Dynamic Reconfiguration with Binary Translation: Breaking the ILP Barrier with Software Compatibility

Antonio Carlos S. Beck
Universidade Federal do Rio Grande do Sul
Instituto de Informática - Av. Bento Gonçalves, 9500
Campus do Vale - Porto Alegre, Brasil
caco@inf.ufrgs.br

Luigi Carro
Universidade Federal do Rio Grande do Sul
Instituto de Informática - Av. Bento Gonçalves, 9500
Campus do Vale - Porto Alegre, Brasil
carro@inf.ufrgs.br

ABSTRACT
In this paper we present the impact of dynamically translating any sequence of instructions into combinational logic. The proposed approach combines a reconfigurable architecture with a binary translation mechanism, being totally transparent for the software designer. Besides ensuring software compatibility, the technique allows porting the same code for different machines tracking technological evolutions. The target processor is a Java machine able to execute Java bytecodes. Experimental results show that even code without any available parallelism can benefit from the proposed approach. Algorithms used in the embedded systems domain were accelerated 4.6 times in the mean, while spending 10.89 times less energy in the average. We present results regarding the impact of area and power, and compare the proposed approach with other Java machines, including a VLIW one.

Categories and Subject Descriptors
C.1.3 [**Processor Architectures**]: Other Architecture Styles – *adaptable architectures*

General Terms
Performance, Design

Keywords
Java, Reconfigurable Processors, Binary Translation, Power Consumption

1. 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, dictated by market needs, the complexity of these embedded systems is increasing as well, since they are offering more and more functions to the user, like Internet access, color display, audio and video reproduction, videogames, among others [2]. Hence, these systems must have enough processing power to handle these complex tasks.

Therefore, while still sustaining great performance, present days embedded systems must also have low power dissipation and support a huge software library to cope with stringent design times. Consequently, there is a clear need for architectures which can support all the software development effort currently required.

Regarding potential platforms for embedded systems development, reconfigurable fabric has been shown to speed up critical parts of several data stream programs. By translating a sequence of operations into a combinational circuit performing the same computation, one could speed up the system and reduce energy consumption, at the obvious price of extra area. Using a reconfigurable array one is able to have exactly this kind of hardware substitution. Nevertheless, its wide spread use is still withhold by the need of special tools and compilers, which clearly preclude software portability. To handle these problems, recent works have already proposed dynamic analysis of the code to reconfigure the array at run-time [3][4]. However, these works used a fine-grain array, which brings a huge control overhead that increases the complexity of dynamic detection, and also increases reconfiguration time, thus requiring a large cache size to keep the array configurations. As a consequence, just critical parts of the software, like the most executed loops, with some restrictions, can benefit from using the reconfigurable array.

On the other hand, our work uses a coarse-grain granularity array that, in turn, is not limited to the complexity of fine-grain configurations. The algorithm for the dynamic detection and configuration of the array becomes simpler, and less memory for keep these configurations is necessary. Hence, we can optimize any part of the algorithm that, because of the limitations of fine-grain nature, was infeasible. Therefore, we can significantly increase the performance of any kind of software as well as reducing the energy consumption, not being limited to just DSP-like or loop centered applications, as is [3][4].

Furthermore, coupling this coarse-grain array with binary translation (BT) [5], speedups are obtained even in algorithms that do not present a high level of parallelism, since the application of this technique can also transform in combinational logic sequences of instructions that could not present a high instruction level parallelism (ILP). This is a very useful characteristic, since the amount of parallelism during the execution usually varies [6]. In order to demonstrate that, we compare the processor coupled with the reconfigurable array with VLIW versions with the same instruction set.

Thus, in this work we demonstrate the performance and energy gains of using a coarse-grain reconfigurable array with BT to explore every part of the algorithm, even in those which do not present a high level of parallelism, using a Java processor as case study. Thanks to the coupling of dynamic binary translation and Java, we can assure software compatibility in any level of the design cycle, without requiring any tools for the hardware/software partitioning or special compilers, allowing easy

DAC 2005, June 13–17, 2005, Anaheim, California, USA.

software porting for different machines tracking technological evolutions. Algorithms used in the embedded systems domain were accelerated 4.6 times in the mean, while spending 10.89 times less energy in average.

This paper is organized as follows: Section 2 shows a brief review of the existing reconfigurable processors and some other approaches regarding dynamic translation of instructions. In Section 3 we discuss the different architectures of Java machines that will be evaluated. Section 4 explains the advantages of applying binary translation technique to work with the reconfigurable array and some more details about them. Section 5 shows the results regarding power consumption, performance and area. The last Section draws some conclusions and introduces future work.

2. RELATED WORK

Reconfigurable systems have already showed to be very effective, implementing some parts of the software in a hardware reconfigurable logic. Huge software speedups [8][9] as well as a reduction in system energy have been achieved [10][11]. These systems can be implemented together with processors like Chimaera [12] and ConCISe [13], with a tightly coupled reconfigurable array in the processor core, limited to combinational logic. The array is, in fact, an additional functional unit in the processor pipeline, sharing the same resources of the other ones. This makes the control logic simpler, diminishing the overhead required in the communication between the reconfigurable array and the rest of the system. Nevertheless, as the majority of reconfigurable systems, it is necessary additional tools at the design time or special instructions in the processor to make the array works.

Another approach to cope with high performance allied to low energy dissipation is the use of binary translation. In this approach, the system itself monitors the binary of executing program, detects the frequently executed software kernels, and optimizes them. Existing optimizations include dynamic recompilation and caching of previous binary translation results [5]. The Transmeta Crusoe is based on a VLIW processor where the application is analyzed at runtime in order to find the best parts of the software for the binary translation to better explore the ILP [14]. One of the advantages of using this technique is that the partitioning process is transparent, requiring no extra designer effort, and causing no disruption to the standard tool flow.

In [15] Stitt, Lysecky and Vahid presented the first studies about the benefits and feasibility of dynamic partitioning using reconfigurable logic. In [3], a modified place and route algorithm is used, supporting a larger range of benchmarks and requiring less computation time and memory resources, with the same objective: optimize the execution by dynamically moving critical software kernels to configurable logic at runtime, a process called warp processing. However, this technique is limited to critical parts of the software, as some loops, and a fine-grain configurable fabric was used.

Most of the presented works on dynamic optimization focus on single loops within an application. However, as it has been demonstrated by [6], the amount of available parallelism during the execution of a program largely varies. For this reason, the allocation of loop optimizations at compile time does not explore all the reconfiguration potential. However, when applied to a coarse grain reconfigurable substrate, binary translation brings in a new feature: since the optimizations can be developed for any sequence of instructions, we are not limited to optimize only critical loops or to the parallelism available in the application, but rather we can run the whole program faster.

Using binary translation and Java, we ensure at the same time software compatibility and no extra efforts or tools at design time, which means that the underlying hardware can be changed without the need for recompilation or to write a new compiler. Also, there are three main advantages of using a tightly coupled coarse-grain array: it allows a quick reconfiguration; the huge power dissipation and control overhead in a fine grain architectures is avoided; and finally the overhead of the communication between the system and the array is minimal, consequently saving power.

3. JAVA EXECUTION

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 [16]. Nevertheless, it is predicted that at least 80 percent of mobile phones will support Java by 2006 [17]. 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.

Our Low-Power Java processor [18] has a five stages pipeline: instruction fetch, instruction decoding, operand fetch, execution, and write back, as shown in figure 1. The VLIW processor [19] is an extension of the pipelined one. Basically it has its functional units and instruction decoders replicated. The VLIW packet has a variable size, avoiding unnecessary memory accesses. The search for ILP in the Java program is done at the *bytecode* level.

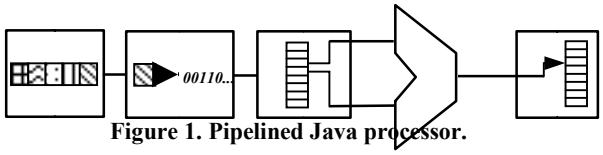

Figure 1. Pipelined Java processor.

4. BT AND RECONFIGURABLE ARRAY

By transforming any sequence of bytecodes into a single combinational instruction in the array using BT one can achieve great gains. Although the delay for the reconfiguration might be higher, if the sequence of instructions is going to be repeated a certain number of times, performance and energy gains are meaningful, since less access to program memory and less iterations on the datapath are required. In this section we explain the potential of using combinational logic, the architecture of our coarse-grain array and how the detection, the reconfiguration and execution work.

4.1 Combinational Circuit Advantage

There are always potential gains when passing the execution from sequential to combinational logic. This concept is better explained with a simple example. Let us have an nxn bit multiplier, with input and output registers. By implementing it with a cascade of adders, one might have in the worst case

$$T_{mult\_combinational} = t_{ppFF} + 2*n*tcell + t_{setFF} \quad (1)$$

where *tcell* is the delay of an AND gate plus a 2-bits full-adder. The area of this multiplier is

$$A_{combinational} = n^2 * Acell + Aregisters \quad (2)$$

If one could do the same multiplier by the classical shift and add algorithm, and assuming a carry propagate adder, the multiplication time would be

$$T_{mult\_sequential}=n * (t_{ppFF} + n*tcell + t_{setFF}) \quad (3)$$

And the area given by

$$A_{sequential}=n*Acell + Acontrol + Aregisters \quad (4)$$

Clearly, by using a sequential circuit one trades area by performance. Any circuit implemented as a combinational circuit will be faster than a sequential one, but will most certainly take much more area. By the advances obtained by technology integration and in reconfigurable arrays architectures, one could imagine the use of a reconfigurable array as a programmable combinational area. However, the use of this array for just a single algorithm would have a prohibitively high cost. In our approach, by coupling the array with a BT mechanism, we can reuse the array in a SW compatible environment. Also, as we translate sequences of instructions into combinational logic, we do not depend on the available application parallelism to speed up the code to be executed, but rather on sequences of instructions that appear several times in the code.

4.2 Dynamic Detection

A separated unit, which is a simple multicycle Java processor [20], is responsible for dynamic analyzing the instructions in order to find the sequences that can be executed in the array. This is done concurrently while the main processor fetches valid instructions. When this unit realizes that there are a certain number of instructions which are worth to be executed in the array, the configuration for this sequence is saved in a reconfiguration cache. The next time this sequence is found, the array will execute it instead of the normal execution in the processor.

The search for the sequence of instructions in the Java program is done at the *bytecode* level, similarly to what the VLIW static analyzer does, classifying sequence of instructions that depend on each other in an *operand block*. The detection operation to find these blocks is very simple: when the stack pointer returns to the start address previously saved, an operand block is found. This explains why the instruction analysis can be developed at run time with a small overhead.

4.3 Architecture of the Array

The reconfigurable array is tightly coupled to the processor. It is implemented as a functional unit in the execution stage, using the same approach of Chimaera and ConCISe, cited before.

The array is divided in blocks, called cells. The operand block (a sequence of *bytecodes*) previously detected is fitted in one ore more of these cells in the array. The cell can be observed in Figure 2. The initial part of the cell is composed by three functional units (ALU, shifter, ld/st). After the first part, six identical parts follow in sequence. This number was chosen based on time delays presented in [21], where it is compared the delay of the booth multiplier (used in our processor) with adders. Each cell of the array has just one multiplier and takes exactly one processor cycle to complete execution, being limited to its critical path, bringing no delay overhead in the processor pipeline.

At the end of each cell, there are two additional functional units: a branch unit and an extended ld/st one, made for the execution of the *iastore* (fetches a static value from memory, taking the address from the stack) instruction, since this instruction needs three operands, instead of two, as usual.

For each cell in the array 327 reconfiguration bits are necessary. Consequently, if the array is formed by 3 cells, 971 bits in the reconfiguration cache are necessary. To these one must add 58 bits of additional information, like as how many cycles the execution takes and what is the initial ROM address that this sequence is located, totalizing 1029 bits for each configuration of the array.

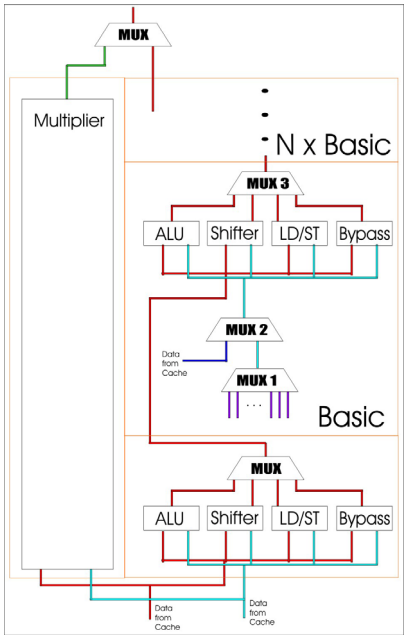

Figure 2. Architecture of the array

4.4 Reconfiguration and Execution

The array is reconfigured using the data of the cache specially designed for it. Each cell in the array has its own reconfiguration cache. Therefore, the reconfiguration of each cell can be done in parallel. While the program executes, when an address of a reconfigurable instruction group is found, the reconfigurable unit detector sends information for the main processor. Then, its control unit configures the array as the active functional unit, stops the rest of the processor while the array is performing it functions, and upgrades the Program Counter with the new address, in order to continue the normal operation after the execution of that sequence of instructions by the array. At the same time, the configurations in the reconfiguration cache for that address are sent to the reconfigurable array, since they are indexed by addresses of the instruction memory.

As the detection for the address that will be used in the reconfiguration is done in the first stage of the pipeline, and the reconfigurable array is in the fourth stage, there are 3 cycles available between the detection and the use of the array. As one cycle is necessary to find the cache line that has the array configuration, two cycles are available for the reconfiguration.

Considering the follow example, we show the potential gains using a reconfigurable array in a Java machine: if there are five simple arithmetical or logic operations, one needs at least 11 cycles for the execution: 6 for pushing the operands into the stack, plus 5 cycles for the operations themselves. This is the optimal execution, without considering pipeline stalls due to data dependency. On the other hand, the array would execute in just one cycle everything, after the first time it finds this sequence and saves it in the reconfiguration cache. If this sequence is repeated a certain number of times, meaningful gains can be achieved.

For the execution, the first two operand values will be used as the first operation. After that, for each basic part of the cell, the first operand is always the result of the previous operation. This is another characteristic of stack machines, and makes the control and the hardware of the array simpler. In each cell it is possible to make 7 simple operations (arithmetic, logical, shift). If a multiplication is found, the result of the current cell is bypassed until the end. This multiplication will be executed in the next cell.

Special attention must be given to some *bytecode* instructions, whenever they are in the reconfiguration or in the execution phases. If a *getstatic* instruction is found (loads a value from main memory), the pointer field in the cache is used. This pointer tells where the access is made. The value of the *getstatic* instruction will be fetched from a special cache (dual-ported with 1kB of size) during the reconfiguration phase, since the access address is static. The value fetched is saved in the operands field.

It is during the reconfiguration phase that the local variables of the method are fetched as well. These local variables are kept in a dual-ported register bank in the processor, and can be fetched at the same time the static values from the memory are.

Some unexpected actions can be taken during these fetches. Cache misses can occur in the case of *getstatic* accesses and other instructions could be accessing the register bank making impossible the load of local variables. In this case, more cycles to treat the cache misses or to wait the instructions that are accessing the register bank are necessary for the reconfiguration of the array.

During the execution, for instructions that save a value in the main memory, a buffer is used in order not to retard the execution of the cells. In the case of instructions that load/store values in main memory, or the local variable storage of the method, values can be bypassed. One example of this is when there is a load instruction in a local variable soon after a store in the same local variable. This avoids unnecessary accesses in the register bank or in the main memory, accelerating the execution and saving power.

Moreover, if there is an instruction *iaload*, which makes an access in the memory and calculates the access address dynamically, the value is fetched from the same cache of the *getstatic*. If there is a cache miss, the penalty is paid and that cell in the array starts to be executed again.

5. RESULTS

Our experiments are supported by simulation, where different versions of the Java Processor execute algorithms used in embedded system domain, as presented before. The tool utilized to provide data on the energy consumption, memory usage and performance is a configurable compiled-code cycle-accurate simulator [22].

Different types of algorithms were implemented and simulated over the architectures described in Section 3 and 4, from simple ones to a complex full MP3 player. Sin computation using the cordic method, as a representative of arithmetic libraries; sort – bubble, select and quick, in a array of 10 or 100 elements – and search (binary and sequential), used in schedulers; IMDCT (Inverse Modified Discrete Cosine Transformation) plus more three unrolled version in order to expose the parallelism, an important part present in various decompression algorithms; a library to emulate sums of floating point numbers, since the Java processors can be configured without a floating point unit in order to save area; and finally a complete MP3 player that executes 4 frames of 40kbit, 22050Hz, joint stereo, where the algorithm is divided in six parts.

Initially, in Table 1 we evaluate the performance of all our benchmark set in the Low Power architecture and in the different versions of the VLIW, and compare those to the Java processor coupled to the reconfigurable array. As can be observed in this table, for the VLIW processor better results are found when unrolled versions are used (IMDCT u1, IMDCT u2 and IMDCT u3). The reason for this is that there are less conditional branches, which reduces the number of cycles lost because of braches miss predictions, and mainly because there is more parallelism exposed. On the other hand, algorithms like the floating point sums emulation do not show performance improvements when the number of instructions available per packet in the VLIW grows. This occurs because there is no more parallelism available in the application to be explored, so increasing the size of the VLIW packet does not matter.

In the same table, in the column *Reconfigurable Array – Sequential*, we show the greatest advantage of using an array with BT to explore every part of the algorithm. Even in algorithms that do not present a high level of parallelism to be explored like the floating point sums emulation, or in the sort or search ones, great gains are achieved. Furthermore, in algorithms which show a good performance in the VLIW architecture because of the high level of parallelism available, like de unrolled versions of IMDCT, the array presents even better results. A good example of how the array with BT can be better exploited is in the sort family of algorithms. When we ran the versions that sort 100 elements, more array configurations are reused, bringing an even better result with no area overhead (the number of different reconfigurations and cells in the array do not increase). It is important to mention that we explored the available parallelism in the array as well. This data can be observed in the column *Reconfigurable Array – Parallel*. Significant improvements were obtained just in a few algorithms, mainly in the unrolled versions of IMDCT, which were changed to have their parallelism more exposed. This reinforces our idea of exploring any sequential part of the software, and not being dependent of the parallelism available.

Table 1. Performance of the architectures

Algorithm	Number of cycles						Data about the array				
	Low-Power	VLIW (instructions per packet)			Reconfigurable Array		#dif. reconf.	#Seq. reused	#max rec.	#max Seq. cells	#max Par. cells
		2	4	8	Sequencial	Paralel					
Sin	755	599	592	583	383	383	8	64	3	2	2
BubbleSort 10	2424	2013	1923	1923	712	600	7	177	3	4	4
SelectSort 10	1930	1689	1689	1689	532	514	8	182	3	3	6
QuickSort 10	1516	1246	1246	1246	496	496	13	132	3	2	2
BuubleSort 100	339797	268610	268610	268610	61541	47840	7	22458	3	4	6
SelectSort 100	134090	127466	127533	127533	30700	30502	8	15280	3	3	6
QuickSort 100	13239	10649	10649	10649	5007	5007	13	2804	3	2	2
Binary Search	403	369	365	365	176	176	5	33	3	2	2
Sequencial Search	1997	1776	1774	1774	658	658	2	253	3	2	2
IMDCT	40306	33128	33071	32994	9399	4287	7	2407	4	10	15
IMDCT u1	31500	18062	12191	9604	7624	2512	16	825	4	10	15
IMDCT u2	30372	17329	11546	9114	6972	2436	13	804	4	10	15
IMDCT u3	18858	11230	9838	7807	2852	2780	7	745	3	4	6
Floating Point Sums	14531	12475	12314	12296	6760	6729	37	660	4	3	4
MP3 part 1	242153	210818	200721	183818	103549	102936	140	12317	5	4	6
MP3 part 2	109396	92735	92735	92735	65010	65010	11	8138	3	3	3
MP3 part 3	64488	49346	49346	49346	45525	45525	22	9190	3	2	2
MP3 part 4	41587	33860	34471	31436	22097	22097	5	2876	4	3	3
MP3 part 5	35895	34405	15905	8959	9016	9016	5	1212	3	3	3
MP3 part 6	159017	103441	73482	51124	36405	31485	53	6005	7	11	15

In the second part of this table we present data concerning the reconfigurable array coupled to the Java architecture. In the first column of this second part we show how many instruction sequences were saved to the cache and were reused in the array. In the second, we demonstrate the amount of reuse obtained for these sequences. The next column shows the maximum number of cycles necessary to reconfigure the array from the cache. The forth column exhibits the maximum number of cells (showed in section 4.3) that these sequences occupied when there is no cells in parallel. When the parallelism among the cells is explored, sometimes are necessary more cells to allocate the parallel sequences. The last column shows these values.

In figure 3 we compare the energy consumption in the ROM and RAM of the Low-Power version with and without the array with the 4 instruction/packet VLIW version, since the values of energy spent in RAM and ROM accesses in this architecture are very similar to the 2 and 8 instructions/packet ones. Because of space restrictions, we grouped the algorithms in categories. We present the total sum of energy of all algorithms in each group.

As can be observed, the array saves energy in ROM accesses, since instructions that would be fetched in the memory are executed in the array, because the dataflow equivalent of this sequence is saved in the reconfiguration cache. In the same way, power consumed in the RAM memory and in the register bank are saved, because now there are a specific cache for loads of static values and the bypass of operands inside the array. Regarding the energy spent in the core, presented in Figure 4, even with the increment of the reconfiguration cache on it, there are still gains in terms of energy consumption in some algorithms. This occurs because, even consuming more power because of the cache, a considerable amount of instructions that would use the five stages of the pipeline of the processor and its sequential logic are now being executed in a combinational logic in the array.

In Figure 5 we show the total energy consumption of the system considering the RAM, ROM, the core and the additional processor that makes the dynamic code analysis. It is important to note that great gains where achieved in energy consumption in all algorithms, proving our technique effectiveness.

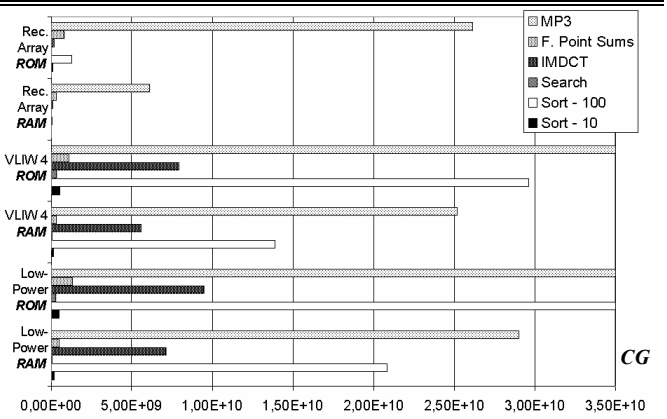

Figure 3. Energy spent in ROM and RAM accesses

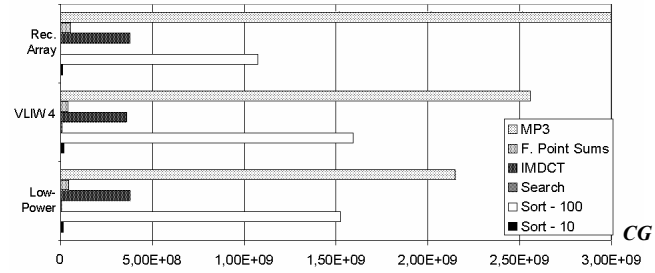

Figure 4. Energy spent in the core

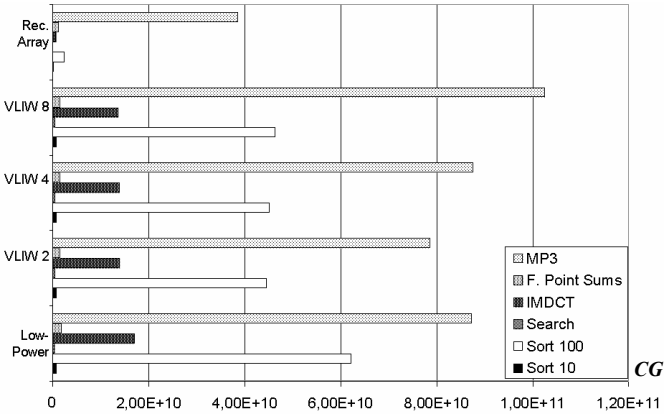

Figure 5. Total energy spent by the architectures

Table 2 shows the area occupied by the Low Power and VLIW versions of our Java processors. In Table 3 we present the area occupied by the Low-Power version with different configurations of the reconfigurable array (the maximum number of reconfigurations allowed versus the total number of cells available in the array), counting its cache and the processor responsible for the detection of the sequences of instructions and to make the reconfiguration. As can be observed in this table, the reconfigurable array, when coupled to the Java processor, even in its simpler version, brings area overhead when compared to the 8 instructions/packet VLIW architecture. However, this was expected, since reconfigurable arrays are very area-intensive due to their great number of functional units. The area was evaluated using the Leonardo Spectrum for Windows [24] and was computed in number of gates, after synthesis of the VHDL versions of them.

Table 2. Area occupied by the architectures

PROCESSOR	*LOW POWER*	*VLIW (INSTRUCTIONS/ PACKET)*		
		2	*4*	*8*
Area (gates)	131215	213850	367675	675395

Table 3. Area occupied by the reconfigurable array

# Cells / *# Reconf.*	*2*	*3*	*4*	*7*	*10*
5	757091	993999	1230907	1941630	2652353
10	1039631	1406301	1772971	2872981	3972990
15	1322172	1818604	2315036	3804332	5293628
20	1604712	2230906	2857100	4735683	6614265
40	2734873	3880116	5025358	8461087	11896815

Finally, table 4 compares the Java processor with the reconfigurable array with all other architectures, in terms of energy and performance. This table shows how faster the version with the reconfigurable array is, and how much less energy it spends. As it can be observed, huge energy savings are achieved when compared to any architecture (10.89 times less energy against the low-power version), and there are meaningful performance gains even when comparing to the 8 instructions/packet VLIW version (2.77 times faster in the mean).

Table 4. Comparison among the architectures

	Energy				*Performance*			
Reconfigurable Array vs.	*Low Power*	*VLIW 2*	*VLIW 4*	*VLIW 8*	*Low Power*	*VLIW 2*	*VLIW 4*	*VLIW 8*
Sin	1.89	1.93	1.79	1.83	1.97	1.56	1.55	1.52
Sort - Bubble 10	3.98	4.35	4.21	4.29	4.04	3.35	3.21	3.21
Sort - Select 10	8.09	7.76	7.76	7.91	3.76	3.29	3.29	3.29
Sort - Quick 10	3.19	3.18	3.18	3.24	3.05	2.51	2.51	2.51
Sort - Bubble 100	34.59	19.95	20.04	1.60	7.10	5.61	5.61	5.61
Sort - Select 100	26.23	24.17	24.17	24.67	4.40	4.18	4.18	4.18
Sort - Quick 100	5.74	5.73	5.72	5.82	2.64	2.13	2.13	2.13
Search - Binary	2.00	2.31	2.31	2.35	2.29	2.10	2.08	2.08
Search – Seq.	15.04	16.44	16.45	16.71	3.03	2.70	2.69	2.69
IMDCT	28.33	24.14	24.15	24.65	9.40	7.73	7.71	7.70
IMDCT u1	19.89	16.34	15.70	15.02	12.54	7.19	4.85	3.82
IMDCT u2	21.72	17.93	17.04	16.35	12.47	7.11	4.74	3.74
IMDCT u3	26.68	20.76	21.38	20.24	6.78	4.04	3.54	2.81
F. Point Sums	1.53	1.26	1.25	1.27	2.16	1.85	1.83	1.83
MP3 Part 1	1.87	0.79	0.82	0.86	2.35	2.05	1.95	1.79
MP3 Part 2	3.42	2.74	2.99	3.05	1.68	1.43	1.43	1.43
MP3 Part 3	1.19	1.95	1.95	1.99	1.42	1.08	1.08	1.08
MP3 Part 4	2.71	3.00	2.80	2.84	1.88	1.53	1.56	1.42
MP3 Part 5	4.91	4.67	7.94	13.96	3.98	3.82	1.76	0.99
MP3 Part 6	4.71	5.39	6.34	8.36	5.05	3.29	2.33	1.62
Average	***10.89***	***9.24***	***9.40***	***8.85***	***4.60***	***3.43***	***3.00***	***2.77***

6. REFERENCES

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

[2] Nokia N-GAGE Home Page, available at http://www.n-gage.com

[3] Stitt, G., Lysecky, R., Vahid, F., "Dynamic Hardware/Software Partitioning: A First Approach". In *Design Automation Conference (DAC)*, 2003

[4] Lysecky, R., Vahid, F., "A Configurable Logic Architecture for Dynamic Hardware/Software Partitioning". In *Design Automation And Test in Europe Conference (DATE),* 2004

[5] Gschwind, M., Altman, E., Sathaye, P., Ledak, Appenzeller, D., "Dynamic and Transparent Binary Translation". In *IEEE Computer,* vol. 3 n. 33, 2000, 54-59

[6] Bingxiong Xu, Albonesi, D., "Runtime Reconfiguration Techniques for Efficient General-Purpose Computation". *In Design & Test of Computers*, vol. 17, n. 1, Jan.-Mar. 2000, 42 – 52

[7] 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

[8] Henkel, J., Ernst, R., "A Hardware/Software Partitioner using a Dynamically Determined Granularity". In *Design Automation Conference*, 1997

[9] Venkataramani, G., Najjar, W., Kurdahi, F., Bagherzadeh, N., Bohm W., "A Compiler Framework for Mapping Applications to a Coarse-grained Reconfigurable Computer Architecture. Conf. on Compiler". In *Architecture and Synthesis for Embedded Systems (CASES)*, 2001

[10] Henkel, J., "A low power hardware/software partitioning approach for core-based embedded systems". In *Design Automation Conference*, 1999

[11] Stitt, G., Vahid F., "The Energy Advantages of Microprocessor Platforms with On-Chip Configurable Logic". In *IEEE Design and Test of Computers,* 2002

[12] Hauck, S., Fry, T., Hosler, M., Kao, J., "The Chimaera reconfigurable functional unit". In *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, 1997, 87–96.

[13] Kastrup, B., Bink, A., Hoogerbrugge, J., "ConCISe: a compiler-driven CPLD-based instruction set accelerator". In *Proc. 7th Annu. IEEE Symp Field-Programmable Custom Computing Machines*, Napa Valley, 92–100.

[14] Klaiber, A., "The Technology Behind Crusoe Processors". In *Transmeta Corporation White Paper*, 2000.

[15] Stitt, G., Vahid, F., "Hardware/Software Partitioning of Software Binaries". In *IEEE/ACM International Conference on Computer Aided Design*, 2002

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

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

[18] 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

[19] 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

[20] Beck, A.C.S., Carro, L. "A VLIW Low Power Java Processor for Embedded Applications". In *17th Brazilian Symp. Integrated Circuit Design (SBCCI 2004)*, Sep. 2004

[21] Callaway, T. K., Swartzlander Jr, E. E., "Power Delay Characteristics of Multipliers". In *IEEE 13th Symposium on Computer Arithmetic (ARITH '97)*, Mar. 1997

[22] 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*, Sep. 2003

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

[24] Leonardo Spectrum, available at homepage: http://www.mentor.com