Measuring the Efficiency of Cache Memory on Java Processors for Embedded Systems

Antonio Carlos S. Beck, Mateus B. Rutzig and Luigi Carro

Department for Computer Science, Federal University of Rio Grande do Sul-Porto Alegre, Brazil,
e-mail: {caco, mbrutzig, carro}@inf.ufrgs.br

ABSTRACT

Java, with its advantages as being an overspread multiplatform object oriented language, has been gaining popularity in the embedded system market over the years. However, because of its extra layer of interpretation, it is also believed that it is a slow language while being executed. Nevertheless, when this execution is done directly in hardware, Java advantages caused by its stack nature start to appear. One of these advantages concerns memory utilization, impacting in less accesses and cache misses. In this work we analyze this impact in performance and energy consumption, comparing a Java processor with a RISC one based on a MIPS architecture with similar characteristics.

Index Terms: Cache Memory, Java, Power consumption, MIPS, Stack Machines

1. INTRODUCTION

While the number of embedded systems does not stop to grow, new and different applications, like cellular phones, mp3 players and digital cameras keep arriving at the market. At the same time, embedded architectures are getting more complex, smaller, more portable and with more stringent power requirements, posing great challenges to the design of this kind of systems. Additionally, another issue is becoming more important nowadays: the necessity of reducing the design cycle.

This last affirmative is the reason why Java is becoming more popular in embedded environments, replacing day by day traditional languages. Java has an object oriented nature, which facilitates the programming, modeling and validation of the system. Furthermore, being multiplatform, a system that was built and tested in a desktop, for instance, can migrate to different embedded systems with a small number of modifications. Moreover, Java is considered a safe language, and has a small code size, since it was built to be transmitted through internet.

Not surprisingly, recent surveys revealed that the presence of Java in devices such as consumer electronics (digital TV, mobile phones, home networking) as well as industrial automation (manufacturing controls, dedicated hand held devices) is increasing day by day. It is estimated that more then 600 million devices will be shipped with Java by 2007 [1][3]. Furthermore, it is predicted that more then 74% of all wireless phones will support Java next year [2][4]. This trend can be observed nowadays, where most of the commercialized devices as cellular phones 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.

However, Java is not targeted to performance or energy consumption, since it requires an additional layer in order to execute its bytecodes, called Java Virtual Machine (JVM), responsible for the multiplatform feature of Java. And that is why executing Java through the JVM could not be a good choice for embedded systems.

A solution for this issue would be the execution of Java programs directly in hardware, taking off this additional layer, but at the same time maintaining all the advantages of this high level language. Using this solution highlights again another execution paradigm that was explored in the past [5]: stack machines. Since the JVM is based on a stack machine, obviously the hardware for native Java execution should follow the same approach, in order to maintain full compatibility.

Additionally, embedded applications today are not as small to fit in the cache and, at the same time, they are not as large to use configurations of traditional desktop environments either. Nevertheless, in nowadays embedded systems they can consume up to

50% of the total energy of the system [6] and occupy a significant part of the total area of the chip. Adding these facts to all the constraints cited before explains why cache memories have gained more importance in the embedded domain.

Hence, in this paper we show the advantages of Java machines regarding energy consumption and performance of the cache memory when comparing to traditional RISC ones, considering the particularities of embedded systems. We demonstrate that, thanks to the particular execution method based on a stack, these machines have less memory accesses and less cache misses concerning the instruction memory. This way, the designer can take advantage of all the benefits of a high level language such Java, shrinking the design cycle, and at the same time increasing the overall performance – proving that Java can also be a high performance and low power alternative when executed directly in hardware.

This work is organized as follows: Section 2 briefly shows a review of the existing Java processors and some recent works concerning cache memory for embedded systems. In Section 3 we discuss the processors used in the evaluation and its particularities. Section 4 presents the simulation environment and the results regarding performance of the cache memory on the systems with various configurations. The last section draws conclusions and introduces future work

2. RELATED WORK

In the literature, one can rapidly find a great number of Java processors aimed at the embedded systems market. Sun's Picojava I [7], a four stage pipelined processor, and Picojava II [8], with a six stage pipeline, are probably the most studied ones. Even though the specifications of such processors allow a variable size for the data and instruction caches, there is no study about the impact of stack execution on these cache memories.

Furthermore, we can cite some works regarding cache memory specifically for embedded systems. In [9], compiler techniques, memory access transformations and loop optimizations are used in order to decrease the number of cache accesses, hence increasing performance and saving power. In [10] a technique aimed at reconfiguring the cache by software in order to change its associativity configuration with the objective of saving power is presented. In [11], taking advantage of the frequent values that widely exist in a data cache memory, a technique is proposed to reduce the static energy dissipation of an on-chip data cache. Other approaches, such as the use of scratchpads in the embedded domain [12], have been applied as well.

In this specific work, we study the effect of cache memories in Java based embedded systems, and why it differs from traditional ones because of its stack machine architecture.

3. ARCHITECTURES EVALUATED

It is very hard to compare two different architectures, even when they are of the same family. Comparing two different hardware components that execute instructions in a different way is even more difficult. As a consequence, in this work we try to make general characteristics, such as number of pipelines stages, equivalent.

A. Architecture Details

The RISC processor used is based on the traditional MIPS-I instruction set [13]. It has a five stages pipeline: instruction fetch, decode and operand fetch, execution, memory access and write back. This MIPS implementation has 32 registers in its bank.

The Java processor used is the Femtojava [14], which implements a subset of Java instructions. It does not support dynamic allocation of objects neither garbage collection. Consequently, all objects are statically allocated in Java programs. This processor has the same number of pipeline stages that the RISC one has. However, the structure of the pipeline is a little different, as shown in Figure 1.

The first stage is instruction fetch, as in the MIPS processor. The second stage is responsible for decoding instructions. The next one is the operand fetch. It is important to note that, in opposite to the MIPS processor, operand fetch cannot be performed at same time as decoding, because it is not known previously which operands to fetch (this data is not intrinsically available in the opcode), as in MIPS architecture. On the other hand, there is no instruction that accesses the register bank and the memory at the same time. As a consequence, the memory access, which is a separated pipeline stage in the MIPS, is made in the forth stage together with the execution. The write back to the register bank is the last stage, in both processors.

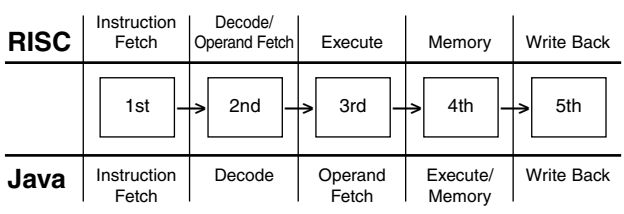

Figure 1. Differences in the pipeline stages between the two processors

The stack in the Java processor is implemented in an internal register bank. In our benchmark set, the stack does not increase more than 32 values. This way, there are 32 registers in the bank implementing the stack, the same number of registers used in the MIPS processor. However, it is important to mention that local variables in the Java processor are also saved in the stack (hence, in the internal register bank). It facilitates passing parameters between methods, since Java uses the concept of frames. On the other hand, local variables in the MIPS processor are saved in the main memory, not in the register bank, increasing the number of memory accesses for this processor and, as a consequence, giving some advantage to the the Java Processor. In order to keep the comparison as fair as possible, in all our benchmark set we implemented these local variables as global. This way, there will be accesses to the main memory in both processors. Moreover, data in the memory will also be accessed in static variables (such as vectors) and information about methods.

B. Computational Methods

To better illustrate the difference between these two paradigms, let's start with an example. In RISC machines, to make a sum of two operands, just one instruction is needed:

add r1, r2, r3

where r2 and r3 are the source operands and r1 is the target. In stack machines, however, this operation needs three different instructions:

push OP1
push OP2
add

At first sight, this characteristic could be considered as a disadvantage. However, stack machines keep the operands always in the stack. This means that the next instructions will use operands that are already present in the stack, generated by the instructions executed before. Hence, the push instructions are not needed anymore. As more operations are executed, more data is reused from the stack. Since it is well known that each basic block usually has more than just on single operation, stack machines can be very economic concerning the number of instructions. This characteristic will reflect in the instruction cache hits and misses, as it will be shown in next section.

Furthermore, in Java, instructions have a variable length: 1, 2 or 3 bytes; in opposite to the MIPS instructions, with a fixed size of 4 bytes. In order to avoid bubbles in the pipeline structure, the Java processor is implemented to fetch 4 bytes at each cycle. This also makes the comparison impartial: the size of each word in the cache memory has exactly the same number of bits for both processors. As we show in the next section, the smaller size of the instructions together with the stack paradigm leads to an incredible difference in the number of instruction cache misses.

4. RESULTS

Three steps were necessary to gather the results. Firstly, using a SystemC description of both processors, traces of memory accesses for each application were generated. Then, another simulator, which in turn uses the traces generated before, was used. It has as inputs the cache size, associativity and spatial locality, in order to better explore the design space. After that, the cache simulator gives as output the number of cache misses and hits depending on the input configuration. Finally, using this information, the ECACTI tool [15] was used for the power consumption evaluation. It is very important to mention that this simulation also takes into account the static power. Statistics for the off-chip memory consumption as the bus were taken from [17], considering this external memory working at 50 Mhz, 2V, implemented in a 0.25 technology.

Different types of algorithms were chosen and simulated over the architectures described in Section 3: Bubble (sort 100 elements); IMDCT (plus three unrolled versions); an algorithm to solve the Crane problem; three algorithms that belong to the Java Grande Benchmark set [16]: Lufact, that solves a N x N linear system using LU factorization followed by a triangular solve; SOR, performs 100 iterations of successive over-relaxation on a N x N grid; Sparse which is a sparse matrix multiplication; and a complete MP3 player that executes 1 frame at 40kbit, 22050Hz, joint stereo. Each algorithm has two versions, in Java and C, compiled by Sun´s Java Compiler [18] and GCC [19], respectively.

First of all, we analyze the number of memory accesses of both processors. It is important to remember that, as explained earlier, the stack in our Java processor is implemented in a register bank instead of using the main memory. Table 1 demonstrates the number of accesses in the data and instruction memories. Dividing the table in data and instruction memories, the first and second columns show the number of accesses of each architecture, and the last column demonstrates the relative difference of accesses that

Table 1. Number of Accesses in the Data and Instruction Memories

	Data Memory			*Instruction Memory*		
Algorithm	***Java***	***MIPS***	***Difference***	***Java***	***MIPS***	***Difference***
Bubble 100	112933	113412	1.00	96508	314240	3.25
Crane	8097	9877	1.21	7700	23703	3.07
IMDCT N	10741	8664	0.80	13475	39241	2.91
IMDCT 1	6305	4231	0.67	8326	22173	2.66
IMDCT 2	6003	3912	0.65	7848	19740	2.51
IMDCT 3	4234	2143	0.50	3824	5998	1.56
Lufact	12617	22927	1.81	14233	60658	4.26
MP3	475887	746186	1.56	503251	2513397	4.99
SOR	197112	231980	1.17	204164	777951	3.81
SPARSE	115549	131753	1.14	165725	602572	3.63
Average			***1.05***			***3.26***

the RISC architecture has, when compared to the Java processor. As it can be seen, in the average, the Java processor has fewer accesses in instruction memory and almost the same in data one. The enormous difference concerning the instruction memory was explained in the last section. In the next sub-sections we analyze the impact of this fact in the performance and power consumption.

For the experiments with the cache, we analyzed instructions and data caches separately, varying their size (64, 128 and 2048 lines); spatial locality (one and two words per line); and associativity (direct mapped, 2-way and 4-way associative).

A. Performance

Firstly, we demonstrate in table 2 (at the end of this work) the number of instruction and data cache misses in both processors, using different cache configurations (because of space limitations, we are considering 64 and 2048 words without exploring spatial locality). As can be observed, concerning instruction cache accesses, the configuration that benefits the most the Java processor (the one that results in the bigger difference of cache misses between the Java and MIPS processors, when dividing the total number of memory accesses by the number of cache misses) is the follow: 64 lines; 2 words per line and 4-way associative. For the MIPS processor: 256 lines, 2 words per line and direct mapped. Even in this configuration there is a huge advantage in instruction cache misses for the Java processor. This proves that, no matter the cache characteristics, stack machines will always present advantages in instruction memory accesses.

We repeat the same analysis for the data cache. The best configuration for the Java processor is: 256 lines; 2 words per line and 2-way associative; and for MIPS: 64 lines, 1 word per line and direct mapped. In this case, depending on the algorithm, there is a small advantage for the Java processor. Another important thing to point out is that there is almost no difference between different cache configurations: the proportion of misses in data caches is almost the same.

Finally, figures 2 and 3 show the average number of data and instruction cache misses for each algorithm of the whole benchmark set considering all possible configurations (18 in the total). This graphic summarizes what was affirmed before: an expressive difference in instruction cache misses and an equivalence in the data memory misses, with a small advantage for the Java processor in some algorithms.

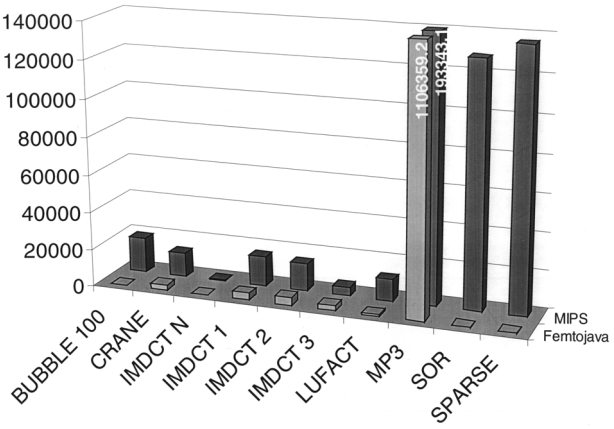

Figure 2. Average number of instruction cache misses

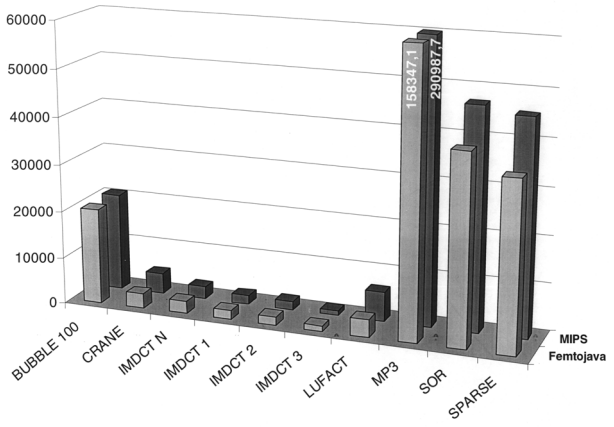

Figure 3. Average number of data cache misses

B. Power and Energy

In this sub-section we analyze the power spent by cycle and the total energy consumption caused by cache misses. Table 3 (at the last page)shows the energy consumption in both processors following the same format of the previous table. As it could be expected, the energy consumption is directly proportional to the number of misses. This way, huge energy savings concerning the instruction memory is achieved, and a very similar consumption in data memory is obtained.

Finally, we demonstrate the average power and energy consumption for each algorithm considering the whole set of cache configurations. As it is demonstrated in Figure 4, considering the instruction cache, there is less power consumption in the Java processor. Since there are less cache misses, less accesses to the off-chip memory are necessary. Moreover, as there are less overall memory accesses, the difference in energy consumption between the two processors is even higher (figure 5).

The same process occurs in data accesses. Power is saved because there are less off-chip accesses per cycle (figure 6) and less energy is spent because there is less overall accesses (figure 7) – although in a different proportion when comparing to the instruction memory.

In all results demonstrated before no compiler optimization flags were used, in order to make the

comparison as fair as possible. Optimizations as forcing values to be in the registers, loop unrolling and method inlining are not available for Java Compilers yet – although they are possible to be made in java bytecodes. However, in order to analyze the impact of such optimizations in the results, in the figure 8 and 9 we make a first analysis, comparing the number of memory accesses and cache misses (instruction and data memory, respectively) of the Femtojava Processor with the MIPS processor when the benchmarks are compiled with the higher possible level of optimization (-O3). As it can be observed, in certain cases there is a huge impact in the number of accesses.

It is important to remember that Java technology is relatively new, and the development of new compilers and the improvement of them is an active area, in opposite to C compilers. Moreover, most compilers optimizations are not used in Java because of the interpreted nature of most applications. As Java hardware machines become available, it is very likely that these common optimizations will be included in future versions of Java compilers and, as a consequence, the advantages of stack machines concerning memory accesses will remain, as we demonstrated when we compared both architectures with the same level of compiler resources.

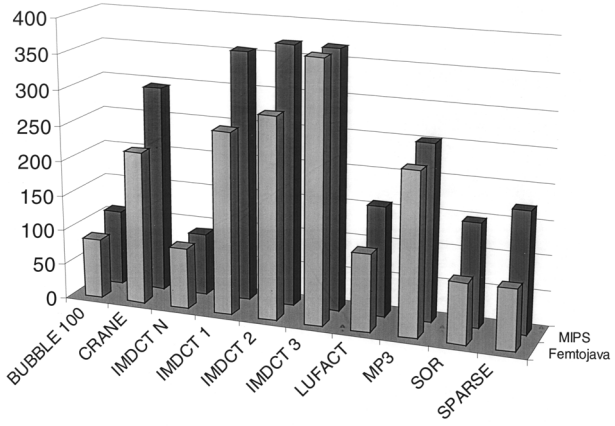

Figure 4. Average power consumption: instruction memory

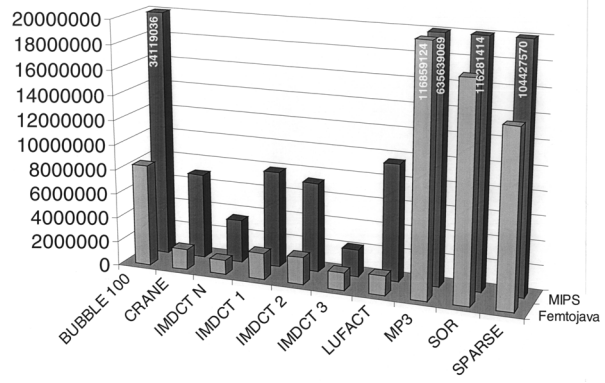

Figure 5. Total energy consumption: instruction memory

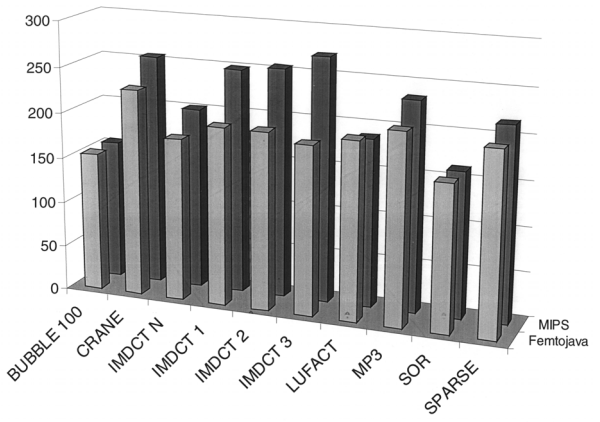

Figure 6. Average power consumption: data memory

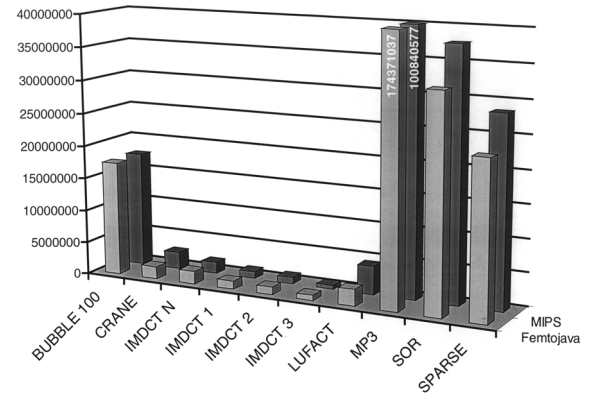

Figure 7. Total energy consumption: data memory

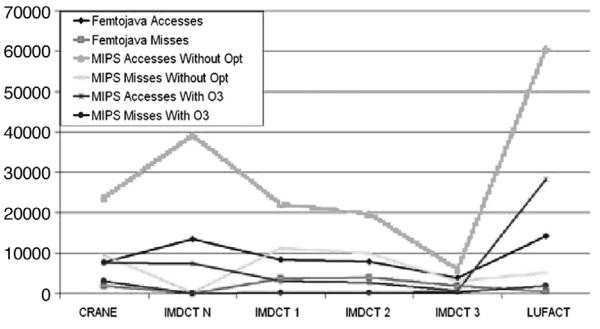

Figure 8. Comparison: number of instruction accesses when using GCC optimization

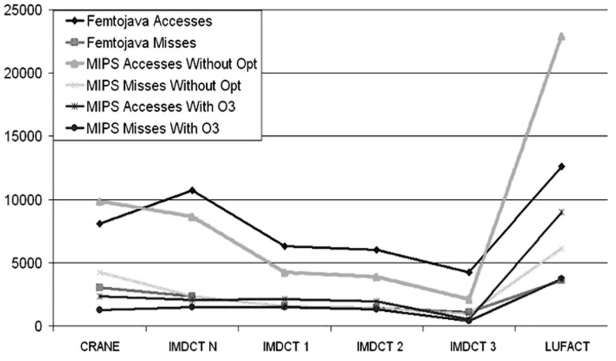

Figure 9. Comparison: number of data accesses when using GCC optimization

Table 2. Cache Misses: Data and Instruction Caches

Algorithm	Femotojava						MIPS					
	Direct Mapped		2-Way Associative		4-Way Associative		Direct Mapped		2-Way Associative		4-Way Associative	
	Size 64	Size 2048	Size 64	Size 2048	Size 64	Size 2048	Size 64	Size 2048	Size 64	Size 2048	Size 64	Size 2048
Instruction Misses												
BUBBLE 100	46	46	46	46	46	46	42210	161	56742	161	93577	161
CRANE	5221	1986	5414	1376	5492	1233	20287	13157	19817	13586	19780	12703
IMDCT N	86	85	86	85	86	85	551	246	701	246	1001	246
IMDCT 1	7410	534	7454	534	7544	534	22173	19033	22173	19033	22173	19033
IMDCT 2	7847	1065	7847	1065	7847	1065	19740	19740	19740	19740	19740	19740
IMDCT 3	3824	3824	3824	3824	3824	3824	5998	5998	5998	5998	5998	5998
LUFACT	2025	753	2104	402	1355	398	26497	6727	30409	6542	32331	4353
MP3	339157	177658	348338	126327	327887	115892	1619910	1155028	1591398	1238030	1595987	1252137
SOR	212	208	212	208	212	208	407200	1793	487906	1715	568243	1510
SPARSE	837	189	327	189	266	189	474916	1098	541970	1134	547256	1156
Data Misses												
BUBBLE 100	21746	18767	23239	18767	23167	18902	22363	18909	23355	19370	23367	19154
CRANE	3137	2865	3258	3016	3176	2972	4434	4350	4582	4236	4840	4215
IMDCT N	3017	2925	2931	2908	2933	2906	2961	2904	2922	2903	2925	2902
IMDCT 1	2185	2114	2124	2103	2126	2101	2127	2096	2109	2098	2122	2098
IMDCT 2	2096	2028	2034	2014	2038	2012	2042	2012	2027	2011	2034	2009
IMDCT 3	1489	1360	1081	980	1137	980	1192	1111	1201	984	1026	977
LUFACT	4328	3123	4351	3302	4403	3364	6877	5627	6662	5804	6635	5673
MP3	183366	159031	177133	157871	170852	155330	349299	278758	332907	276059	311434	272444
SOR	56742	41887	48171	33043	38993	27759	58144	38855	54569	40690	51532	40120
SPARSE	48527	28501	47378	33527	40011	28940	48392	43165	43913	43268	45285	43265

Table 3. Energy Consumption: Data and Instruction Memories

Algorithm	Femotojava						MIPS					
	Direct Mapped		2-Way Associative		4-Way Associative		Direct Mapped		2-Way Associative		4-Way Associative	
	Size 64	Size 2048	Size 64	Size 2048	Size 64	Size 2048	Size 64	Size 2048	Size 64	Size 2048	Size 64	Size 2048
Instruction Misses												
BUBBLE100	4622500	9093762	5985222	9315584	9293107	11703380	31401292	29614310	41461715	30336657	66444558	38111566
CRANE	2396929	1446902	2578221	1251341	2867143	1388874	9017035	7017051	9160023	7313817	9939053	7572986
IMDCT N	676353	1298316	866589	1329787	1328378	1663183	2086467	3780314	2698561	3871952	4159112	4842844
IMDCT 1	3277721	977414	3409005	999917	3722066	1205903	9677175	9011353	9980190	9183079	10718838	9731127
IMDCT 2	3424789	1125681	3532040	1150457	3793481	1344601	8615317	9039787	8885083	9210416	9542680	9698246
IMDCT 3	1668944	1751173	1721204	1784227	1848592	1878728	2617765	2746740	2699733	2798585	2899544	2946813
LUFACT	1466261	1612687	1696992	1520186	1892724	1870845	13194239	8153040	15557640	8266489	18351883	8956713
MP3	155850840	111980692	166367564	95259123	175360694	103843738	749623160	656697406	773306823	700529723	859700972	767902453
SOR	9823544	19278256	12706350	19748227	19704116	24799638	195407580	73822337	237543484	75590596	294853991	94762686
SPARSE	8232485	15655977	10374209	16037592	16030695	20137947	213362901	57074229	247690881	58477708	269870020	73394632
Data Misses												
BUBBLE 100	13841556	17450956	16006035	17829329	19826680	20672943	14104256	17547680	16080693	18098734	19950178	20824238
CRANE	1605763	1804099	1765653	1896779	2008290	2080738	2194871	2511890	2389788	2519971	2823847	2756449
IMDCT N	1685268	2074613	1802174	2111544	2168286	2376472	1564401	1871620	1670252	1909573	1965569	2123483
IMDCT 1	1150195	1362274	1214547	1386094	1429387	1541291	1028694	1160654	1080482	1184413	1228508	1289035
IMDCT 2	1101189	1302575	1160929	1324046	1366279	1471772	980432	1100084	1028919	1121467	1163766	1217458
IMDCT 3	780828	893115	681523	770778	847296	875507	565610	605839	598823	570791	603341	621193
LUFACT	2284397	2323109	2469519	2438169	2917931	2773195	3767177	4203988	4004314	4357870	4773290	4876462
MP3	93985029	102628840	98201343	104301335	111909632	115130406	171384300	171618926	175397484	174102160	192336542	191217904
SOR	31461830	33781470	30891475	31225477	34046601	34145275	33670458	35957658	35531631	37416209	42254204	42943643
SPARSE	24376877	21239089	25540304	23546070	26601941	24705859	25097529	28099203	25196879	28713799	30201961	31971252

5. CONCLUSIONS

Java became popular mainly because it is an object oriented language and for its ease of use. It is also common sense that it is a slow language while being executed. In this paper we showed that when executing Java directly in hardware, advantages start to appear. In the case of the memory sub-system, specifically instruction caches, stack machines can have a higher cache hit ratio that, besides increasing the overall performance, reflects in less dynamic and static power consumption. The data memory presents very similar results, when the comparison is made with equivalent configurations. This way, besides speeding up the design cycle, Java can also bring advantages concerning performance, area and power consumption. Our next step is to compare complete architectures of both processors.

REFERENCES

[1] The Embedded Software Strategic Market Intelligence. Java in Embedded Systems. http://www.vdc-corp.com/
[2] S. McAteer. Java will be the dominant handset platform. www.microjava.com/articles/perspective/zelos/.
[3] D. Mulchandani. Java for Embedded Systems. Internet Computing, 31(10):30–39, May 1998.
[4] Lawton, "Moving Java into Mobile Phones", Computer, vol. 35, n. 6, 2002, pp. 17-20.
[5] P. Koopman, Stack Computers: The New Wave, Halsted Press, 1st edition, 1989
[6] S. Segars, "Low power design techniques for microprocessors," Int. Solid-State Circuits Conf. Tutorial, 2001.
[7] J. M. O'Connor, M. Tremblat, "Picojava-I: the Java Virtual Machine in Hardware", IEEE Micro, vol. 17, n. 2, Mar-Apr. 1997, pp. 45-53
[8] Sun Microsystems, PicoJava-II Microarchitecture Guide, Mar. 1999.
[9] W. Shiue, C. Chakrabarti, "Memory Design and Exploration for Low Power, Embedded Systems", The Journal of VLSI Signal Processing - Systems for Signal, Image, and Video Technology, Vol. 29, No. 3, Nov. 2001, pp. 167-178
[10] C. Zhang, F. Vahid, W. Najjar, "A highly configurable cache architecture for embedded systems", Proceedings of the 30th annual international symposium on Computer architecture (ISCA), 2003
[11] C. Zhang, J. Yang, F. Vahid," Low Static-Power Frequent-Value Data Caches", Proceedings of the. Design, Automation and Test in Europe Conference (DATE), 2004
[12] R. Banakar, S. Steinke, B. Lee, M. Balakrishnan, P. Marwedel, "Scratchpad Memory: A Design Alternative for Cache On-chip memory in Embedded Systems", Proc. of the 10th International Workshop on Hardware/Software Codesign, CODES, 2002
[13] J. L. Hennessy, D. A. Patterson, Computer Organization and Design: The Hardware/Software Interface, Morgan Kaufmann Publishers, 3th edition, 2005
[14] A.C.S.Beck, L. Carro, , "Low Power Java Processor for Embedded Applications". In: IFIP 12th International Conference on Very Large Scale Integration, Germany, 2003
[15] G. Reinman and N. Jouppi. Extensions to cacti, 1999. Unpublished document.
[16] D. Gregg, J. Power, "Platform Independent Dynamic Java Virtual Machine Analysis: the Java Grande Forum Benchmark Suite", Joint ACM Java Grande - ISCOPE Conf. Proc., 2001
[17] K. Puttaswamy, K. Choi, J. C. Park, V. J. Mooney, A. Chatterjee, P. Ellervee. "System Level Power-Performance Trade-Offs in Embedded Systems Using Voltage and Frequency Scaling of Off-Chip Buses and Memory", ISSS'02., October, 2002
[18] Java Tecnology Homepage, http://java.sun.com/
[19] GCC Homepage, http://gcc.gnu.org/.