Reconfiguration of Embedded Java Applications

João Cláudio Soares Otero, Flávio Rech Wagner, Luigi Carro

Universidade Federal do Rio Grande do Sul – Instituto de Informática
Porto Alegre, Brazil
{jcotero,flavio,carro}@inf.ufrgs.br

Abstract

This work presents the development of a coarse grain reconfigurable unit to be coupled to a native Java microcontroller, which is designed for an optimized execution of the embedded application. Code fragments to be accelerated through this unit are identified by profiling the application. The unit is able to explore ILP in a simple way and allows for Java compatibility, while also reducing the number of executed instructions, thus improving the performance with simultaneous energy savings. In many cases, as demonstrated by experiments, it also allows for smaller power consumption.

1. Introduction

The growing of the embedded systems market, together with the increasing popularity of Java applications for the consumer electronics industry [1], gives margin to a new focus on the embedded computation landscape, which must increasingly consider the issues of design complexity, time-to-market, and software compatibility. Besides that, the requirement of energy savings with simultaneous performance improvement is imposed by the current mobile systems. Traditional RISC processors achieve high performance by applying techniques that result in high power consumption and are not appropriate for the embedded systems domain [2]. In this context, we are developing a large project around FemtoJava [3], a microcontroller designed for the native execution of bytecodes of embedded Java applications.

Since energy savings are one of the main objectives of an embedded system design, we present in this paper Javarray, a coarse grain reconfigurable unit to be coupled to the FemtoJava microcontroller, which is designed for the optimized execution of operand blocks from the more representative basic blocks of embedded applications. These blocks are identified by static profiling of the application.

With profiling analysis applied to the Java code and the use of reconfiguration, we can explore ILP in a simple way and reduce the number of executed instructions, thus improving the performance with simultaneous energy savings. At the same time, software compatibility with the Java application is maintained and a better fit of the application to the embedded systems domain is achieved.

The remaining of this paper is organized as follows. An analysis of related work can be found in Section 2, concerning other architectures that also look for energy savings with simultaneous performance improvements. Section 3 analyzes the properties of embedded Java applications that allow us to derive the main architectural needs of the Javarray reconfigurable unit. Experimental results are presented in Section 4. Section 5, finally, draws main conclusions and introduces future work.

2. Related Work

Many reconfigurable architectural designs have been proposed for various goals. Some architectures have a more general goal, trying to explore distinct levels of parallelism for desktop applications, like Piranha [4] and TRIPS [5]. Dealing with the embedded market, the Javarray architecture takes another approach and uses alu-sized processor elements, proposing a reconfigurable architecture for obtaining energy savings through optimization of critical code. Similar ideas are being explored with very promising results in related works.

In the XiRisc processor [6], the algorithms' portions with more intensive computations are mapped to the reconfigurable unit, allowing for performance speedups from 4.3 to 13.5, while at the same achieving reduction in energy consumption of up to 92%. The work in [7] proposes the development of a reconfigurable logic architecture specifically devoted to dynamic hardware/software partitioning. Through this architecture, and by programming the FPGA to deal specifically with

1-4244-0054-6/06/$20.00 ©2006 IEEE

the performance improvement of software kernels, the authors claim to have reached performance gains between 2 and 4 times, with energy savings with an average of 33% and up to 74%. The ADRES architecture [8] is specifically designed for embedded systems and uses a VLIW processor tightly coupled to a coarse-grained reconfigurable array, which is intended to efficiently execute only computationally intensive kernels of applications.

The Javarray design considers the same idea of optimization of software parts that are dynamically mapped to a reconfigurable architecture, but differs from [6] and [7] by the use of a coarser granularity for the processing elements and also by its optimization strategy, which is chosen through the analysis of the application profile. Differently from the ADRES architecture, Javarray maintains native software compatibility with Java and requires no special compilers.

3. Javarray Architecture

The Javarray architecture acts only at interesting and repetitive points of the dynamic trace of the application, leaving less attractive tasks to the Java microcontroller, to which the array is coupled. In this sense, Javarray is a functional unit tightly coupled with the host processor.

The FemtoJava microcontroller is a stack-based microcontroller specifically designed for the embedded system market, which natively executes Java bytecodes. In this work, two versions of the Java microcontroller are used. The first one is a multi-cycle version, which takes from three to fourteen cycles to execute an instruction. The other is a low-power pipelined version – a classical 5-stage processor, but with the additional presence of registers playing the role of the operand stack and local variable storage (used to keep values of the local variables of a method), instead of using the main memory for this purpose, as in other stack machines. In this work, we compare the use of these two FemtoJava versions as the host processor for the Javarray reconfigurable unit.

Through the analysis of application profiles, we have defined interesting points for applying reconfiguration. We have identified several basic blocks – parts of code without incoming or outcoming branches – where some of them were very repetitive on the execution of the application.

A basic block is composed by one or more *operand blocks*, used as units for reconfiguration, which are well defined by their first and last instruction addresses. After the block mapping to the reconfigurable substrate has been performed, the fetch of the first address corresponding to the first instruction of an operand block fires the execution of the reconfigured block. After the identification of an operand block starting address, the fetch and decoding of the other instructions of the block are not necessary anymore.

The application kernels were selected from various projects of our group. They do not represent current embedded systems benchmarks, but correspond to distinct categories of applications. Thus, SortBubble is an instance of the well known bubble sort algorithm; IMDCT is the Inverse Modified Discrete Cosine Transformation, used for instance in the MP3 decompression algorithm; and Crane is a control algorithm used to control a crane, proposed as a benchmark on the area of system-level modeling and synthesis [9].

In all these application kernels, profiling revealed that a small part of the existing basic blocks stands for most of the executed code. For the IMDCT, 2 out of 17 existing basic blocks represent 40% of the total application instructions, but 94% of the executed code. For the SortBubble, 2 out of 12 basic blocks represent 47% of the code, but 80% of the execution. And for the Crane, 7 out of 117 basic blocks contain 20% of the application code, but stand for 34% of the executed instructions.

Even if a broader analysis of other applications would be desirable, it is not common, because of the existence of branches, that the size of the basic blocks is larger than the ones found on these selected kernels. Furthermore, the format of the Java code, based on a stack machine, produces bytecodes' dependence graphs that are very similar to binary trees. From this knowledge on the code properties, we have noted that a 3x8 structure of processing elements is sufficient to support most of the interesting blocks for reconfiguration. We have also observed a maximum number of two memory reads in a same depth level of this 3x8 structure, as well as a maximum number of 6 instructions that used variable values (from the stack) or constant values, for every two levels. These observations lead to the definition of the general topology of the Javarray architecture, shown in Figure 1.

Since the Java Virtual Machine is based on a stack machine, potentially parallel instructions are serialized, and many data input instructions occur repeatedly in the normal Java instruction flow. The reconfiguration of these instructions in a dependence graph can eliminate this repetition and also explore the whole ILP potential of the application. Thus, for repeated instructions, the result is calculated only once and then used many times when required. Besides that, many data input instructions refer to constants, such that after the instructions have been reconfigured, they do not need to be executed at every basic block's occurrence anymore. It has been found that up to 80% of the input instructions can be eliminated.

Also, when the original instruction flow is transformed into a dependence graph, many instructions in pairs like *istore-iload* and *putstatic-getstatic* that are internally present in the graph can also be eliminated, thus reducing

even more the number of required instructions for the execution of the reconfigured basic block. Adding the savings on the repeated and constant data input instructions to those achieved by the suppression of internal instruction pairs, a reduction of up to 42% on the number of total required instructions has been achieved, in comparison to the complete original basic block code, thus saving processing time and energy.

Based on the previous considerations about the application dependence graphs, the general Javarray topology (Figure 1) consists of an array of 3x8 processing elements, each of them capable of executing the fundamental operations of the Java *bytecodes*, interleaved with interconnections. The array is able to execute one operand block configuration in a unidirectional processing flow. At every two lines, the processing elements are organized in a way to share six input data registers and four memory read registers. There is also a special processing element that is capable of executing operations with three inputs, to perform *iastore* operations – a *bytecode* that commonly appears as the root of many dependence trees in Java applications.

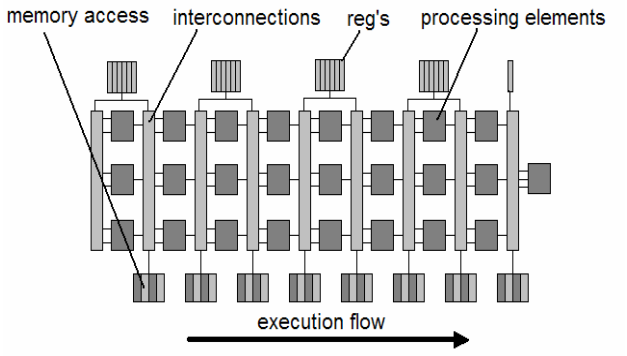

Figure 1: Javarray topology.

4. Experimental Results

To analyze energy and power consumption and performance, a cycle-accurate model of the Javarray coupled to the FemtoJava microcontroller was developed. The cycle-accurate power simulator CACO-PS [10] performs a high-level approximation considering the bit transitions for simulating dynamic power consumption at the gate level, but also considers static power. The results are given in numbers of switching *gate capacitances* and can be calibrated for the current technology. The estimates collected by simulation compare the use of the Javarray, coupled to the two versions of the Java microcontroller as the host processor, against the same execution without the Javarray.

The energy savings shown in Table 1(a) reveal that, for the IMDCT application, with the reconfiguration of 2 basic blocks that represent 40% of the application code and 94% of the executed instructions, we can achieve global energy savings of 64% when Javarray is coupled to the multi-cycle version of FemtoJava. When it is coupled to the low-power version, 41% of energy savings have been achieved. The table also shows that we can obtain energy savings from 47% to 36% for the SortBubble and from 22% to 17% for the Crane application (a control-based application), depending upon the FemtoJava version to be used. Even with energy savings of only 22% and 17%, it must be considered that the reconfigured blocks of the Crane application represent only 34% of the total executed instructions, such that the obtained savings are very close to the potential limit. These energy savings are mainly due to the reduction in the number of executed instructions and to the reduction in the stack and memory accesses allowed by the code reorganization into a dependence tree.

Regarding performance, Table 1(a) shows that the Javarray coupled to the multi-cycle FemtoJava version gives cycle savings between 8% and 63% when compared to the same applications executed without Javarray. Although the low-power pipelined FemtoJava version has a better performance than the multi-cycle one, global savings of up to 53% have been obtained in comparison to the number of required cycles to execute the applications without Javarray. The performance gains are also due in part to the reduction in the number of instructions and in part to the ILP exploration made possible by the instructions' reconfiguration.

Table 1(b) shows the power increase/reduction in the execution of the basic blocks when we couple Javarray to the two versions of FemtoJava, in comparison to the execution of the same blocks without the use of Javarray. Thus, 100% corresponds to the original power consumption of the blocks without Javarray, and the table shows the differences obtained by the use of Javarray when compared to this original consumption. Thus, results smaller than 100% represent power savings.

Although a power increase could be expected, because of the Javarray, it can be observed in Table 1(b) that this increase, in average, is small. In many cases, a simultaneous reduction in power consumption can be obtained together with the energy and performance gains. This is not an absurd and essentially happens when the energy savings are larger than the performance gains. This fact can be observed mainly when the low-power pipelined version is used, because it does not give a large margin to further acceleration of applications as the multi-cycle version does.

ENERGY AND PERFORMANCE			Multicycle	Low-Power	Multicycle	Low-Power
blocks	% code	% exec.	energy savings	energy savings	cycle savings	cycle savings
bb 12 imdct	25%	81%	55%	35%	54%	46%
bb 14 imdct	15%	13%	9%	6%	8%	7%
total	**40%**	**94%**	**64%**	**41%**	**63%**	**53%**
bb 6 sortbubble	14%	36%	23%	11%	23%	0%
bb 7 sortbubble	33%	43%	24%	25%	24%	28%
total	**47%**	**79%**	**47%**	**36%**	**47%**	**28%**
bb 8 crane	1%	10%	6%	5%	6%	0%
bb 31 crane	2%	3%	2%	0%	1%	0%
bb 35 crane	1%	3%	2%	1%	2%	1%
bb 39 crane	1%	7%	4%	4%	4%	0%
bb 44 crane	3%	4%	3%	2%	1%	2%
bb 51 crane	3%	3%	2%	2%	2%	2%
bb 62 crane	9%	4%	3%	2%	3%	2%
total	**20%**	**34%**	**22%**	**17%**	**18%**	**8%**

(a)

POWER		
blocks	Multicycle	Low-Power
bb 12 imdct	**97%**	**45%**
bb 14 imdct	**85%**	**41%**
bb 6 sortbubble	100%	**69%**
bb 7 sortbubble	102%	119%
bb 8 crane	184%	**47%**
bb 31 crane	**66%**	**94%**
bb 35 crane	113%	106%
bb 39 crane	**78%**	**42%**
bb 44 crane	**44%**	126%
bb 51 crane	**89%**	105%
bb 62 crane	**85%**	**98%**

(b)

Table 1: Savings in energy, number of cycles (a), and power (b).

5. Conclusions and Future Work

In this work, we have presented a coarse grain reconfigurable unit aimed at performance increase and energy savings of Java-based embedded applications. The proposed approach explores the intrinsic JVM characteristics in the design of a reconfigurable unit that presents high locality of resources to accelerate the most representative application' basic blocks, selected by static profiling.

Qualitatively, the basic blocks' reconfiguration into the Javarray architecture allows the reduction in the number of instructions to be executed, the exploration of ILP, a smaller number of accesses to the stack and to the memory, a smaller need for fetching and decoding of instructions, and a smaller interaction with the host processor datapath. This way, performance and energy gains have been reached, with simultaneous reduction on power consumption in some cases.

The selection of the most representative basic blocks to be optimized depends on some prior (static) or run-time (dynamic) application analysis. In the case of embedded applications, where there is a previous knowledge of the application, the static profiling analysis is not an obstacle. However, although the present work deals with static reconfiguration, a dynamic approach was always intended and is being investigated. Our group has already obtained results regarding dynamic binary translation for reconfigurable arrays [11].

References

[1] G.Lawton, "Moving Java into Mobile Phones". In: Computer, Vol. 35, n. 6, June, 2002, pp. 17-20.

[2] K.Wilcox, S.Manne. "Alpha processors: A history of power issues and a look to the future". In: Coolchips Tutorial, An Industrial Perspective on Low Power Processor Design in conjunction with Micro-33, Dec. 1999.

[3] S.A.Ito, L.Carro, R.P.Jacobi. "System Design Based on Single Language and Single-Chip Java ASIP Microcontroller". In: Proceedings of Design Automation and Test in Europe. Paris, France, February 2000. IEEE Computer Society Press, 2000. pp. 703-707.

[4] L.A.Barroso et al.. "Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing". In: Proceedings of the 27th Annual International Symposium on Computer Architecture, Vancouver, Canada, June, 12-14, 2000.

[5] K.Sankaralingam et al.. "Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture". In: Proceedings of the 30th Annual International Symposium on Computer Architecture – ISCA-2003. San Diego, California, USA, June, 9-11, 2003. IEEE Computer Society Press, 2003.

[6] A. Lodi et al.. "A VLIW Processor with Reconfigurable Instruction Set for Embedded Applications". In: IEEE Journal of Solid-State Circuits, Vol. 38, n. 11, November 2003.

[7] G. Stitt, F. Vahid. "The Energy Advantages of Microprocessor Platforms with On-Chip Configurable Logic". In: IEEE Design and Test 19, 6, November 2002. pp 36-43

[8] B.Mei, S.Vernalde, D.Verkest, H.De Man, R.Lauwereins. "ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix". In: Proceedings of 13th International Conference on Field Programmable Logic and Application – FPL 2003, Lisbon, Portugal, September 1-3 2003. Lecture Notes in Computer Science 2778 Springer 2003, pp. 61-70.

[9] E.Moser, W.Nebel. "Case Study: System Model of Crane and Embedded Control". In: Proceedings of Design, Automation and Test in Europe, 1999. IEEE Computer Society Press, 1999. pp 721-723.

[10] A.C.Beck, J.C.B.Mattos, F.R.Wagner, L.Carro. "CACO-PS: A General Purpose Cycle-Accurate Configurable Power Simulator". In: Proceedings of the 16th Symposium on Integrated Circuits and Systems Design – SBCCI. São Paulo, Brazil, September 2003. IEEE Computer Society Press, 2003. pp. 349-354.

[11] A.C.Beck, L.Carro. "Dynamic Reconfiguration with Binary Translation: Breaking the ILP Barrier with Software Compatibility". In: Proceedings of the 42nd Annual Conference on Design Automation – DAC'05. San Diego, California, USA, June 13-17, 2005. ACM Press, New York, NY, pp. 732-737.