System Design Based on Single Language and Single-Chip Java ASIP Microcontroller

Sérgio Akira Ito
Inst. of Computer Science
UFRGS - Brazil
saito@inf.ufrgs.br

Luigi Carro
Electrical Engineering
UFRGS - Brazil
carro@iee.ufrgs.br

Ricardo Pezzuol Jacobi
Dept. of Computer Science
UnB - Brazil
rjacobi@cic.unb.br

Abstract

Microcontrollers have been playing an important role in the embedded market. However, the designer of microcontroller based systems must deal with different languages and tools in the hardware and software development, despite of their distinct design process. This paper presents a new design strategy to implement embedded applications described uniquely in Java, while maintaining software compatibility throughout the design process. Moreover, the target hardware is a single chip FPGA, taking benefit from their low cost and easy reconfiguration to customize the microcontroller. This papers presents the environment and some results of system synthesis.

1 Introduction

The requirements of new applications (such as multimedia processing) are driving the market of embedded processors to the range of powerful 32-bit devices [1]. On the other side, traditional 8-bit microcontrollers are advancing into new products boosted by their low cost and new capabilities, making the estimated shipments grow up [2].

However, time-to-market pressures and the proliferation of incompatible devices make the design of embedded software hard for consumer device developers. Strategies for reducing costs on system design based on embedded microcontrollers are still welcome, since the development of software for these devices presents some odd challenges.

1.1 The need for a single language

In most applications of microcontrollers, there are at least three design languages: the assembler of the chosen processor; some high level language like C; and more recently an abstract system specification language to simulate the proposed solution. The assembly language is needed because some functions must still be written by hand, either because of the required speed, or because of the limited memory. This means that whenever a new microcontroller comes into the market, the decision to change the processor must be carefully taken, because the availability of a new efficient C compiler is not granted, and the cost in training engineers in the new assembler development is high.

Embedded system developers have embraced Java over the past few years because this technology can provide high portability and code reuse for their applications [3]. Also, if one could learn a single language to specify a system and to program it, then the software design cycle would certainly shrink.

1.2 The need for a single reprogrammable chip

While most people in the desktop market worries about Java's performance, another challenging problem is to fit an embedded Java application into an available Field Programmable Gate Array (FPGA). The system to be designed could then benefit from the decreasing costs of FPGAs, increasing densities, application scalability, and high performance attained in a single chip solution. The standardization of a single FPGA as the hardware and a single language to develop all modeling and embedded software could bring design costs down for all companies working with applications such as embedded control and home electronics.

However, we have seen that most current solutions focus at embedded systems with enough resources to fit a Real Time Operating System (RTOS), garbage collection, a specific Java Virtual Machine (JVM), multithreading support, and so forth [4, 5]. A methodology to make Java technology available to microcontroller-like embedded applications, based on low cost circuits and boards is missing. This kind of application demands integrated functions, small memory footprint (no more than a few KB of RAM) and low-power constraints (e.g. portable equipment). Therefore, dynamic linking and object management can be costly to be supported by the runtime environment.

1.3 SASHIMI proposal

To address these resource constrained embedded applications we propose a new approach, based on a design environment for specific applications and on the generation of Java microcontrollers, SASHIMI (Systems As Software and Hardware In MIcrocontrollers). In this environment the designer provides a Java application to be analyzed and optimized to run on a Java Application Specific Instruction Set Processor (ASIP), plus an optional Application Specific Integrated Circuit (ASIC), both synthesized in single FPGA chip. This approach is also characterized by highly integrated functions, simpler runtime environment, no new compiler development, software compatibility and cross-platform design possibilities.

This paper is organized as follows: some current solutions for specific Java applications are presented in section 2. Section 3 gives an overview of the SASHIMI methodology and embedded Java application design. Some concepts and rules to application modeling are presented in section 4. Section 5 discusses some characteristics of the design and implementation of the Java microcontroller. Finally, future work and conclusions are presented in section 6.

2 Related work

The suitability of a Java platform to implement embedded applications for devices with 4 to 16-bit CPUs and with limited amount of memory is the first issue when considering Java as a design architecture [3]. The Java Card Platform is targeted to develop applications that run in environments as small as 512 bytes of RAM, 16K of ROM and 8-bit CPUs. This platform has a two-part JVM, a reduced API, and supports dynamic object creation. It is exclusively targeted to applications like smartcard, because it depends on card acceptance devices to run the applications [6].

Some lightweight JVMs have been designed to run dynamically loaded applications in embedded systems, as the Hewlett-Packard's JVM [7], Kaffe [4] and Spotless System [8]. However, as reported by SunLabs in the Spotless project, designing a Java platform to devices with a few kilobytes of RAM available both to the runtime environment and to applications may be an arduous task. Sun also claims that PicoJava microprocessor can be configured to embedded market. However, since its microarchitecture incorporates sophisticated mechanisms to gain performance [9], it seems impossible to PicoJava to be smaller than a classic microcontroller.

Compiling Java source or bytecodes to native code could overcome poor performance and could help to maintain the runtime environment smaller [10]. However, this approach requires a compiler (or compiler back-end) for each new device, as it is today in microcontroller market, making software portability harder to attain.

The JASIP architecture [5] is an interesting solution to native execution of multithreaded Java applications. This architecture requires a processing element for each thread, early class hierarchy resolution, and supports object allocation at global memory and thread scheduling. However, its prototype needs an FPGA board with 100K gates and a module of memory to be implemented, mismatching with the single chip and low-power applications concept.

Java has also been studied as a specification language for embedded systems [11], and to specify hardware-software systems [12]. In this research area, Cardoso presented an approach to synthesize hardware from system specifications using Java and some restrictions in the application modeling [13]. However, this codesign methodology is targeted to generate pieces of hardware to accelerate Java applications, making no assumption of resource constraints.

The SASHIMI approach shares some concepts with these works as system specification using a subset of Java, CPU customizing, and early resolution of references, as will be presented in the next section. However, our focus is to provide a general methodology to support the development of embedded applications, based on a single language and single chip approach to reduce costs.

3 Designing with SASHIMI methodology

The next subsections describe the SASHIMI methodology to design (model, simulate and program) Java applications targeted to microcontrollers. Some modeling constraints are discussed bellow, because the full support for Java language is not available for the current environment.

3.1 Modeling constraints

The SASHIMI environment assumes that the modeling of the system will be made in the Java language, agreeing with the constraints below. The automated tasks performed by the development environment are code analysis, performance estimation and critical routine identification. A set of tools can help the designer to predict the final system performance and costs.

As mentioned early in this paper, not all Java applications can be synthesized using the current SASHIMI environment. The nature of target applications induces some constraints in the coding style. These restrictions are similar to those that make the synthesizable version of VHDL smaller than the full language. Therefore, a Java application to be synthesizable in SASHIMI must conform to the following conditions:

- the *new* operator is not allowed, since it would require virtual memory management;

- only *static* methods and variables are supported, for the same reason above;
- no recursive methods are allowed, since this would require dynamic memory management;
- interfaces are not supported, because dynamic binding represents additional cost at runtime;
- floating-point arithmetic is not allowed, although one could enable it at the price of a larger FPGA;
- multiple threads are not supported, because most of microcontroller applications can be described in a single thread, lowering design and hardware costs.

The conditions above make SASHIMI targeted to handle typical microcontroller applications, like intelligent sensors, garage doors, no-break controllers, identification and security systems and the like. This type of applications forms the core of the microcontroller business in small and medium size companies.

3.2 The goal of code analysis

Figure 1 illustrates the entire design flow from the Java source code to the synthesized chip of the microcontroller. In the SASHIMI environment the user will begin with Java files representing the application source code. In this phase, the development process follows the traditional edit-compile-run cycle in a desktop computer with a standard Java Development Kit. In this scenario, running the application is equivalent to simulating it in a still unavailable hardware, emulated by the Java interpreter in this case. During the simulation phase, the designer can use some predefined classes (within threads are allowed) to model the behavior of some necessary peripherals. Later, in the synthesis step these classes will be substituted by some code providing the interface with real components.

When the designer considers the application is ready, the input vectors provided by the user and the output vectors obtained by the simulation are saved together to be used later by the bytecode validation phase. The *Code Analyzer* tool takes the executable code (class files) to perform some analysis like performance and size occupied by the hardware. Performance is estimated using method call information, number of instructions per method, cycles per instruction, and probable frequency to be reached after synthesis. FPGA area and frequency are estimated by using previous results obtained by synthesis with different number of instructions. Finally, the designer can verify whether the reports generated agree with system requirements or not. It is necessary to keep in mind that the full Java instruction set is not supported, and the code analysis must provide the information to ASIP generation and code adaptation.

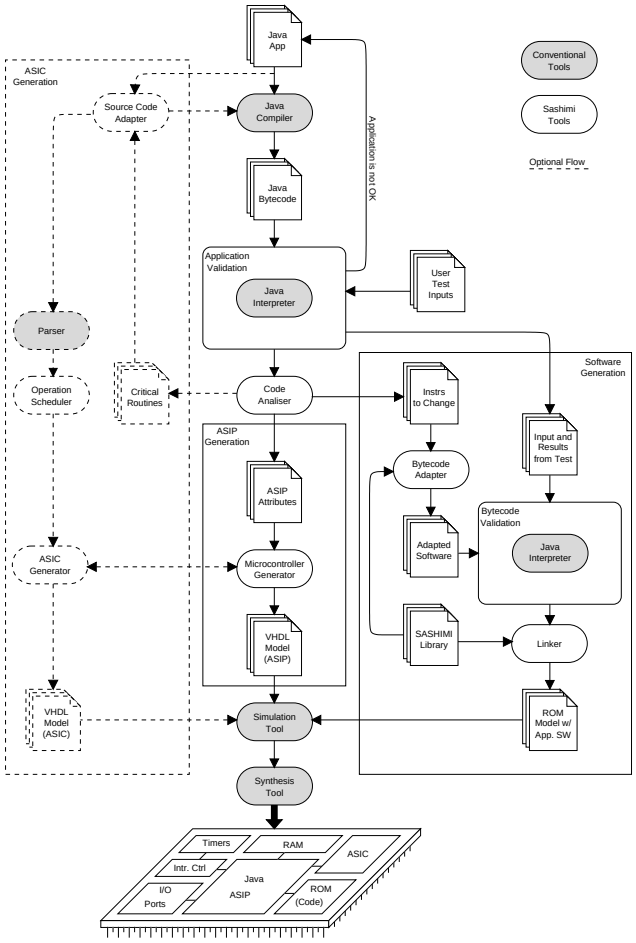

Figure 1. Design flow of SASHIMI

Adapting bytecodes involves some transformations inside the class files, while semantic modifications are not allowed. This process transforms complex instructions (e.g. *tableswitch*, *lookupswitch*) in a sequence of simpler instructions. Class files structures keep inside the maximum stack size for each method in a class [14], that must be recalculated to correct execution of the adapted code. After this step, a new simulation phase is needed, to validate the bytecode transformation. But now, this process uses the previously stored output test vectors, and can be automatic.

The criteria that drives the bytecode substitution process are the instructions supported by the target microcontroller; the size and speed requirements; the amount of use of each instruction (perhaps one can change a rarely used hardware multiplier by a software routine saving circuit area); and the memory size available to store the application code.

At this point of the design flow one has only a pool of class files adapted to be compatible with the specific instruction set that will be implemented in the Java ASIP. The next

step removes unnecessary information (e.g. line numbers structures), resolves class hierarchy, links and converts the application code and system level code (libraries to access hardware resources and to execute complex instructions) in a unique ROM image.

When performance estimation does not match with the application requirements, the critical part of the code can be identified by the *Code Analyzer* tool, and the designer can choose whether some instructions will be discarded or not. Alternatively, the designer can provide an ASIC specification to be integrated into the hardware in order to improve performance. The communication between the ASIC and the Java ASIP depends of the ASIC behavior, but can be made through the stack or heap.

3.3 System model and microcontroller generation

The ASIP must be able to run the code stored in ROM. From attributes (e.g. instructions to implement and RAM size) extracted by *Code Analyzer*, a VHDL model of the optimized microcontroller is generated. Finally, this model is simulated using the target application and synthesized in FPGA by a synthesis tool as Maxplus-II [15] or Synopsys [16]. The code is stored in a non-volatile memory and only the necessary classes are linked. Updating of the application is made by replacing this memory module or reprogramming it if using erasable memories.

The microcontroller can also be updated because it is synthesized in FPGA. Updating the processor functions can be done either by creating new instructions required by that specific application or by synthesizing specific hardware functions, like timers, watchdogs or even digital filters.

4 Modeling an application

We have described a Java code implementing the integration algorithm of the PODOS system [17], as illustrated in figure 2. This system is an integrated circuit that measures the distance a person walks or runs. It is placed on the shoe and communicates with a display on the person's wrist. The complete model of PODOS system has nine classes modeled in Java. Some of these classes are only for simulation purposes, as Display (representing a LCD display) and Keyboard (simulating user input). The software is represented by classes that perform some processing (as the class *Integrator* in figure 2) or by classes that control user interaction.

For the piece of code presented in figure 2, an unsupported instruction is the integer division (*idiv*). In this case, a loop executing subtractions or alternatively an arithmetical right shift could solve the problem. Because the computation of speed and distance are frequently made in this system, the *ishr* instruction was the preferred solution. Other instructions like *tableswitch* were changed to a sequence of

```
public class Integrator {
   . . .
   public static void receiveNewValue(int newValue) {
      if (evenTime) { acceleration0 = newValue; evenTime = false; }
      else { acceleration1 = newValue; evenTime = true; }
      calculeSpeed();
      calculeDistance();
   }
   public static void calculeSpeed() {
      if (evenTime) { speed0 = (acceleration1 + acceleration0)/2; }
      else { speed1 = (acceleration0 + acceleration1)/2; }
   }
   public static void calculeDistance() {
      distance = distance + (speed0 + speed1)/2;
   }
}
```

Figure 2. Valid Java code to SASHIMI

conditional branches. The code was placed in ROM, and a RAM map was generated because some instructions as *putstatic* and *getstatic* can access some positions of memory representing the constant pool entries.

5 Java microcontroller characteristics

The techniques used to store code and class information in ROM play an important role on microarchitecture design. For example, in methods calls the information about the number and type of each parameter and local variable must be easily accessed to allow simple hardware structures. The organization of runtime information in RAM also can help decrease the footprint memory of an application because earlier address resolution allows implementing this type of instruction directly in hardware.

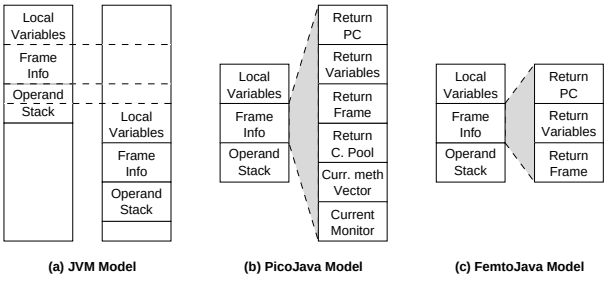

Figure 3. Frame allocation onto stack.

Frames to method calls must have a structure similar to figure 3(a), where *local variables* are locals of the method, *frame info* holds return program counter value and frame pointers, while *operand stack* is used to store intermediate results of operations and parameters to method calls. As we can see in figure 3(b), the PicoJava processor must manage several information on method calls. The model used by our microcontroller, called FemtoJava, is simpler because stores fewer information in the frame allocation process, while still maintain the JVM model behavior.

Table 1. Synthesis results

	Version	Logic Cells	Freq. (MHz)	Supported Instructions
FemtoJava	PODOS	488	7.97	23
	V1	838	6.11	40
	V2	917	6.13	43
MCS8051	No multiplier	500	3.16	12
	Multiplier	695	2.46	13
Device: EPF10K20R240-4 (Altera Flex 10K20)				

To validate our ideas we have implemented an 8-bit stack-based microcontroller to run Java bytecode. The major characteristics of this device are reduced bytecode instructions set, Harvard architecture, orthogonality of execution, small size, and easy introduction and removing of instructions. A first version of this microcontroller was accomplished by the authors getting a core with 43 instructions implemented [18]. The instructions supported are basic integer arithmetic and bitwise operations, conditional and unconditional jumps, load/store instructions, stack operations, and four extended bytecodes to arbitrary load/store. In this core all instructions were executed in 7 cycles, because several instructions are memory bounded.

In order to make some tests, the integration algorithm of the PODOS system was implemented and simulated. Results in table 1 suggest that some applications will need only some few instructions and will demand fewer logic cells to be implemented. The V2 version is the largest one, because it has shift instructions, occupying about 79% of EPF10K20R240-4 Altera's FPGA. The gain regarding the occupied area is more evident between the PODOS and V2 version of the FemtoJava. The relationship between these results and the methodology applied reflect the possibility of excluding instructions with high cost in hardware.

Table 1 also presents a rough comparison of FemtoJava with the 8051 processor with reduced instruction set (12 instructions). The MCS8051 core including mul instruction is smaller than the V1 version of FemtoJava with 40 instructions (including imul). Although with more area, speed of the FemtoJava is much greater than the 8051 in the same FPGA with much more instructions.

6 Conclusions

We presented a new system design approach where only the Java language is necessary for the tasks of system modeling, simulating and programming, based on a single chip Java microcontroller. The SASHIMI methodology allows high integration, confirmed by synthesis results, and easy maintenance, because of the use of FPGAs as the underlying hardware, and Java as the only language interface for the entire design flow.

For the future, we can also consider multithreading, while still keeping the runtime environment as small as possible. To maintain applications with bounded resource consumption some restrictions are also necessary - as preallocation of all objects, no *new* operators inside loops, and no structures dynamically extensible.

The availability of FPGAs chips with greater capacity will allow new features to be included in our microcontroller. Therefore a family of cores can be designed to optimize some features, depending of application needs.

References

[1] M. Schlett. Trends in Embedded-Microprocessor Design. *Computer*, 31(8):44–49, Aug. 1998.

[2] R. Grehan. 8-bit microcontrollers grow up... and down. *Computer Design*, 36(5):72–78, May 1997.

[3] D. Mulchandani. Java for Embedded Systems. *Internet Computing*, 31(10):30–39, May 1998.

[4] M. Barr. A Free Java Virtual Machine for Embedded Systems. In *Proceedings of ESC'98*, San Jose, CA, Nov. 1998. ESC On-line.

[5] M. Mrva, K. Buchenrieder, and R. Kress. A Scalable Architecture for Multi-threaded JAVA Applications. In *Proceedings of DATE'98*, pages 868–874, Paris, Feb. 1998. IEEE.

[6] J. Card. Java Card Applet Developer´s Guide. Technical Report Revision 1.12, Sun Microsystems, Inc., Aug. 1998.

[7] D. Clark. HP Enters the Java Fray. *Computer*, 31(6):19, June 1998.

[8] A. Taivalsaari, B. Bush, and D. Simon. The Spotless System: Implementing a Java™ System for the Palm Connected Organizer. Technical Report TR-99-77, SunLabs, Feb. 1999.

[9] H. Mcghan and M. O'Connor. Picojava: A Direct Execution Engine for Java Bytecode. *Computer*, 31(10):22–30, Oct. 1998.

[10] P. Bothner. Compiling java for embedded system. available at http://www.cygnus.com/news/whitepapers/ compiling.html, Jan. 1999.

[11] J. S. Young et al. Design and Specification of Embedded Systems in Java Using Successive, Formal Refinement. In *Proceedings of DAC'98*, pages 70–75, San Francisco, CA, June 1998. ACM.

[12] R. Helaihel and K. Olukotun. Java as a Specification Language for Hardware-Software Systems. In *Proceedings of ICCAD'97*, pages 690–697, San Jose, CA, Nov. 1997. IEEE.

[13] J. Cardoso and H. Neto. An Approach to Hardware Synthesis from a System Java™ Specification. In *Proceedings of WDTA'98*, pages 149–152, Dubrovnik, June 1998.

[14] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, 1997.

[15] Altera. Altera Corporation. http://www.altera.com, 1999.

[16] Synopsys. Synopsys, Inc. http://www.synopsys.com, 1999.

[17] L. Carro and L. B. de Nale. Podos - Distance Measurement Device. In *Proceedings of SIM'98*, pages 165–168, Bento Gonçalves, RS, May 1998. UFRGS.

[18] S. A. Ito, L. Carro, and R. Jacobi. Issues on Designing Embedded Applications Targeted to a Java Microcontroller. In *Proceedings of ICMP'99*, pages 180–184, Campinas, SP, Aug. 1999. SBMicro.