When an object is accessed, the related operand descriptor is read from the operand descriptor table to verify whether the object is resolved or not, then the specific object header is accessed through the direct address pointer stored in the CDS of current class. Along with the object accessing, the bound control checks, such as access permission, boundary, and data type, are also carried out by hardware. Moreover, both class variables and instance variables are stored in the related data spaces, therefore they are accessed by their references directly and not accessed through an intermediate object handle as Sun's JDK 1.0 and 1.1 [28].

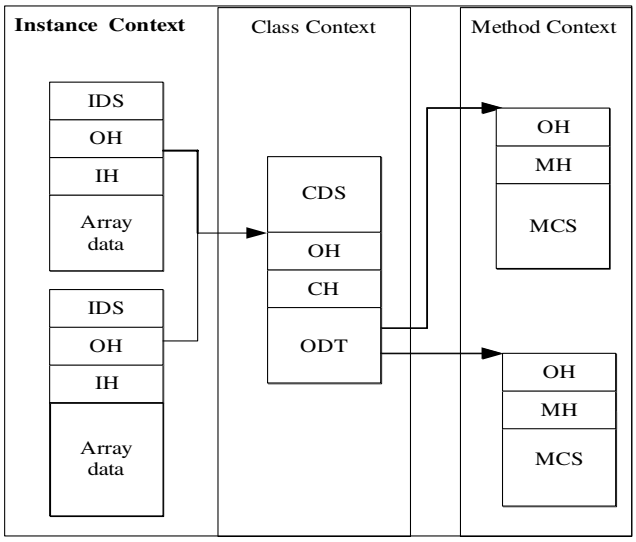

Fig. 3. Different object structures and their relations.

However, in the stack-based Java processors, such as PicoJava II, the information about object and its fields is dispersed and loaded into the system stack serially. Due to their stack architecture, retrieving and verifying these information is a time-consuming procedure. For example, when the Bytecode *putfield* is executed by software trap in PicoJava II, it takes about 113 clock cycles to prepare stack and verify the related information, such as whether the object is resolved or not, access right, type, and so on [9]. But in jHISC, the corresponding verifications are performed by hardware along with the object accessing. The details are shown in Table 1, where the clock cycles are obtained by assuming all data are hit in the data cache during execution.

3.3. Method manipulation

Method invocation/revocation affects the system performance in the object-oriented system because objects communicate each other through methods and each method invocation may consume several tens of clock cycles by software trap. Thus, an object-oriented processor should provide a fast and secure method manipulation in hardware directly. In the Java virtual machine, two basic kinds of methods, i.e., instance method and static method, are provided and invoked by two different Bytecodes. The Bytecode *invokevirtual* is used to invoke an instance method and the Bytecode *invokestatic* invokes a static method. By contrast, in jHISC, the instruction *ivkclass* is provided to invoke a static method. For an instance method, if it is within the same class context as the invoker, the internal method invocation instruction *ivkinternal* is used to invoke it; otherwise, the instance method invocation instruction *ivkinstance* is used instead.

During each invocation, the processor requires locating the method code, checking access control, pushing the current object context data into the system stack, and passing

Table 1
Executing the Bytecode *putfield* in PicoJava II and jHISC

Step	PicoJava II		jHISC	
	Operations	Clock cycles	Operations	Clock cycles
1	Initialize stack to prepare for the operation and read the reference	19	Read the operand descriptor to obtain the field's reference	1
2	Verify whether the field is resolved or not	8	Get the address of the class which owns the field	1
3	Extract the pointer to the field_info_table entry and go to the access verification	11	Access the object and class headers	1
4	Verify access permission, check the field's size, type, and determine the quick instruction type	75	Read data and the class property descriptor about the instance field	1
5	Rebuild the trap frame and restore the stack frame	13	Determine the field's address	1
6	Executing the related quick format instruction	4	Write data into the destination address	1

control to the invoked method. The more object contexts are switched, the more data are needed to push into the system stack. Generally, an internal method invocation only requires switching of the involved method contexts. A class method invocation requires one more, i.e., involved class context. And an instance method invocation requires switching of all the three kinds of contexts, namely involved method context, class context, and instance context. These are the reasons why we divide the Bytecode *invokevirtual* into the instructions *ivkinternal* and *ivkinstance* to speed up its execution.

For example, as illustrated in Fig. 4(a), class *Internal_Method_Example* includes two methods, *Caller( )* and *DoSomething( )*. Inside the method *Caller( )*, an internal method invocation occurs by asserting *DoSometing( )*. The corresponding object context switches and the object structures are given in Fig. 4(b).

In the current method space, instruction *ivkinternal ODT #1* triggers an internal method invocation and then directly accesses the #1 operand descriptor inside the ODT of current class to retrieve the invoked method's reference, which provides a direct address in the current CDS to locate the OH of method *DoSomething( )*. With the information inside the OH and MH of method *DoSomething( )*, the processor saves and updates the current method context, then accesses the MCS of method *DoSomething( )* to fetch instructions to execute until meeting a method revocation instruction if no exception occurs.

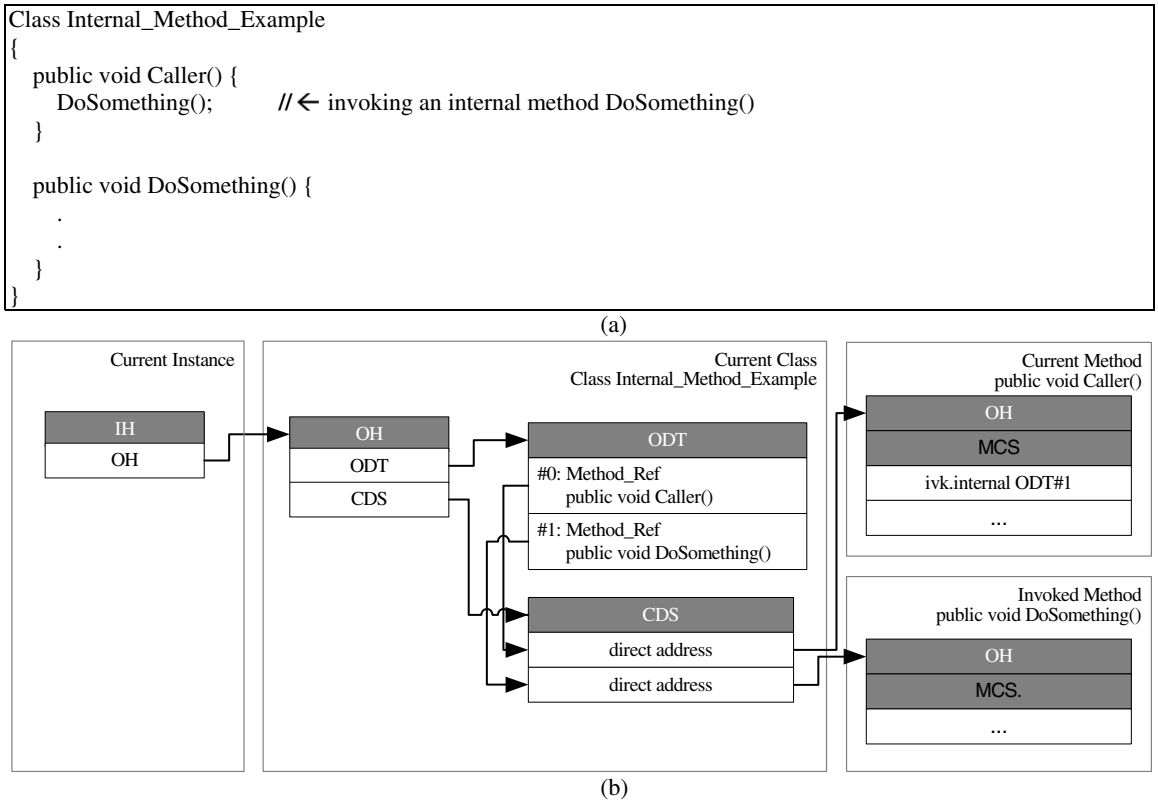

Fig. 4. An internal method invocation (a) Java program (b) Object context switching.

Once meeting a method revocation instruction, the processor will restore the object contexts by popping the previously stored context contents according to the stage register. If we assume that all data are hit in the data cache and the object is resolved, this internal method invocation will consume five clock cycles. Among them, reading the operand descriptor from the operand descriptor table, obtaining the direct address from the CDS of current class, accessing the object header of the invoked method, storing the current method context, and allocating the new object frame for the invoked method take one clock cycle, respectively.

However, in PicoJava II, the method invocation in Fig. 4(a) is performed by the Bytecode *invokevirtual* through software trap first. It will consume 195 clock cycles in the case of the same assumption. Among them, 31 clock cycles are taken by reading the object reference and verifying whether the object is resolved; 29 clock cycles are consumed by extracting the pointer to the field_info_table entry and checking the access flag; 107 clock cycles are spent in accessing the invoked method and checking the method index to determine the quick instruction type; 13 clock cycles are used to rebuild the trap frame and return; 15 clock cycles are taken by the quick instruction.

3.4. Instruction set

In jHISC, the instruction set is compatible with the MIPS32 except for the object-oriented related instructions and data transfer instructions between memory and registers. In traditional computers, data transfer operations between memory and registers allow application programs to access memory directly through the *load*/*store* instructions, which may result in the illegal data accessing and cause a security problem. For example, some viruses may cast an integer as a memory address where the system data are stored and then crash the host system by operating the data. In jHISC, the instructions *load* and *store* are replaced by the instructions *array.load* and *array.store*, which are executed with tough checks, such as boundary, data type, access permission, and so on, to prevent the malicious accesses. However, these checks increase some overhead, for example, the instruction *array.load* takes three clock cycles while the instruction *load* consumes only one clock cycle. Due to the limitation to access memory directly, the instructions *oo_set_header*, *oo_cod_setreference*, and *oo_cod_setresolved* are added to access memory for the object creation and reference resolution.

In jHISC, all data are encapsulated into objects and described by the operand descriptors. Each object associates with a pair of memory boundaries (upper and lower boundary), which can be calculated through the base address of object header and the field *DsSize* or *arraySize*. When a program accesses data, it needs to read the related operand descriptors and pass the access control checks. Generally, the instructions *gifld*, *gfld*, *pifld*, and *pfld* are provided to operate on the instance fields. The instructions *gsfld* and *psfld* are added to access the static fields. The instructions *array.store* and *array.load* are used to access the elements of an array. To speed up execution as the Bytecode *invokevirtual*, the Bytecode *getfield* is divided into two instructions, i.e., *gifld* and *gfld*, which are used to get the value of an instance field within and outside the current class context, respectively. The conversions are carried out at run-time. If the Bytecode *aload_0* immediately precedes the Bytecode *getfield*, the Bytecodes *aload_0* and *getfield* will be folded into the instruction *gifld*, otherwise the Bytecode *getfield* will be converted into the instruction *gfld*. Similar way is also applied to the Bytecode *putfield,* which is divided into the instructions *pifld* and *pfld*. If the Bytecode *aload_0* immediately precedes the Bytecode that is directly followed by the Bytecode *putfield*, these three consecutive Bytecodes will be folded into the instruction *pifld*. Otherwise, they are folded into the instruction *pfld*.

Six groups of instructions are defined in jHISC. They are logical instructions, arithmetic instructions, branching instructions, array manipulation instructions, object-oriented instructions, and data manipulation instructions. After the instructions for floating-point operations are excluded, 94% of all Bytecodes and 83% of the object-oriented related Bytecodes are implemented in hardware directly. The other complex Bytecodes, such as *new*, *newarray*, and so on, are implemented by software traps due to their assistance requirements of the operating system. The corresponding details are shown in Tables 2 and 3.

3.5. System architecture

The architectural block diagram of jHISC is shown in Fig. 5. The whole system, which contains 4 kbyte instruction cache and 8 kbyte data cache, is implemented by six

Table 2
Bytecodes supported in jHISC

Number of Bytecode	226
Number of Bytecodes after the instructions for floating-point operations are excluded	167
Number of Bytecodes supported in hardware	157
Number of Bytecodes implemented by software traps	10
Number of object-oriented related Bytecodes	41
Number of object-oriented related Bytecodes supported in hardware	34
Percentage of Bytecodes supported in hardware	94
Percentage of object-oriented related Bytecodes supported in hardware	83

Table 3
Bytecodes implemented through software traps in jHISC

new	newarray	anewarray	multianewarray
new_quick	athrow	anewarray_quick	multianewarray_quick
monitorenter	monitorexit		

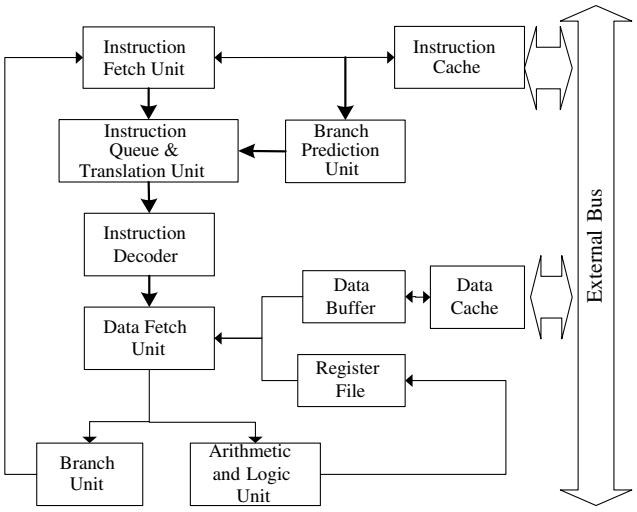

Fig. 5. Block diagram of system architecture.

pipeline stages: instruction fetch (IFETCH), instruction folding and translation (ITRANSLATION), instruction decoding (IDECODE), data fetch (DFETCH), execution (EXEC), and write-back (WBACK). The pipeline control flow is shown in Fig. 6.

In Fig. 5, the Instruction Queue & Translation unit consists of an instruction buffer, an instruction folding manager, and a stage controller. The instruction buffer is used to store the Bytecodes fetched from the instruction cache or external memory. The instruction folding manager classifies the Bytecodes in the instruction buffer according to their opcodes and the type definitions, then checks their foldability, and converts the Bytecodes to the jHISC instructions by one to one or N to one. The folding model and algorithm are based on the EPOC model, which was proposed by Ton et al. [35,36]. For example, the Bytecode stream *aload_4*, *getfield #5*, *istore_3* can be translated into the jHISC instruction *gfld #5*, *R4*, *R3*. Stage controller is used to control the whole stage of instruction pushing, popping, and folding. Data buffer unit consists of 16 multi-port registers so that data can be read or written synchronously to reduce the accessing time.

4. System performance and analysis

The system with 4 kbyte instruction cache and 8 kbyte data cache was designed with VHDL and implemented through a Xilinx Virtex FPGA XCV800. We analyzed the Bytecode distribution in the CaffeinMark [34] and clock cycles taken by each Bytecode, then normalized them to obtain the weighted average number of clock cycles per Bytecode to estimate the system performance. During estimation, we compared jHISC with PicoJava II and JOP in the occupied hardware resources and execution performance. PicoJava II and JOP are two open sources. PicoJava II is a full functional Java processor and faster than the JIT compiler and interpreter [7]. Some subsequent Java processors are based on it. JOP, which executes Bytecodes through microcode translation, is a compact Java processor and designed for the embedded real-time systems [22,37]. Just like most of the other existing Java processors, PicoJava II and JOP are stack architecture. The object-oriented related instructions are implemented by software traps in PicoJava II while they are realized through microcode in JOP.

4.1. Hardware resource needed

We synthesized the three systems with the same functional components under the consistent optimization conditions. Hence, we needed to delete the redundant logic in PicoJava II, such as Powerdown, Clock and Scan unit, and Floating-Point unit. The caches were generated by Xilinx CORE Generator and implemented by the internal block RAMs of FPGA in jHISC. As a result, the synthesized JOP core occupied 2271 LUTs (Look-Up Table), 13 block RAMs, and 21 32×1 ROMs in FPGA

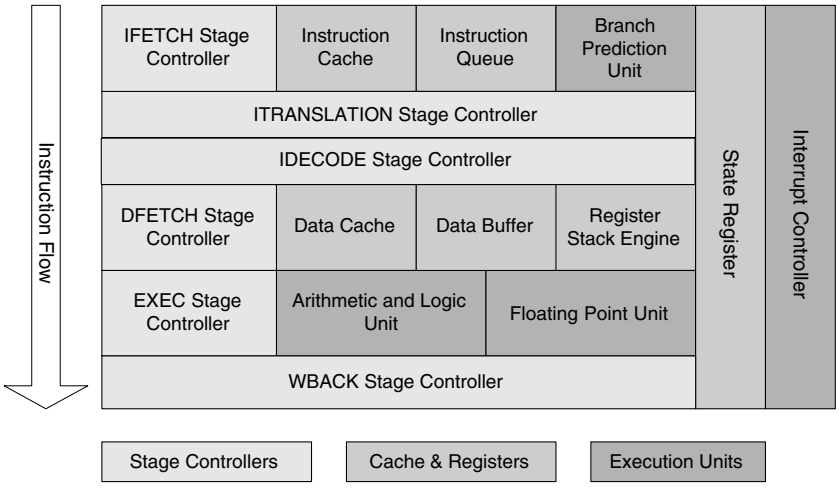

Fig. 6. Pipeline control flow.

(xcv800bg432-6); the PicoJava core needed 43,053 LUTs, 42 single-port RAMs (RAM128X1S), 6 128×1 ROMs, and 122 256×1 ROMs whereas jHISC consumed 15,803 LUTs and 28 block RAMs. If we deleted the microcode unit in PicoJava II, which is mainly used to realize the quick variations of object and array manipulation instructions, the synthesized core occupied 41,207 LUTs and 42 single-port RAMs. The details are shown in Tables 4 and 5.

We can find that the total LUTs, which are used to implement the logic circuits in FPGA, occupied by jHISC are about 37% of those needed by PicoJava II. As mentioned before, the quick variants of object-oriented related bytecodes are realized by microcode, which needs many ROMs (6 128×1 ROMs and 122 256×1 ROMs) so that the chip area and complexity increase. In JOP, the caches are implemented by the internal block RAMs of FPGA, and the logic circuits (total LUTs) are much smaller than those needed by jHISC and PicoJava II because (1) the data cache (substituted by the stack cache) size is only 1 kbyte in JOP while it is 8 kbyte in jHISC and PicoJava II; (2) there are no security and exception checks in JOP, which are critical features of Java virtual machine and will increase the system complexity; (3) the simple instructions are implemented by hardware directly in JOP, and the complex Bytecods are translated into sequences of these simple instructions through microcode. Thus, the logic circuits are less required, but more ROMs (21 32×1 ROMs) are needed; (4) there are no register files in JOP.

4.2. Execution performance

Tables 6 and 7 show the distribution of operations in the CaffeinMark executed in the K Virtual Machine (KVM) [5], which is a highly portable Java virtual machine designed for the memory-constraint small embedded devices. The tables show that the load/store operations, which can be executed in one or two cycles, account for about 52% of all operations, and the object and array manipulation operations are about 18% of all operations. Moreover, in order to speed up execution, the object-oriented related Bytecodes are mostly executed by their quick variants because the quick variants are about 9.05% of all operations whereas the original formats are about 0.91% of all operations. In addition, the Bytecode *aload_0* pushes the *"this"* reference onto the operand stack for constructors and instance methods in the Java virtual machine. It indicates whether the object operation is within the current class context or not. For example, if a Bytecode *aload_0* is previously adjacent to the Bytecode *invokevirtual*, it means that the system will invoke an internal method of the current class. From Table 7, the operation *aload_0* is 8.43% of all operations, and the operations for constructors and instance methods (getfield, putfield, invokevirtual, and invokespecial) are about 9% of all operations, thus most of the instance operations occur within the same class context.

As in the KVM, the object-oriented related Bytecodes are executed originally by software traps in PicoJava II. Once the specific entries in the constant pool are resolved, the object-oriented related Bytecodes are replaced by their corresponding quick variants, which are executed via microcode. In JOP, there are no quick variants, and the object and array manipulation instructions are translated into a series of native instructions through microcode. Table 8 shows the execution time in clock cycles of some main object and array manipulation instructions. In Table 8, the cycle counts for jHISC are based on its RTL model. The cycle data for JOP are obtained from [37] by assuming the number of clock cycles to access memory is one. In PicoJava II, the cycles consumed by the original formats of object-oriented related Bytecodes are estimated by totaling all the clock cycles taken by the relevant Bytecodes in the software traps, and the cycles needed by the quick variants are quoted from the data sheet [9]. During estimation, the time consumed by the object-oriented related instructions handler, such as exception handling, is ignored in PicoJava II.

Similar to PicoJava II and JOP, jHISC executes the simple instructions in one or two cycles, but for the object and array manipulation instructions, their original formats are executed much faster and quick variants are also performed a bit faster in jHISC. The main reason has been analyzed in Section 3.2. In addition, the branch operations are also executed much faster in jHISC than in PicoJava II

Table 4
Functional components and optimization situations

jHISC	PicoJava II	JOP
• Instruction Cache (4 kbyte)	• Instruction Cache Unit (4 kbyte)	• Instruction Cache Unit (4 kbyte)
• Instruction Fetch Unit	• Integer Unit (IU)	• Bytecode Fetch unit
• Instruction Queue Unit	• Data Cache Unit (8 kbyte)	• Native Instruction Fetch Unit
• Instruction Decoder	• Stack Manager Unit (SMU)	• Stack
• Data Fetch Unit		• Stack Cache (1 kbyte)
• Data Buffer		• Instruction Decoder
• Register File (32 general-purpose registers)		
• Data Cache (8 kbyte)		
• Arithmetic and Logic Unit		

Optimization conditions during synthesis: (1) FSM compiler (2) resource sharing.

Table 5
Synthesized results

jHISC (xcv800bg432-6)		PicoJava II (microcode unit) (xc2v4000bf957-6)		PicoJava II (no microcode unit) (xc2v4000bf957-6)		JOP (xcv800bg432-6)	
Cell usage		*Cell usage*		*Cell usage*		*Cell usage*	
FDC	139 uses	BUF	4 uses	BUF	2 uses	FDC	244 uses
FDCE	3301 uses	FD	578 uses	FD	584 uses	FD	90 uses
FDP	3 uses	FDE	34563 uses	FDE	34561 uses	FDCE	653 uses
FDPE	2 uses	FDR	475 uses	FDR	472 uses	FDE	228 uses
FDR	45 uses	FDRE	934 uses	FDRE	921 uses	FDP	54 uses
FDRE	465 uses	FDRS	2 uses	FDRS	2 uses	FDPE	14 uses
GND	26 uses	FDS	146 uses	FDS	143 uses	FDP_1	1 use
MULT_AND	161 uses	FDSE	25 uses	FDSE	30 uses	FDR	33 uses
MUXCY	213 uses	GND	11 uses	GND	11 uses	FDRE	9 uses
MUXCY_L	3798 uses	LD	85 uses	LD	85 uses	FDS	19 uses
MUXF5	1052 uses	MULT_AND	1 use	MULT_AND	1 use	FDSE	1 use
MUXF6	82 uses	MUXCY	5 uses	MUXCY	5 uses	GND	13 uses
RAMB4_S16	2 uses	MUXCY_L	205 uses	MUXCY_L	205 uses	MULT_AND	43 uses
RAMB4_S2	16 uses	MUXF5	9412 uses	MUXF5	9162 uses	MUXCY	19 uses
RAMB4_S4	8 uses	MUXF6	2386 uses	MUXF6	2394 uses	MUXCY_L	417 uses
RAMB4_S8	2 uses	MUXF7	175 uses	MUXF7	187 uses	MUXF5	188 uses
VCC	23 uses	MUXF8	64 uses	MUXF8	64 uses	MUXF6	53 uses
XORCY	612 uses	RAM128X1S	42 uses	RAM128X1S	42 uses	RAMB4_S4_S4	8 use
XORCY_L	16 uses	VCC	4 uses	VCC	4 uses	RAMB4_S16_S16	2 uses
		XORCY	113 uses	XORCY	113 uses	RAMB4_S4	3 uses
I/O primitives	174	*I/O primitives*	162	*I/O primitives*	162	VCC	8 uses
IBUF	75 uses	IBUF	42 uses	IBUF	42 uses	XORCY	266 uses
OBUF	99 uses	IBUFG	1 use	IBUFG	1 use	*I/O primitives*	61
BUFGP	1 use	OBUF	118 uses	OBUF	118 uses	IBUF	1 use
		OBUFT	1 use	OBUFT	1 use	IOBUF	32 uses
		BUFG	5 uses	BUFG	5 uses	BUFGP	1 use
RAM/ROM usage summary		*RAM/ROM usage summary*		*RAM/ROM usage summary*		*RAM/ROM usage summary*	
Block RAMs: 28 of 28 (100%)		RAM128X1S: 42 128 × 1 ROMs (ROM128X1): 6 256 × 1 ROMs (ROM256X1): 122		RAM128X1S: 42		ROM32X1: 21 Block RAMs : 13 of 28	
Mapping summary		*Mapping summary*		*Mapping summary*		*Mapping summary*	
Total LUTs: 15803 (83%)		Total LUTs: 43053 (93%)		Total LUTs: 41207 (89%)		Total LUTs: 2271 (12%)	

Table 6
Distribution of all operations

Total tested Bytecode instructions: 836,202,789 Operation type	Percentage
Instructions that push a constant onto the stack	8.25
Instructions that load a local variable onto the stack	38.36
Instructions that store a value from the stack into local variable	5.90
Stack operations	0.46
Integer, floating-point and logic operation	10.86
Type conversion	0.01
Control flow	17.37
Array operations	8.83
Object access, method invocation and revocation	0.91
Quick object-oriented related operations	9.05

and JOP. In PicoJava II, to execute a branch instruction, the system pops data off the operand stack and compares them. If the branch condition is met, the virtual machine forms a signed 16-bit offset, calculates a target address, and then jumps to the target address. In JOP, the execution of a branch instruction is similar to that in PicoJava II, apart from that the branch offset is obtained from the branch table, not operands. In jHISC, the compared data are stored in registers. The 24-bit branch offset is determined in advance and as an immediate operand of the branch instruction. If the branch condition is met, a branch instruction takes four cycles in PicoJava II and JOP while it takes two cycles in jHISC. This also contributes to the performance improvement in jHISC because the branch operations are about 14% of all operations.

Using N_i, W_i to represent the clock cycles consumed by a Bytecode and its distribution weighting, respectively, and N to denote the average clock cycles for each Bytecode execution, we have

$$N = \sum_{i=1}^{M} (N_i \times W_i) \ (M \text{ is the number of Bytecodes}). \quad (1)$$

Thus, we can estimate the overall performance of PicoJava II, JOP, and jHISC, which are presented in Table 9. During estimation, in jHISC, the cycles taken by the Bytecode *getfield* are the average of those taken by the instructions *gfld*

Table 7
Distribution of some object and array manipulation Bytecodes in the CaffeinMark

Bytecodes	Percentage
ireturn	0.9012
lreturn	0.0016
areturn	0.0014
return	0.0041
getfield_quick	2.2526
agetfield_quick	5.3544
getfield2_quick	0.0020
putfield_quick	0.4408
putfield2_quick	0.0009
getstatic_quick	0.0003
agetstatic_quick	0.0014
putstatic_quick	0.0
invokevirtual_quick	0.8784
invokespecial_quick	0.1135
invokestatic_quick	0.0083
invokeinterface_quick	0.0009
new_fast	0.0007
iaload	3.0366
aaload	1.7316
caload	1.8490
iastore	0.4792
new	0.0
newarray	0.0013
anewarray	0.0
multianewarray	0.0
monitorenter	0.0
monitorexit	0.0
aload_0	8.4352

Table 8
Cycles needed by some main object and array manipulation Bytecodes

Bytecodes in PicoJava II	Cycles		JOP	Instruction in jHISC	Cycles
	Original format	Quick variant			
getfield	114	4		gfld	6
			12	gifld	2
agetfield_quick	4				
putfield	130	4	15	pfld	6
				pifld	2
getstatic	103	3	6	gsfld	6
agetstatic_quick	3				
putstatic	103	3	7	psfld	6
invokestatic	86	11	67	ivkclass	9
invokevirtual	195	15	88	ivkintance	9
				ivkinternal	5
invokespecial	208	17	67	ivkintance	9
invokeinterface	203	184	96		
checkcast	97	6		checkcast	3
instanceof	100	7		instanceof	4
ireturn			19		
return	8		17	oo_rvk	5
areturn			19		
return			19		
iaload					
aaload	5		24	arrayload	3
caload					
iastore	7		26	arraystore	3

Table 9
Overall performance

	Cycles
The average clock cycles for each Bytecode execution in PicoJava II	2.59
The average clock cycles for each Bytecode execution in JOP	5.58
The average clock cycles for each Bytecode execution in jHISC	1.97

and *gifld*. The same way is also applied for the Bytecodes *putfield* and *invokevirtual*.

As indicated in Table 9, when the three systems have the same clock frequency, jHISC improves the overall performance about 30% (2.59/1.97 − 1) against PicoJava II and 183% (5.58/1.97-1) against JOP. If we assume all the object-oriented related Bytecodes are executed by software traps, the average clock cycles for each Bytecode execution in PicoJava II will be increased to 13.25, and the overall performance will be speeded up about 570% (13.25/1.97-1) by jHISC. Compared with the quick variants replacement scheme in PicoJava II, although the overall performance is only speeded up 30%, the hardware resource needed is reduced significantly in jHISC. More important, we can apply some techniques of instruction level parallelism to further improve the performance. And if we adopt a cache to store the direct references of objects after they are resolved, the execution time consumed by the object-oriented related Bytecodes will be further cut down by half in jHISC. In addition, Tables 6 and 7 show that jHISC is very effective for the object-intensive computing. The speed-up gains 30% and 183% are much smaller in our estimation because: (1) most of the instance operations are actually within the same class context, thus the cycles taken by the Bytecodes *invokeinstance*, *getfield*, and *putfield* in jHISC are smaller than those in our estimation; (2) the object-oriented operations are relatively few and infrequent in the CaffeinMark (about 10% of all operations).

5. Conclusion

Small embedded devices with Java applications, such as mobile phones, Pocket PCs, PDAs, and so on, are becoming more and more popular. It is necessary for their processors to have small chip area and good performance for execution of Java applications. In view of these, jHISC offers an attractive solution for these upcoming devices. First, it uses operand descriptors to describe object and each field of an object context is stored with a constant address offset, which make the complex object-oriented related Bytecodes be implemented by hardware directly and object information be accessed parallel to improve the system performance. Second, because most of the instructions are implemented by hardware directly and no ROMs are required for microcode, the chip area is reduced, which results in the decrease of power consumption.

Acknowledgements

This work has been supported by the Strategic Research Grant 7001847 of the City University of Hong Kong. The authors thank Mok Pak Lun, Lo Kai Man, and Yu Wing Shing for their invaluable consultation.

References

[1] J. Gosling, B. Joy, G. Steele, The Java™ Language Specification, Addison Wesley, Reading, MA, 1996.
[2] http://au.sun.com/news/localpress/2004/06/11_print.html.
[3] M. Grand, Java Language Reference, O'Reilly, 1997.
[4] M.W. El-Kharashi, F. Elguibaly, Java Microprocessors: Computer Architecture Implications, in: 1997 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, August 1997, pp. 277–280.
[5] Sun Microsystems: White Paper on KVM and the Connected, Limited Device Configuration, Sun Microsystems White Paper, May, 2000.
[6] N. Vijaykrishnan, N. Ranganathan, R. Gadekarla, Object oriented architectural support for a Java processor, in: ECOOP'98, 1998, pp. 330–355.
[7] J. M. O'Connor, M. Tremblay, PicoJava-I: The Java virtual machine in hardware, IEEE MICRO, March 1997, pp. 45–53.
[8] H. McGhan, J.M. O'Connor, PicoJava: A direct execution engine for Java Bytecode, Computer (1998) 22–30.
[9] Sun Microsystems: PicoJava-II: Java Processor Core, Sun Microsystems Data Sheet, April 1998.
[10] aJile Systems Inc.: aJ-100 Real-Time Low Power Java™ Processor, aJ-100™ Reference Manual, Version 2.1, December 2001.
[11] Aurora VLSI Inc.: AU-J2000: Super High Performance Java Processor Core (Data Sheet), Aurora VLSI Inc., 2000.
[12] ARM: Jazelle Technology for Java Application, ARM Data Sheet, May 2001.
[13] NanoAmp, Solutions, Inc.: The MOCA-J™ Accelerator: Performance Boosting Solutions for J2ME Software, White Paper on MOCA-J™, 2003.
[14] E. Lattanzi, A. Gayasen, M. Kandemir, V. Narayanan, et al., Improving Java performance using dynamic method migration on FPGAs, in: The 18th International Parallel and Distributed Processing Symposium, April 2004, pp. 134–141.
[15] H. Yajun, R. Hipik, et al., Adding hardware support to the HotSpot virtual machine for domain specific applications, in: Lecture Notes in Computer Science, vol. 2438, September 2002, pp. 1135–1138.
[16] J. Parnis, G. Lee, Exploiting FPGA concurrency to enhance JVM performance, in: Australasian Computer Science Conference, January 2004, pp. 223–232.
[17] K.B. Kent, M. Serra, Hardware/Software co-design of a Java virtual machine, in: IEEE International Workshop on Rapid Systems Prototyping, June 2000, pp. 66–71.
[18] NAZOMI Communications Inc.: JA108 – Multimedia Application Processor (Product Brief), 2003.
[19] R. Radhakrishnan, R. Bhargava, L. John, Improving Java performance using hardware translation, in: ACM International Conference on Supercomputing, June 2001, pp. 427–439.
[20] J. Glossner, S. Vassiliadis, The Delft-Java engine: an introduction, in: The Third International Euro-Par Conference on Parallel Processing, August 1997, pp. 766–770.
[21] J. Glossner, S. Vassiliadis, Delft-Java link translation buffer, in: The 24th Conference on EuroMicro, August 1998, pp. 221–228.
[22] M. Schoeberl, JOP: a java optimized processor, in: Lecture Notes in Computer Science, vol. 2889, October 2003, pp. 346–359.
[23] A. Kim, M. Chang, Designing a Java microprocessor core using FPGA technology, Comput. Control Eng. J. 11 (3) (2000) 135–141.
[24] K.B. Kent, M. Serra, Hardware architecture for Java in a hardware/software co-design of the virtual machine, in: The Euromicro Symposium on Digital System Design, September 2002, pp. 20–27.
[25] K.B. Kent, M. Hejun, M. Serra, Rapid prototyping of a co-design Java virtual machine, in: IEEE International Workshop on Rapid System Prototyping, June 2004, pp. 164–171.
[26] M.W. El-Kharashi, F. Elguibaly, K.F. Li, A quantitative study for Java microprocessor architectural requirements. Part II: High-level language support, Microprocess. Microsyst. 24 (2000) 237–250.
[27] M.W. El-Kharashi, F. Elguibaly, K.F. Li, A quantitative study for Java microprocessor architectural requirements. Part I: Instruction set design, Microprocess. Microsyst. 24 (2000) 225–236.
[28] N. Vijaykrishnan, N. Ranganathan, Supporting object accesses in a Java processor, in: IEE Proceedings – Computers and Digital Techniques, vol. 147, No. 6, November 2000, pp. 435–443.
[29] P.L. Mok, A.S. Fong, K.W. Hau, Object-oriented processor requirements with instruction analysis of Java programs, ACM SIGARCH Comput. Archit. News 31 (5) (2003) 10–15.
[30] P.L. Mok, C.L. Li, A.S. Fong, Method manipulation in an object-oriented processor, ACM SIGARCH Comput. Archit. News 31 (4) (2003) 18–25.
[31] A.S. Fong, A computer architecture with access control and cache option tags on individual instruction operands, ACM SIGARCH Comput. Archit. News 31 (3) (2003) 1–5.
[32] A.S. Fong, HISC: a high-level instruction set computer, in: The Seventh European Simulation Symposium, October 1995, pp. 406–410.
[33] T. Lindholm, F. Yellin, The JAVA Virtual Machine Specification, second ed., Addison Wesley, Reading, MA, 1999.
[34] http://www.benchmarkhq.ru/cm30/index.html.
[35] L.R. Ton, L.C. Chang, C.P. Chung, An analytical POC stack operations folding for continuous and discontinuous Java Bytecodes, J. Syst. Archit. 48 (2002) 1–16.
[36] L.R. Ton, L.C. Chang, J.J. Shann, C.P. Chung, Design of an optimal folding mechanism for Java processors, Microprocess. Microsyst. 26 (2002) 341–352.
[37] M. Schoeberl, JOP: A Java optimized processor for embedded real-time systems, PhD thesis, http://www.jopdesign.com.