Design and Implementation of Software Objects in Hardware

Fu-Chiung Cheng Hung-Chi Wu
Department of Computer Science and Engineering
Tatung University
Taipei, 104 Taiwan, R.O.C.
fccheng@ttu.edu.tw wuandy@ms10.hinet.net

Abstract

This paper proposes a novel approach to implement software object in hardware. Data-Memory mapping schemes are investigated and four hardware object design schemes are proposed and implemented on a CAD tool. Performance evaluation is carried out in Altera Quartus tool in terms of speed, logic elements and number of transitions. The result of experiments shows that object-reference scheme is much better than the other 3 schemes in terms of hardware cost, energy consumption and speed for FPGA implementation.

Keywords *: Java software objects, hardware objects, self-timed system.*

1. Introduction

SIA roadmap for semiconductor design [1] shows that for the last two decades potential design complexity has grown at a rate of 58% per year, while the designer productivity at the same time has only raised 21% per year. This has led to a growing design productivity gap between manufacturing capability of chips and the functionality that designers can implement in unit time. The manufacturing capability is predicted to grow at the same rate for another decade and hence the gap must be filled by increasing designer productivity rate of growth.

To fill in the gap, future design methods must provide a major productivity leap [2]. This is likely to require a paradigm shift. Hardware design will have to happen at a much higher level of abstraction. Reuse of previously designed components will have to take place at a large scale and in an organized way. Time-to-market is also pressing issue. With product cycles shortening, the timely delivery of a design becomes more and more important.

New design methodologies and tools are quite necessary. Object-oriented (OO) design, successfully used for several years by the software community, is a rather different approach to complexity management compared to traditional hardware design methodologies. In OO methodologies, however, the designer first distinguishes the main data type (or classes) present in the system and the operations that should be applied to them. The whole system is composed of data (or objects) of these types which are interacting by calling one another's methods. In other words, OO methodology suggests modeling the system in terms of its constituting data objects, while traditional methodologies concentrate on structurally decomposing the target architecture of the system.

By utilizing object-orientation in hardware design, we can benefit from its abstraction and reuse techniques [3, 4]. Moreover, object-orientation would then have the potential to unify software and hardware design.

This paper proposes a novel approach to implement reusable software objects in hardware. Four hardware object design schemes are proposed and implemented on a CAD tool. And, performance evaluation, in terms of speed, power consumption and cost, against these schemes is carried out by using Altera Quartus II. The result of experiments shows that object-reference scheme is much better than the other 3 schemes in terms of hardware cost, energy consumption and speed.

This section describes the motivation of this paper. The rest of the paper is organized as follows: Section 2 introduces some background knowledge, including previous work and the concepts of software class and object. Section 3 describes the data-memory mapping analysis. Section 4 presents four designs of hardware object implementation schemes. The implementation and performance evaluation of hardware objects are described in Section 5 and 6, respectively. Section 7 summarizes this paper and presents some suggestions for future work.

2. Background Knowledge

This section introduces related works and the concepts of software objects and classes.

2.1. Related Works

Radetzki [2, 3] proposed a synchronous hardware object implementation of a non-derived class. The structure of the object circuit is guided by the model of a synchronous finite state machine implementation. That is there is a memory element for state storage and a controller to control state transition and output logic, and a feedback of the next state into the state memory. No concurrent methods can be invoked at the same time due to the multiplexer implementation. This work has also been used in the ODETTE project [4], which

introduces SystemC-Plus as an extension to SystemC with synthesizable object-oriented features [5].

Goudarzi and Hessabi proposed object implementation with inheritance and polymorphism [6]. Objects are stored in a global memory and an Object Management Unit (OMU) maps the virtual address on function units to physical address in global memory.

2.2. Object Implementation in Java

Java is an object-oriented language that operates on variable sized contiguous lists of words called objects [7]. Objects are comprised of many fields. The fields of an object are accessed and manipulated by class's methods. These fields can be classified into class (or static) and object (or instance) data. Class data are created during the class load time and have only one copy in the memory called *method area*. The object data are created in the memory called *heap* whenever a "new" operator is performed.

For example, a Stack class, which skeleton source code is shown in Program list 1, has 4 methods (push, pop, isFull, isEmpty), two *class data* (numOfObjects and errorFlag) and three *object data* (statckTop, size and stackData).

Program list 1 Stack

```
public class Stack {
 private byte stackTop=0;
 private byte size=4;
 private byte stackData[];
 private static byte numOfObjects =0;
 public static boolean errorFlag=false;
 public Stack() {stackData=new byte[size];}
 public Stack(byte stackSize) {
   size = stackSize;   this();
 }
 void push(byte newdata) {
  if (stackTop < size)   {
    stackData[stackTop] = newdata;
    stackTop ++;
  }  else {
    errorFlag = true;
 }}}
 public byte pop() {…}
 public boolean isFull() {…}
 public boolean isEmpty() {…}
}
```

2.2.1. Software Object Representation

Sun's Java virtual machine (JVM) uses an indirect address object model [8] in which the handle contains pointers to the object's location in the heap and a pointer to the object's method table.

2.2.2. Sizes of Objects

According to the memory space required to allocate an object, objects can be classified into ***fixed-size*** and ***variable-size*** ones. For example, the objects created based on the Stack class are of variable-size since the size of the stack, stackSize, is determined at run time. Parameter stackSize is passed to the constructor, Stack(byte stackSize), and an array of bytes with size stackSize is allocated to stackData.

A RSA encryption/description class [10], on the other hand, can create ***fixed-size*** objects only. To allocate space for fixed-size objects in hardware is easier than for variable-size ones.

3. Data-Memory Mapping Analysis

To implement software objects in hardware, we first analyze how to store object's data and then map these data to either global or local memories.

A class has two types of data: class (i.e. static) data and object (i.e. instance or non-static) data. These data can be allocated in either a global memory (e.g. main memory) or a local memory (e.g. an embedded memory in FPGAs or reconfigurable processors). Here we assume the capacity of the global memory is much larger than that of the local memory.

For software object implementation in JVM, both class data and object data are stored in the global memories called "method area" and heap, respectively. For hardware IP implementation in FPGAs/VLSI chips, class data and object data can be stored in the embedded memory (i.e. local memory) of FPGAs or in SRAM or DRAM (i.e. global memory) outside the FPGAs/VLSI chips.

Based on types of data and types of memories to hold these data, there are four possible data-memory mapping schemes as follows: CLOL (**C**lass data and **O**bject data in **L**ocal memories), CLOG, CGOL, and CGOG schemes.

1. **CLOL mapping scheme:** In this scheme, both class and object data are stored in IP's (Intellectual Property) local memory. The possible hardware implementation of CLOL scheme is shown in Fig 1.

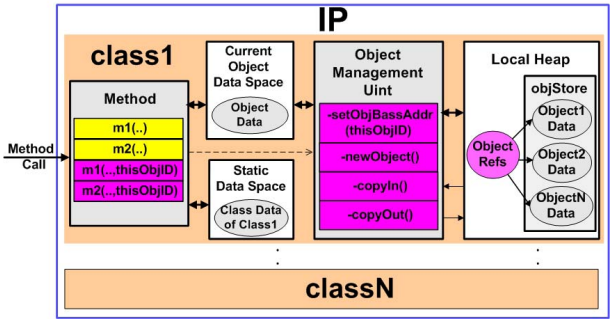

Fig 1 HO implementation of CLOL mapping and CICO scheme

The IP block may contain one or more classes and each class may be used to create multiple objects. Each

class implemented in hardware contains the following parts: (a) the method circuits implementing class's methods directly in hardware; (b) the current object data space holding the current object data that are directly accessed by the method circuits; (c) a static data space holding static data; (d) a local heap holding created objects; and (e) the object management unit (OMU) which copies the object data in local heap to the current object data space through the object references which point to the objects created in the local heap.

The method circuits can directly access the class data and a copy of object data stored in the current object data space. In fact, the method circuits, together with temporary object data and class data, are exactly the same as a software class. Before "this" (i.e. current object reference in Java) object can be manipulated by the method circuits, "this" object data in the local heap have to be copied to the current object data space.

This scheme is particularly suitable for modern FPGAs in which the method circuits and the OMU can be implemented in logic blocks and the static data space, the current object data space and the local heap can be implemented in the embedded memories.

Moving object data between the current object data space and the local heap is not free and may increase hardware cost as well as power consumption. One possible improvement, shown in Fig 2, is to remove the space of the temporary object data so that the method circuits can directly manipulate the object data by using the object management unit.

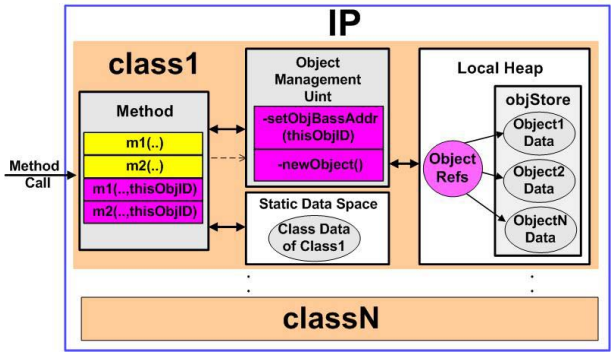

Fig 2 Efficient HO implementation and Object-reference scheme

2. **CLOG mapping scheme:** This scheme is similar to CLOL except that objects are stored in a global memory (e.g. SRAM or DRAM outside the hardware IP) instead of a local embedded memory. Thus, the number of objects that can be stored in the global heap is much larger than that in the local heap.

3. **CGOL mapping scheme:** The current object data are directly accessed by the methods circuits. Objects are stored in the local heap and managed by the object management unit. The class data management unit is required to access class data in the global heap.

CGOG mapping scheme: In this scheme, both class and object data are stored in a global heap such as DRAM. The hardware IP contains the method circuits and the Data Management Unit (DMU) which provides access mechanism for manipulating both object data and class data. This scheme can be implemented by a FPGA with an off-the-shelf memory component. The hardware methods and the DMU are implemented in FPGA's logic blocks.

Table 1 Comparison of data-memory mapping scheme

Scheme	Class Data	Object Data	FPGA w/o embedded Memory	FPGA w embedded Memory	VLSI Impl.	Object Space
1.CLOL	LM	LM	X	Good	Good	Limited
2.CLOG	LM	GM	X	Good	Good	Huge
3.CGOL	GM	LM	X	Good	Good	Limited
4.CGOG	GM	GM	Good	OK	Good	Huge

Table 1 summarizes these data-memory mapping schemes where LM/GM stands for local/global memory. The four schemes discussed above can be implemented in both VLSI and FPGAs with embedded memory. For the FPGAs without embedded memory, only CGOG scheme can be applied to this type of FPGAs. Finally, the number of objects can be stored in a local memory is usually limited; therefore, for the applications which need to store huge amount of objects, CLOG and CGOG schemes are preferred.

This paper focuses on implementing hardware objects in modern FPGAs with embedded memory based on the CLOL scheme. How to implement the other three mapping schemes in either FPGAs or VLSI is reserved for the future work.

4. Hardware Object Design Schemes

Modern FPGAs such as Altera Stratix II family and Xilinx Spatan-III contains a good chunk of embedded memories. Thus, hardware IPs containing many classes can be implemented inside a single FPGA using CLOL hardware object implementation scheme.

The following sub-sessions propose four efficient hardware object implementation (HOI) schemes for FPGAs with embedded memory. They are Copy-In Copy-Out (CICO), CICO with Object ID check (IDC), CICO with Object ID and dirty check (IDCD), and object-reference (ObjR) schemes. Note that these methods can be applied to VLSI implementation as well.

4.1. Copy-In Copy-Out scheme

In CICO scheme, methods and the OMU are implemented in the FPGA's logic blocks and class data, current object data space and the local heap are implemented in the embedded memories as shown in Fig 1. The main idea of CICO scheme is as follows:

Objects are created in the local heap. Before any function call, "copy-in" the object data from the local heap to the current object data space. Whenever completing any function call, "copy-out" the current object data in the current object data space back to the local heap.

4.2. CICO with Object ID check scheme

Due to blindly copy in and move out objects between the current object data area and the local heap, the CICO scheme may suffer from performance and power consumption penalty. This can be improved by checking if updating the current object in the current object data area is necessary. We can add an additional variable, currentObjectID, to track the object stored in the current object data area. If the object ID of the current method call is the same as the current object ID, then there is no need to copy out the current object and to move in the new object from the local heap. This can save huge amount of energy consumption for a high-changing method call sequence.

4.3. CICO with dirty object check scheme

For the CICO with Object ID check scheme, the current object data are copied out even though the object data are not dirty when a new object is going to be used. If the object stored in the local memory is intact, there is no need to carry out the copy-out operation. This can be achieved by adding a dirty bit to check if the current object is clean or dirty. Such scheme is called CICO with Object ID and dirty check scheme.

One of the advantages of this method is to further reduce the communication overhead of copying out object data if the current object is not dirty. The extra hardware cost of this scheme includes checking dirty bit and detecting dirty object data. If object-change rate is high and most method calls are read-only ones than the improvement can be dramatically.

4.4. Object-Reference scheme

The previous three hardware object implementation schemes contain the space to cache the current object (i.e. "this" object) to be accessed by hardware method circuits. If we can merge the current object data into the local heap, there is no need to carry out copy-in and copy-out operations. Such scheme is called object-reference scheme and is shown in Fig 2.

In this scheme, object data are created directly in the local heap and accessed directly by methods through the OMU. Every object created in the local heap is associated with an object reference which points to the object. Object references are stored in an array, objectRefs, and objects are stored in another array, objStore. Both arrays are implemented in two embedded memories inside FPGAs. Since the object data are stored in objectStore array and there is no current object data space, the original object (instance) variables have to be mapped into objectStore's location based on the object reference. Thus, the instance variables shown in all methods are renamed to the objectStore references.

4. Implementation

The hardware object design schemes mentioned in previous section can be applied to both synchronous and asynchronous systems. One of the severe problems of synchronous object implementation is that it is very difficult to reuse these synchronous hardware objects in SOC design due to timing issue. We believe that, for hardware IP reuse, self-timed system technology is a better choice due to the modularity, composibility, low electro magnetic interference (EMI) and low power consumption [11-13].

5.1. Method Circuit Implementation

The design flow of hardware method (API) implementation is shown in Fig 3. Firstly, the functionality of Java classes are tested and verified in an IDE (integrated development environment) tool such as JBuilder, NetBean or Eclipse. Secondly, the Java classes are then translated into VHDL codes by SOCAD [9, 10], a CAD tool for SOC design. Together with the self-timed cell library, the VHDL codes are then simulated and verified in either FPGA design flow (such as Altera Quartus or Xilinx ISE) or VLSI design flow. This paper focuses on FPGA design flow.

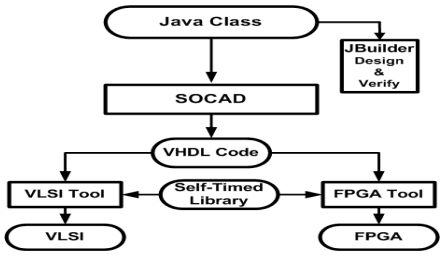

Fig 3 Design flow of method circuit implementation

5.2. Hardware Object Implementation

The current version of SOCAD does compile software methods into hardware VHDL codes; however, it does not support hardware objects, yet. Our methodology proposed in section 4 can add hardware object functionality to SOCAD as shown in Fig 4. That is our hardware object implementation schemes can be applied to Java classes to produce new Java classes with object management units. The Java classes with OMUs

are first verified in a Java IDE tool and then translated into VHDL codes by SOCAD tool.

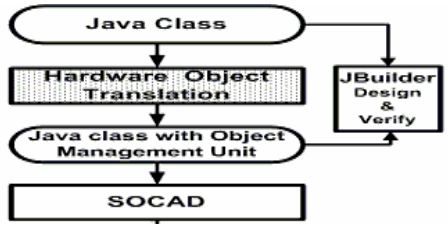

Fig 4 Design flow of hardware object implementation

Hardware object implementation has two cases to be considered: objects with fixed size and objects with variable size. Due to the space limitation, we show only CICO scheme with fixed size object in detail. The translation steps of the CICO scheme for *fixed-size* objects are listed as follows:

For each class do the following steps:

1. Global constants:

(a). Set a constant, MAX_NO_OF_OBJS, to indicate the maximal number of objects that can be created in the local heap.

(b). Compute the object (i.e. instance data) size of the class in interest and set the object size in the constant, SIZE_OF_OBJ.

2. Heap (or object) management:

(a). Create a heap space with its size equal to the maximal number of objects times the size of the object (e.g. objStore= new byte[SIZE_OF_OBJ* MAX_NO_OF_OBJS];).

(b). Create two static variables, heapTop and numberOfObjects, to point to the free space of the local heap and to record the number of objects created in the local heap, respectively.

(c). Add three private methods, *newObject, copyIn* and *copyOut,* to create and initialize an object in the local heap, to copy the data in the current object data space into the local heap and to copy the object data in the local heap into the current object data space, respectively.

(d). For each public constructor, *c*, in the class, add a statement to the end of the constructor, *c*, to call newObject method. The newObject method allocates a space for the object in the local heap and sets the initial values of the instance data.

3. Object reference management:

(a). Create a static variable, objectBaseAddress, to point to the first address of the current object in the local heap.

(b). Create an array of object references to keep tracks of the base (start) address of the objects created in the local heap.

(c). Add a private method, setObjectBaseAddress (thisObjID), to adjust the objectBaseAddress variable to the start address of the object in the local heap based on the thisObjID.

4. **Overloaded method modification:** For each public method, *m*, in the class, add a new public overloaded method with an additional parameter, thisObjId. The new overloaded method performs the following tasks:

(a). Call setObjectBaseAddress method to adjust the objectBaseAddress pointing to the start address of the object in the local heap.

(b). Call copyIn method to move object data based on thisObjID in the local heap to the current object data space.

(c). Call the method *m* and keep the return value in a local variable, returnResult, if necessary.

(d). Call copyOut method to copy the current object data back to the local heap.

(e). Return returnResult value if necessary.

5.3. Verification

The Altera Quartus can be used to verify the correctness of hardware objects in VHDL codes. The source code of Java, Java classes with OMUs, complied VHDL codes, detail verification and simulation results with Altera project can be found from our web site [10]. Also a version of executable SOCAD tool with a set of self-timed cell library can be downloaded from [9].

5. Performance Evaluation

The objectives of this section is to carry out the performance evaluation of the four HO schemes in terms of the number of logic cells (i.e. hardware cost), signal transitions (i.e. energy consumption) and speed. The experiment is carried out by using Altera Quartus II 4.2 Tool with the Stratix- EP1S10B672C6 FPGAs.

1. Hardware cost: Table 2 shows the summary of hardware cost in terms of the number of logic blocks and the embedded RAM used. A RSA/Stack IP is set to hold up to 10/3 hardware objects. For stack/RSA IP columns 2/7 and 3/8 show the number of logic blocks used and the ratio of ObjR scheme, respectively and columns 4/9, 5/10 and 6/- show the embedded memory usage for objStore, objRefs and stackData, respectively.

Table 2 Hardware Cost of IPs

Scheme	Stack IP					RSA IP			
	#N of LBs	ObjR to ratio	Embedded RAM (byte)			#N of LBs	ObjR to ratio	Embedded RAM (byte)	
			Obj Store	Obj Refs	Stack Data			Obj Store	Obj Refs
CICO	3019	1.27	20	3	5	4370	1.02	30	10
IDC	3166	1.32	20	3	5	4479	1.04	30	10
IDCD	3201	1.34	20	3	5	4523	1.06	30	10
ObjR	2383	1	20	3	N/A	4287	1	30	10

The preliminary result shows that object-reference scheme uses much less logic blocks in variable-size

objects (i.e. Stack IP) as shown in columns 2 and 3 of Table 2. However, the improvement is not impressive in fixed-size objects as shown in columns 7 and 8.

2. Number of Signal Transitions: For self-timed circuits, the energy consumption is proportional to the signal transitions. Table 3 shows the average number of transitions applying the same input patterns for the four schemes. Quartus Power Analyzer tool (i.e. PowerPlay) is used to measure the power consumption. Since it is designed for synchronous circuits, the result may be used for reference only. Columns #T, DTPD and SP (speedup) denote the total number of signal transitions, dynamic thermal power dissipation and the ratio of signal transitions, respectively. The result shows that, for Stack, object-reference scheme is about 4/2/2 times less power consumption than CICO/IDC/ICDC hardware object schemes. However, for the RSA IP, the power consumptions of these four schemes are similar. This is because the total number of transitions of copy-in and copy-out occupies only a small portion of the whole transitions of RSA encryption or decryption operations.

Table 3 Comparison of #Transitions

Scheme	Stack IP				RSA IP			
	#T	SP	DTPD (nw)	SP	#T	SP	DTPD (nw)	SP
CICO	133428	4.99	0.12	4	412128	1.01	1.59	1.05
IDC	72087	2.69	0.06	2	409688	1.00	1.53	1.01
IDCD	66834	2.50	0.06	2	411093	1.01	1.53	1.01
ObjR	26715	1	0.03	1	408196	1	1.52	1

3. Speed: Push operations of Stack IP are used to measure the speed performance which is strongly depended on object change rate. Table 4 shows the execution time based on 30% and 100% object change rate. The result shows that object-reference scheme outperforms the other three schemes by a factor of about 6 and 2 for high-object and low-object change rates, respectively.

Table 4 Speedup of different object change rates

	100% object change rate				30% object change rate			
	CICO	IDC	IDCD	ObjR	CICO	IDC	IDCD	ObjR
Min	2.636	2.86	2.868	0.474	2.634	0.187	0.197	0.473
Max	3.069	2.87	2.877	0.478	3.064	2.865	2.877	0.476
Avg	2.853	2.866	2.873	0.476	2.807	0.99	1.001	0.474
spdup	1	0.995	0.991	5.99	1	2.835	2.804	5.921

6. Conclusions and Future Works

This paper proposes a new approach to implement software objects in hardware. Data-memory mapping schemes are investigated and four hardware object implementation schemes based on CLOL scheme are proposed and implemented.

Performance evaluation is carried out by using Altera Quartus tool in terms of speed, logic elements and number of transitions. Result of experiments shows that object-reference scheme is much better than the other 3 schemes in terms of hardware cost, energy consumption and speed.

Our goal is to implement time-to-market reusable hardware objects with adequate performance, less EMI and low power consumption based on self-timed system technology.

There are four possible directions for further work: Firstly, the granularity of dirty check can be important since it is not uncommon that the objects to be created can be large. Secondly, this paper deals with only those classes without inheritance. How to implement software classes with inheritance in classes is still under investigation. Thirdly this paper investigates only on CLOL mapping scheme. It is very interesting to explore and implement the other three mapping schemes. Finally, polymorphism and garbage collection are not yet being considered in this paper.

7. References

[1] International Technology Roadmap for Semiconductors (ITRS) Design, 2001.

[2] Radetzki M., Synthesis of digital circuits from object-oriented specifications. *PhD Thesis*, University of Oldenburg, 2000.

[3] T. Kuhn, W. Rosenstiel, U. Kebschull. *Object Oriented Hardware Modeling and Simulation Based on Java*. Proc. International Workshop on IP Based Synthesis and System Design, 1998.

[4] The ODETTE Project, Within European commission IST Research Program, http://odette.offis.de/.

[5] Grimpe E., Timmermann B., Fandrey T., Biniasch R., Oppenheimer F., SystemC object-oriented extensions and synthesis features. *Forum on Design and Specification Languages*, 2002.

[6] M. Goudarzi, S. Hessabi, "Synthesis of Object-Oriented Descriptions Modeled at Functional-Level," World Scientific and Engineering Academy and Society Transactions on Computers, Athens, 2003.

[7] Sun Microsystems Inc. *The Java Virtual Machine Specification*, 1996

[8] Bill Venners. *Inside the Java 2 Virtual Machine*. McGraw-Hill Companies, Inc., 1999.

[9] SOCAD, A CAD tool for SOC Design, 4C applied technologies lab. of CSE Dept. Tatung University, Taiwan, http://4c.cse.ttu.edu.tw/snipsnap/space/SoCAD.

[10] Paper source code URL: http://homepage.ttu.edu.tw/d9406002/index.html

[11] A. Davis and S.M. Nowick. *An Introduction to Asynchronous Circuit Design*. Computer Science Department, University of Utah, Sep. 1997.

[12] Fu-Chiung Cheng, Stephen H. Unger and Michael Theobald, "Self-timed Carry-Lookahead Adders", IEEE Transactions on Computers, pages 659-672, July 2000.

[13] Fu-Chiung Cheng, "Practical Design and Performance Evaluation of Completion Detection Circuits" ICCD'98, pages 354--359. IEEE Computer Society Press, 1998.