LOW POWER JAVA PROCESSOR FOR EMBEDDED APPLICATIONS

Antonio Carlos S. Beck and Luigi Carro
Universidade Federal do Rio Grande do Sul - Instituto de Informática - Av. Bento Gonçalves, 9500 - Campus do Vale - Porto Alegre, Brasil

Abstract: This chapter presents a low power architecture of a Java processor. We show that the use of techniques like pipeline and the implementation of the stack in a register bank instead of using the main memory allow aggressive reduction of power dissipation, with a very small area overhead. Besides, thanks to the forwarding technique and to the specific stack machine organization, huge power savings can be obtained when applying this technique to a pipelined implementation of the architecture. Several examples of embedded applications are used to show the power savings obtained through the architecture optimization

Key words: Java, Power Consumption, Stack Machines

1. INTRODUCTION

The embedded system market grows day by day. The production of specific processors to be used inside microwaves, videogames, printers, mp3 players, digital cameras, cellular phones and others appliances, is following the same growing path[1]. Particularly in portable embedded systems, where the battery lifetime is a crucial factor, some special care is needed in terms of power consumption of the system.

Java is becoming increasingly popular in embedded environments. It is estimated that devices with embedded Java such as cellular phones, PDAs and pagers will grow from 176 million in 2001 to 721 million in 2005 [2]. Nevertheless, is predicted that at least 80 percent of mobile phones will support Java by 2006 [3]. As one can observe, the importance of Java in

Please use the following format when citing this chapter:
Beck, Antonio Carlos, S., Carro, Luigi, 2006, in IFIP International Federation for Information Processing, Volume 200, VLSI-SOC: From Systems to Chips, eds. Glesner, M., Reis, R., Indrusiak, L., Mooney, V., Eveking, H., (Boston: Springer), pp. 213-228.

embedded systems is growing. This means a careful look on embedded Java processors and their performance versus power tradeoffs must be taken into account.

In this chapter we show a study about three different architectures capable of executing Java bytecodes and discuss their area, performance and mainly power requirements, focusing on embedded systems applications. We demonstrate that by the use of inexpensive techniques, one can optimize the execution of instructions and obtain a drastic reduction in the energy consumption, by a factor of more than 10. Moreover, it is demonstrated that the use of the forwarding technique, besides increasing the performance, allows further benefits to stack-like architectures, since power consumption is reduced because of the small number of writes in the stack. Less writes in the stack means a smaller number of register or memory writes, hence reducing memory accesses, one of the major sources of power dissipation in embedded processors[4,5,6]. Our experiments are supported by simulation, using three different architectures of the Femtojava Processor[7,8], executing different algorithms used in embedded system domain. The area was computed in number of logic cells, after synthesis of different VHDL versions of the processors.

This chapter is organized as follows: Section 2 shows a brief review of the existing Java processors. In Section 3 we discuss the different architectures of Java machines that will be evaluated, and present the advantages of using the forwarding technique in stack machines. Section 4 presents the simulation environment: the power simulator and the test case algorithms executed in the processors. Section 5 shows the results regarding power consumption, performance and area. The last Section draws conclusions and introduces future work.

2. RELATED WORK

A large number of Java processors aimed at the embedded systems market have already been proposed. Sun's Picojava I [9], a four stage pipelined processor, and Picojava II [10], with a six stage pipeline, are probably the most studied ones. Even though the specification of such processors allows a variable size for the data and instruction caches, and the floating point unit is optional, there is no special care on the underlying microarchitecture in order to reduce the area and power consumption of the system.

The same occurs to others Java processors: Komodo[11], a multithreaded Java microcontroller concerned especially with real time applications; and Traja[12], a dual issue pipelined processor that makes use of instruction reordering to avoid data dependencies. All of these and other examples of

native Java execution machines always focus on obtaining the maximum possible performance, in order to leverage Java execution with RISC and VLIW architectures. However, in the domain of embedded systems, not only plain throughput is the correct metric. Other issues like power dissipation and software compatibility play a major role. This way, the research on low power Java processors, able to maintain enough performance to execute the target application with the smallest possible power budget, is the goal of this work.

3. ARCHITECTURE OF THE JAVA PROCESSORS

The Femtojava processor is a stack-based microcontroller that executes Java bytecodes. General characteristics of the Femtojava processor are: reduced instruction set, Harvard architecture and small size. This processor was designed specifically for the embedded system market. The size of its control unit is directly proportional to the number of different instructions used by the application. From an available tool[7], the Java bytecodes of the application are analyzed, and the control unit is generated, supporting only the instructions used by that application.

The first architecture evaluated is a multicycle version of Femtojava[7] that takes three to fourteen cycles to execute an instruction. Its microarchitecture can be observed in Figure 1.

The pipelined version[8] has five stages: instruction fetch, instruction decoding, operand fetch, execution, and write back, as shown in Figure 2. One of the main characteristics of the pipelined Femtojava is the presence of registers playing the role of operand stack and local variable storage.

The first stage, instruction fetch, is composed by an instruction queue of 9 registers of one byte each. The first instruction in the queue is sent to the instruction decoder stage. The decoder has three functions: the generation of the control word for that instruction, to handle data dependencies and to inform to the instruction queue the size of the current instruction, in order to put the next instruction of the stream in the first place of the queue. This is necessary because of the use of variable length instructions: they can have one or two immediate operands, or none at all. When at least 4 registers in the instruction queue are empty, a word of 32 bits comes from the instruction memory, pointed by the program counter.

The operands fetch is done in a variable size register bank, defined a priori in earlier stages of the design. Stack and the local variable pool of the methods are available in the register bank. There are two registers: SP and VARS. They point to the top of the stack and to beginning of the local variable storage, respectively. Depending on the instruction, one of them is

used as base for the operands fetch. Once the operands are fetched, they are sent to the fourth stage, where they will be executed. There is no branch prediction, in order to save area. All branches are supposed to be not taken. If the branch is taken, a penalty of three cycles is paid.

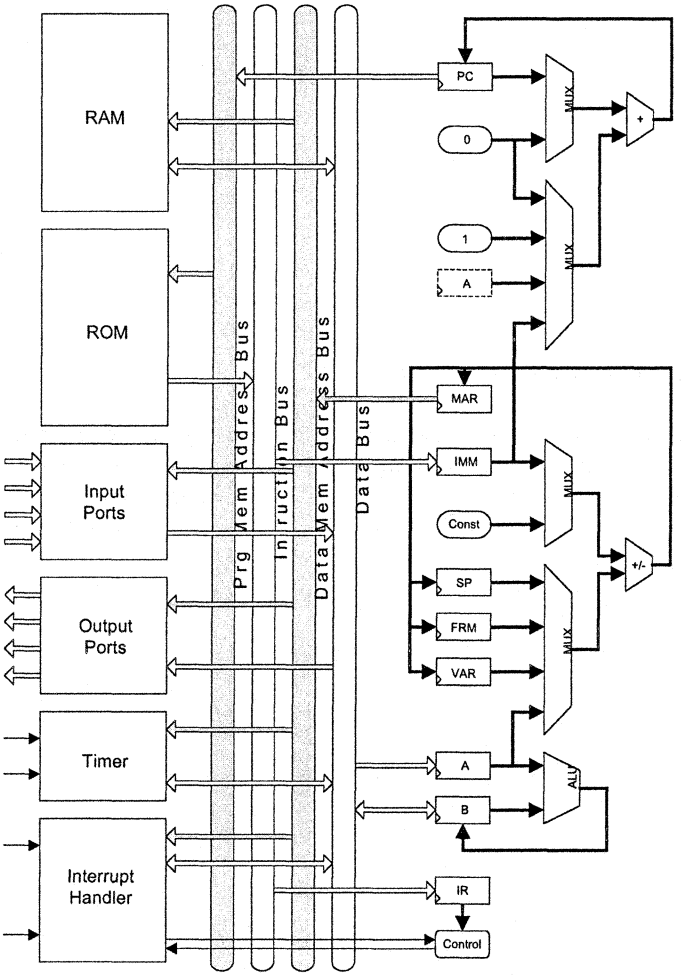

Figure 1. Multicycle Java Processor[7]

The write back stage saves, if necessary, the result of the execution stage back to the register bank, using the SP or VARS as base. As the register bank can not be simultaneously read and written, when an instruction in the fifth stage writes back its result and an instruction in the third stage wants to

read an operand, a bubble is inserted in the pipeline. There is a unified register bank for the stack and local variable pool, because this facilitates the call and return of methods, taking advantage of the JVM specification, where each method is located by a frame pointer in the stack.

Finally, the third architecture evaluated is the same pipelined one described before, plus the increment of the forwarding technique: if there is an instruction in the execution stage that will write its result in the stack, and the following instruction accesses the stack in the operand fetch stage, a true dependency (RAW - read after write) is characterized. One of the solutions is to stall the pipeline, inserting bubbles on it, until the first instruction finishes the write back stage. Another solution is to make use of the forwarding technique[13], passing directly the result from the execution stage to the operand fetch stage.

In operand stack based processors, the use of such technique brings an advantage when comparing to register-like processors: in instructions that manipulate the stack, the operands forwarded to earlier stages will not be used anymore. As a consequence, there is no need to write back these operands to the stack. The result is the reduction on the power consumption, because the diminishing number of writes in the stack.

Two types of forwarding can occur: when the instruction in the execution stage consumes one operand from the top of the stack (like istore, which saves the top of stack in some place of the local variable pool); or when the instruction consumes two operands from the stack (like arithmetic operations: iadd, isub, ior). In the first case, the operand forwarded comes from the execution stage to the operand fetch. In the second case, the second operand comes from the write back stage.

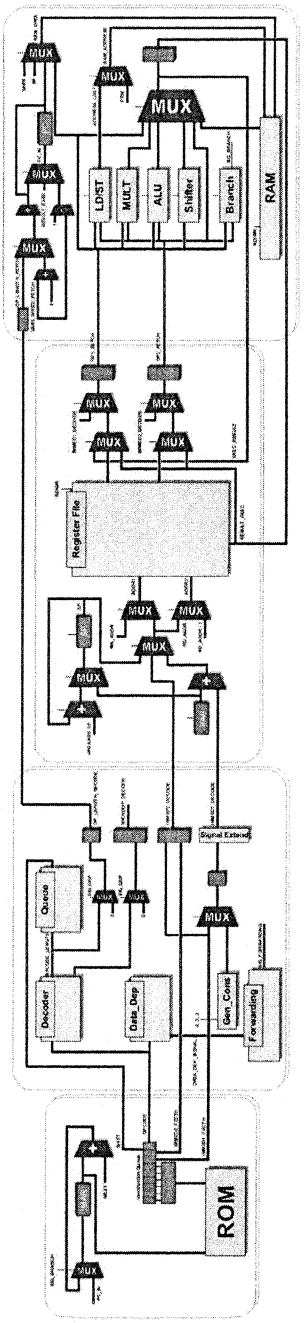

Figure 2. Pipelined Java Processor

4. SIMULATION ENVIRONMENT

4.1 Power Simulator

CACO-PS[8], a compiled-code cycle-accurate simulator, was used to provide data on the energy consumption, memory usage and performance. Power dissipation is evaluated in terms of switching capacitances, and as the processor has separated instruction and data memories, we also included an evaluation module concerning RAM and ROM memories, besides the register bank. This way, one can verify the relative power dissipation of the CPU, instruction memory, and data memory. It is important to measure the impact of each one of these blocks, so that one can better explore the design space.

We used the power simulator to collect the amount of capacitances that switch during the execution of a certain algorithm. This power estimation technique is comparable to the component-based approach [14,15,16].

4.2 Algorithms

Five different types of algorithms were implemented and simulated over the architectures described in Section 3. Sin Calculation, as a representative of arithmetic libraries; sort and search, used in schedulers; IMDCT (Inverse Modified Discrete Cosine Transformation), an important part of the MP3 decompression algorithm; and a library to emulate sums of floating numbers, since the Femtojava processors does not have a floating point unit in order to save area.

Sequential algorithms are used as representative of scalar computations. The first one is Sequential1 that performs the search in a sequential fashion in a non-ordered table. Even if the value has already been found, the algorithm stops just at the end of the table. The second algorithm (Sequential2) stops right after the required value is found in a table that has been previously ordered. The last search algorithm performs a binary search in the vector.

The sort algorithms arrange a set of ten numbers putting them in increasing order. Three different kinds of sort are performed: bubble sort, insert sort and select sort. The floating point sum algorithm makes 20 sums of two floating point numbers and puts its results in a vector in the memory. Finally, the sin algorithm uses the cordic method to calculate the result.

5. RESULTS

Table 1 shows the area occupied by the three different versions of the Femtojava processor. It is important to note that the register bank, used as stack and local variable pool, has 32 registers, the maximum required among all the applications. The computed area includes the control unit that supports all the Femtojava instruction set. The area was evaluated using the Leonardo Spectrum for Windows [17], and it is presented in logic cells. More details about the VHDL implementation can be found in [18].

Table 1. Area occupied by the VHDL version of the architectures

PROCESSOR	*Multicycle*	*Low-Power*	*Low-Power with forwarding*
AREA	1345 LCs	2916 LCs	3016 LCs

Table 2 shows the performance in number of cycles of the processors for each application.

Table 2. Performance of the different architectures

ALGORITHM	NUMBER OF CYCLES		
	Multicycle	*Low-Power*	*Low-Power with forwarding*
Sin	2447	1237	755
Sort – Bubble	6774	3234	2468
Sort – Select	5335	2703	1930
Sort – Insert	4093	2071	1571
Search - Binary	1162	602	403
Search - Sequencial 1	8497	3803	2765
Search - Sequencial 2	7586	2779	1997
IMDCT	140300	61841	40306
Floating Point Sums	30747	18735	14531

Operating at the same frequency, the core of the pipelined version with forwarding is the architecture which has the major power consumption per cycle, since this architecture is the more complex one, with several extra registers. This behavior can be observed in figure 3.

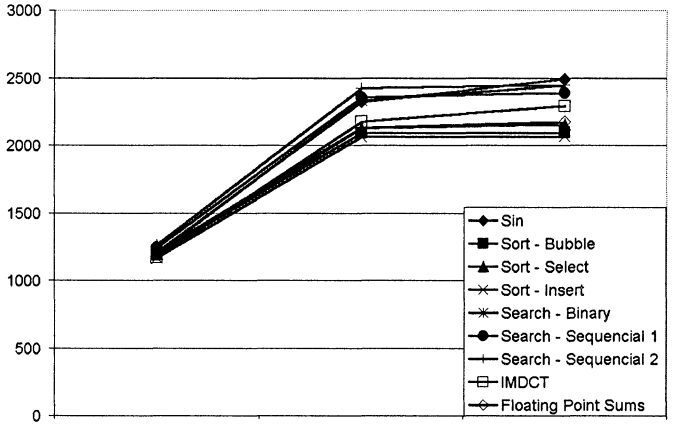

Figure 3. Power consumed per cycle

Even though the power consumed per cycle in the pipelined versions is greater then the multicycle one, the total energy consumption of the system is decreased, because of the high throughput reached by the pipelined versions and the decrease in the number of accesses in the memory.

As it can be observed in figures 4 and 5, the overall energy consumption is drastically reduced in some algorithms, when one changes the architecture of Femtojava. Others, like the algorithm of sums of floating point numbers, do not show such a great gain. The reason is that the implementation of this algorithm makes a lot of method calls, with a consequent increase in the main memory power consumption. Moreover, the algorithm makes an intensive use of the local variable pool of the methods, and although the forwarding technique is applied to the local variable pool as well, the forwarded operands must be written back in the registers, because there is no warranty that they will not be used in the future. The IMDCT and floating point sum algorithms are presented in a separated figure in order to ease the visualization.

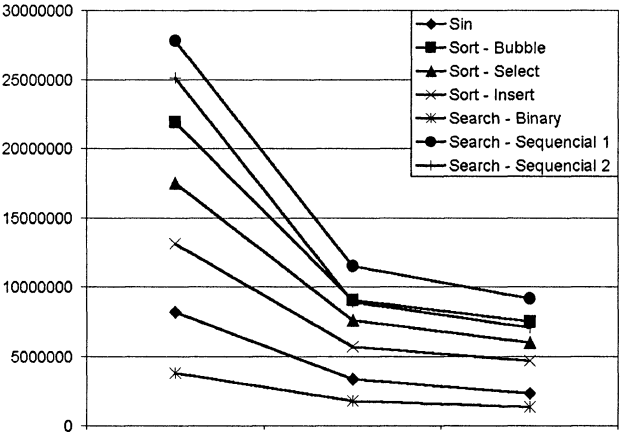

Figure 4. Energy consumption in the three architectures

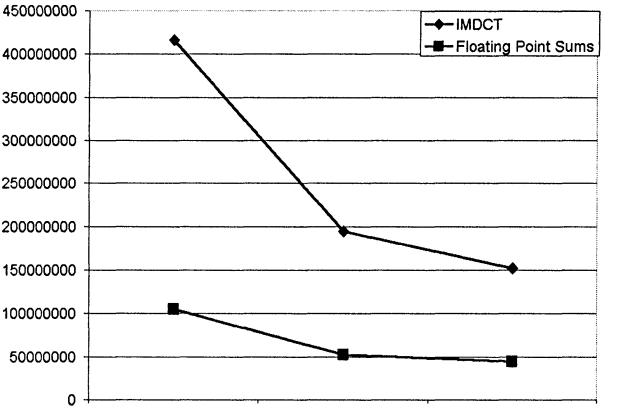

Figure 5. Continuation: Energy consumption in the three architectures

Figures 6 and 7 show the energy consumed because of RAM accesses. As Femtojava Multicycle uses the main memory as stack and variable pool, there is a big difference between this architecture and the pipelined ones. These last, in turn, just make accesses in the main memory in calls and returns of methods, and while executing specific instructions, such as *putstatic* and *getstatic*. Figures 6 and 7 show the advantage of implementing the stack and local variable pool in a register bank instead of using the main memory.

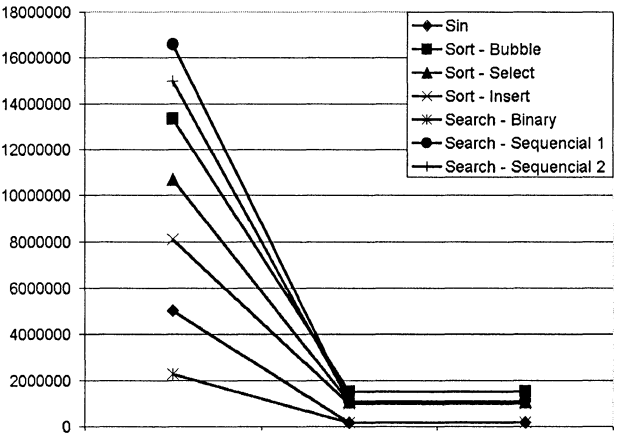

Figure 6. Energy caused by main memory accesses

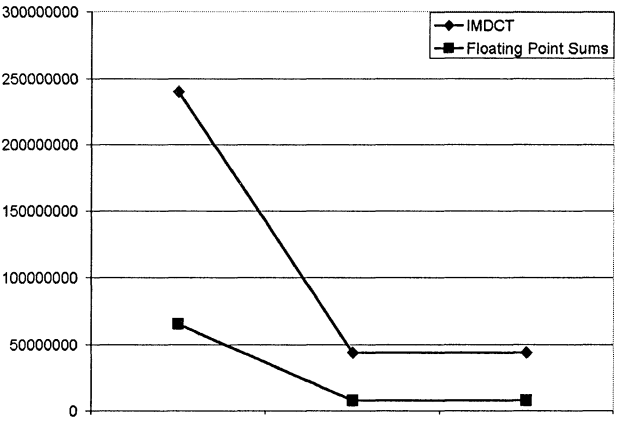

Figure 7. Continuation: Energy caused by main memory accesses

Figures 8 and 9 show the energy consumed by the core of the presented architectures. As one can observe, the power consumed in the core is reduced in the pipelined version with forwarding, in comparison to the version that does not use the technique. The difference on the energy consumption has two reasons: the better utilization of the functional units because of the reduction of pipeline stalls; and the decrease of the number of registers writes, since the average of power consumed in registers writes per

cycle was reduced by almost 70%, showing the advantage of using the forwarding technique.

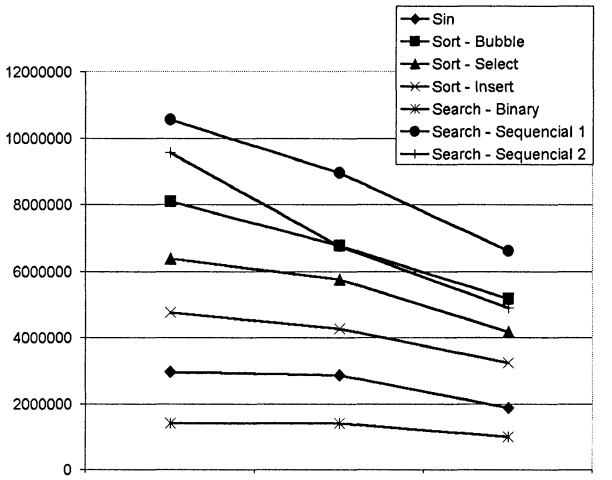

Figure 8. Energy consumed in the cores

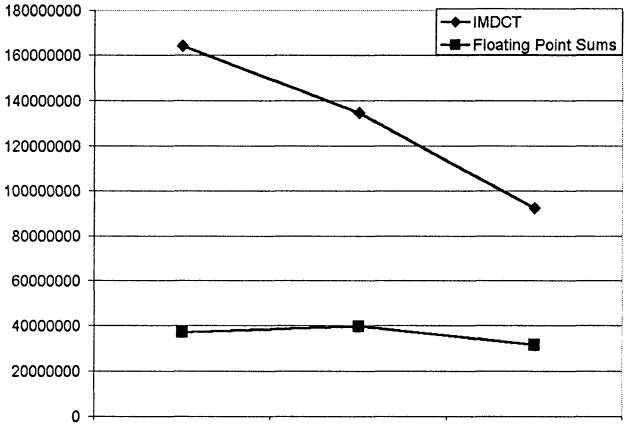

Figure 9. Continuation: Energy consumed in the cores

In embedded applications, many of them with real time requirements, a specific throughput must be warranted for the application. Assuming that this throughput is reached by the multicycle version, the frequency of operation of the pipelined versions can be decreased in order to save power,

since these architectures can execute more instructions per cycle, as was showed in table 1.

Moreover, when assuming that the dynamic power is the dominant in the power consumed in the system, and all the gates of the microprocessor form a collective switching capacitance C with a common switching frequency f, one obtains:

$$P = C.f.Vdd^2 \quad (1)$$

As can be observed in [19], the voltage of the processor Transmeta TM5400 (known as Crusoe) [20], designed for embedded systems, can be decreased by a factor of 4,6% when the operation frequency is reduced by 10%.

Figure 10 shows the relative decrease in the energy consumption when the frequency of the pipelined version (without forwarding) is reduced to reach exactly the same throughput of the multicycle version, and when the voltage is reduced thanks to the decrease in the frequency, using as base the equation (1).

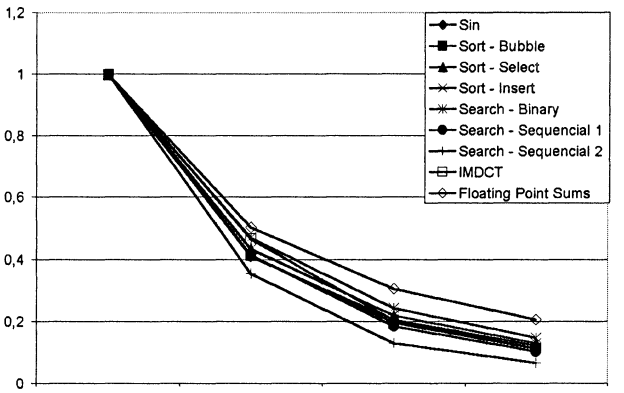

Figure 10. Energy consumed in the cores with the frequency and voltage reduced

Applying the forwarding technique, the energy consumption is reduced even more, as can be observed in figure 11, when it is shown the relative decrease in the power consumption comparing the pipelined version without forwarding with the one that uses the technique. Both had the frequency and voltage reduced to reach the same throughput of the multicycle version.

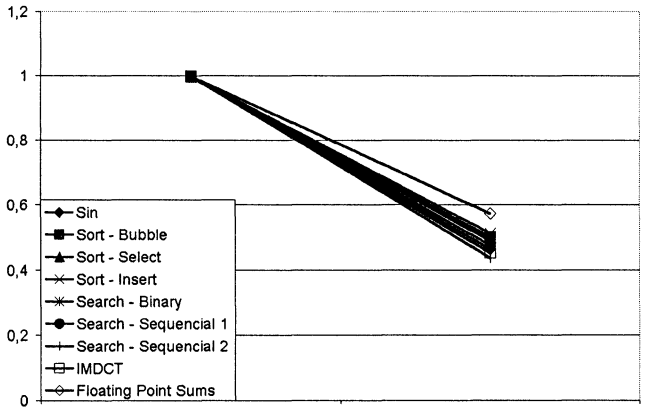

Figure 11. Comparison between the architectures without and with forwarding, both with the frequency and voltage reduced

Finally, we show the advantage of increasing the area of the processor in order to support pipeline and forwarding, trading it for a huge saving in the energy consumption. Figure 12 shows a relative comparison between the area overhead and the average of the power consumed by all the applications.

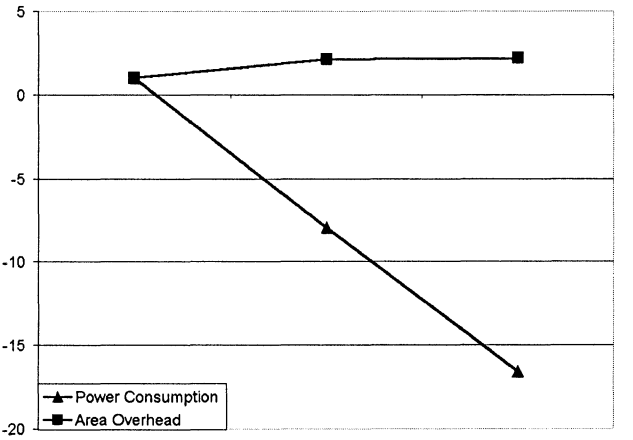

Figure 12. Area versus energy consumption

As can be observed, for twice the area overhead, one obtains a factor of 16 in the energy reduction.

6. CONCLUSION AND FUTURE WORK

We demonstrated that for embedded applications, which need a small stack for its operations, one can obtain a huge decrease in the power consumption with a proportionally small increase in the overall area of the system. Particularly for these machines, the use of the forwarding technique brings a large reduction in the power consumption of the core, taking advantage of the stack architecture and of algorithms that uses the stack intensively, like the IMDCT one. We believe that this behavior is true for others stream-based algorithms, since these are algorithms that make intensive use of the stack.

For future work, we will evaluate more algorithms of the embedded system domain, like an MP3 player. Moreover, we are studying the impact of applying the VLIW technique in Java machines, concerning mainly the power consumption[21], as well as alternatives to increase the VLIW performance, such as the use of techniques to find the instruction parallelism inside the program, like software pipelining, superblocks, and static speculative execution. Other techniques such as the use of a reconfigurable array working together with binary translation to detect the sequence of instructions and reconfigure the array at runtime, and CMP, where a set of processors with the same ISA work together in the same die, will also be evaluated, always giving special concern to the advantages and particularities of stack processors in these techniques.

7. REFERENCES

1. M. Schlett, "Trends in Embedded-Microprocessor Design", *Computer*, vol. 31, n. 8, 1998, pp. 44–49
2. D. Takahashi, "Java Chips Make a Comeback", *Red Herring*, 2001
3. G. Lawton, "Moving Java into Mobile Phones", *Computer*, vol. 35, n. 6, 2002, pp. 17-20
4. V. Tiwari, S. Malik, A. Wolfe, "Power Analysis of Embedded Software: A First Step Towards Software Power Minimization", *IEEE Transactions on VLSI Systems*, vol. 2, n. 4, Dec. 1994, pp. 437–445
5. T. Simunic, G. Micheli, L. Benini, "Energy-Efficient Design of Battery-Powered Embedded Systems", *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED99)*, Aug. 1999
6. G. Chen, R. Shetty, M. Kandemir, N. Vijaykrishnan, M. Irwin, "Tuning garbage collection for reducing memory system energy in an embedded java environment", *ACM Transactions on Embedded Computing Systems*, vol. 1, n. 1, Nov. 2002, pp. 27-55
7. S.A. Ito, L. Carro, R.P. Jacobi, "Making Java Work for Microcontroller Applications", *IEEE Design & Test of Computers*, vol. 18, n. 5, 2001, pp. 100-110
8. A.C.S. Beck, J.C.B. Mattos, F.R. Wagner, L. Carro, "CACO-PS: A General Purpose Cycle-Accurate Configurable Power-Simulator", *16th Brazilian Symp. Integrated Circuit Design (SBCCI 2003)*, Sep. 2003

9. J. M. O'Connor, M. Tremblat, "Picojava-I: the Java Virtual Machine in Hardware", *IEEE Micro*, vol. 17, n. 2, Mar-Apr. 1997, pp. 45-53
10. Sun Microsystems, *PicoJava-II Microarchitecture Guide*, Mar. 1999
11. J. Kreuzinger, R. Marston, Th. Ungerer, U. Brinkschulte, C. Krakowski, "The Komodo Project: Thread-based Event Handling Supported by a Multithreaded Java Microcontroller", *25th Euromicro Conference (EUROMICRO)*, Sep. 1999, pp. 2122-2128
12. N. Shimizu, M. Naito, "A Dual Issue Queued Pipelined Java Processor TRAJA-Toward an Open Source Processor Project", *Proceedings of Asia Pacific Conference on ASIC (AP-ASIC)*, 1999, pp. 213-216
13. J. L. Hennessy, D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, 3th edition, 2003
14. V. Dalal, C. P. Ravikumar, "Software Power Optimizations in an Embedded System". *VLSI Design Conference*, IEEE Computer Science Press, Jan. 2001, pp. 254-259
15. K. Choi, A. Chatterjee, "Efficient Instruction-Level Optimization Methodology for Low-Power Embedded Systems". *International Symposium on System Synthesis*. Montréal, ACM, Oct. 2001, pp 147-152
16. R. Chen, M. J. Irwin, R. Bajwa, "Architecture-Level Power Estimation and Design Experiments". *ACM Transactions on Design Automation of Electronic Systems*, vol. 6, n. 1, Jan. 2001, pp 50-66
17. Leonardo Spectrum, available at homepage: http://www.mentor.com/synthesis
18. V. Gomes, A.C.S. Beck; L. Carro, "A VHDL Implementation of a Low Power Pipelined Java Processor for Embedded Applications". *X Workshop Iberchip*. Cartagenas, mar. 2004.
19. J. Pouwelse, K. Langendown, H. Sips, "Dynamic Voltage Scaling on a Low-Power Microprocessor", *The Seventh Annual International Conference on Mobile Computing and Networking*, 2001, pp. 251-259
20. Transmeta Corporation, *Tm5400 processor specifications*, http://www.transmeta.com
21. A.C.S. Beck, L. Carro, "A VLIW Low Power Java Processor for Embedded Applications", *17th Brazilian Symp. Integrated Circuit Design (SBCCI 2004)*, Sep. 2004