A Hardware/Software Co-design and Co-verification on a Novel Embedded Object-Oriented Processor

Yau Chi Hang, Tan Yi Yu, Mok Pak Lun, Yu Wing Shing, Fong Anthony S.

Department of Electronic Engineering, City University of Hong Kong
Tat Chee Avenue, Hong Kong
Anthony.Fong@cityu.edu.hk

Abstract. In the past, programming language are procedural, the design concept is based on the module and scope which are difficult to manage, but nowadays, the programming trend is Object-Oriented Programming (OOP), where objects are the key elements to build up application and the communications between different objects are through method invocation. A novel object-oriented processor offers an opportunity to enhance the system security, performance and provides a more effective way to manipulate OOP instead of using a software Virtual Machine. jHISC is a novel object-oriented processor which provides a natural way to map the concept of OOP into architectural level through the hardware object data structure. Our solution is to design secure hardware object data structures on a novel processor with Just-In-Time compilation for Java which then makes it possible to implement complex OO related bytecodes at hardware level and access some fields of object in parallel to improve the execution speed. It mainly targets J2ME and implements about 93% bytecodes and 83% OO related bytecodes in hardware directly.

1. Introduction

Computer hardware becomes quite mature, however, software trend is moving to Object-Oriented Programming (OOP) with advantages on reusable design and better security. In traditional systems, processor architecture like Complex Instruction Set Computer (CISC) or Reduced Instruction Set Computer (RISC) does not support object manipulation directly on hardware. To support object technology in nowadays system, there are mainly three categories, which are compiling objects into native (Fig. 1), developing software-based object virtual machine (Fig. 1) and developing an object-oriented processor (Fig. 2).

For the first approach, since traditional computer system does not support object-oriented programming, applications written in object-oriented programming languages like C++ are compiled into native instructions for execution. Application processes will be created and executed in their own spaces, and they are invisible from each other through the use of virtual memory system. Through the specific compiler, objects will be translated into subroutines, which are machine readable for direct execution. Compilation approach can fully optimize the generated codes, but object-oriented features, such as dynamicity behavior, will be removed. Modification of a single class requires the whole application to be re-compiled.

In the second approach, an object virtual machine application is built on top of the traditional operating system. From the view of operating system, the application is just a usual process like other applications. Protection of different processes is the same as in compilation approach and objects are manipulated on this software virtual machine. The advantages of this method are that without the modification of hardware platform and current system, object technology can be supported through software emulation. But the two layers of software, virtual machine and operating system, also introduce much overhead to the overall computing system and increase the memory footprint of the system.

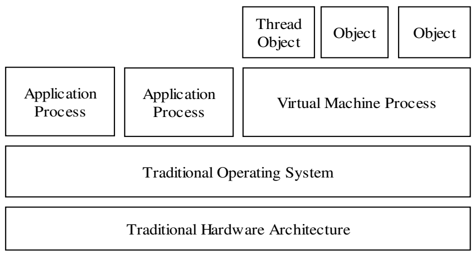


Fig. 1. System Architecture of Compilation Approach and Virtual Machine Approach

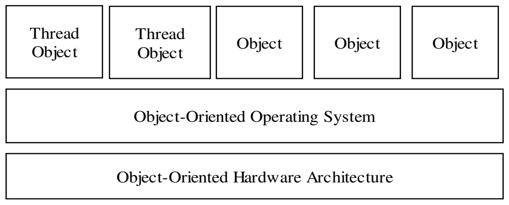

Thread Object
Thread Object
Object
Object
Object
Object-Oriented Operating System
Object-Oriented Hardware Architecture

Fig. 2. System Architecture of Object-Oriented Hardware Approach

In our solution, we will use the third approach where the objects are directly manipulated on an object-oriented processor, and an object-oriented operating system will be built to assist hardware in managing the heap (Fig 2). In this approach, protection of objects is governed by the object-oriented operating system with the protection features offered by the object-oriented processor. This approach can manipulate objects directly and the efficiency of the computing system is increased because no page table updating is required during context switching.

In the following sections, we will firstly discuss our secure hardware object architecture on section 2, their overall security features on section 3. Moreover, we will show the hardware/software co-verification methodology which verifies the Java compatibility on our novel architecture, through the Just-In-time compilation on Java bytecode and demonstrate some co-verification results on section 4. Finally, a conclusion will be presented on Section 5.

2. jHISC v3 Architecture with Object Representation

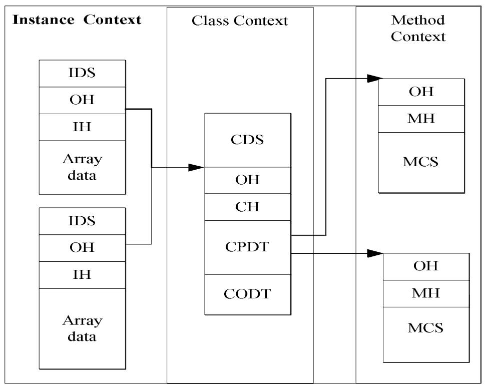


Fig. 3. Object Data Structure relationship

In Java, objects are created with some class templates, which provide information of the object instance, such as methods, fields, etc. Object instances of the same class share the same method codes, but each instance maintains its own copy of instance variables.

There are different kinds of objects in an object-oriented system. An object can have three contexts which are the instance context, the class context, and the method context. In jHISC v3, it maps the three contexts to architecture. The information about objects in different contexts is stored in different data structures which are defined by the architecture for object representation. Generally, all the objects share the same extendable object data structure. Array and thread objects, which require additional support by hardware, are represented differently.

Three headers are defined. They are object, class, and method headers. These headers store information for describing the object, class, and method itself. A header may associate with other data structures for describing different aspects of the context.

Descriptor tables store descriptors. Each descriptor contains information about an element within the class. Two kinds of descriptor table are defined. They are Class Operand Descriptor Table (CODT), and Class Property Descriptor Table (CPDT).

Data spaces provide spaces for the storage of data. There are three types of data space. They are the Class Data Space (CDS), the Instance Data Space (IDS) and the Local Variable Frame (LVF). Unlike CDS and IDS, LVF are dynamically created inside the register file upon method invocation and destroyed upon method revocation.

Inside the method context, Method Code Space (MCS) is defined to store the instructions. Fig 3 shows the relationships between different data structures. On the following of this section, we will show each of our hardware data structure in details.

2.1 Descriptor Format

jHISC uses a 32-bit operand descriptors to describe properties own by a class or resources accessed by the class. Descriptor is a data structure which stores the information about a variable, reference or any data structures. In jHISC v3, a single descriptor format is defined, which combined both fields in Class Operand Descriptor (COD) and Class Property Descriptor (CPD). The descriptor length is 32-bit, with five fields defined according to the specification of Java virtual machine. These fields are, Address Field, Access Modifier, Type Field, Static Flag, Read Only Flag, and Resolved Flag.

2.2 Instruction Set

jHISC is a RISC based core with some object-oriented features enhancement processor. Its Instruction length is 32-bit and supports up to three operands. Each instruction uses 8-bit opcode for defining instruction operation and references data in current LVF with 7 bits, addressing up to 128 registers. In addition, jHISC provides object-oriented manipulation instructions to handle the object-oriented related processing.

In traditional computers, memory-register data transfer instructions allow program to access memory directly which may result in security problems. In jHISC, all data are encapsulated into objects, and each object associates with a pair of memory boundaries (see Section 3). A program needs to pass the bound control checks before it accesses the data and out-of-bound accesses are prohibited. Two instructions "gifld" and "pifld" are introduced in our architecture to perform data transfer operations between memory and register with rigid memory access checking.

There are totally seven groups of instruction defined in jHISC v3, they are: logical instructions, arithmetic instructions, branching instructions, array Instructions, Object-Oriented (OO) instructions and data manipulation instructions. Excluding the float-point and 64-bit operation instructions, jHISC implements 93% bytecodes and 83% OO related bytecodes in hardware directly (Table 1). The rests are executed through software traps, such as "new", "newarray".

Table 1. The bytecodes supported by jHISC

Number of bytecodes	226
Number of bytecodes excluding the float-point and 64-bit operations	140
Number of bytecodes supported by the hardware directly	130
Number of bytecodes done by the software traps	10
Number of bytecodes for OO operations	40
Number of OO bytecodes supported by the hardware directly	33
Percentage of bytecodes supported by the hardware directly	93%
Percentage OO bytecodes supported by the hardware directly	83%

3. Security Features

Nowadays computing systems are multi-tasked. Multiple applications are executed on the same machine simultaneously. As more parties are involved in a computing system, it raises some concern of information privacy between different applications.

For object-oriented hardware approach, different objects share the same addressing space. Protection features for object access should be provided by the processor and used by the operating system in order to guarantee security between objects. Unlike traditional processor architecture, object-oriented processor has built-in protection logics to ensure security, in order to minimize execution errors.

Operations in a processor can be roughly categorized into two groups. The first group performs real calculation and data movement. They are used for generating the results of execution. Another group provides the support for control flow. Such operations include branching, subroutine calling and returning. They let a sequential execution machine to change the order of instruction execution according to the flow of program.

The switch of control from one domain to another provides some possible back holes for hacking programs and viruses. In order to secure the system from damages, checking routines are introduced directly to the architecture. In jHISC v3, the secure control transfer mechanism is based on object-oriented programming and the concept of object-oriented programming features like data encapsulation, messaging, etc, is used to restrict access of information from intrusion.

3.1 Strong type and Structural Memory Access

jHISC v3 is a strong typed system based on the definition of objects. Every data is associated with a type field, which indicates the data type. Data are encapsulated inside an object. All data types are specified upon compilation and they cannot be changed at run-time.

Direct memory accessing operations is prohibited in the jHISC v3 system. Instead of using direct memory accessing operations like load/store, jHISC v3 provides indirect memory accessing operations such as oo.getfield, oo.putfield, array.load and array.store.

3.2 Local Variable Frame and Object Data Structure Out of Bound Checking

Local variables are maintained inside Register Stack Engine (RSE). The RSE contains embedded logics for checking the out of boundary access of local variables. Any out of boundary access to local variables would cause the system a protection violation. This feature removes the potential bugs on the software or corrupted executable programs.

Pair of registers is maintained in the processor for object data structure access. A base register is used to locate the address of the object data and a size register records its size for out of boundary checking. Logics are embedded to do this kind of checking every time the data is accessed through the object-oriented instructions. Since every object has only bounded access to other objects, the sharing of address spaces in object-oriented hardware approach could be achieved with this protection features.

3.3 Intra-Method and Inter-Method Control Transfer

Intra-method control transfer instructions are defined for conditional statements in programming languages, such as if-then-else, while-loop, for-loop, etc. In jHISC v3, these instructions are based on branching instructions with an additional checking to avoid branching outside of the current method space. A pair of register is used to store the base address and size of current running method. The branch target is an offset to the base address (Fig. 4). The avoidance of branching outside the current method space enables all objects running in the same addressing space.

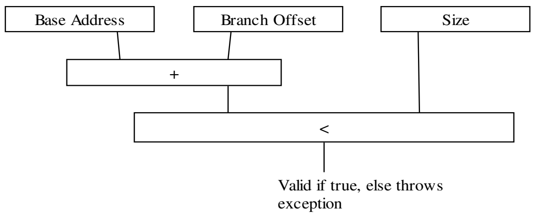


Fig. 4. Branch target checking in jHISC v3

Inter-method control transfer instructions include method invocation and method revocation. Method invocation is a procedure to call a method, while method revocation is a procedure to return from a method. They provide functionality for messaging between objects. Each method in jHISC v3 is defined to have only single entry point of calling. This feature prohibits the code bypassing in the same addressing space (Fig 5).

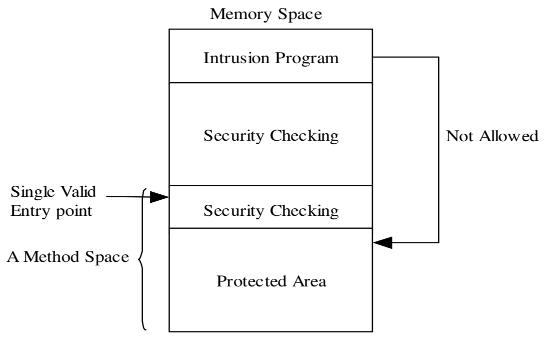


Fig. 5. Single entry point of Method in jHISC v3

4. Co-Verification Methodology and Result

The jHISC V3 with 4KB instruction cache and 8KB data cache has already been implemented on a Xilinx Virtex XCV800 FPGA to verify our concept. During implementation, the caches are generated by Xilinx CORE Generator and the whole system costs about 600K equivalent gates in FPGA.

In order to test the compatibility of running Java programs on our architecture, we built a software verification platform to compile all the JVM instructions into jHISC instructions and verify the functionality of the secure architecture through the Just-In-Time compilation of bytecode stream and run it on the FPGA. With the software testing implementation of Just-in-time compilation for jHISC, compatibility of Java program can be assured and tested. With some I/O display operations that have been implemented, the bytecode can be compiled and loaded into the FPGA board through the PCI interface and the required results can be dumped back for display. The linker has also been built to link and demonstrate the executable results that were obtained in our architecture. As shown in Fig. 6, the executable memory map is loaded into the FPGA board through the linker by calling the DimeJavaAPI.

All implemented instructions are tested to verify whether they are operated correctly. The related results are shown on Table 2. Besides, assembly program are written to infringe the system and test whether the jHISC core can detect such infringement and throws an exception to the system. The results are shown on Table 3.

Finally, some testing Java programs are compiled through our JIT compilation and loaded into the system for execution. With the verification of all the data structures and resolutions which are built through the compiler with GUI display (Fig. 7), the compiled code are successfully run in the jHISC core and the results are dumped back through the linker to the Console (see Fig. 8). Table 4 shows results of testing on some simple Java program.

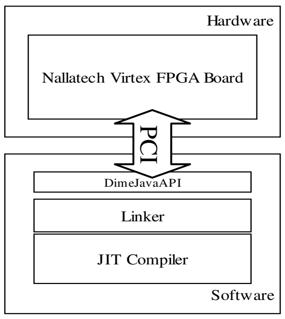


Fig. 6. The Flow of the Co-verification Methodology

Table 2. Result of Instruction Verification

Instruction Group	Result
Logical Instructions	passed
Arithmetic Instructions	passed
Branching Instructions	passed
Array Instructions	passed
Object-Oriented Instructions	passed
Data Manipulation Instructions	passed

Table 3. Result of Protection Features Verification

Tests	Result
Index a register that is out of current LVF	Causes Exception
Modification of a reference type register	Causes Exception
Operation on two different typed (Float and Int) register	Causes Exception
Branch outside the current MCS	Causes Exception
Access an element index that is larger than the array size	Causes Exception
Invoke a non reference type CPD	Causes Exception
Invoke with an index that is larger than CPDTS	Causes Exception
Invoke a non-static method in a static method	Causes Exception
The address of CPB is larger than CDSS	Causes Exception
Invoke a non-method typed object	Causes Exception
Allocate registers (In + Local + out) larger than 32	Causes Exception
Incompatible argument size for invoked method	Causes Exception
Invoke with an index that is larger than CODTS	Causes Exception
Invoke with a non property type COD	Causes Exception
Invoke an unresolved method	Causes Exception
Invoke with a class reference that is not class typed	Causes Exception
Revoke with IVKTYPE = 00b	Causes Exception
Access non-static field in static method	Causes Exception
Access a static field that out of CDS	Causes Exception
Access a field that is out of IDS	Causes Exception
Modify a read only field	Causes Exception
Access an unresolved field	Causes Exception

Table 4. Results of testing on some Java programs

Tests	Result
Helloworld – display a 'helloworld' text	Correct
Interactive sorting programs – input a sequence of numbers and sorted the numbers with different sorting	Correct
Perform some arithmetic calculation	Correct
Program to convert all input lower case characters to upper case	Correct
Min/Max – input a sequence of numbers and the minimum and maximum value are displayed	Correct
Message Program – an interactive program that can let user to store message, remove message and read message	Correct

5. Conclusion

In this paper, we have discussed the secure hardware data structure on jHISC, which has been built with FPGA technology, and a software platform to co-verify the compatibility, through Just-In-Time compilation, on some Java class files. Through this hardware/software co-design, some resolution burdens in JVM have been compensated through our novel architecture. Besides, comparing with the traditional system where the security and boundary checking of methods are controlled by software, our approach in putting the security and boundary checking jobs into hardware would move the burden of security checking into some pipeline stages and greatly improve the overall system security and throughput of the processor. Moreover, a secure method invocation can safeguard any error or bad reference produced by erroneous programming. Furthermore, our co-verification process verifies the functionalities of the instructions set, exception handling of our hardware protection features and the execution of some Java program examples. Eventually, Java can be run as a native language in the jHISC architecture and all Object-Oriented and Java's protection features are penetrated from software to hardware in order to increase the overall system security and performance.

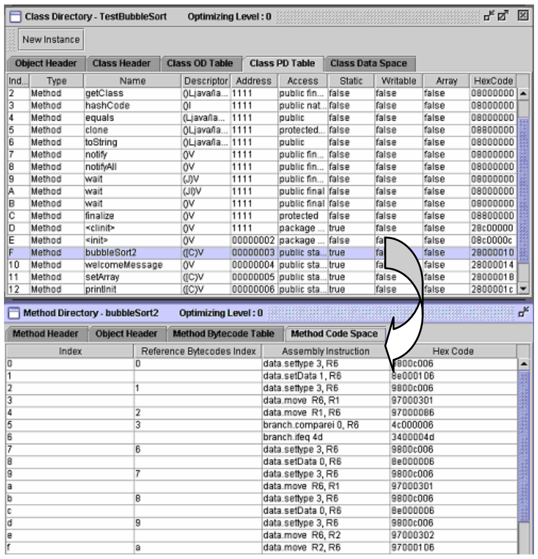
Acknowledgements

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

1. Andrew S. Tanenbaum. Structured Computer Organization, Forth Edition. Prentice Hall, 1999.
2. Fong, A., S., "HISC: A High-level Instruction Set Computer". In 7th European Simulation Symposium, pages 406-410. The Society for Computer Simulation, Oct 1995.
3. Mok Pak Lun, Richard Li, Anthony Fong. "Method manipulation in an object oriented processor". ACM SIGARCH Computer Architecture News archive. Volume 31, Issue 4, September 2003.
4. Bill Venners. Inside the Java 2 Virtual Machine. Mc Graw Hill 1999.
5. Tim Lindholm, Frank Yellin The Java(TM) Virtual Machine Specification (2nd Edition). Addison-Wesley Pub Co; 2nd edition,1999
6. Austin Kim, Yang Qian and J. M. Chang, " Designing a Memory System Using a Static Loader For Embedded Java Architectures" , The Second International Workshop on Compiler and Architecture Support for Embedded Systems (CASES'99) , Washington, D.C., October 1-3, 1999
7. Chang, L.-C., Ton, L.-R., Kao, M.-F., Chung, C.-P., "Stack operations folding in Java processors", Computers and Digital Techniques, IEE Proceedings- Volume 145, Issue 5, Sept. 1998, pp. 333 – 340

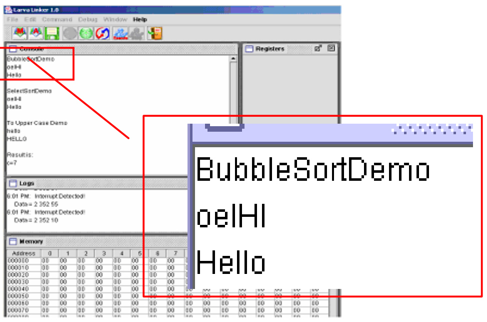
8. Kim, A.; Chang, M, "Advanced POC Model-Based Java Instruction Folding Mechanism", Euromicro Conference, 2000. Proceedings of the 26th Volume 1, 5-7 Sept. 2000, pp. 332 - 338 vol.1
9. Sun Microsystems, Inc. The Java HotSpot performance engine architecture.
10. M. W. El-Kharashi, "Java Microprocessors: Computer Architecture Implications", IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, Vol. 1, August, 1997, pp. 277-280.
11. D. Mulchandani, "Java for Embedded Systems", IEEE Internet Computing, June, 1998, pp. 30-39.



Ind...	Type	Name	Descriptor	Address	Access	Static	Writable	Array	HexCode
2	Method	getClass	()Ljava/la...	1111	public fin...	false	false	false	08000000
3	Method	hashCode	()I	1111	public nat...	false	false	false	08000000
4	Method	equals	(Ljava/la...	1111	public	false	false	false	08000000
5	Method	clone	()Ljava/la...	1111	protected...	false	false	false	08800000
6	Method	toString	()Ljava/la...	1111	public	false	false	false	08000000
7	Method	notify	()V	1111	public fin...	false	false	false	08000000
8	Method	notifyAll	()V	1111	public fin...	false	false	false	08000000
9	Method	wait	(J)V	1111	public fin...	false	false	false	08000000
A	Method	wait	(JI)V	1111	public final	false	false	false	08000000
B	Method	wait	()V	1111	public final	false	false	false	08000000
C	Method	finalize	()V	1111	protected	false	false	false	08800000
D	Method	<clinit>	()V	1111	package ...	true	false	false	28c00000
E	Method	<init>	()V	00000002	package ...	false	fa	false	08c0000c
F	Method	bubbleSort2	([C)V	00000003	public sta...	true	fa	false	28000010
10	Method	welcomeMessage	()V	00000004	public sta...	true	fa	false	28000014
11	Method	setArray	([C)V	00000005	public sta...	true	false	false	2800001B
12	Method	printInit	([C)V	00000006	public sta...	true	false	false	2800001c

Index	Reference Bytecodes Index	Assembly Instruction	Hex Code
0	0	data.settype 3, R6	9800c006
1		data.setData 1, R6	8e000106
2	1	data.settype 3, R6	9800c006
3		data.move R6, R1	97000301
4	2	data.move R1, R6	97000086
5	3	branch.comparei 0, R6	4c000006
6		branch.ifeq 4d	3400004d
7	6	data.settype 3, R6	9800c006
8		data.setData 0, R6	8e000006
9	7	data.settype 3, R6	9800c006
a		data.move R6, R1	97000301
b	8	data.settype 3, R6	9800c006
c		data.setData 0, R6	8e000006
d	9	data.settype 3, R6	9800c006
e		data.move R6, R2	97000302
f	a	data.move R2, R6	97000106

Fig. 7. Class and Method Directory Frame (GUI) displaying information of Class and Method Context which are built from a bubble sort class program and verified through our JIT compiler



```
BubbleSortDemo
oelHI
Hello

SelectSortDemo
oelHI
Hello

To Upper Case Demo
hello
HELLO

Result is:
c=7
```

```
6:01 PM:   Interrupt Detected!
   Data = 2 352 55
6:01 PM:   Interrupt Detected!
   Data = 2 352 10
```

```
BubbleSortDemo
oelHI
Hello
```

Fig. 8. Executing results of some Java programs.