

An Approach to Hardware Synthesis from a System Java[®] Specification*

João M P Cardoso
University of Algarve/INESC
ESDA (Electronic Systems Design Automation) Group
{Joao.Cardoso, hcn}@inesc.pt

Horácio C Neto
IST/INESC

Abstract

In this paper we present a new approach to HW/SW co-design starting from a system specification using the Java programming language. A novel compiler front-end is described that extracts all the needed information of the given specification and represents it in an Object-Oriented (OO) intermediate representation graph. It exploits different levels of parallelism to permit efficient binding onto an architecture of HW resources. A number of compiler transformations are then performed in order to generate the appropriate VHDL behavioural specification of the OO model. Finally High-Level Synthesis (HLS) techniques are used to generate the most adequate HW architecture.

1. Introduction

HW/SW co-design addresses the concept of meeting system-level objectives through the concurrent design of two types of system-components (HW and SW) [1]. The use of OO technology to model HW/SW systems makes possible the management of designs of increased complexity because it permits higher levels of abstraction and facilitates the reuse of specifications (critical to minimise the time-to-market). Starting from an OO model has the further advantage that each class encapsulates methods and data with tightly coupled affinity relationships and, therefore, its instances are good candidates to bind into a component (HW or SW).

Some HW/SW co-design research authors use an OO language but translate it to an intermediate representation model where the OO initial structures are lost. Our approach preserves the initial class hierarchy structure and, therefore, a partition at the class level such as the one proposed in [2] is feasible. As the specification language we adopt the Java language [3] which has suitable properties to specify at the system-level [4].

* This work has been partially supported by the Ph.D. program of the Prodep 5.2 action and the program Praxis XXI under the scope of Project PRAXIS/2/2.1/TIT/1643/95.

Java has a wide range use: in specification of HW/SW systems [4], embedded, reactive and real-time systems [5][6], programming of applets, and generic applications [3]. It has syntax mechanisms to support concurrency (critical to embedded systems), is architectural independent, is a pure and simple OO language, is strongly typed, contains exception handling mechanisms, is safe and robust, is partially dynamic, etc.

We describe some of the work done to automate a co-design environment starting with an OO system specification using the Java language. In Section 2 we summarise the Java OO model. Then, Section 3 describes the computations and transformations performed by the front-end compiler. Section 4 describes the implementation of OO concepts in HW. Finally, in Section 5 we present results in a proof of concepts base, and in Section 6 conclusions and future work to be done are presented.

2. The Java OO Model

The Java technology has had a fast dissemination, is available for free, and a large number of support tools has been developed. The use of it for the system specification further supports the use of an abstract model by compilation to the JVM (Java Virtual Machine) [7] and makes available multithreaded concurrency.

The abstract model used to represent the textual system specification does not lose the class relationships and structures, and thus embodies the use of a real OO intermediate representation model (as shown in Figure 1) in the HW/SW co-design phases. Furthermore, the use of the Java *classfile* intermediate representation makes possible the support of multi-language specifications, and permits a system specification executable.

Java is easier to analyse than C++ and makes feasible a more powerful analysis. An object reference in Java points only to the beginning of the object and two references correspond exactly to the same memory location or not. The memory locations containing references can be determined exactly using type information, because of the static and strong typing propriety of the language.

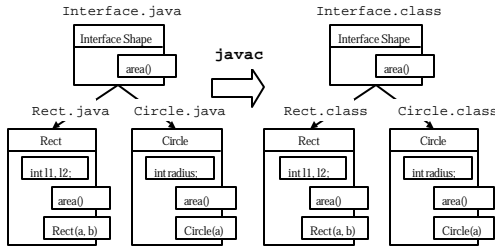


Figure 1. Java specification and the resulting classfiles.

3. The Front-End Environment

Figure 2 illustrates the proposed HW/SW co-design environment. The SW part of the specification is furnished in *classfiles* to be executed directly by a Java processor, by an interpreter or JIT compiler, or by the translation of the *classfile* representation to native code (e.g. C with later compilation to the target processor). The HW part is furnished in behavioural VHDL processes suitable to be synthesised by an HLS system. Timing constraints can be specified in the source code permitting the identification of the constrained or time-consumption code regions by the front-end.

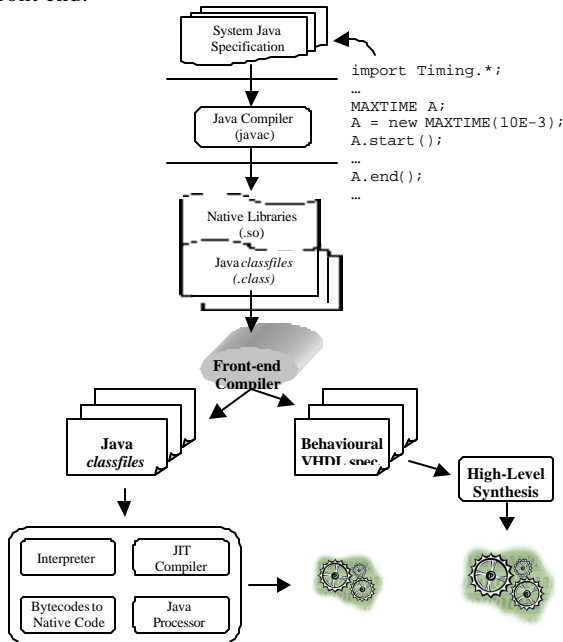


Figure 2. The Java environment co-design flow.

From the *classfiles* the front-end performs the following transformations [8]:

- Construction of the Method Calling Sequence Graph (MCSG+) that is a Call Graph type with extensions to the OO model. The root of the graph is the main method, each edge represents a message sent or a field access, and each vertex represents the set of methods, that can be called by

the message, or the field accesses. Figure 3 illustrates the graph for an array sorting application (where each ellipse represents an access to a field and a box represents an invoked method). The class relations and the access to class members are also represented by this graph (but are omitted in the figure).

- For each method in the MCSG+, a Control Flow Graph (CFG) representing the sequence of *bytecode* instructions is constructed.
- For each CFG, *bytecode* instructions are grouped in Basic Blocks (BBs) and a new CFG where each vertex is a BB of JVM instructions is obtained. From the new CFG the Dominators Tree is constructed and the existent natural loops are identified [9] and extracted.

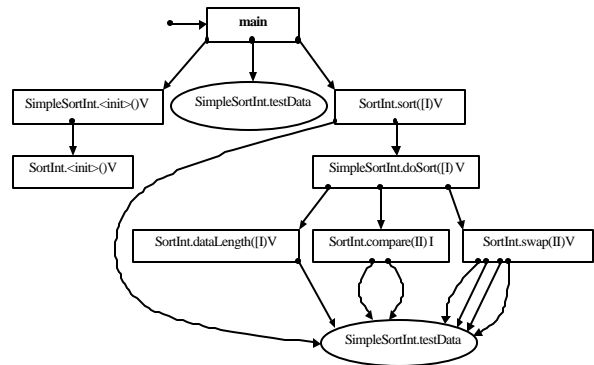


Figure 3. An example of the MCSG+.

An intra-procedural analysis is performed on each BB of the CFG to extract the implicit parallelism at the BB level. Then, a Control Dependence Graph (CDG) and a Data Dependence Graph (DDG) are produced, which are fundamental to evaluate the implementation of parallel parts of the graph by different components. The DDG construction considers the utilisation of local variables, the use of the operand stack, the memory alias, the access to array elements, access to fields, and the semantic dependencies (like the dependencies between a manipulation of an object and the call of its constructor).

Because of the semantic gap between the OO model and the VHDL model [10], additional compiler transformations are performed before the generation of the correspondent VHDL specification: object inlining (methods and data); flattening of the classes hierarchic structure; and implementation of dynamic dispatching.

Because of the partial dynamic nature of the Java language, the compiler determines which class instances will be created by the application by performing a static analysis of the memory object allocation. At the moment object and array creations inside loop bodies are not permitted. In the case of small and deterministic loops a static allocation will be considered in the near future. In the other cases pre-allocation techniques, such as described in [4], are performed.

OO programming encourages the use of small methods which typically results in an high invocation density.

Hence, the compiler considers the migration of the methods completely to HW and also the possibility of duplication of methods of the same class to be executed in parallel when invoked from different objects (in different memory locations).

4. Object-Oriented Concepts in HW

An invocation of a method needs the resolution of the right method definition. This is done with a bottom-up search of the method definition on the class hierarchy, represented in the MCSG+, starting in the set of classes that reaches the point where an object sends a message. The analysis of the whole program can avoid the dynamic dispatching when it knows exactly which method will answer a specific message. Without this analysis all the set of methods must be synthesised and a conditional selection based on run-time information can allocate the HW circuit that implements the invoked method (see Figure 4).

Java	VHDL
<pre> if(option == 2) S = new Rect(l1, l2); else S = new Circle(l1); ... int a = S.area(); ... VHDL </pre>	<pre> if(option = 2) then newRect(l1, l2, Rect_l1_l1, Rect_l2_l1); Class_1 := Rect; else NewCircle(l1, Circle_radius_1); Class_1 := Circle; end if; ... if(Class_1 = CLASSES'(Rect)) then Rect_area(Rect_l1_l1, Rect_l2_l1, area); else Circle_area(Circle_radius_1, area); end if;... </pre>
<pre> type CLASSES is (Circle, Rect); variable Class_1: CLASSES; ... </pre>	

Figure 4. Dynamic dispatching in VHDL.

The sequence of messages sent to a given object is maintained as in the original CFG. The extraction of possible parallelism in this sequence requires a more in-depth inter-procedural analysis. The intra-procedural implicit parallelism of field accesses for a given object is determined by computing the reaching definitions and the use-definitions chains [9]. However, when these fields are allocated to a single one-port RAM and parallel accesses from different HW components are possible, a synchronisation scheme is needed to avoid memory access conflicts.

The reference alias problem is resolved by data-flow computation over the CFG of BBs of the given method. This is done with a reaching definition analysis that includes object/array creations and assignments. The object references of the same class type (or a type in the matching class hierarchy) passed as method arguments are identified as alias references to force data-dependencies by default, because currently a whole program analysis of memory alias is not performed. In the case of exploitation of parallel execution of a method invoked by different class instances an analysis of possible static fields accesses by that method must be considered. An access of the method to a static field (class variable) can create a data dependency.

At the moment, there are some restrictions in the allowed *bytecode* segments to be translated to VHDL. How-

ever, some of these limitations will be resolved in a near future. The limitations are the following:

- Object members cannot reference an object.
- The implementation of the `Object` class (the root of all the Java classes) is not considered.
- The use of threads is not considered.
- The object creation inside loops is not permitted.
- The `double` type is not considered.
- The exception handling mechanism (`try - catch - finally`) is not allowed.
- Recursively methods are not considered.
- An HW entity cannot invoke an SW method.
- The utilisation of the API libraries is set to SW.
- Break statements inside loops are not allowed. All loops are described by VHDL `while` loops.
- All the switch constructors must have `break` statements, because VHDL does not support flow to the next switch branch directly.

The designer has to select the allocation of large intermediate arrays in the HW board if it has RAM, thus avoiding bus traffic, in the main system memory or in the data-path of the synthesised HW as registers. In the case of the use of the main memory system, the SW specification must send to HW the addresses of arrays or objects.

Bytecode ALU operations with some of the primitive data types (like `boolean`, `byte`, `short`, `int`, etc.) use the same JVM instructions [7] and a conversion instruction is used before the result is stored into a local variable. The behavioural VHDL generated uses a 32 bit minimum word length for ALU operations. A data-flow analysis is needed to resolve this problem through extraction of the possible type information [9]. The JVM instructions for type conversions like `i2s` (`int` to `short`) are translated to equivalent VHDL statements that use conversion functions. The access to array elements is type differentiated (except the arrays of `booleans` that are treated like arrays of `bytes`) [7].

<pre> a) package HwInterface; public class InOut { final static public void setup() { System.loadLibrary("InOutNat"); } final static public native void write (int Address, int Data); final static public native int read(int Address); ... } </pre>
<pre> b) #include <jni.h> #include "HwInterface_InOut.h" JNIEXPORT void JNICALL Java_HwInterface_InOut_write (JNIEnv *Env, jobject c, jint a, jint b) {...} JNIEXPORT jint JNICALL Java_HwInterface_InOut_read (JNIEnv *Env, jobject c, jint a) {...} </pre>
<pre> c) ... HwInterface.InOut.setup(); ... HwInterface.InOut.write(Address1, Data1); ... Data2 = HwInterface.InOut.read(Address2); ... </pre>

Figure 5. SW-HW communication class library.

5. HW/SW Communication

A call to a native method [3] is inserted in the *bytecodes* every time the SW needs to communicate with the HW. Figure 5 shows a simple interface (using a memory

mapped based concept) class library (5a), the file of the C functions (5b), and the use of these functions in a Java program (5c). The JVM representation of the Java specification (5c) is inserted in the method *bytecode* of the *classfile*. The *setup* method must be invoked in the *startup* of an application because of the class and library loading overhead (€31ms executed in a Sun SPARC 10 under SunOS 4.1.4 with the Java interpreter [11]).

6. Results

All the code of the framework front-end has been developed in Java and the current version specification contains over 9,000 lines of Java source code.

Experimental results validate the approach described. The HW solutions obtained using the CADDY-II [12] HLS system use at the back-end the Synopsys™ Design Compiler [13] with the IMS SOG gate-array [14] as the target library. Table 1 presents the experimental results between HW and SW implementations and the area of the HW solutions. The speedups shown do not take into account the communication overhead.

Table 1. HW results (area in equiv. gates - eqG)

Ex.	Lines of Java	JVM Inst.	BBs	Lines of VHDL	SW exec. Time	HW exec. time (@20MHz)	HW area (eqG)	Speed Up
E1	48	267	60	271	3ms	280µs	7,563	11
E2	27	121	6	173	6µs	150ns	910	40
E3	41	53	30	68	7.3µs	100ns	5,235	73

The example E1 is a filter image algorithm (input image of 16 x 16 pixels) and has intensive memory accesses [15]. The HW follows the DMA principle and stores the intermediate arrays in memory. The increase of the input image will improve the achieved performance. The unrolling of the loops increase the speedup but furnishes an impractical HW solution in terms of area.

E2 (a hamming decoder) is an example with shifts and bit logical operations. The HLS solution is obtained unrolling all the loops in the VHDL code (responsible to the shift functions) and executes in 3 cycles while the rolled one executes in 47 cycles.

Example E3 corresponds to a simple Java application (VHDL specification of the Figure 4) with three classes, and one interface. The SW execution time, including the class loader overhead, is about 4ms.

7. Conclusions and Future Work

We have described the techniques used in order to provide an automatic prototyping path from the abstract model of Java specifications to HW. The developed front-end makes possible, with the utilisation of an HLS system back-end, the system-level synthesis approach starting from a system Java specification. It has the capability to

determine intra-procedural implicit parallelism, which is an effective way to exploit HW/SW trade-offs, especially when the overall system consists of multiple components, thus having the possibility to map the parallel blocks into independent devices.

Ongoing work focuses on the full automation co-design process, especially the estimation phases and the use of a partitioning algorithm. Other envisaged work will be towards the integration of memory managing techniques on the HW side to deal with the partial dynamic nature of the Java language, like the data allocation in the specification of systems that use dynamic data structures (e.g. linked lists) proposed in [16].

From the resulting parallel graph, obtained with the implicit parallelism extraction, the scheduling of memory accesses and the synchronisation between partitions must be performed in order to make feasible the mapping of each partition onto an HW component.

References

- [1] G. De Micheli, R. K. Gupta, "Hardware/Software Co-Design", Proc. of the IEEE, vol. 85, no. 3, March 1997, pp. 349-365.
- [2] W. H. Wolf, "Object oriented co-synthesis of distributed embedded systems", ACM TODAES, vol. 1, no. 3, July 1996, pp. 301-314.
- [3] Ken Arnold, and James Gosling. The Java Programming Language. Addison-Wesley, Reading, Massachusetts, 1997.
- [4] R. Helaihel, and K. Olukotun, "Java as a Specification Language for HW-SW Systems", Proc. of the IEEE ICCAD, San Jose, California, Nov., 9-13, 1997, pp. 690-697.
- [5] C. Passerone, et al., "Modeling Reactive Systems in Java", Proc. of the 6th IEEE/ACM Int. Workshop on HW/SW Co-Design, Seattle, Washington, USA, March 15-18, 1998.
- [6] P. Chou, G. Borriello, "Software Architecture Synthesis for Retargetable Real-Time Embedded Systems", Proc. of the 5th IEEE/ACM Int. Workshop on HW/SW Codesign, Braunschweig, Germany, March 24-26, 1997, pp. 101-105.
- [7] Tim Lindholm and Frank Yellin. The Java Virtual Machine Specification. Addison-Wesley, Reading, Massachusetts, 1996.
- [8] J. M. Cardoso, and H. C. Neto, "Towards an Automatic Path from Java Bytecodes to Hardware Through High-Level Synthesis", 5th IEEE ICECS, Lisbon, Sept. 7-10, 1998 (to appear).
- [9] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, Compilers: Principles, Techniques and Tools, Addison Wesley, 1986.
- [10] IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1993.
- [11] Sun Microsystems™, The Java developer's Kit, 1997. version 1.1. See <<http://java.sun.com/products/jdk/1.1/>>.
- [12] CADDY-II, FZI, Karlsruhe, version 5.10. See <<http://www.fzi.de/sim/people/bringmann/caddy/caddy.html>>.
- [13] Synopsys Inc., "Design Compiler v3.3a", April 8, 1995.
- [14] IMS, "1.2 µm CMOS Gate Forest Static Cells", 15-Dec-93.
- [15] _____, "3rd Annual VIUF Design Contest", 1997. See <http://rassp.scrs.org/Fall97_VIUF/Design_Contest/>.
- [16] G. de Jong, et al., "Background memory management for dynamic data structure intensive processing systems", Proc. of the IEEE ICCAD, San Jose CA, Nov. 1995, pp. 515-520.