Exploiting Java Through Binary Translation for Low Power Embedded Reconfigurable Systems

Antonio Carlos S. Beck, Victor F. Gomes, Luigi Carro
Universidade Federal do Rio Grande do Sul
Instituto de Informática - Av. Bento Gonçalves, 9500
Campus do Vale - Porto Alegre, Brasil
{caco, vfgomes, carro}@inf.ufrgs.br

ABSTRACT

In this paper we present a Binary Translation algorithm to detect, completely at run-time, sequences of instructions to be executed in a reconfigurable array, which in turn is coupled to an embedded Java processor. By translating any sequence of operations into a combinational circuit performing the same computation, one can speed up the system and reduce energy consumption, at the obvious price of extra area. We show what are the costs to implement this translation algorithm in hardware, and what are the performance and energy gains when using such technique. Furthermore, we demonstrate that this translation algorithm is particularly easy to be implemented in a stack machine, because of its particular computational method. Algorithms used in the embedded systems domain were accelerated 4.6 times in the mean, while spending almost 11 times less energy.

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 diffusion of embedded systems devices seems to be far from ending. While new products like PDAs, smart cellular phones and mp3 players keep arriving on the market, traditional consumer electronics like televisions, VCRs and game consoles are providing new capabilities [1]. Nevertheless, the continuous growing demand for more functional, more portable and more complex appliances also poses great challenges to the design of embedded systems, since these systems must have enough processing power to handle these tasks.

SBCCI'05, September 4–7, 2005, Florianópolis, Brazil.

In the same way, Java is becoming increasingly popular in embedded environments. Recent surveys reveal that Java is present in devices such as consumer electronics (digital TV, mobile phones, home networking) as well as industrial automation (manufacturing controls, dedicated hand held devices). It is estimated that more then 721 million devices will be shipping with Java by this year [2]. Furthermore, it is predicted that 80% of mobile phones will support Java by 2006 [3], but even today most of the commercialized devices already provide support to the language. This means that current design goals might include a careful look on embedded Java architectures, and their performance versus power tradeoffs must be taken into account.

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 that can support all the software development effort currently required.

The reconfigurable fabric is one of these potential platforms, and 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 [4][5]. However, in these approaches, 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, in a previous work [6] we have already shown the potential of a Java software compliant architecture supporting a reconfigurable array. Coupling a coarse-grain array with dynamic binary translation (BT) [7], which is used to detect potential sequences of instructions at run time to be executed in the array, we could significantly increase the performance of any kind of software as well as reduce the energy consumption, not being limited to just DSP-like or loop centered applications.

This is a very useful characteristic, since the amount of parallelism during the execution of complex programs usually varies [8]. Furthermore, even if one considers perfect conditions and uses a large range of techniques such as trace scheduling, branch prediction and loop unrolling, the parallelism is limited [9]. With the BT mechanism we could assure software compatibility in any level of the design cycle, without requiring

any tools for the hardware/software partitioning or special compilers, with high performance and low-power execution of Java applications.

This work shows the details on how this binary translation works, and reveals that such system can be easily implemented if one considers the nature of stack machines such as the Java architecture. Moreover, we evaluate the costs of this analysis and present the area overhead, performance improvements and energy consumption due to the use of this technique, for several real world examples.

This paper is organized as follows. Section 2 discusses related work in the field of dynamic binary translation for performance improving. Section 3 presents the Java processor and the reconfigurable array. Section 4 demonstrates details of the BT algorithm and the advantages of using a Java processor as target architecture. Section 5 presents the simulation environment and the results regarding the use of this technique. Finally, the last section draws conclusions and introduces future work.

2. RELATED WORK

The Binary Translation technique was first proposed for translating at run-time the assembly code of an application (i.e. its binary code) from an old (legacy) machine to an equivalent code for a newer machine. However, new advantages were found in its use. Although counterintuitive, BT has been used to achieve high performance allied to low energy dissipation [10-11]. One approach consists in monitoring of the program binary execution, detecting frequently executed software kernels and optimizing them. Existing optimizations include dynamic recompilation and caching of previous BT results.

Concerning recent BT examples, the HP Dynamo is based on software that analyzes the application at runtime in order to find the best parts of the software for the binary translation [10]. The Transmeta Crusoe is based on a VLIW processor that uses binary translation at runtime to better exploit the ILP of the application, where the source machine is the x86 instruction set [11]. 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.

Another technique for performance increasing is the use of reconfigurable systems, implementing some parts of the software in a hardware reconfigurable logic. Huge software speedups [12] as well as a system energy reduction have been achieved [13]. Processors like Chimaera [14] and ConCISe [15], have 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 as 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. However, the use of reconfigurable arrays is always limited to some kind of static analysis of the code. This means that there is no total software compatibility and special tools are needed in the design cycle.

In [4], Stitt, Lysecky and Vahid presented the first studies about the benefits and feasibility of dynamic partitioning using reconfigurable logic, combining these both techniques cited before. In [5], 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, these works use 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, this technique is limited to critical parts of the software, as some loops.

On the other hand, we use a tightly coupled coarse grain reconfigurable array, which has four main advantages: it allows a quick reconfiguration; the huge power dissipation and control overhead of a fine grain architecture is avoided; the overhead of the communication between the system and the array is minimal, consequently saving power; and finally, a relative small amount of memory for keeping the configurations of the array is necessary.

Adding to this last advantage, the hardware to implement the Binary Translation, used to detect at run-time the sequence of instructions to be executed in the array in a Java processor, becomes simpler, thanks to its stack machine nature, as we will explain in details later. This two main characteristics, simple combinational logic and small amount of memory required, allows the construction of a machine to detect and optimize all sequences of instructions at real time of a software executing in a Java processor. As a consequence, one can explore every part of the algorithm, even in those which do not present a high level of parallelism, since this technique can explore vertical sequences of instructions, which are not necessarily data independent. In order to demonstrate that, we compare the processor coupled with the reconfigurable array with VLIW versions with the same instruction set.

Furthermore, 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.

3. JAVA ARCHITECTURES AND THE RECONFIGURABLE ARRAY

The architecture used is a Java processor [16], which has a five stages pipeline: 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 storage (used to keep values of the local variables of a method), instead of using the main memory for this purpose, as done in other published stack machines.

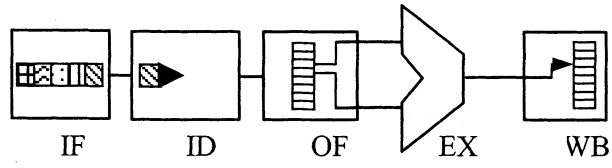

Figure 1. Pipelined Java Processor [16]

The used reconfigurable array is tightly coupled to the processor. It is implemented as an ordinary functional unit in the execution stage, using the same approach as Chimaera, cited before. The array is divided in blocks, called cells. The operand block (a sequence of Java 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. 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.

For each cell in the array, 327 reconfiguration bits are needed. Consequently, if the array is formed by 3 cells, 971 bits in the reconfiguration cache are necessary. To these reconfiguration bits one must add 58 extra bits of additional information, such 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.

A separated unit is responsible for dynamic analysis (Binary Translation) of 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 is a certain number of instructions which are worth being 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.

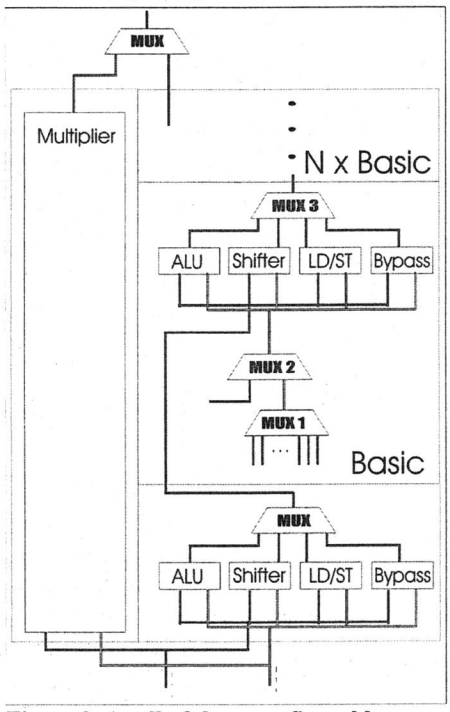

Figure 2. A cell of the reconfigurable array

For performance comparisons we have used a VLIW version of the same Java processor [17], which 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 and the search for ILP in the Java program is done at the bytecode level.

4. THE BT ALGORITHM

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.

The search for the sequence of instructions in the Java program is done at the bytecode level, 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.

In the sequence of instructions, observed in figure 3a, 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 variable 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, forming two operand blocks (Figure 3b). Hence, their operation can occur in the reconfigurable array, and will be saved in the reconfiguration cache (figure 3c).

When an operand block is found, a write command for the reconfigurable cache is sent. This command saves the content of the buffer to this cache. The content of the buffer is the list of the decoded instructions of the operand block. This list is made in real time, as the instructions are fetched from memory. When a basic block limit is found, as well as the end of an operand block, this buffer is cleaned waiting for a new operand block. The size of the buffer is of 20 eight-bit registers long, since this number is enough to keep each array configuration.

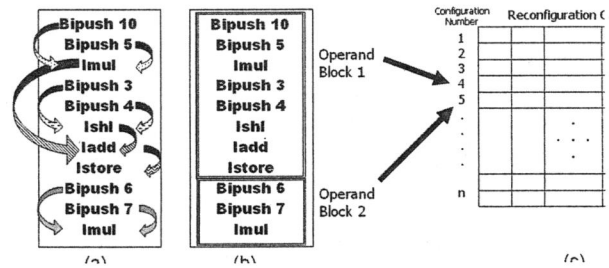

Figure 3. The process of building an operand block

Additionally, when an operand block is found, besides its list of instructions, a Program Counter (PC) value must also be saved. This is how the detector will know when a sequence of instructions will be configured to execute in the array with the configuration previously saved in the cache. The PCs are saved in a bitmap list. This way, both write and read are fast, and as just one bit for address is necessary, no large amounts of memory are needed.

The organization of the reconfiguration cache is fully associative, so any address can go to any place in the cache. In this first approach, we are always considering that we have enough space in the cache memory to save all configurations. In the future, however, we can use traditional replacement cache techniques to manage the cache.

As explained before, the detection can be done at run time. The main advantage of the run-time analysis is that next time that the sequence of instructions is detected it can already be executed in the array. If this work was not done at real time, some cycles

could be lost for the detection of sequences, and as consequence, the sequence that is being processed could be executed again, and it would not be configured in the array.

5. RESULTS

Our experiments are supported by simulation, where different versions of the Java Processor execute algorithms used in the domain of embedded systems, 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 [18].

Different types of algorithms were implemented and simulated over the architectures described in Section 3, from simple ones to a complex full MP3 player, as can be seen in Table 1. In the IMDCT (Inverse Modified Discrete Cosine Transformation) example we have also developed three unrolled versions, in order to expose the parallelism. We also use 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. As a more complex example, we have a complete MP3 player that executes 4 frames of 40kbit, 22050Hz, joint stereo.

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 version, 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, 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 the 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). 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 different configurations of the array were used more than once, or, in other words, how many instruction sequences were saved to the cache and were reused in the array. In the second column we demonstrate the amount of reuse obtained for these sequences. The next column shows the maximum number of cycles that were necessary to reconfigure the different configurations of the array from the cache. The forth column exhibits the maximum number of cells that these sequences occupied in the array.

Table 1. Performance (number of cycles) of the architectures and data about the reconfigurable array

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

In Figure 4 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.

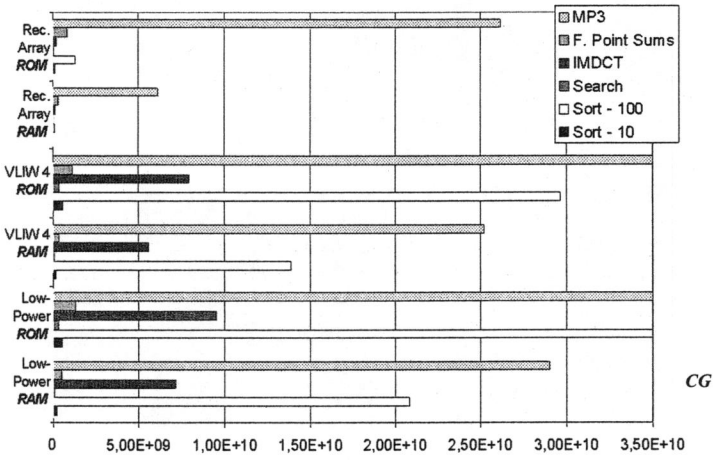

Figure 4. Energy consumption in RAM and ROM of the Java processor with and without the reconfigurable array

As can be observed, the array saves energy in ROM accesses, since instructions that would be fetched from 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 is saved, because now there is 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 5, even with the increment of the additional logic of BT to detect the sequence and the reconfiguration cache on it, there are still gains in terms of energy consumption in some algorithms. Even with more power being consumed by the additional cache, savings are achieved from the great number of instructions that would normally use all five processor pipeline stages and its sequential logic, and are now being executed in the array.

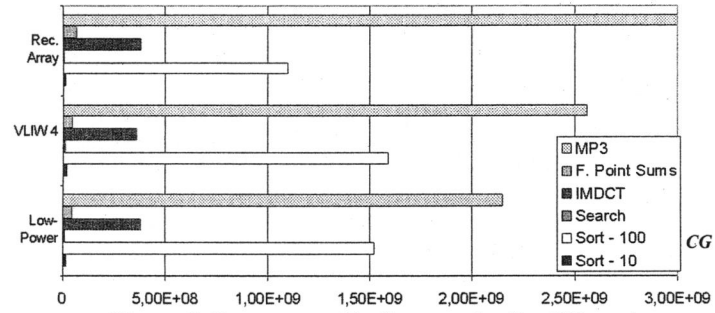

Figure 5. Energy spent in the cores by the different architectures

Finally, in Figure 6, we show the total energy consumption of the system considering the RAM, ROM, the core and the additional logic that makes the dynamic code analysis. It is important to note that great gains were achieved in energy consumption in all algorithms, proving the effectiveness of the proposed technique.

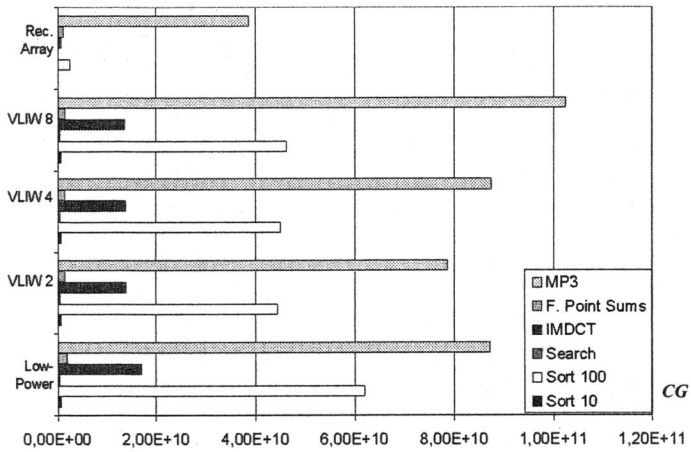

Figure 6. Total energy consumption of the architectures

Table 2 shows the area occupied by the Low Power and VLIW versions of our Java processors, and 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). Table 2 also shows the cache and the BT logic responsible for the detection of the sequences of instructions and to make the reconfiguration. As can be observed in this table 3, 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 Leonardo Spectrum for Windows [19]. The area taken by the processors was computed in number of gates, after synthesis of the VHDL versions of these processors.

Table 2. Area occupied by the architectures

Processor	Low-Power	VLIW (instructions per packet)		
		2	*4*	*8*
Area	131215	213850	367675	675395

Table 3. Area occupied by the Java processor and the array logic

#Cells / *#Reconfs*	*2*	*3*	*4*	*7*	*10*
5	723141	960049	1196957	1907680	2618403
10	1005681	1372351	1739021	2839031	3939040
15	1288222	1784654	2281086	3770382	5259678
20	1570762	2196956	2823150	4701733	6580315
40	2700923	3846166	4991408	8427137	11862865

Huge energy savings are achieved when compared to any architecture (almost 11 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).

6. CONCLUSIONS AND FUTURE WORK

We showed in this paper the costs of implementing binary translation to work with a coarse-grain array in a native Java processor in order to boost performance and reduce energy consumption. The use of such technique is totally transparent for the software designer, since the search of the potential sequence of instructions is done at run-time. Furthermore, we demonstrated that there is no need for huge available parallelism in the application, such as it is in VLIW and superscalar architectures, to achieve good results. Moreover, the implementation of this technique in a Java processor shows great advantages because of the specific stack-like architecture.

For future work, more algorithms concerning the embedded system domain and optimizations aimed at the reconfigurable arrays will be evaluated. Furthermore, starting a Chip-Multiprocessing approach, we will use another Java processor for the analysis of instructions instead of a dedicated hardware, and we will evaluate the costs and real-time constraints of using such technique.

7. REFERENCES

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

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

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

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

[7] Beck, A. C. S.., Carro, L. "Dynamic Reconfiguration with Binary Translation: Breaking the ILP Barrier with Software Compatibility". In *Design Automation Conference (DAC)*, 2005

[8] 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

[9] Wall, D. W. "Limits of Instruction-Level Parallelism". *In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991, 176 – 189

[10] Bala, V., Duesterwald, E., Banerjia, S., "Dynamo: A Transparent Dynamic Optimization System". In *Conf. on Programming Language Design and Implementation*, 2000

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

[12] Gupta, R. K., Micheli, G. D., "Hardware-software co-synthesis for digital systems". In *IEEE Design and Test of Computers*, Vol. 10, N. 3, 1993, 29-41.

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

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

[15] 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, CA, 1999, 92–100.

[16] 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)

[17] 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

[18] 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

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