A Novel Just-In-Time Compiler on an Embedded Object-Oriented Processor

Yau Chi Hang, Tan Yiyu, Fong Anthony S.
Department of Electronic Engineering, City University of Hong Kong
Tat Chee Avenue, Hong Kong
jerry.yau@student.cityu.edu.hk

Abstract

As software trend is moving to Object-Oriented Programming (OOP), a novel object processor for running OOP offers an opportunity to enhance the system security, performance and to provide an alternative for a Virtual Machine to suit OOP. We select Java as our targeting language due to its security, re-usability and portability. However, Java’s severe restrictions of its real-time behavior become the barrier on embedded devices.

Our solution is to build a Java Just-In-Time compiler on jHISC, a descriptor processor targeting on Java, in order to excel in running Java language. Through this hardware and software co-development, Java can run as a native language in embedded devices similar to the performance of running C language on RISC or CISC and penetrate all Java’s features to increase overall system security and performance.

Keywords: Object, Java, Just-in-time (JIT) compiler, access control, operand descriptor.

1. Introduction

Java is a successful OOP example, popular in Internet, mobile and other applications. It inherits all advantages of OOP and has enhancement in security, portability and re-usability. In addition, with the small size of Java bytecode, and the platform independent characteristics, these factors provide a standard for the mobile technology to grow.

Despite Java’s popularity, it still has not shown great success on the embedded Java market. It is mainly due to its severe restrictions of its real-time behavior (Figure 1 shows the traditional running process of Java class files with its overhead):

**Extra access control checking** – A lots of access control checking on the modifiers are required on JVM layer in order to obey the rules of protection in Java.

The modifiers are private, public, protected and package.

**Resolving overheads of objects** -- Great overheads on resolving the object and repeatedly referencing the JVM’s constant pools are inevitable beforehand to execute the calling objects, its fields or operations.

On the following sections, we will firstly introduce jHISC architecture on Section 2, then follows with our compiler’s operation flow on Section 3. Section 4 and 5 will discuss the compilation techniques use on our compiler. Finally, testing methodology and result will be shown on Section 6, follows with a conclusion on Section 7.

Figure 1. Severe restrictions in real-time behavior on JVM and security problem due to the OS written in language other than Java

2. The Overview of jHISC architecture

HISC is a general-purpose computer which extends conventional descriptor computer architecture to include hardware support for system attributes [2]. jHISC is a computer architecture design using High
Level Instruction-Set Computer (HISC) concept to provide hardware support on object-oriented computing targeting for Java’s object-manipulation. Different kinds of object in Java such as instance and class are mapped to the hardware readable object structures in jHISC. An object is represented by an Operand Descriptor Table (ODT). Each entry on the ODT corresponds to a persistent variable or method. Besides the Operand Descriptor (OD) for every instance variable or method in each object context of the ODT, object ID, class hierarchy are presented in the specific entries of the ODT for certain object operations such as class hierarchy checking, and casting. In addition, basic object operations in jHISC are defined through object instructions. They include ivk (method invocation), rvk (return from method invocation), gfld (get field), and pfld (put field). The details of these instructions are described in [3].

jHISC is now on version 3 with code name Larva. It is targeted for small mobile devices. Actually, jHISC is a RISC based core with some object-oriented features enhancement. jHISC provides Object-Oriented manipulation instructions to handle the object-oriented related processing. There are totally seven groups of instructions in jHISC v3, they are: Logical, Arithmetic, Branching, Array, Object-Oriented, Data Manipulation and Miscellaneous.

On our architecture, all 202 JVM opcodes [4][5] on stack operations will be correctly compiled into the 57 three-addressing jHISC v3 Instruction. As a result, jHISC provides an alternated solution to support object-oriented programming without losing all compatibility with existing computer architectures.

3. Compiler’s Components and Operation Flow

The modules of the compiler and the interaction between components are shown on Figure 2. The followings are the explanations of the flow:
1. Select the Java class file to compile and run in jHISC. Then the Class Loader will be invocated to load in the required class files
2. Through the Bytecode Reader to read in the bytecodes.
3. The selected class file and its corresponding parent classes and referencing classes will also be loaded recursively.
4. The selected class files and its corresponding parent classes will be loaded into the Core Structure through the class loader where the information on each context will be extracted and the corresponding structures, the Operand Descriptor Tables (ODTs) will be built inside the Core Structure. All the linking between classes will also be performed in the Core Structure.
5. During the extracting and building of the class structures through the hierarchical, each method will be passed into the Method Compiler to compile each method.
6. Through the Optimizer to reduce the compiled code length. The optimization is performed using the instruction folding techniques.
7. Code Generator will be invoked by the Method Compiler to emit the jHISC machine code through the assistance of the Optimizer to perform optimization.
8. jHISC machine code will be generated and stored back into the Core Structure for further processing and display.
9. Memory Allocator will be invoked by the Core Structure to allocate memory address for each tables and code spaces and
10. finally built the memory map and the linker is invoked to load in the executable memory map.
11. The executable memory map will be load into the FPGA board and execute through the hardware and the final result will dump back into the console of the linker for checking and display.

Figure 2. General Components and Modules of the compiler and the program flow
4. Method Compilation Mechanism

Method compiler compiles the method codes into machine code. It performs the compilation of JVM method’s bytecode into jHISC machine code which will be stored in the Method Code Space (MCS) inside the Core Structure.

In Java bytecode, all opcodes are based on a stack machine, so on our compiler’s point of view, it has to determine the number of words required by the method frame in order to map precisely on the jHISC register files. In the jHISC hardware, there is no stack operation. The size of the Local Variable Frames (LVFs) that are to be allocated by the hardware requires the compiler to calculate and place all the size parameters on the hardware-readable headers in order to let the hardware know how to allocate the LVF during the method invocation.

Single Index Resolution, Double Index Resolution and the Triple Index Resolution are the main resolution burdens in JVM’s Constant Pool Structure. The details of these resolutions are described in [6]. The sequential searching on the Constant Pool for resolution produces a great burden in JVM. In our architecture, the Fast Indexing Resolution Architecture is introduced in order to reduce the resolution burdens during initial running time.

Out of the 202 standard bytecode instructions, 53 of them refer to items in the constant pool. The purpose of fast index resolution is to replace these 53 symbolic references with direct references to memory or indexing to relative Operand Descriptor Tables (ODTs) entries during initial loading time. It finally forms the fast indexing resolution architecture instead of an offset in the JVM system. Figure 3 and Figure 4 shows the resolution burden on constant pool with triple index resolution. They are the reassembling result of a sample class file, TestBubbleSort.class file (original in binary format), using “javap –verbose” -- The Java Class File Disassembler. While Figure 5 shows the corresponding resolution architecture built on our system.

![Figure 3. The reassemble of the TestBubbleSort.class file using javap](image)

On the contrary, Fast Indexing Resolution Architecture built on initial loading time eliminates all the sequential searching, shown in Figure 5. Firstly, the invocation on IO_Class.output() inside the Method Code Space (MS) of welcomeMessage() “oo.invokeclass CODT #1” is an index to the Class Operand Descriptor Table (CODT)’s index #1. This index #1 entry maintains the class #0 and property #0 which are the smart indexing that will direct to the IO_Class’s index #0 inside its Class Property Descriptor Table (CPDT) through the direct address inside the Class Data Space (CDS). Eventually, the IO_Class.output() have been reached and can be invoked through the hardware invocation instruction.
straightly. As a result, all the sequential searches inside the constant pool are eliminated.

Figure 5. An example of Fast indexing Architecture maintains inside the Core Structure

5. Optimization

An Optimizer module is designed and implement in the compiler for optimizing the code length. This optimizer follows the instruction folding schemes that are introduced in [7], [8] and [9].

Instruction Folding is a way of omitting the unnecessary loads or write back operations to the stack. It has the ability to detect some related contiguous instructions in the instruction flow of a stack machine, and execute them collectively in some way like a single, compound instruction [7]. The following shows an example of optimization in the optimizer on Type II (PPPOCPOC) sequence which was defined in [9].

As shown on Table 1, without optimization, compiled code length equals to 9. After performing the optimization, all the stack operations are eliminated and there is no register used on stack operation and the compiled code length is reduced to 3.

Summary of how the instruction folding has been applied are as follows.

The corresponding register allocation for the jHISC machine:

- R0–R8: local variable frame – registers for storing local variable declared in method
- R9–Ra: temporary frame – registers for temporary storage for compatibility of JVM instructions e.g. swap operation
- Rb–Rd: operation frame – registers allocated for stack operations

As shown in Table 1, due to the dependency problem, we must first fold group B before folding group A.

Table 1. Compare the compilation result between with optimization and without optimization

<table>
<thead>
<tr>
<th>Original Bytecode</th>
<th>Compiled jHISC Assembly code (without optimization)</th>
<th>Compiled jHISC Assembly code (with optimization)</th>
</tr>
</thead>
<tbody>
<tr>
<td>iload 3</td>
<td>data.move R3, Rb</td>
<td>arith.add R7, R3, R8</td>
</tr>
<tr>
<td>iload 4</td>
<td>data.move R4, Rc</td>
<td>arith.add R6, R4, R3</td>
</tr>
<tr>
<td>iload 6</td>
<td>data.move R6, Rd</td>
<td>arith.add Rb, Rb, Rb</td>
</tr>
<tr>
<td>istore 3</td>
<td>data.move Rc, R3</td>
<td>data.move Rb, R8</td>
</tr>
<tr>
<td>istore 7</td>
<td>data.move R7, Rc</td>
<td>arith.add Rd, Rc, Rb</td>
</tr>
<tr>
<td>iadd</td>
<td>arith.add Rb, Rc, Rb</td>
<td>arith.add Rd, Rc, Rb</td>
</tr>
<tr>
<td>return</td>
<td>oo.rvk</td>
<td>oo.rvk</td>
</tr>
</tbody>
</table>

*Bolded registers are the stack operation registers, the total stack operations without optimization is 8

6. Testing Methodology and Result

The jHISC version 3’s architecture has already been implemented on a Xilinx Virtex XCV800 FPGA. With the novel Just-in-time compiler for jHISC, compatibility of Java program can be assured and tested. With some I/O display operations that have been implemented, the compiled program can be loaded into the hardware and the required result can be dumped back for display. The linker has also been built to link and demonstrate the executable result that was run on our architecture. As shown in Figure 6, the executable memory map is load through the linker and calling the DimeJavaAPI to load into the FPGA board.

As you can see in Figure 7, some testing Java programs are successfully compiled and run in the jHISC processor and the results are dumped back through the linker to the Console. The initial disordering of ASCII characters “oelHl” is bubble-sorted. The sorted result “Hello” is displayed in the
Console in ascending order. Besides some other testing Java programs such as a “selection sort” demo, a “To Upper Case” demo and a calculation demo have also been demonstrated.

7. Conclusion

This paper has introduced a new design of Just-In-Time compiler architecture on jHISC, an object-oriented descriptor computing targeting on Java for embedded computing. Some resolution burdens in JVM have been compensated through our Fast Indexing Resolution Architecture. Instruction Folding techniques have also been applied to optimize the compiled code length. Furthermore, the implementation of the optimization on the bytecode and memory has been clearly verified and analyzed with a series of Java programs. Eventually, Java can be run as a native language in the jHISC architecture and many Object-Oriented and Java’s features are supported from the hardware.

Acknowledgements

The work described in this paper was partially supported by a grant from City University of Hong Kong (Strategic Research Grant Project No. 7001548).

References