Design and implementation of a Java processor

Y.Y. Tan, C.H. Yau, K.M. Lo, W.S. Yu, P.L. Mok and A.S. Fong

Abstract: Java is widely applied in current embedded systems due to its object-oriented features and advantages such as security, robustness, and platform independence. A Java virtual machine is needed to execute Java programs. However, in most of the existing solutions to Java virtual machines, the overhead of executing object-oriented related instructions is significant and becomes the bottleneck of system performance. To solve this problem, a novel Java processor called jHISC is proposed, which mainly targets J2ME and embedded applications. In jHISC, the object-oriented related instructions are implemented by hardware directly, as a hardware-readable data structure is used to represent the object. The complete system with 4 kB instruction cache and 8 kB data cache is described by VHDL and implemented in a Xilinx Virtex FPGA. It occupies 601 859 equivalent gates and the maximum clock frequency of the system is about 30 MHz. Compared with PicoJava II, the overall performance is speeded up 1 to 7.4 times and the execution efficiency of object-oriented related bytecodes is improved by 0.91 to 13.2 times for the same clock frequency.

1 Introduction

Java was introduced in the mid-1990s by Sun Microsystems and is now widely applied in network applications and embedded devices, such as PDAs, mobile phones, TV set-top boxes and Palm PCs [1]. Java claims to be more robust, secure and portable in addition to its inherited common advantages of object-oriented programming languages such as encapsulation, polymorphism, dynamic binding and inheritance. Its increasing robustness and security can be attributed to automatic garbage collection, static and run-time type checking, exception handling mechanism, array boundary checking and restrictive object reference management [2, 3] while its enhanced portability is realised through the compilation and execution of Java machine instructions called bytecodes instead of the particular processor binaries. To meet the demand of the rapidly developing embedded devices market, Sun Microsystems extended the scope of Java technology with the introduction of *Java 2 Platform Micro Edition (J2ME)*. With J2ME, applications can be shared for a wide range of devices and downloaded dynamically [4]. Since then, it has become the universal standard environment for the downloadable services and mobile entertainments running on mobile phones and PDAs.

Java bytecodes are originally executed in a virtual machine by interpretation where operations are emulated by using loops to fetch, decode and execute. Interpreter is applied for its simplicity, relatively easy implementation and small memory requirement, although its performance is affected significantly by time-consuming loops during software emulation. Instead of dynamically interpreting each bytecode at run-time, a just-in-time (JIT) compiler converts Java bytecodes into native instructions on the fly and caches them to eliminate the future redundant translations. It offers significant speedup over interpreter while also introducing additional compilation overhead and consuming much more memory, a precious resource in embedded systems [5, 6]. In contrast to the JIT compiler, an offline compiler translates Java source codes or bytecodes to native instructions or intermediate languages like C, and applies some time-consuming techniques to optimise the generated codes. However, it results in loss of portability, which is a critical Java feature.

Executing bytecodes by software emulation is inefficient; for example, the average number of instructions needed to emulate a bytecode in the UltraSPARC platform is 35 for interpreters and 20 for JIT compilers [7, 8]. An alternative solution to improve execution performance uses the Java processor, which implements the Java Virtual Machine (JVM) by hardware and combines the advantages of interpreters and JIT compilers. It potentially delivers much better performance than a general-purpose processor for Java applications by tailoring hardware support for some Java special features such as security, multithreading and garbage collection. Compared with other methods, Java processors appear to be more suitable for embedded devices.

A number of researchers and companies have focused on developing Java processors in recent years [6, 9–26]. In this paper, we propose a novel Java processor called jHISC, which is a 32-bit processor and mainly targets J2ME applications in embedded devices. The rest of this paper is structured as follows. The related work is introduced in Section 2, as jHISC architecture is described in Section 3. In Section 4, system implementation and performance estimation results are presented. Finally, conclusions are made in Section 5.

2 Related work

From the proposed solutions to the Java processor in recent years [6, 9–26], three approaches may be summarised: dedication, acceleration and hardware translation to support bytecodes in hardware. A dedicated Java processor takes bytecodes as its native instructions and executes them

IEE Proceedings online no. 20050074

doi:10.1049/ip-cdt:20050074

Paper first received 13th June 2004 and in revised form 27th July 2005

The authors are with the Department of Electronic Engineering, City University of Hong Kong, Tat Chee Avenue, Kowloon, Hong Kong

E-mail: anthony.fong@ee.cityu.edu.hk

directly. The most popular and simplest way is to replace the JVM with a hardware stack machine, because JVM is fundamentally a stack-based machine implemented by software. In processors of this type, such as PicoJava I and II from Sun Microsystems, aJ-100 from aJile Systems, Inc. [6, 9–11], most of the simple bytecodes are implemented by hardware directly and the rest are employed by software traps or microcode. However, they also inherit all the weaknesses of the stack machine. For example, all operands such as temporary data, intermediate values, and method arguments are pushed onto or popped frequently from the stack during execution, so that the execution efficiency is quite low. In particular, the stack-based implementation of JVM imposes data dependency among the consecutive instructions so that any techniques of instruction-level parallelism are prohibited. Moreover, because they are pure Java processors, they will not execute the application programs written by other programming languages without specialised compilers to produce Java bytecodes.

In the translation approach, a small hardware unit is added between the instruction fetch and decoding units of a general-purpose processor core to convert most of the simple bytecodes into native instructions at run-time, and for the other complex bytecodes like object-oriented related bytecodes, the system invokes software traps. ARM Jazelle and JA108 are two well-known Java processors using this method [12, 13]. In JA108, hardware units were added to convert multiple stack-based bytecodes into a register-based native instruction, but for the complex bytecodes without direct hardware assistance, it took the traditional JVM interpreter to perform them. Several researchers also finished some works by adopting this method. Radhakrishnan *et al.* [16] and Schoeberl [26] accelerated Java performance by hardware interpretation, Glossner and Vassiliadis [24, 25] developed Delft-Java by translating most of the bytecodes into the Delft-Java instructions directly. The hardware translation method ensures Java processors maintain the integrality and instruction-level parallelism of the general-purpose processor. Processors of this type can also execute application programs written by other programming languages that host architecture supports. However, some features of Java language, such as security and object-oriented programming features, may be compromised if the general-purpose processor core does not support them at the hardware level.

A Java accelerator is a coprocessor attached to a host general-purpose processor to execute Java bytecodes so that the system can execute the application programs written by both Java and other programming languages supported by the host processor. Traditionally, accelerators are integrated inside the general-purpose processor core, such as in the AU-J2000 from Aurora VLSI Inc. [14], or function independently outside the host, as in the MOCA-J™ from NanoAmp Solutions, Inc. [15]. Some researchers have also adopted several techniques into coprocessors to speed up Java execution. Lattanzi *et al.* [18] and Ha *et al.* [20] have proposed schemes to speed up the execution of Java applications by dynamically migrating the most heavily used methods on a configurable hardware device. Kent *et al.* presented a software/hardware co-design method to complement the host processor with a FPGA-based Java coprocessor to execute most of the simple bytecodes [17, 27, 28]. Parnis and Lee built a multithreaded JVM based on FPGA to enhance its performance by exploiting the parallelism of FPGA [19]. Zheng Liang *et al.* invented a Java accelerator based on asynchronous circuits for low-power applications [22]. Such coprocessors provide good support to Java without affecting the compatibility of the host general-purpose processors, but chip area and power consumption increase significantly, which are critical factors in embedded devices.

Object-oriented operations constitute about 15% of all operations in the profiled benchmarks [29–31]. Hence, they have significant impact on the execution speed of Java programs. However, almost all existing Java processors execute object-oriented related bytecodes by software traps or microcode where an object-oriented-related operation may consume tens of clock cycles, sometimes more than 100 clock cycles. For example, the execution of bytecode 'invokevirtual' takes about 195 clock cycles by software trap in PicoJava II [10]. Although, in some solutions, quick version replacement schemes of object-oriented-related bytecodes have been introduced to speed up execution, they also increased the chip area and power consumption because the quick version of the related object-oriented bytecode was performed by microcode, which took a lot of ROMs [9, 10]. The performance penalty of these schemes does not fit well with the requirements of embedded devices, such as real-time operations, and low power consumption. In addition, many application programs written by other object-oriented programming languages are now available, which makes it desirable to have a general-purpose processor with enhanced architecture features to support object-oriented programming in hardware directly. To address these problems, we have developed jHISC, a novel Java processor supporting object-oriented operations at the hardware level.

3 System design

jHISC is a 32-bit object-oriented processor based on the High Level Instruction Set Computer (HISC) architecture, which extends typical computer architecture to support object-oriented programming in hardware by using hardware-readable data types called operand descriptors (OD) to describe objects [32–34]. It mainly targets such J2ME applications as smart cell phones, PDAs and other embedded devices, but floating-point operations are not supported in the current version.

3.1 HISC architecture overview

High Level Instruction Set Computer (HISC) architecture is a 64-bit processor proposed by Anthony Fong [34]. It extends typical computer architecture to support object-oriented programming at the hardware level by introducing 128-bit operand descriptors to describe both object references and variables. Each operand descriptor, residing in an operand descriptor table, is maintained by operating system and read only to user programs. Figure 1 shows the descriptor format in HISC, which contains *Address, Type, Size, Vector, Access Rights, Caching*

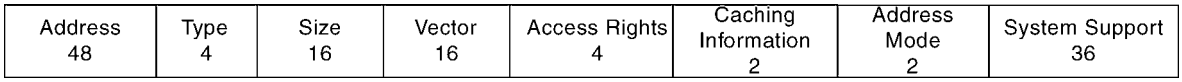


Fig. 1 *Operand descriptor format in HISC*

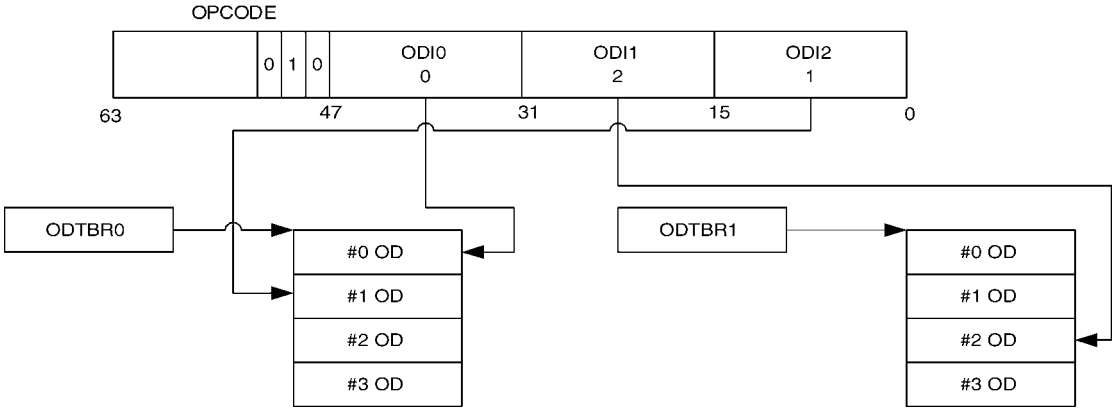


Fig. 2 *Accessing operand descriptors in the operand descriptor tables*

Information, Addressing Mode and *System Support* [34]. The details of each field are explained as follows:

- *Addressing Mode* defines the addressing mode, including direct addressing and stack pointer relative addressing.
- *Address* provides the physical location of the operand in the direct addressing mode or an offset to the stack pointer in the stack pointer relative addressing mode.
- *Type* defines the operand type. Eleven types, such as logic, integer, floating-point, binary-coded-decimal, character string, bit-string, object reference, signed and unsigned fixed-point, function and method, are defined. Besides the basic data type, HISC also supports object reference and method types.
- *Size* specifies the size of the operand.
- *Vector* is the count of the remaining elements of a vector. This is used to represent an array.
- *Access Rights* stores the access control information for each operand descriptor. Within it, three bits are used to define read, write and execution privileges and one bit specifies the operation mode, such as supervisor mode or user mode.
- *Cache Information* indicates the data coherency requirement in multiprocessing.

In HISC, each instruction is generally divided into four parts: a 16-bit opcode and three 16-bit operand descriptor indices (ODIs). The least three bits of opcode are used to select operand descriptor tables. Operand descriptor indices are applied to locate and retrieve operand descriptors in the operand descriptor table. HISC accesses objects by maintaining two operand descriptor tables (ODT0, ODT1) referenced through two operand descriptor base registers (ODTBR0, ODTBR1). Figure 2 illustrates an instruction with the least three bits of opcode being '010' and the three operand descriptor indices being 0, 2, 1 to look up operand descriptors in the operand descriptor tables.

The least three bits of opcode '010' indicate that ODI0, ODI2 will index into the operand descriptor table 0 and ODI1 will index into the operand descriptor table 1. According to the values of ODI0, ODI1 and ODI2, the processor can locate and access the #0 and #1 operand descriptors in the operand descriptor table 0 and #2 operand descriptor in the operand descriptor table 1.

3.2 jHISC architecture

3.2.1 Descriptor format: In HISC, a 128-bit operand descriptor makes the architecture complicated to implement using hardware. In jHISC, we simplified the operand descriptor to 32 bit according to the Java specification [35] and the uniform format is shown in Fig. 3, which includes *Address Field, Type Field, Static Flag, Access Modifier, Read-Only Flag*, and *Resolved Flag*. The function of each field is introduced as follows:

- *Address Field* provides a byte offset to locate data in the corresponding data spaces.
- *Access Modifier* is used for security control, and four access modifiers (public, private, protect and package) are defined in the current system.
- *Type Field* stores the data types defined for both primitive and reference, such as byte, integer, word, and reference, and so on. The primitive data are stored inside data space and, for the reference, a direct address is stored to locate the described resource.
- *Static Flag* indicates where the data are stored. For non-static fields, data are stored in the instance data space (IDS). For static fields, data are stored in the class data space (CDS). When a static field is inherited from a class extension, a direct address pointing to it is stored in the CDS.
- *Read-Only Flag* denotes whether the target can be written.
- *Resolved Flag* indicates whether the reference is resolved or not. If not, the system will be trapped to the operating system routines for the dynamic reference resolution.

Two kinds of operand descriptors, class operand descriptor and class property descriptor, are also defined in jHISC to assert the resources accessed by the class and the properties owned by the class, respectively. A class operand descriptor contains the *Address Field, Type Field*

Table 1: Bytecodes supported in jHISC

Total number of bytecodes	226
Number of bytecodes except the floating-point operation instructions	167
Number of bytecodes supported by hardware	156
Number of bytecodes performed by software traps	11
Number of object-oriented-related bytecodes	41
Number of object-oriented-related bytecodes supported by hardware	34
Percentage of bytecodes supported by hardware	93%
Percentage of object-oriented-related bytecodes supported by hardware	83%

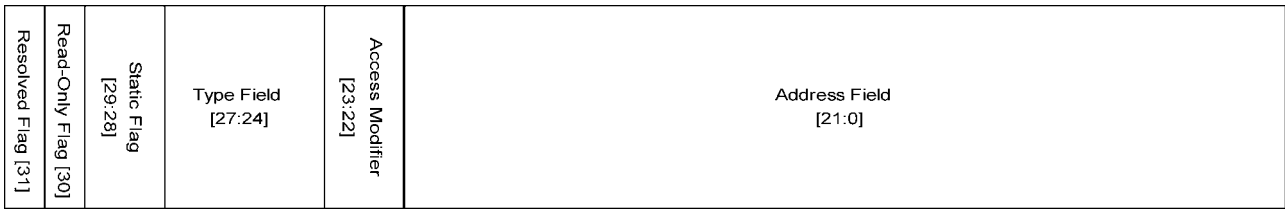


Fig. 3 *Operand descriptor format in jHISC*

and *Resolved Flag*; in a class property descriptor, only the *Resolved Flag* is not included.

3.2.2 Object representation: The object representation method is critical in an object-oriented programming system because of its effects on the speed of accessing objects. In jHISC, an object is represented by the hardware-readable data structure–object context, which consists of object header, data space and the corresponding descriptor tables, and so on. Three kinds of contexts, namely instance, class, and method contexts, are mapped to the hardware architecture and distinguished by the object header (OH), which is shown in Fig. 4.

Inside an object header, the function of each field is introduced as follows.

- *objType* stores the object type, such as instance, class, method and array.
- *dsSize* specifies the size of related data space, for example, CDS, IDS and Method Code Space.
- *gcInfo* is reserved to give hardware support for real-time garbage collection in the future; garbage collection is performed by the operating system in the current version.
- *class* links an instance object with its affiliated class through a reference pointer.
- *arraySize* and *arrayType* specify the number and type of elements in an array, respectively, when the object is an array.

Other than object header, an instance context also includes instance header (IH) and instance data space (IDS); a class context also contains class header (CH), class operand descriptor table (CODT), class property descriptor table (CPDT) and class data space (CDS); a method context consists of method header (MH), method code space (MCS) and local variable frame (LVF). When used to represent an array, an instance context also contains an array data area under the instance header. And inside class context, CODT and CPDT store class operand descriptors and class property descriptors, respectively. The different object context structures and their relations are shown in Fig. 5.

Typically, each object has a unique object context and a reference always points to the base address of the object header after the object is resolved. In an object context, all components are stored continuously and each is stored with a constant address offset to the object header, thus allowing the access of some components in parallel to reduce the accessing overhead. For example, as illustrated in Fig. 6*a*, a method *Caller*( ) in the class *Class_Method_Example* invokes a static method *About_Apple*( ) in the class *Apple*. During the method invocation, the processor requires the location of the method code, checking access control, pushing the contents of current object context onto the system stack, and passing the control from one object to another. The corresponding object context switches and the object structures are given in Fig. 6*b*.

In the current method space, instruction 'ivkclass' triggers a class method invocation and then accesses the #1 operand descriptor in the CODT of the current class to obtain the property reference *Apple·About_Apple*( ). The reference provides two offsets, one for getting the reference of class *Apple*, which provides the direct address inside the current CDS to locate the OH of class *Apple*, and another for accessing the method reference *public static void About_Apple*( ) inside the CPDT of class *Apple* to get the OH of method *About_Apple*( ) which is pointed by a direct address in the CDS of class *Apple*. Once the OH of method *About_Apple*( ) is located, the processor accesses the OH and MH of method *About_Apple*( ) synchronously and with their contents, the processor saves and updates the current class and method contexts, then accesses the MCS of method *About_Apple*( ) to fetch instructions to execute until meeting a method revocation instruction if no exception occurs. Once meeting a method revocation instruction, the processor will restore the corresponding contexts by popping the previously stored contents according to the stage register.

3.2.3 Instruction set: jHISC is a RISC processor with some architectural enhancements for object-oriented operations. Its instruction set supports a three-operand mode and is compatible with MIPS32 except for the memory-register data transfer and object-oriented related instructions. Memory-register data transfer instructions allow programs to access memory directly by the load/store

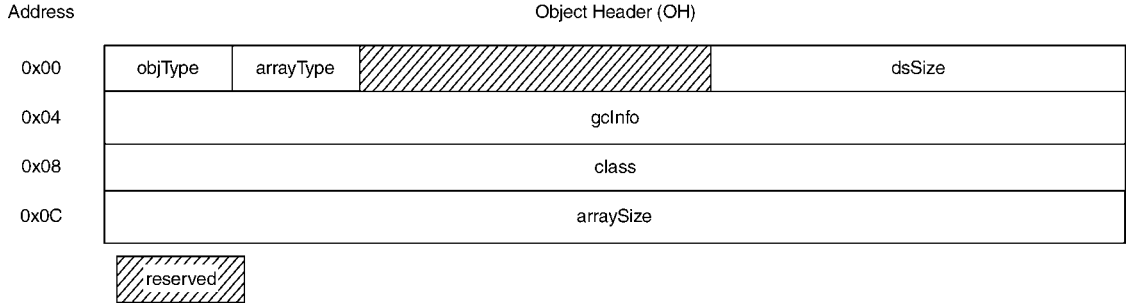


Fig. 4 *Object header format*

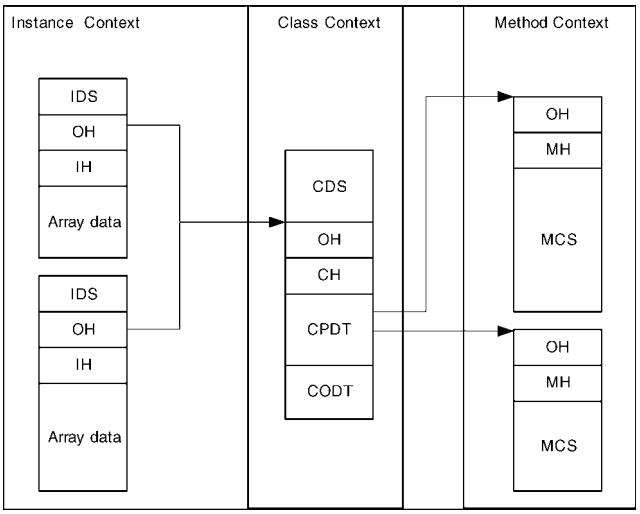


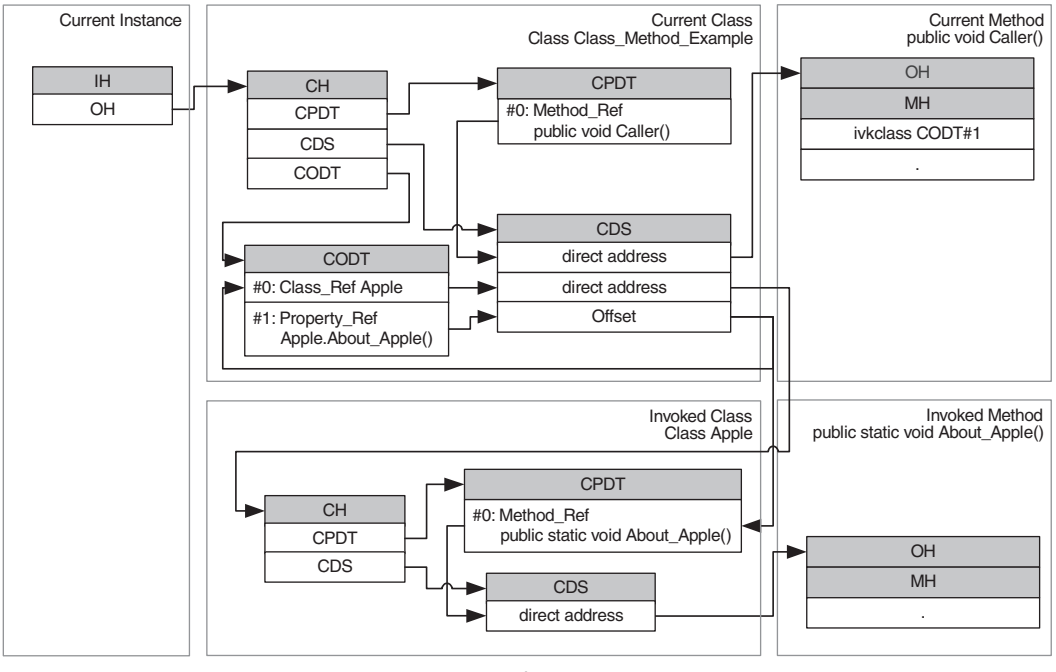
Fig. 5 *Different object structures and their relations*

instructions in traditional computers. Thus, application programs may access the data that do not belong to them; for example, some viruses may change an integer as a memory address where data used by OS are stored, and then take control or crash the host system by accessing these data. In jHISC, the load/store instructions are replaced by the instructions 'array.load' and 'array.store', respectively. During their execution, some secure checking is carried out, such as boundary, data type, and so on, to forbid virulent accesses, which also increases some overheads; for example, instruction 'array.load' consumes three clock cycles for its execution, whereas instruction 'load' needs only one clock cycle in traditional computers. Owing to the limitation to access memory directly, the instructions 'oo_set_header', 'oo_cod_setreference', 'oo_cod_setpropertyindex' and 'oo_cod_setresolved' are added to access memory with rough checking for object creation and reference resolution.

In jHISC, all data are encapsulated into objects and described by operand descriptors. Each object associates with a pair of memory boundaries (upper and lower boundary), which can be calculated through the base address of its object header and the field *DsSize* or *arraySize*. Before a program accesses data, it needs to read the related operand descriptors, access the specific object header and then pass the bound control checking, such as access right, boundary and data type checking, and so on. If it fails to pass the checking, the access will be prohibited. Generally, the instructions 'array.store' and 'array.load' are used to access the elements of an array; the instructions 'gifld', 'pifld', 'gfld' and 'pfld' are added to access data into an instance object, and the instructions 'gsfld' and 'psfld' are provided to perform data operations into a class

```
Class Apple
{
    public static void About_Apple() {
        .
    }
}
Class Class_Method_Example
{
    public void Caller() {
        Apple.About_Apple(); // <-- invoking class method About_Apple()
    }
}
```

a



b

Fig. 6 *Static method invocation*

a Java program
b Object context switching procedure

Table 2: Bytecodes implemented through software traps in jHISC

new	newarray	anewarray	multianewarray
new_quick	athrow	anewarray_quick	multianewarray _quick
monitorenter	monitorexit	lookupswitch	

object. Additionally, jHISC provides object and array manipulation instructions to handle the related operations. To improve the execution efficiency, bytecode 'getfield' is divided into two instructions, 'gifld' and 'gfld', which are used to get an instance variable value within and outside the current class context, respectively. A similar procedure is applied to the bytecodes 'putfield' and 'invokevirtual'.

Excluding the instructions for floating-point operations, 93% of all bytecodes and 83% of the object-oriented-related bytecodes are implemented in hardware, directly, in jHISC. Many of the performance-sensitive instructions, such as 'new' and 'newarray', not implemented in hardware due to their complexity and assistance requirements of the operating system, are executed through software traps. The corresponding details are shown in Tables 1 and 2.

3.2.4 System architecture: Figure 7 shows the architectural block diagram of jHISC. The whole system with 4 kB instruction cache and 8 kB data cache is implemented by five pipeline stages: instruction fetch, instruction decoding, data fetch, execution and write-back (described below in the occurring sequence).

Instruction Fetch: Instruction Fetch Unit controls Instruction Cache, Instruction Queue Unit and Branch Prediction Unit to fetch bytecodes either from instruction cache or external memory according to the program counter, which is calculated based on the state register and Branch Prediction Unit. The fetched bytecodes are put into the Instruction Queue Unit, where the bytecodes are folded into jHISC instructions. Instruction Cache is a read-only cache with 4 kB size and direct mapping. Inside Instruction Cache, a data RAM is provided for the temporary storage of instructions and a status RAM is used to store the validity and tag information of instructions. There are 256 cache lines, each storing 16-byte data in order to improve the hit ratio by fetching the continuous instructions to fill cache from the external memory when data miss occurs. Instruction Queue Unit is used to fold and translate bytecodes into jHISC native instructions. It will be described in detail in Section 3.2.5. Branch Prediction Unit predicts the branch results in order to maximise the efficiency of the pipeline.

Instruction Decoding: Instruction Decoder gets instructions from Instruction Queue Unit and decodes them to generate the related information, such as opcode and operands.

Data Fetch: Data Fetch Unit fetches data from Register File, Data Buffer, Data Cache or external memory according to the operands; at the same time, data access right and type checking are also carried out. Data Cache is a write-back cache with 8 kB size and direct mapping. Its basic structure is similar to the instruction cache, and the difference is that there are 256 cache lines, each storing 32-byte data. Data Buffer Unit consists of 16 multiport registers in order to make it possible to read or write data in parallel to reduce accessing time. When data are requested, Data Buffer will check whether there are copies in it. If not, it will send the request signal to Data Cache to fetch data and fill up the buffer. Additionally, 32 general registers are provided to implement the local variable frames of the JVM and store the object context contents. Typically, these registers are arranged as a circular stack like the operand stack in JVM and each stack entry is mapped into a register in jHISC. They are accessed with virtual register indices, and the real register indices are calculated based on the local variable frame. Once a new local variable frame is allocated, the register file controller will check whether there are enough free registers. If not, data in the used registers will be flushed to the memory until enough registers are available.

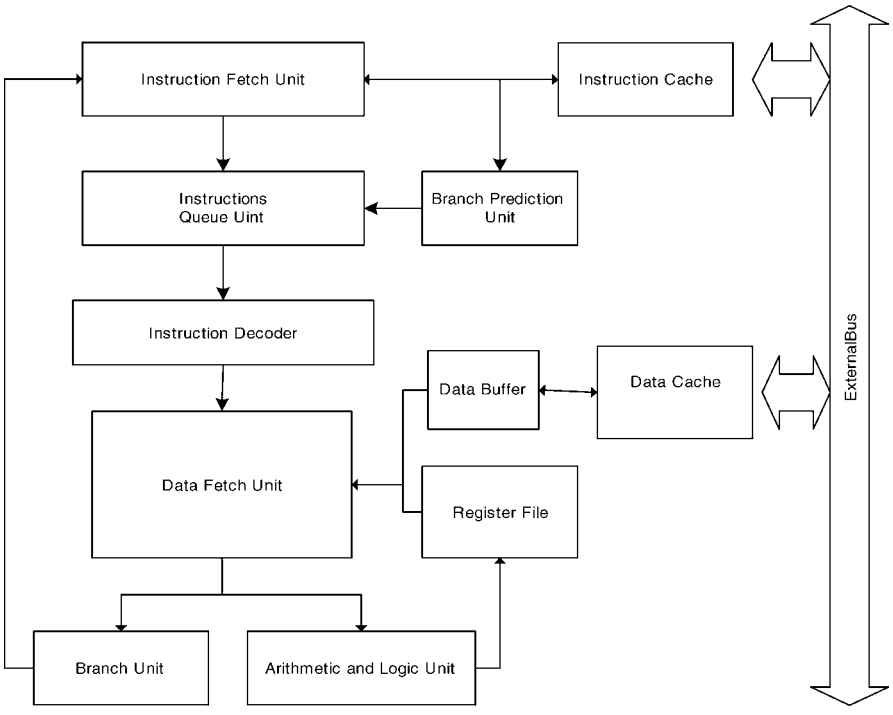


Fig. 7 *Block diagram of system architecture*

Execution: The Arithmetic and Logic Unit performs all arithmetic and logic operations, such as AND, OR, XOR, ADD, SUB, and so on. The object manipulation operations are implemented by a finite state machine and supported through managing the stage register and pipeline stage controller in the execution stage.

Write-back: The execution results are written back to the register file according to the register indices.

3.2.5 Instruction folding and translation: In stack machines, instruction folding is a technique used to eliminate the unnecessary loads or write-back operations to the stack by detecting some contiguous instructions and executing them collectively like a single, compound instruction. In Java processors, this technique was first introduced in PicoJava I and II by Sun Microsystems [6, 9, 10], where all bytecodes were classified into six types and combined into nine foldable patterns. In order to improve the folding efficiency, Chang *et al.* [36], Ton *et al.* [37, 38] and Kim *et al*. [39, 40] proposed folding algorithms based on POC, EPOC and Advanced-POC models, respectively, to fold the continuous bytecodes without patterns and discontinuous bytecodes. In jHISC, similar to the POC model, all bytecodes are classified into five types according to their characteristics. The type definitions are shown as follows [36, 40]:

- *Producer (P):* instructions that get data from constant registers or local variables and push them onto the operand stack, such as 'iconst_1', 'iload_3'.
- *Operator (O):* instructions that pop data from the operand stack and perform operations. This type is further divided into two subtypes, Producible Operators (O_P) such as 'iaload', 'iadd', which push its operation result onto the operand stack, and Consumable Operator (O_C), which does not push the operation result, such as 'if_icmpeq'.
- *Consumer (C):* instructions that pop data from the operand stack and store them back into local variables, such as 'istore', 'istore_0'.
- *Termination (T):* instructions that do not operate on the stack, such as 'goto', 'return'.
- *Temporary (Tp):* instructions that perform operations without popping data from the operand stack, but push the results onto it, such as 'getstatic'.

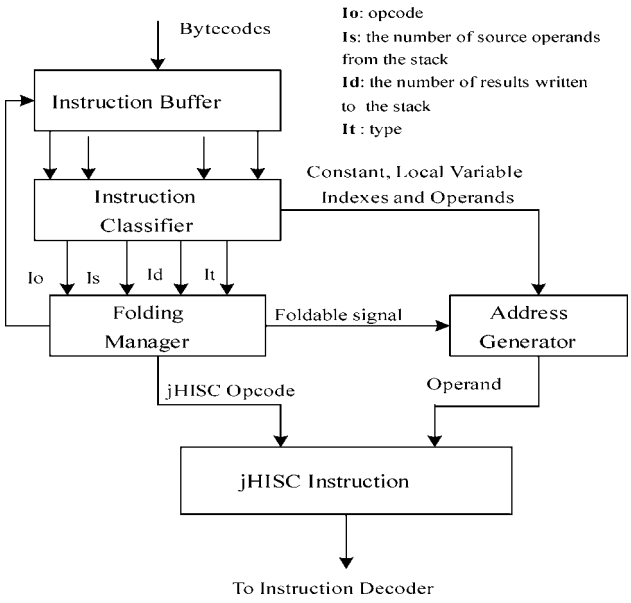


Fig. 8 *Block diagram of instruction folder*

The Folding rules can be simply summarised into the following three rules:

1. P type bytecode can be folded into the following adjacent C or O type bytecode.
2. C type bytecode can be folded into previous adjacent O_P or Tp type bytecode.
3. T type bytecode cannot be folded.

The block diagram of instruction folding and the translation unit is illustrated in Fig. 8. Bytecodes are fetched into the Instruction Buffer, which has eight entries, each storing one bytecode. The Instruction Classifier classifies bytecodes according to their opcodes and the type definitions. The corresponding bytecode types and opcodes are stored in an instruction tag register and opcode buffer inside the Folding Manager. The constant registers, local variable indices or operands are stored in an operand buffer inside the Address Generator. The Folding Manager checks foldability of bytecodes according to their types, folding rules, the source operands count popped and results count pushed onto the stack for each bytecode. The foldable checking flow is similar to that introduced in the Ton *et al.* papers [37, 38]. If bytecodes can be folded, the Folding Manager will generate the relevant jHISC opcode and a foldable signal to tell the Address Generator which bytecodes in the Instruction Buffer are folded; otherwise, it will only translate the bytecode into the jHISC instruction. Finally, the Address Generator produces the source and destination addresses for jHISC opcode according to the foldable signal and the information in the operand buffer.

In jHISC, the constant registers and local variable frames are implemented by the register file. Typically, all bytecodes can be mapped into jHISC instructions one to one if no folding occurs. For example, if a bytecodes stream is 'iload_3, iload 4, iload 6, iadd, istore 3, iload 7, iadd, istore 8, return', their corresponding types will be 'P, P, P, O_P, C, P, O_P, C, T'; the one-to-one mapping results are shown in Table 3. In the table, registers Rb, Rc and Rd are temporary registers allocated by the register file control engine. After the types of these bytecodes are obtained, the Folding Manager will detect the first O or C type bytecode. Because it is O type (*iadd*), the Folding Manager will check whether the next bytecode to the bytecode 'iadd' is C type or not. If it is, the two previous P type bytecodes adjacent to the bytecode 'iadd' and the C type bytecodes will be folded into the O type operation instruction (*arith.add R6*, *R4*, *R3*); otherwise, only the two previous adjacent P type bytecodes are folded into the O type operation instruction. In the same way, bytecodes in group 2 can also be folded into a jHISC instruction (*arith.add R7*, *R3*, *R8*). The results are illustrated in Fig. 9. From Fig. 9, we find that the instruction length is

Table 3: One-to-one mapping results

Original bytecode	jHISC assembly code
iload_3	data.move R3, Rb
iload 4	data.move R4, Rc
iload 6	data.move R6, Rd
iadd	arith.add Rd, Rc, Rc
istore 3	data.move Rc, R3
iload 7	data.move R7, Rc
iadd	arith.add Rc, Rb, Rb
istore 8	data.move Rb, R8
return	oo.rvk

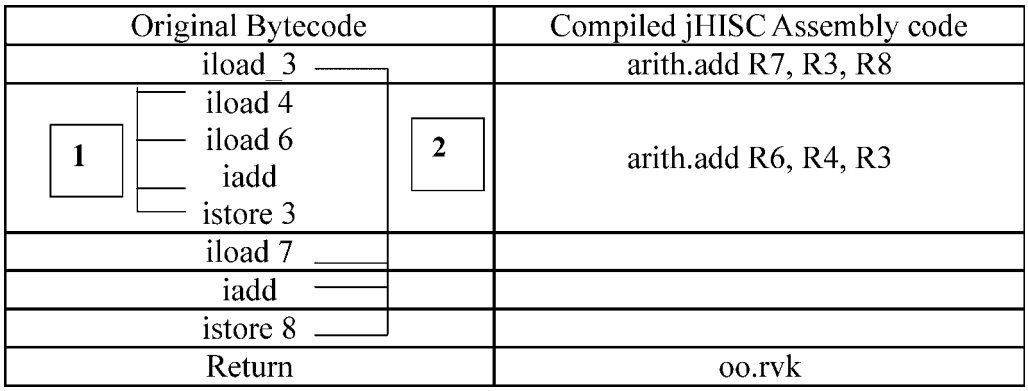


Fig. 9 *Folded results*

reduced from 9 to 3 after folding and the temporary registers are not needed. Because the instruction set of jHISC supports three-operand mode, most of the bytecodes can be folded. For example, bytecode stream 'aload_3, getfield #5, istore_2' can be translated into the jHISC instruction 'gfld #5, R3, R2'. If 'aload_3' is changed to 'aload_0', the jHISC instruction will be 'gifld #5, R2' because it obtains a field within a class context.

4 System implementation and performance estimation

4.1 System implementation

The complete system with 4 kB instruction cache and 8 kB data cache was described by VHDL and implemented in a Xilinx Virtex FPGA (XCV800-BG432; speed level 6). The corresponding chip is currently under development. During implementation, the caches were generated by Xilinx CORE Generator and implemented by the internal block RAMs of FPGA. The whole system occupied 601 859 equivalent gates in FPGA and its maximum clock frequency was about 30 MHz. Table 4 shows the mapping results reported by Xilinx ISE 6.0 and Table 5 presents the individual hardware resources needed by each component.

Table 5 indicates that about 92% of hardware resources are occupied by storage units such as Instruction Cache, Data Cache, Data Buffer and Register File. Data Fetch Unit also requires more hardware resources due to the complexity of the data fetch controller for execution of object-oriented-related bytecodes. If we implement all object-oriented-related bytecodes through software traps, the equivalent gate count needed by the Data Fetch Unit and the whole system will be reduced to 21 856 and 567 150, respectively, and the maximum clock frequency of the system will be increased to 33.2 MHz. Thus, the implementation of object-oriented-related instructions requires around 34 709 (601 859 − 567 150) equivalent gates in FPGA and the maximum clock frequency of the system is decreased by 3.2 MHz (33.2 to 30.0).

Table 4: Mapping results of the whole system

Logic utilisation:
Number of slice flip flops: 3951 out of 18 816
Number of 4 input LUTs: 13 918 out of 18 816 (73%)
Logic distribution:
Number of occupied slices: 8326 out of 9408
Total number 4 input LUTs: 15 573 out of 18 816
Number used as logic: 13 918
Number used as a route-through: 1655
Number of block RAMs: 28 out of 28
Total equivalent gate count for design: 601 859
Timing summary (post-map static timing): constraints cover 190 639 996 928 paths, 0 nets, and 85 468 connections
Design statistics:
Minimum period: 33.296 ns (maximum frequency: 30.034 MHz)
Minimum input required time before clock: 4.305 ns

4.2 Performance estimation

The performance of a processor can be defined as the time to execute a specific program, which is the product of three elements: the weighted average number of cycles per instruction, the cycle time and the number of instructions executed. Because the execution time of a program is not precise in the prototype machine due to the internal interconnect delays in FPGA, we only used the prototype machine to verify our concept and some simple programs have been tested on it. We analysed the distribution of bytecodes in the profiled benchmarks and clock cycles needed for the execution of each bytecode, and then normalised

Table 5: Hardware resources needed by each component

Component name	Equivalent gate count in FPGA	Percentage of the resources in the whole system, %
Instruction Cache	165 176	27.44
Instruction Fetch Unit	887	0.15
Instruction Queue Unit	5796	0.96
Instruction Decoder	908	0.15
Data Cache	298 206	49.55
Data Fetch Unit	33 789	5.61
Data Buffer	35 702	5.93
Simple Interrupt Control Unit	1551	0.26
Execution Unit	6414	1.07
Write-back Unit	650	0.11
State Register Control Unit	4224	0.70
Register File	53 784	8.94
The whole system	601 859	

Table 6: Bytecode size analysed in benchmarks

Benchmarks	Description	Bytecode size
SPEC JVM98 benchmark suits	a collection of benchmark programs for Java Virtual Machine [41]	28 829 139 706
Xbrowser V4.2	an open source Java application for browsing web [42]	275 250 622
Java2Demo	a demo program packaged with J2SDK	468 650 441

them to obtain the weighted average number of cycles per bytecode to estimate the system performance. We also chose PicoJava II as a comparison for two reasons: (1) it is an open source and full functional Java processor with six instruction pipeline stages, and (2) it is faster than the JIT compiler and interpreter [6] and some subsequent Java processors are based on it. During estimation, because the clock cycles needed for the execution of bytecodes were taken by assuming cache hits and no pipeline stalls or exceptions in PicoJava II, we used the same assumptions for jHISC. In addition, the instruction folding factor wasnot considered.

Tables 6–8 show the total bytecode size analysed, the distribution of all operations and the distribution of some main object-oriented-related bytecodes in the profiled benchmarks. The tables indicate that more than 50% of operations are load/store operations, which are executed in one or two cycles, and the object manipulation operations comprise about 14.72% of all operations. In PicoJava II, the object-oriented-related bytecodes are executed originally by software traps. Once the specific entries in the constant pool are resolved, the object-oriented related bytecodes will be executed by their quick formats implemented by microcode to speed up execution. Similar to PicoJava II, in jHISC, most of the simple instructions, such as load/store and logic operations, are executed in one or two cycles, but for object-oriented-related bytecodes, their original formats are executed much faster and quick formats performed a little faster than those in PicoJava II. Table 9 shows the clock cycles consumed by some object and array manipulation instructions in jHISC and PicoJava II. In Table 9, the results of jHISC were based on the simulation of its RTL model. The data for the original format of bytecodes in PicoJava II were estimated by totalling all the clock cycles consumed by the relevant bytecodes in software traps, and the data for the quick format of bytecodes were taken directly from the data sheet of PicoJava II.

Table 7: Distribution of all operations

Operation type	Percentage
Instructions that push a constant onto the stack	8.38
Instructions that load a local variable onto the stack	44.49
Instructions that store a value from the stack into local variable	7.87
Stack operations	2.02
Integer, floating-point and logic operations	11.10
Type conversion	0.65
Control flow	10.65
Object access, method invocation and revocation	14.72
Others	0.12

Using N_i, W_i to represent the clock cycles needed for execution of a bytecode and its distribution weighting, respectively, and N to denote the average clock cycles for each bytecode execution, we have

$$N = \sum_{i=1}^{M} (N_i \times W_i) \quad (1)$$

where M is the number of bytecodes.

We can estimate the overall performance gain for PicoJava II and jHISC by using (1). The details are shown in Table 10. In the table, the results for quick format are calculated by assuming all the object-oriented-related bytecodes are executed by their quick formats; in other words, their distribution weightings are the same as those of their original formats.

Executing all the object-oriented-related bytecodes in the form of their quick formats is the optimal case, so we find that when the two systems have the same clock frequency, the performance of executing a bytecode is speeded up from 1 (4.2/2.1 − 1) to 7.4 (17.6/2.1 − 1) times and the performance of executing an object-oriented-related bytecode is improved by 0.91 (13.6/7.1 − 1) to 13.2 (100.9/7.1 − 1) times by jHISC. When an object-oriented-related bytecode is executed by software trap in PicoJava II, most of the time is spent in preparing for stack operations, and locating and checking the related fields of object in sequence. However, in jHISC, the related fields of object are encapsulated into its context and accessed in parallel due to their constant offset to the object header.

Table 8: Distribution of some object-oriented-related bytecodes in the profiled benchmarks

Bytecode	Percentage in all operations	Percentage in object-oriented operations
getstatic	1.10	4.63
putstatic	2.11	17.14
getfield	0.37	4.66
putfield	0.02	0.11
invokevirtual	3.92	21.00
invokestatic	0.32	2.65
invokespecial	0.84	7.62
invokeinterface	0.21	1.93
ireturn	1.62	14.55
return	0.98	7.28
areturn	1.21	7.09
checkcast	0.47	3.53
instanceof	0.13	0.84
new	0.157	0.89
newarray	0.05	0.30
anewarray	0.03	0.14
multianewarray	0.00	0.00

Table 9: Cycles needed by some object and array manipulation bytecodes in PicoJava II and jHISC

Bytecodes in PicoJava II	Cycles		Instruction in jHISC	Cycles
	Original format	Quick format		
getfield	114	4	gfld	6
			gifld	2
putfield	130	4	pfld	6
			pifld	2
getstatic	103	3	gsfld	6
putstatic	103	3	psfld	6
invokestatic	86	11	ivkclass	9
invokevirtual	195	15	ivkinstance	9
			ivkinternal	5
invokespecial	208	17	ivkinstance	9
invokeinterface	203	184		
checkcast	97	6	checkcast	3
instanceof	100	7	instanceof	4
ireturn				
return				
areturn		8	oo_rvk	5[a]
return				
new	109	99	new	100
newarray		221	new	100
anewarray	210	201	new	100

[a]The clock cycles taken by the method revocation instruction 'oo_rvk' are 3, 5, 7, respectively, in the case of returning from the bytecodes 'ivkinternal', 'ivkclass' and 'ivkinstance'. The value in the table is the average value

Table 10: Overall performance estimation

	Cycles
Average clock cycles for executing a bytecode in PicoJava II (original format)	17.6
Average clock cycles for executing a bytecode in PicoJava II (quick format)	4.2
Average clock cycles for executing a bytecode in jHISC	2.1
Average clock cycles for executing an object-oriented-related bytecode in PicoJava II (original format)	100.9
Average clock cycles for executing an object-oriented-related bytecode in PicoJava II (quick format)	13.6
Average clock cycles for executing an object-oriented-related bytecode in jHISC	7.1

Concurrently, the field and access checking are performed by hardware during access. All these lead to performance improvement in jHISC. Compared with the quick format scheme implemented by microcode in PicoJava II, the execution performance is improved by 100%.

5 Conclusion

jHISC provides an efficient and secure solution for Java applications. First, both the hardware implementation of complex object-oriented-related bytecodes and parallel access of fields inside object contribute to overall performance improvement because it uses a hardware-readable data structure to represent object and each field of object is stored with a constant offset to the object header. Secondly, built-in bound checking forbids unauthorised object accesses to enhance system security because all data are encapsulated into objects and no operations access memory directly. Moreover, in order to speed up the executions of object-oriented-related bytecodes, we can add a method cache to store the direct reference addresses of objects after they are resolved. The well-defined hardware-readable data structure can also be applied to other object-oriented programming languages such as C# and C++.

6 Acknowledgment

This work is partially supported by the City University of Hong Kong under Strategic Research Grant 7001548.

7 References

1 Gosling, J., Joy, B., and Steele, G.: 'The Java™ language specification' (Addison-Wesley, 1996)
2 El-Kharashi, M.W., and Elguibaly, F.: 'Java microprocessors: Computer architecture implications'. IEEE Pacific Rim Conf. on Communications, Computers and Signal Processing, August 1997, Vol. 1, pp. 277–280
3 Grand, M.: 'Java language reference' (O'Reilly, 1997)
4 Sun Microsystems: 'White paper on KVM and the connected, limited device configuration'. Sun Microsystems White Paper, May 2000
5 Mulchandani, D.: 'Java for embedded systems', *IEEE Internet Comput.*, June 1998, **2**, (3), pp. 30–39
6 O'Connor, J.M., and Tremblay, M.: 'PicoJava I: The Java virtual machine in hardware', *IEEE Micro*, March 1997, **17**, (2), pp. 45–53
7 Radhakrishnan, R., Rubio, J., and John, L.K.: 'Characterization of Java applications at the bytecode level and at UltraSPARC-II machine code level'. Int. Conf. on Computer Design, October 1999, pp. 281–284
8 Radhakrishnan, R., Vijaykrishnan, N., John, L.K., Sivasubramaniam, A., Rubio, J., and Sabarinathan, J.: 'Java runtime systems: characterization and architectural implications', *IEEE Trans. Comput.*, February 2001, **50**, (2), pp. 131–146
9 McGhan, H., and O'Connor, J.M.: 'PicoJava: a direct execution engine for Java bytecode', *Computer*, October 1998, **31**, (10), pp. 22–30

10 Sun Microsystems: 'PicoJava-II: Java processor core'. Sun Microsystems data sheet, April 1998
11 aJile Systems, Inc.: 'aJ-100 real-time low power Java™ processor, *aJ-100*™ reference manual'. Version 2.1, December 2001
12 ARM: 'Jazelle technology for Java application'. ARM data sheet, May 2001
13 NAZOMI Communications Inc.: 'JA108 – multimedia application processor (product brief)', 2003
14 Aurora VLSI Inc.: 'AU-J2000: super high performance Java processor core (data sheet)'. Aurora VLSI Inc., 2000
15 NanoAmp Solutions, Inc.: 'The MOCA-J™ accelerator: performance boosting solutions for J2ME software'. White Paper on MOCA-J™, 2003
16 Radhakrishnan, R., Bhargava, R., and John, L.K.: 'Improving Java performance using hardware translation'. ACM Int. Conf. on Supercomputing, June 2001, pp. 427–439
17 Kent, K.B., and Serra, M.: 'Hardware/software co-design of a Java virtual machine'. IEEE Int. Workshop on Rapid Systems Prototyping, June 2000, pp. 66–71
18 Lattanzi, E., Gayasen, A., Kandemir, M., Narayanan, V., Benini, L., and Bogliolo, A. 'Improving Java performance using dynamic method migration on FPGAs'. 18th Int. Parallel and Distributed Processing Symp., April 2004, pp. 134–141
19 Parnis, J., and Lee, G.: 'Exploiting FPGA concurrency to enhance JVM performance'. Australasian Computer Science Conf., January 2004, pp. 223–232
20 Ha, Y.J., Hipik, R., Vernalde, S., Verkest, D, Engels, M., Lauwereins, R., and Man, H.D.: 'Adding hardware support to the HotSpot virtual machine for domain specific applications', *Lect. Notes Comput. Sci.*, 2002, **2438**, pp. 1135–1138
21 Cardoso, J.M.P., and Neto, H.C.: 'Macro-based hardware compilation of Java™ bytecodes into a dynamic reconfigurable computing system'. IEEE Symp. on Field-Programmable Custom Computing Machines, April 1999, pp. 2–11
22 Zheng Liang, Plosila, J., and Sere, K.: 'Asynchronous Java accelerator for embedded Java virtual machine'. IEEE CAS Symp. on Emerging Technologies: Mobile and Wireless Communication, May 2004, pp. 253–256
23 Vijaykrishnan, N., and Ranganathan, N.: 'Object-oriented architecture support for a Java processor'. 12th European Conf. on Object-Oriented Programming, 1998, pp. 330–354
24 Glossner, J., and Vassiliadis, S.: 'The Delft-Java engine: an introduction'. Third Int. Euro-Par Conf. on Parallel Processing, August 1997, pp. 766–770
25 Glossner, J., and Vassiliadis, S.: 'Delft-Java link translation buffer'. EuroMicro 24th Conf., August 1998, pp. 221–228
26 Schoeberl, M.: 'JOP: a Java optimized processor', *Lect. Notes Comput. Sci.*, 2003, **2889**, pp. 346–359
27 Kent, K.B., and Serra, M.: 'Hardware architecture for Java in a hardware/software co-design of the virtual machine'. Euromicro Symp. on Digital System Design, September 2002, pp. 20–27
28 Kent, K.B., Ma, H., and Serra, M.: 'Rapid prototyping of a co-design Java virtual machine'. IEEE Int. Workshop on Rapid System Prototyping, June 2004, pp. 164–171
29 El-Kharashi, M.W., Pfrimmer, J., Li, K.F., and Gebali, F.: 'A design space analysis of Java processors'. IEEE Pacific Rim Conf. on Communications, Computers and Signal Processing, August 2003, pp. 28–30
30 Vijaykrishnan, N., and Ranganathan, N.: 'Supporting object accesses in a Java processor', *IEE Proc., Comput. Digit. Tech.*, 2000, **147**, (6), pp. 435–443
31 Mok, P.L., Fong, A.S., and Hau, K.W.: 'Object-oriented processor requirements with instruction analysis of Java programs', *Comput. Archit. News*, 2003, **31**, (5), pp. 10–15
32 Mok, P.L., Li, C.L., and Fong, A.S.: 'Method manipulation in an object-oriented processor', *Comput. Archit. News*, 2003, **31**, (4), pp. 18–25
33 Fong, A.S.: 'A computer architecture with access control and cache option tags on individual instruction operands', *Comput. Archit. News*, 2003, **31**, (3), pp. 1–5
34 Fong, A.S.: 'HISC: a high-level instruction set computer'. 7th European Simulation Symp., October 1995, pp. 406–410
35 Lindholm, T., and Yellin, F.: 'The JAVA virtual machine specification' (Addison Wesley, 1999, 2nd edn.)
36 Chang, L.C., Ton, L.R., Kao, M.F., and Chung, C.P.: 'Stack operations folding in Java processors', *IEE Proc., Comput. Digit. Tech.*, 1998, **145**, (5), pp. 333–340
37 Ton, L.R., Chang, L.C., and Chung, C.P.: 'An analytical POC stack operations folding for continuous and discontinuous Java bytecodes', *J. Syst. Archit.*, 2002, **48**, pp. 1–16
38 Ton, L.R., Chang, L.C., Shann, J.J., and Chung, C.P.: 'Design of an optimal folding mechanism for Java processors', *Microproc. Microsyst.*, 2002, **26**, pp. 341–352
39 Kim, A., and Chang, M.: 'Java bytecode optimization with advanced instruction folding mechanism', *Lect. Notes Comput. Sci.*, 2000, **1940**, pp. 268–275
40 Kim, A., and Chang, M.: 'Advanced POC model-based Java instruction folding mechanism'. 26th EUROMICRO Conf., September 2000, pp. 332–338
41 Standard Performance Evaluation Corporation, http://www.specbench.org/osg/jvm98
42 Armond Avanes, http://xbrowser.sourceforge.net