A VLIW Low Power Java Processor for Embedded Applications

A.C. S. Beck and L. Carro
Universidade Federal do Rio Grande do Sul
Instituto de Informática – Av. Bento Gonçalves, 9500 – Porto Alegre/RS – Brazil
e-mail: caco,carro@inf.ufrgs.br

Abstract—**This paper presents a pioneer VLIW architecture of a native Java processor. We show that, thanks to the specific stack architecture and to the use of the VLIW technique, one is able to obtain a meaningful reduction of energy consumption, with small area overhead, when compared to other ways of executing Java in hardware. The underlying technique is based on the reuse of memory access instructions, hence reducing power during memory or cache accesses. The architecture is validated for some complex embedded applications like IMDCT computation and other data processing benchmarks. Up to 3 times of savings in energy consumption is presented, when comparing the pipelined Java architecture against a VLIW version of the same processor**

Index Terms—**VLIW, Java, Embedded Systems, Power Consumption**

I. Introduction

The embedded system market is expanding. The research and production of specific processors to be used inside cellular phones, mp3 players, digital cameras, microwaves, videogames, printers and others appliances is following the same growing path [1]. Moreover, the complexity of these embedded systems, which are offering more and more functions to the user, like Internet access, color display, audio and video reproduction, among others, is increasing as well [2]. These applications require systems with enough processing capabilities to handle with their tasks.

In the same way, 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 [3]. Nevertheless, it is predicted that at least 80 percent of mobile phones will support Java by 2006 [4]. As one can observe, the presence of Java in embedded systems is becoming more significant. This means that current design goals might include a careful look on embedded Java processors, and their performance versus power tradeoffs must be taken into account.

In this paper we show a pioneer Java VLIW architecture, comparing it with different architectures capable of executing Java bytecodes and discussing their area, performance and mainly power requirements, focusing on embedded systems applications. We demonstrate that by the use of the VLIW technique, one can optimize the execution of instructions and obtain a meaningful reduction in the energy consumption when the algorithm presents a high level of parallelism. Moreover, it is shown that the use of the VLIW technique further benefits stack-like architectures and reduces the power consumption, thanks to the reduction of memory accesses, one of the major sources of power dissipation in embedded processors [5]. Furthermore, the technique here presented can be used in other areas, such as Java compilers and virtual machines. Depending on the VLIW version employed and the level of parallelism available in a given application, up to 4 times in performance gains is presented when comparing to the simple pipelined processor. In the same way, up to 3 times of energy savings is demonstrated.

This paper is organized as follows: Section 2 shows a brief review of the existing Java and VLIW processors. In Section 3 we discuss the different architectures of Java machines that will be evaluated, and present the advantages of using the VLIW 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.

II. Related Work

A great number of Java processors aimed at the embedded systems market has already been proposed. Sun's Picojava I [6], a four stage pipelined processor, and Picojava II [7], with a six stage pipeline, are probably the most studied ones. Even though the organization 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, like Komodo [8], a multithreaded Java microcontroller concerned especially with real time applications.

All of these and other examples of native Java execution machines always focus on obtaining the maximum possible performance. 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.

Concerning VLIW machines, [9] proposed a stack processor based on the VLIW technique for real time multimedia network system and data processing. Sun Microsystems proposed in [10] the MAJC architecture, which exploits the parallelism in multiple levels: instruction, data, thread and process, through vertical and speculative multithreading, chip multiprocessing and VLIW. Other VLIW processors aimed at DSP were developed, like Viper [11], Fujitsu FR500 [12] and Texas TMS320C6x [13].

However, none of these examples of VLIW executes di-

rectly the Java bytecodes. As a consequence, they do not explore the advantages of the search of parallelism and execution of Java bytecodes in a stack-like architecture using the VLIW technique. This way, the research of architectures 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.

III. THE JAVA PROCESSORS

The Femtojava processor [14] is a stack-based microcontroller that executes Java bytecodes. General characteristics of the processor are: reduced instruction set, Harvard architecture and small size. This processor was designed specifically for the embedded system market. The first architecture evaluated is a multicycle version [14] that takes from three to fourteen cycles to execute an instruction.

The second architecture is the pipelined version [15], which has five stages: instruction fetch, instruction decoding, operand fetch, execution, and write back, as shown in Figure 1. One of the main characteristics of this architecture is the presence of registers playing the role of operand stack and local variable pool (used to keep values of the local variables of a method). We call this architecture of Low-Power, for reasons to become clear next.

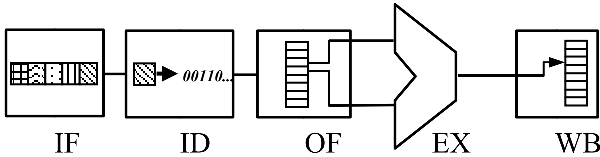

Fig. 1. Pipelined Femtojava Processor [15]

The first stage, instruction fetch, is composed by an instruction queue of 9 registers. The first instruction in the queue is sent to the instruction decoder stage. The decoder has four functions: to generate the control word for that instruction, to handle data dependencies, to analyze the forwarding possibilities and to inform to the instruction queue the size of the current instruction, in order to allocate 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.

Operands fetch is done in a variable size register bank, defined a priori in earlier stages of the design. The operand stack and the local variable pool of the methods are available in the register bank. There are two registers: SP and VARS, which 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 operand fetch. Once the operands are fetched, they are sent to the fourth stage, where the operation is 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 must be paid.

The write back stage saves, if necessary, the result of the execution stage back to the register bank, again, using the SP or VARS as base. 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.

The Low-Power Java processor uses the forwarding technique [16], which brings an advantage when comparing to Load-Store based 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 number of writes in the stack is reduced. In [15] we show a gain factor of 8 concerning energy consumption with a minimal area overhead, thanks to the use of the forwarding technique.

The VLIW processor is an extension of the pipelined one. Basically, it has its functional units and the instruction decoders replicated. The additional decoders do not support the instructions for call and return of methods, since they are always in the main flow. The local variable storage is placed just in the first register file. When the instructions of other flows (instructions located in any slot but the first one in the VLIW packet, which are executed in parallel) need the value of a local variable, they must fetch it from the register bank in the main flow. Each instruction flow of the VLIW packet has its own operand stack, which has less registers than the main stack, since the stacks for the secondary flows do not grow as much as the one in the main flow does.

The VLIW packet has a variable size, avoiding unnecessary memory accesses. A header in the first instruction of the word informs to the instruction fetch controller how many instructions the current packet has. The search for ILP in the Java program is done at the bytecode level. The algorithm works as follows: all the instructions that depend on the result of the previous one are grouped in an operand block. The entire Java program is divided in these groups and they can be parallelized respecting the functional unit constraints. For example, if there is just one multiplier, two instructions that use this functional unit cannot be operated in parallel. An example of this procedure can be observed in Figure 2.

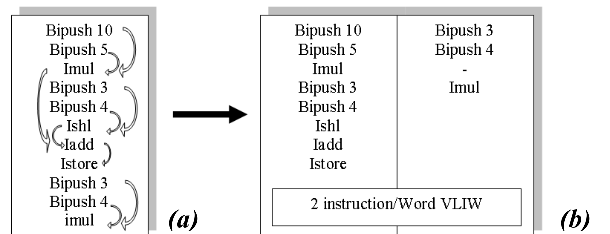

Fig. 2. Building the VLIW packets

In the sequence of instructions, observed in Figure 2a, the first *imul* instruction will consume the operands pushed previously, by the instructions *bipush 10* and *bipush 5*. After that, the *ishl* instruction will consume two more operands produced before by the previous *bipush*. The *iadd* instruction will consume the results of *imul* and *ishl.* Finally, the *istore* will save the result of the *iadd* in the local va-

riable pool. After that, there are two more *bipush* instructions, which operands will be used by the last *imul*. However, they do not use any result of the set of instruction previously executed. In other words, their operand stacks are independent. Hence, their operation can occur in parallel, as can be observed in Figure 2b. It is important to notice that the two *imul* instructions ca not stay at the same VLIW packet, since we are considering that in this example the VLIW processor has just one multiplier.

One of the main advantages of a stack processor is the manner of how the operand blocks communicate with each other. In conventional VLIW architectures, usually some kind of communication system among the functional units of the many flows is necessary. There are many possibilities: the use of crossbars, buses or a shared register bank. In the case of the latter, additional instructions are necessary to synchronize the communication of the flows in the register bank to maintain the data consistency.

In the Java language, when an operation block gets to the end, its result is saved in the local variable pool. This variable pool is shared among all the flows in a register bank. When an operand block in a determined flow needs a result of another operand block that is in another flow, it is only necessary to access the register bank of the main flow, where the local variable pool is located. No extra instructions are necessary or synchronizations mechanisms, because this communication is intrinsically found in a stack machine based language, such as Java.

The code sequence in Figure 3 illustrates this procedure. One can note that, after the parallelization, the third operation block in the first flow needs a result from the second operation block, which is in the second VLIW flow. As the result was written in the register bank (through the instruction *istore_1*), the third operation block reads this result using the *iload_1* instruction in the local variable pool that is shared in a register bank among all the flows.

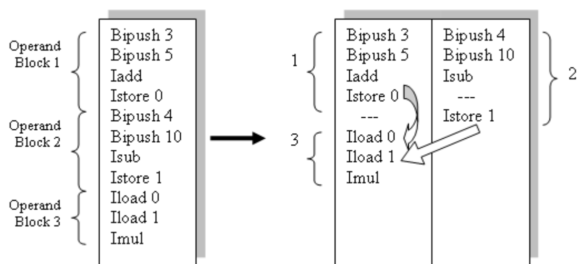

Fig. 3. Sharing data among different flows

It is important to notice that is really easy to build a program to analyze the parallelism in Java bytecodes. There is no special care to handle the communication among the flows and, to build the operands blocks, it is just necessary the information of how much each instruction consumes or produces operands to or from the stack.

As mentioned before, one of the major sources of energy consumption are the memory accesses. Hence, another optimization at the *bytecode* level is made concerning this problem. After the search for parallelism, another search is done: instructions that read the main memory (i.e. *getstatic*) are aligned in the same VLIW packet. If they fetch a value at the same address in the memory and between them this value is not changed (i.e., there is no *putstatic*), one can align these *getstatic* instructions. Hence, the processor, instead of make all the *getstatic* accesses, just need to perform one operation, passing the value to the other flows. Figure 4 illustrates this procedure.

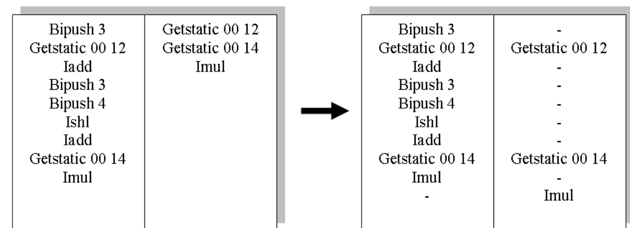

Fig. 4. Memory accesses – alignment process

IV. Simulation Environment

The tool utilized is a configurable compiled-code cycle-accurate simulator [17]. It was used to provide data on the energy consumption, memory usage and performance. Its power estimation technique is comparable to the component-based approach used in [18].

Power dissipation is evaluated in terms of switching activity, 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.

Five different types of algorithms were implemented and simulated over the architectures described in Section 3. Sin computation, 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 point numbers, since our Java processors can be configured without a floating point unit in order to save area.

There are two search algorithms: one executes the search in a sequential fashion and the other performs a binary search in the same 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. Additionally, three loop unrolled versions of the IMDCT algorithm were implemented, to make the search for ILP in the java program easier.

V. Results

Our experiments are supported by simulation, where different versions of a Java Processor execute algorithms used in embedded system domain. The area taken by the processors was computed in number of FPGA's logic cells, after synthesis of the VHDL versions of these processors.

Table 1 shows the area occupied by the three different versions of our Java processors. It is important to note that the register bank in the Low-Power version (as the main

flow of the VLIW one), used as stack and local variable pool, has 32 registers, the maximum required among all the applications (note that they are counted in the table, even though the FPGA's memory could be used for this purpose). The area was evaluated using the Leonardo Spectrum for Windows [19], and it is presented in logic cells.

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

PROCESSOR	*MULTICYCLE*	*PIPELINE*	*VLIW (# instructions per word)*		
			2	*4*	*8*
Area (LCs)	1365	3749	6110	10505	19297

Table 2 shows the performance in number of cycles of the processors for each application. As can be observed in this table, better results are found when unrolled versions are used (IMDCT u1, IMDCT u2 and IMDCT u3). One reason for this is that there are less conditional branches in these versions. Therefore, the number of cycles lost because of braches miss predictions is reduced as well (in the case of our Java Processor, 3 cycles for each branch miss prediction). For the VLIW version, another advantage in the use of unrolled versions is that the parallelism is more exposed in the bytecodes for the analyzer, since the size of basic blocks increases significantly. The drawback when using this technique is the increase in the memory footprint.

Table 2. Performance of the architectures, in number of cycles

Algorithm	*Number of cycles*				
	Multicycle	*Low-Power*	*VLIW*		
			2	*4*	*8*
Sin	2447	755	599	592	583
Ord./Bubble	6950	2424	2104	1967	1967
Ord./Select	5335	1930	1707	1670	1670
Ord./Insert	5111	1934	1601	1331	1331
Binary Search	1162	403	368	365	365
Sequencial Search	7586	1997	1775	1775	1775
IMDCT	140300	40306	33050	32994	32994
IMDCT u1	97354	31500	19325	12313	9944
IMDCT u2	92882	30369	18689	11737	9432
IMDCT u3	51345	18858	12789	8929	7741
Floating Point Sums	30747	14531	12474	12313	12313

Operating at the same frequency, the VLIW versions are the ones that have the major power consumption per cycle in the core, because their architectures are more complex, with more functional units and registers available. This behavior can be observed in Figure 5 (where VLIW 2 means two instructions per VLIW packet and so on). It is important to note that VLIW processors with more instructions per packet consume even more power, since there are more sequential components that spent energy even if they are not used.

However, as can be observed in Figure 6, when considering the total energy consumption instead of the power consumption per cycle, the difference among versions is small for the majority of the benchmark set. The reason for that is the high IPC average achieved by the VLIW processors: even though they spend more power per cycle, they need fewer cycles to finish the execution of a given benchmark.

The multicycle version uses the main memory for the operand stack and local variable storage. There is a good difference in terms of energy consumption between this architecture and the Low-Power version. This version, in turn, just make accesses in the main memory in method calls and returns or in specific instructions, like *getstatic* and *putstatic*. Figure 7 demonstrates the advantage of implementing the operand stack and local variable storage in a register bank instead of using the main memory for this purpose.

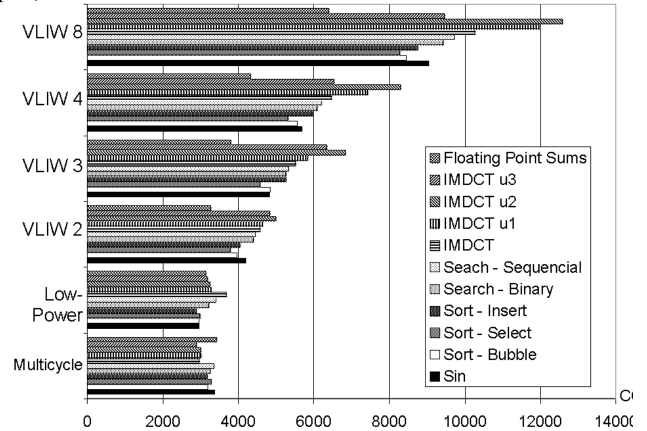

Fig. 5. Power consumed per cycle in the core

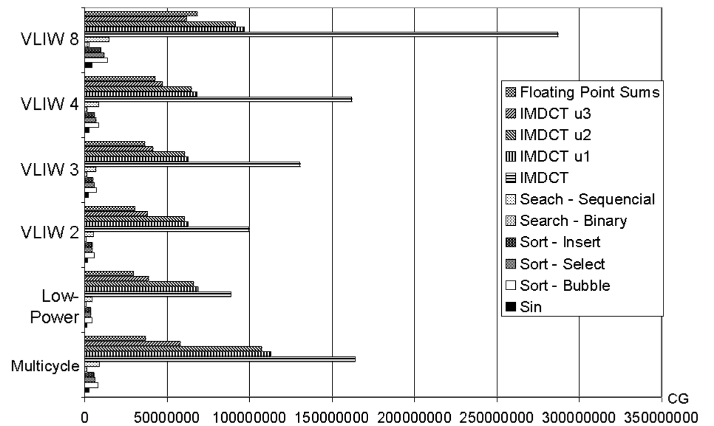

Fig. 6. Total of energy spent in the core per each application

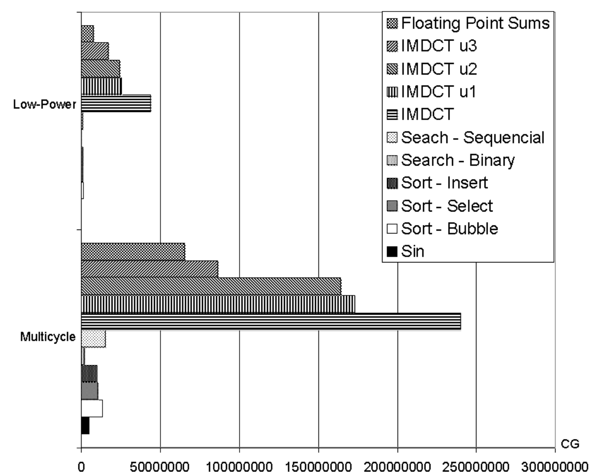

Fig. 7. Total of energy spent in memory accesses

In Figure 8, we show the gains when using the VLIW technique with and without *getstatic* alignment, as explained in Section 3. As one can observe, there is an expressive reduction in some algorithms, mainly in the IMDCT filter. It is important to notice that there is a small loss in the parallelism when this technique is used, because some instructions are moved in order to align the getstatic instructions in the same VLIW packet. This loss is in average of 1,5%, considering our benchmark set.

In Figure 9 we show the total energy consumption, considering the core, RAM and ROM of all architectures analyzed. The difference of consumption between the multicycle version and the pipelined one has mainly three reasons: the use of the register bank in the core avoiding accesses in the main memory, the reduced number of stack writes due to the use of the forwarding technique and the better IPC average.

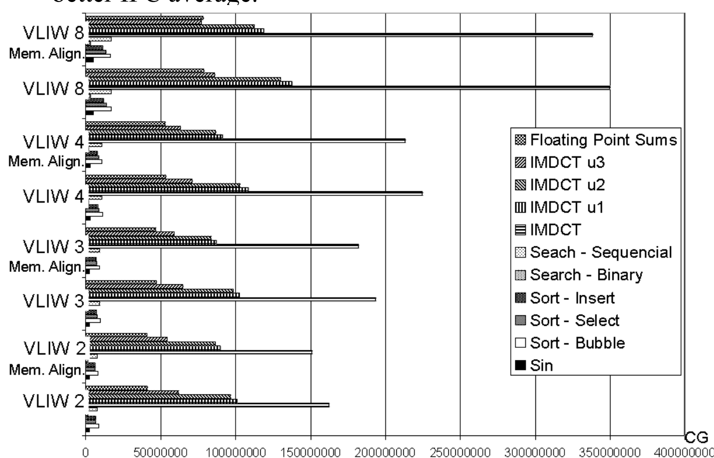

Fig. 8. Power savings in memory accesses alignments

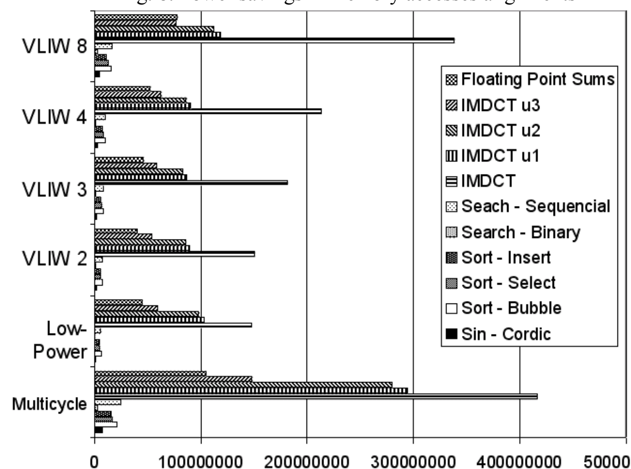

Fig. 9. The total energy consumption of each algorithm in the different versions of the processors

The gain in the VLIW version is basically because of the use of smaller register banks in the operand stacks of the secondary VLIW flows, avoiding, for some instructions, the use of the one in the main flow, which has more registers; the decrease in number of memory accesses due to the *getstatic* alignment; and finally the even better IPC average. However, these advantages in terms of power consumption can be observed only when these secondary flows are intensively utilized, which is exactly the case of the unrolled versions of the IMDCT.

In embedded applications, many of them with real time requirements, a specific throughput must be warranted for the application. Assuming that this task has already been accomplished by the multicycle version, the frequency of operation can be reduced in the Low-Power and VLIW versions with the purpose of saving more power, since these architectures can execute more instructions per cycle, as was shown in table 2. Moreover, when assuming that the dynamic power is the dominant in the total power consumed by the system, and that all the gates of the microprocessor form a collective switching capacitance C with a common switching frequency f, one obtains:

$$P = C \cdot f \cdot V_{DD}^2 \qquad (1)$$

As can be observed in [20], the voltage of the processor Transmeta TM5400 (known as Crusoe) [21], 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, and consequently, the voltage, of the Low-Power and VLIW versions are reduced to reach exactly the same throughput of the multicycle version, using as base the Equation (1).

It is very important to mention that the IMDCT versions with loop unrolled are those that show more benefits, bringing a high rate of instruction executed per cycle. Using the loop unrolled technique a high level of parallelism is exposed for the analyzer. Therefore, the frequency and voltage can be reduced even more when compared to others algorithms. As can be observed in Figure 10, the IMDCT u2 demonstrates good savings in terms of energy consumption, in the VLIW 4. When there is a high level of parallelism, the secondary flows are massively used, so the registers of these flows can be always occupied by instructions. However, there is a limit. The execution of floating point sums algorithms, for instance, allows up to 4 instructions per VLIW packet. Beyond that, as there is no more parallelism available in the application, all the extra sequential components of the secondary flows keeping spending power without any purpose. In the search and sort algorithms, always there is a disadvantage in terms of energy consumption, because of the lack of parallelism available.

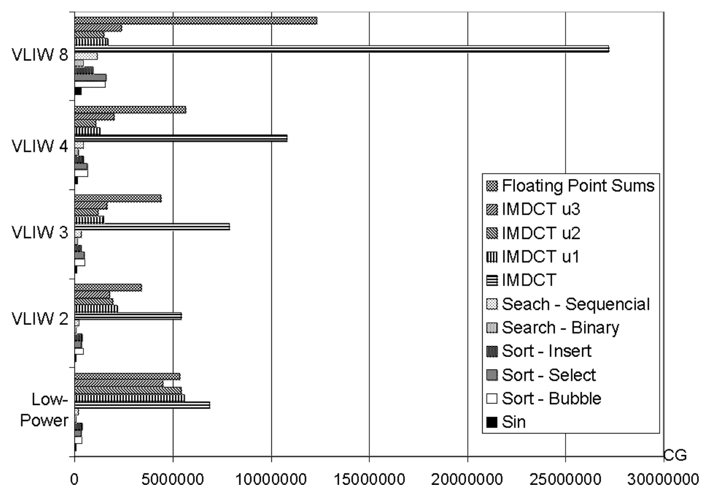

Fig. 10. The total energy consumed by the Low-power and VLIW versions considering that they have the same performance

The memory accesses are responsible for approximately 20% of the overall energy spent by the whole system. However, depending on the structure, size and level of the memory, it can be responsible for 50% of the energy consumption or even more [22][23]. In Figure 11 we show the difference in energy consumption when applying the *getstatic* alignment technique on the VLIW 4 processor (average of the three unrolled versions of IMDCT), considering different proportions of energy spend by memory accesses (X-axis). As one can observe, when the memory accesses is responsible for 50% of the energy consumption of the whole system, when using the *getstatic* alignment one can save almost 15% of the overall consumption of the system. This approach has almost no cost in hardware and performance, and it is a particular characteristic of stack-like processors.

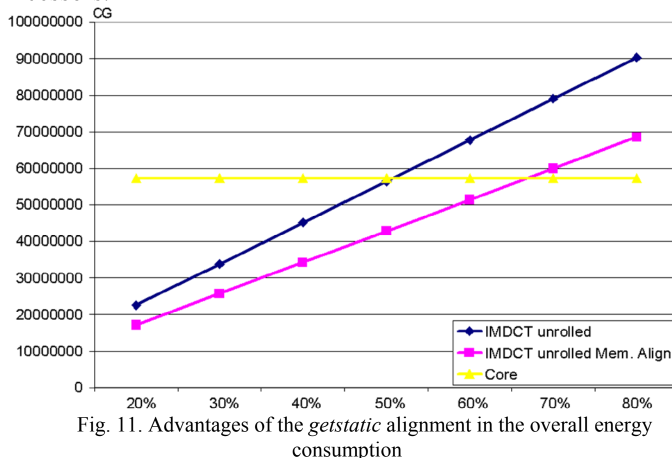

Fig. 11. Advantages of the *getstatic* alignment in the overall energy consumption

In none of the studied examples we found an algorithm that spends less energy in the VLIW 8. However, we have not used more sophisticated techniques, like software pipelining and superblock, to better expose the parallelism of the applications. For loop intensive programs, software pipelining provides performance gains ranging from 50% to 300% and superblock from 10% to 25% [24]. Even though none of those techniques was used, a gain of 3 times in energy consumption was shown in the VLIW 4, compared to the low-power version (considering the unrolled versions of IMDCT), demonstrating the potential of this Java VLIW architecture.

VI. Conclusions and Future Work

We demonstrated that in algorithms that present a high level of parallelism, one could obtain a meaningful decrease in the energy consumption, taking advantage of the VLIW technique in stack-like processors. Besides the opportunity of using the memory accesses alignment, the search for the parallelism and the communication among flows are facilitated, because of the stack-like architecture which Java is based. For future work, more algorithms concerning the embedded system domain and optimizations aimed at the VLIW architecture will be evaluated. Other alternatives to increase the VLIW performance, such as the use of other techniques to find the instruction parallelism inside the program, like software pipelining, superblocks, and static speculative execution, will be studied, bringing even better results.

VII. References

[1] Schlett, M. Trends in Embedded-Microprocessor Design. In *Computer*, vol. 31, n. 8, 1998, 44–49

[2] Takahashi, D. Java Chips Make a Comeback. In *Red Herring*, 2001

[3] Lawton, G. Moving Java into Mobile Phones. In *Computer*, vol. 35, n. 6, 2002, 17-20

[4] Tiwari, V., Malik, S., Wolfe, A. Power Analysis of Embedded Software: A First Step Towards Software Power Minimization. In *IEEE Transactions on VLSI Systems*, vol. 2, n. 4, Dec. 1994, 437–445

[5] Chen, G., Shetty, R., Kandemir, M., Vijaykrishnan, N., Irwin, M. Tuning garbage collection for reducing memory system energy in an embedded java environment. In *ACM Transactions on Embedded Computing Systems*, vol. 1, n. 1, Nov. 2002, 27-55

[6] O'Connor, J. M., Tremblat, M. Picojava-I: the Java Virtual Machine in Hardware. In *IEEE Micro*, vol. 17, n. 2, Mar-Apr. 1997, 45-53

[7] Sun Microsystems. In *PicoJava-II Microarchitecture Guide*, Mar. 1999

[8] Kreuzinger, J., Marston, R., Ungerer, Th., Brinkschulte, U., Krakowski, C. The Komodo Project: Thread-based Event Handling Supported by a Multithreaded Java Microcontroller. In *25th Euromicro Conference*, Sep. 1999, 2122-2128

[9] Nakamura, K., Sakai, K., Ae, T. Real-Time Multimedia Data Processing using VLIW Hardware Stack Processor, In *Proceedings IEEE Workshop on Parallel and Distributed Real-Time Systems*, 1995, 84-89

[10] Tremblay, M., Chan, J., Chaudhry, S., Conigliaro, A., Tse, S. The MAJC Architecture: A Synthesis of Parallelism and Scalability. In *IEEE Micro*, vol. 20, n. 6, 2000, 12-25

[11] Gray, J., Naylor, A., Abnous, A., Bagherzadeh, N. VIPER: A 25MHz, 100 MIPS Peak VLIW Microprocessor. In *Proceedings of the 1993 IEEE Custom Integrated Circuits Conference*, San Diego, 1993.

[12] Suga, A., Matsunami, K. Introducing the FR500 embedded microprocessor. In *IEEE Micro*, Jul-Aug. 2000, 21-27

[13] Seshan, N. High VelociTI Processing. In *IEEE Signal Processing Magazine*, vol. 15, n.2, March 1998, 86-101

[14] Ito, S.A., Carro, L., Jacobi, R.P. Making Java Work for Microcontroller Applications. In *IEEE Design & Test of Computers*, vol. 18, n. 5, 2001, 100-110

[15] Beck, A.C.S., Carro, L. Low Power Java Processor for Embedded Applications. In: *IFIP 12th International Conference on Very Large Scale Integration*, Germany, December (2003)

[16] Hennessy, J. L., Patterson, D. A. *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, 3th edition, 2003

[17] Beck, A.C.S., Mattos, J.C.B., Wagner, F.R., Carro, L. CACO-PS: A General Purpose Cycle-Accurate Configurable Power-Simulator. In *16th Brazilian Symp. Integrated Circuit Design (SBCCI 2003)*, Sep. 2003

[18] Chen, R., Irwin, M. J., Bajwa, R. Architecture-Level Power Estimation and Design Experiments. In *ACM Transactions on Design Automation of Electronic Systems*, vol. 6, n. 1, Jan. 2001, 50-66

[19] Leonardo Spectrum, available at homepage: *http://www.mentor.com/synthesis*

[20] Pouwelse, J., Langendown, K., Sips, H. Dynamic Voltage Scaling on a Low-Power Microprocessor. In *The Seventh Annual International Conference on Mobile Computing and Networking*, 2001, 251-259

[21] Transmeta Corporation, *Tm5400 processor specifications*, *http://www.transmeta.com*

[22] Montanaro J. and et. al. A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor. In *IEEE Journal of Solid-State Circuits*, vol. 31, n. 11, Nov. 1996, 1703 - 1714

[23] Inoue, K., Ishihara, T., Murakami, K. Way-predicting set-associative cache for high performance and low energy consumption. In *Proceedings on Low Power Electronics and Design*, Aug. 1999, pp. 273 - 275

[24] Lee, M., Tirumalai, P., Ngai, T. Software Pipelining and Superblock Scheduling: Compilation Techniques for VLIW Machines. In *Proceedings of the 26th Hawaii International Conference on System Sciences*, Jan. 1993