A Performance Analysis of an Object-Oriented Processor

Tan YiYu, Lo Kai Man and Fong Anthony S.
Department of Electronic Engineering
City University of Hong Kong
Tat Chee Ave, Kowloon, Hong Kong

Abstract
Java is relatively a new object-oriented programming (OOP) language. It has the advantages of object-oriented languages such as encapsulation, polymorphism, dynamic binding and inheritance. Consequently programs developed by Java are more reliable and secure and Java is widely used in embedded system and the other application areas. However, as traditional computer architectures RISC and CISC [4] do not provide much hardware support for OOP, their performance for OOP is notoriously poor. In this paper, a performance analysis of an object-oriented processor jHISC is presented, which supports most object oriented instructions in hardware level. By comparing the cycle count of related instructions with PicoJava II, it can speed up the execution of object-oriented related instructions from 700% to 1700% and overall performance from 390% to 630%.

Keywords: Object-oriented programming, Java, operand descriptor, bytecode, performance.

1. Introduction
Due to Java’s OOP characteristics, Java has become popular in mobile devices, such as mobile phone, palm device, PDA etc. However, Java’s performance is a critical problem since the processing units in the mobile devices do not execute OOP operations well. Consequently many processor development groups are trying to develop Java enhancements in silicon. Unfortunately, these Java enhanced processors fail to achieve good OOP performance because of their traditional von Neumann computing architecture [8].

In most cases, OOP features are not supported well in hardware but instead by either microcode or software traps. A summary for key features of current famous Java processors is shown in table 1 [1][2][4][7][13][14][15].

To implement OOP features in hardware level, jHISC is proposed by Mok [11]. This paper describes performance analysis of jHISC. The rest of this paper is structured as follows. Section 2 is an overview to jHISC and PicoJava. In section 3, the performance comparison and analysis related object-oriented operations between jHISC and PicoJava is described. Finally, conclusion is presented in section 4.

Table 1. The key features of current famous Java processors

<table>
<thead>
<tr>
<th>Processor</th>
<th>ARM (v7, 9 and 11)</th>
<th>highfoot</th>
<th>ALL-J2000</th>
<th>J200</th>
<th>PicoJava II</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td></td>
<td>32-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic Architecture</td>
<td>Stack Machine on RISC Core</td>
<td></td>
<td>Harvard Machine + RISC engine</td>
<td>Stack Machine + Translator</td>
<td>Stack Machine + Translator</td>
</tr>
<tr>
<td>Bytecodes Support</td>
<td></td>
<td>Translates to native instruction</td>
<td>Directly</td>
<td>Directly</td>
<td>Directly</td>
</tr>
<tr>
<td>Instructions Support</td>
<td>Microcode Software Trap</td>
<td>Software Trap</td>
<td>Software Trap</td>
<td>Software Trap</td>
<td>Software Trap</td>
</tr>
</tbody>
</table>

2. jHISC and PicoJava overview

2.1. jHISC - An Object-Oriented Processor
jHISC is a 32-bit object-oriented processor which mainly targets to Java [11]. It is based on High Level Instruction Set Computer (HISC) architecture proposed by Fong [5][6] which provides hardware readable data types to represent objects called Operanad Descriptor...
(OD) [6]. jHISC provides hardware support to object-oriented programming. It is targeted to J2ME KVM. Floating point unit is not implemented in jHISC. The markets it focuses on are mobile devices, such as cell phone, PDAs and other embedded systems.

The structure of Operand Descriptor (OD) is shown in figure 1. The descriptor length is 32-bit with five fields. Each OD describes a property in an object and a set of ODs describes an object.

![Figure 1. Operand descriptor structure](image1.png)

![Figure 2. Operand descriptor structure](image2.png)

![Figure 3. Operand descriptor structure](image3.png)

Every object associates with an object header. An additional object type specified header is defined following the object header. The object type specified headers are included class context header, instance context header and method context header.

![Figure 4. Object header](image4.png)

The instruction set of jHISC is based on MIP32, with different memory-register data transfer instructions. These instructions allow program to access memory directly and potentially cause the security holes of memory. In addition, jHISC provides object-oriented manipulation instructions to handle the object-oriented related processing [11].

![Figure 5. jHISC architecture block diagram](image5.png)

The instructions of jHISC are optimized for Java. Java bytecodes are 100% supported. Excluding 64-bit data type related operations, jHISC executes 94% of Bytecodes in hardware level and handles 6% of them by software traps. The details are shown in table 2. In addition, Java bytecodes can be converted into jHISC instructions from 1-to-1 to N-to-1.

<table>
<thead>
<tr>
<th>Table 2. The instructions supported by jHISC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Bytecode</td>
</tr>
<tr>
<td>Number of Bytecodes after the instructions for floating-point operations are excluded</td>
</tr>
<tr>
<td>Number of Bytecodes supported in hardware</td>
</tr>
<tr>
<td>Number of Bytecodes done by software traps</td>
</tr>
<tr>
<td>Number of object-oriented related Bytecodes</td>
</tr>
<tr>
<td>Number of object-oriented related Bytecodes supported in hardware</td>
</tr>
<tr>
<td>Percentage of Bytecodes supported in hardware</td>
</tr>
<tr>
<td>Percentage of object-oriented related Bytecodes supported in hardware</td>
</tr>
</tbody>
</table>

2.2. PicoJava II

PicoJava II is a 32-bit stack based Java processor purposed by Sun Microsystems. The target Java
specification is J2SE and its targeted markets are consume and embedded products such as Mobile phones, digital set-top boxes, pervasive computing, PDA, portable games devices etc. The original plan for J2SE is to serve as a hardware Java Virtual Machine with the security and object-oriented features but not the performance penalty due to the software JVM [9].

In PicoJava II, only the most common instructions that most directly impact program execution are implemented in hardware. Some complicated but performance critical instructions are implemented by microcode or software trap, such as object-oriented and array related instructions. Thus the hardware design is simple since the complex instructions are not implemented by hardware. Additionally, in JVM, an instruction execution usually needs to copy data from local variable to the top of stack to precede an instruction that uses these data. The PicoJava folds these two instructions into one by accessing local variable directly. These folding operations can speed up Java bytecode executions.

Since PicoJava II core is released, it has been embedded into some Java processors, such as MB86799, which is a Java processor proposed by Fujitsu [7][13]. However, it has the following disadvantages:

- **Incompatibility to the existing systems.** PicoJava II is a stack machine [8] and uses Bytecodes as its own native instructions. It is not compatible to the popular RISC instruction sets. Therefore the system software development cost increases.

- **The performance improvement of the Java program execution is not substantial.** In PicoJava II, object-oriented related operations are handled by software traps and microcode, including complex arithmetic computing on microcode [12][13]. The performance gain of the related operations is thus quite limited.

- **Inflexible system architecture.** As PicoJava II targets to act as a hardware JVM to execute Java program, the architecture of PicoJava II is exactly the same as JVM [9]. PicoJava II is developed as a basic stack machine. The simple stack operations cannot provide complex computing efficiently.

- **Large core size.** In order to support object-oriented operations, PicoJava II uses a lot of ROM to store microcode and software traps object-oriented operations handlers and complex arithmetic computing operations [12][13]. Consequently, the silicon size of the core becomes large.

**Large power consumption.** For the same reason of large core size, the high usage of ROM introduces relatively high power consumption in PicoJava II. It is a critical problem for mobile devices and other power sensitive applications.

3. Performance Comparison and Analysis

As jHISC is still under research in FPGA level, in order to analyze the performance, PicoJava II is used for this objective since it is an open source and a full functional Java processor.

From a Bytecodes distribution analysis, average of 15% of Bytecodes is object-oriented related [10]. In traditional software JVM, a load or store operation require a few CPU cycles while an object-oriented related operation could use more than 100 cycles.

For variable load/store instructions, the fastest one can be executed in 1 cycle in PicoJava II. However, for object-oriented related operations, significant large of CPU cycles are needed in PicoJava II. Clearly, object-oriented related operations are the most costly operation in the program execution. Since jHISC is still under development in FPGA, the exact instruction times cannot be determined.

To compare the performance of executions of the instructions between the two systems, the numbers of cycles used by several related instructions of both systems are counted. During counting, we assume all the objects are resolved and they are either public or private [9]. Moreover, for PicoJava II, we assume all data in cache is 100% hit, and all instructions included into microcode or software trap in PicoJava II consume only 1 cycle and the time used by software traps (includes exception handling, context switching, etc.) for OOP instructions handler is not included. The comparison results are shown in table 3 in detail

According to the research of Bytecodes distribution analysis in Xbrowser, Java2D Demo and SPEC JVM98, the object-oriented operations distribution ratio of above instructions are calculated [10]. The results demonstrate that jHISC speeds up the execution of object-oriented related operations significantly comparing with PicoJava II. By assuming the overall CPU time used by object-oriented operations is directly proportional to the Bytecodes distribution and let B be the total number Bytecodes, then,

$$\text{the total no. of cycle for the Bytecodes executions} = \text{Sum}(\% \text{ of distribution in Bytecode } X \text{ no. of cycle used by the Bytecode})$$
Assuming all non-OO operations executed in PicoJava II and jHISC are consumed 1 cycle only, the overall performance gain can be estimated and shown in Table 4.

### Table 3. The number of cycles cost by some OO related bytecodes

<table>
<thead>
<tr>
<th>Instruction Function</th>
<th>Instruction Name in PicoJava II</th>
<th>No. of cycle with access modifier checking</th>
<th>Instruction Name in jHISC</th>
<th>No. of cycle(100 % cache hit)</th>
<th>No. of cycle(100 % cache miss)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get field</td>
<td>getfield-internal</td>
<td>107-</td>
<td>getfield</td>
<td>6-</td>
<td>18-</td>
</tr>
<tr>
<td>Put field</td>
<td>putfield-instance</td>
<td>98-</td>
<td>putfield</td>
<td>6-</td>
<td>18-</td>
</tr>
<tr>
<td>Get static field</td>
<td>getstatic-internal</td>
<td>80-</td>
<td>getstatic</td>
<td>6-</td>
<td>18-</td>
</tr>
<tr>
<td>Put static field</td>
<td>putstatic-internal</td>
<td>75-</td>
<td>putstatic</td>
<td>6-</td>
<td>18-</td>
</tr>
<tr>
<td>Invoke static</td>
<td>invokevirtual-internal</td>
<td>58-</td>
<td>invokevirtual</td>
<td>6-</td>
<td>27-</td>
</tr>
<tr>
<td>Invoke instance</td>
<td>invokevirtual</td>
<td>150-</td>
<td>invokevirtual</td>
<td>6-</td>
<td>27-</td>
</tr>
</tbody>
</table>

### Table 4. Overall performance estimation

<table>
<thead>
<tr>
<th></th>
<th>No. of cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>The average no. of cycle for the Bytecodes executions in picoJava II</td>
<td>10.20-</td>
</tr>
<tr>
<td>The average no. of cycle for the Bytecodes executions in jHISC in worst case</td>
<td>2.64-</td>
</tr>
<tr>
<td>The average no. of cycle for the Bytecodes executions in jHISC in best case</td>
<td>1.62-</td>
</tr>
</tbody>
</table>

From these figures, we estimate that the overall performance can be increased from 390% to up to 630%. As we have assumed that all data is 100% cache hit in PicoJava II and the time CPU cycle used by object-oriented operations is directly proportional to its distribution, the performance gain may be further increases in actual cases.

### 4. Conclusion

In the foreseeable future, Object-Oriented Programming will dominate the software development field. Java is one of the popular OOP languages and its market is growing fast. However, the poor performance of Java is a critical problem need to be solved. jHISC provides a solution to speed up the Java program execution with the security benefit of OOP and program compatibility to existing systems. The well defined hardware readable OOP data structure of jHISC can also be applied to other OOP languages such as C#.NET and C++.

### 5. Acknowledgement

This work is partially supported by the City University of Hong Kong, Strategic Research Grant 7001847.

### 6. References

September 2003, pp. 18-25.


