Object and Method Exploration for Embedded Systems Applications

Júlio C. B. Mattos
Federal University of Rio Grande do Sul
Informatics Institute
Porto Alegre - RS - Brasil
julius@inf.ufrgs.br

Luigi Carro
Federal University of Rio Grande do Sul
Informatics Institute
Porto Alegre - RS - Brasil
carro@inf.ufrgs.br

ABSTRACT
The growing complexity of embedded systems has claimed for more software solutions in order to reduce the time-to-market. However, while this software decrease the development time of the embedded system functionalities , it must at the same time help the handling of the embedded systems tight constraints, like energy, power and memory availability. Object orientation is now a common technique to write maintainable code, but its application to the embedded systems domain is withhold by the overhead in terms of memory, performance and code size. This paper introduces a methodology to explore object-oriented embedded software improving different levels in the software design, while dealing with different embedded systems requirements (power, memory area and performance). The proposed approach transforms the original OO code into an optimized code allowing the automatic configuration of a solution for a specific application. Experimental results with an MP3 player show a large design space exploration with different solutions in terms of performance, energy and memory.

Categories and Subject Descriptors
C.3 [**Special-Purpose and Application-Based Systems**]: *Real-time and embedded systems;* D.1.5 [**Programming Techniques**]: Object-oriented Programming.

General Terms
Languages, Measurement, Performance.

Keywords
Embedded Software, Object-Oriented, Design Space Exploration.

1. INTRODUCTION
Nowadays, the embedded system market does not stop growing, and new products with different applications are available. These systems are everywhere, for example, mobile telephones, cars, videogames and so on. In embedded applications, requirements like performance, reduced power consumption and program size, among others, must be considered. Moreover, the embedded systems complexity is increasing in a considerable way.

SBCCI'07, September 3–6, 2007, Rio de Janeiro, Brazil.

Embedded systems are heterogeneous systems that cover a broad range of algorithms implemented on hardware and software. In the past, hardware configurations dominated the field while today most of the applications are implemented in a mixed configuration where software constitutes the main part [1,2]. Probably in the future even more products will have most of their characteristics developed in software. In many cases software is preferred to a hardware solution because it is more flexible, easier to update and can be reused. Hence, software is more and more becoming the major cost factor for embedded devices [3, 4].

Over the years, embedded software coding has been traditionally developed in assembly language, since there are stringent memory and performance limitations [5]. On the other hand, the best software technologies use large amounts of memory, layers of abstraction, elaborate algorithms, and these approaches are not directly applicable in embedded systems [5]. However, hardware capabilities have been improved, and the market demands more elaborate products, increasing software complexity. Thus, the use of better software methodologies is clearly required, for example object orientation. Nevertheless these abstract software techniques require a high price in the embedded domain, and the problem of embedded software development for this market still exists.

One of the main stream software methodologies is the object-oriented paradigm. In the last decades the object-oriented mechanism has become the dominant programming paradigm. Object-oriented programming scales very well, from the most trivial problems to complex ones. In spite of object orientation advantages, the acceptance in the embedded world has been slow, since embedded software designers are reluctant to employ these techniques due the memory and performance overhead [6, 7, 8].

Moreover, over the past few years embedded developers have embraced Java, because this technology can provide high portability and code reuse for their applications [9, 10, 11]. In addition, Java has features such as efficient code size and small memory footprint, that stand out against other programming languages, which makes Java an attractive choice as the specification and implementation language of embedded systems. However, developers should be free to use any object oriented coding style and the whole package of advantages that this language usually provides. In any case, one must also deal with the limited resources of an embedded system.

As mentioned, the existing software methodologies are not adequate for embedded software development, because embedded software development should address different constraints from desktop software. In this way, this paper introduces a methodology to explore object-oriented embedded software improving different tasks in the system design. Our approach is

divided into two main parts where the embedded software exploration methodology can be improved.

The first part, called method exploration level, aims to improve the methods implementation (the algorithms that implement these methods). This exploration phase introduces a mechanism for the automatic selection of software and hardware IP components for embedded applications, which is based on a software IP library and a design space exploration tool.

The second part, called object exploration level, aims to explore the object organization to improve the dynamic memory management through the use of a design space exploration tool to allow an automatic selection of the best object organization. This approach is also compliant with classical OO techniques and physical embedded systems requirements. The overall goal is to provide high level object orientation support, while at the same time support optimized memory, power and performance for embedded systems.

Moreover, the tools here presented (one based on method and another based on object exploration) are mutually orthogonal, that is, their execution is independent. The designer can first use the method exploration tool and after the object exploration tool or vice-versa. Since the optimizations performed by each tool are orthogonal, this makes the complexity of the exploration simple.

This paper is organized as follows. Section 2 discusses related work in the field of embedded software optimization. Section 3 presents our approach to design space exploration of object oriented embedded software. Section 4 presents some results using a case study. Finally, section 6 draws conclusions and introduces future work.

2. RELATED WORK

Our method approach is similar to [12], but instead of aiming at a partitioning between software and hardware, it concentrates on algorithmic variations of software routines that are commonly found in a wide range of embedded applications. This way, it provides design space exploration for given platforms.

The optimizations (in method and object level) that we proposed are different from most of studies. Most approaches on embedded software optimization are based on traditional compilation optimization. There are lots of works that deal with compilation optimizations concerning performance, memory, energy [13, 14, 15]. Our approach differs from these, because our approach intends to improve the code before the compiler optimization. We believe that design decisions taken at higher abstraction levels can lead to substantially superior improvements.

There are several works that agree with our statement that object-orientation and Java produce huge overhead [6, 7, 16]. However, these papers only measure this overhead. The main problem with Java is the overhead produced by memory management. In this way, there are lots of works that proposed software and hardware solutions to reduce the overhead as well to make the CG more predicable [17, 18, 19]. These strategies are very different from ours. Our object exploration level works before the execution – when the memory management and garbage collector act.

The Real-Time Specification for Java (RTSJ) [20] introduces the concept of immortal memory to improve real-time aspects. The objects in immortal memory can be created and accessed without GC delay, but there is no mechanism for freeing those objects during the application execution. This approach is similar to our strategy of transforming dynamic objects into static ones. However, it is necessary to use the RTSJ (during application code) and modify the virtual machine to adopt this technique.

There are other approaches to improve memory behavior of Java programs, for example [21]. However, these techniques improve the data locality during the execution time. Another paper [22] proposes a region analysis and transformation for Java programs. But, in this approach, the code must be modified and it is necessary to extend the virtual machine to support region annotations and to provide region run-time support.

Shaham [23] presents a similar strategy. But, our proposed approach starts from a more radical point of view. Instead of trying just to improve the code written by the programmer, we try to automatically transform as many dynamic objects into static ones, in the goal to reduce execution time, while maintaining memory costs as low as possible. Thus, it provides a large design space exploration for a given application.

3. DESIGN SPACE EXPLORATION OF OO EMBEDDED SOFTWARE

This section shows the two main parts that our methodology is divided. The first part, called method exploration level, aims to improve the method implementation (the algorithms that implement these methods). The second part, called object exploration level, aims to explore the object organization to improve the dynamic memory management.

3.1 Method exploration level

This section shows the method exploration approach. The main idea is to explore different algorithm solutions (method implementations) for a certain application according to the embedded system requirements. This approach consists in the use of a software library, a set of different processor cores (but with the same instruction set), and a design space exploration tool to allow an automatic software and hardware IP selection. The software IP library contains alternative algorithmic implementations for routines commonly found in embedded applications, whose implementations are previously characterized regarding performance, power, and memory requirements for each processor core.

In our approach, the designer receives the application specification and after coding it in Java language using the software IP library, he/she submits the application to the design space exploration tool. This methodology is compliant with the component-based development, where a component is a self-contained part or subsystem that can be used as a building block in a larger system. Using component-based development style, the reuse become easier and increase the software productivity.

The software IP library contains different algorithmic versions of the same function thus supporting design space exploration. Considering a certain core (HW IP) and for each algorithmic implementation of the library functions, it measures the performance, the memory usage, and energy and power dissipation. This way, the characterization of the software library is performed according to physical related aspects that can be

changed at an algorithmic abstraction level. On hardware level, this approach uses different implementations of the same Instruction Set Architecture providing range solutions on performance, power and memory area.

Thus, this methodology allows the automatic selection of software and hardware IPs to better match the application requirements. Moreover, if the application constraints might change, for example with tighter energy demands or smaller memory footprint, a different set of SW and/or HW IPs might be selected.

Using this methodology, the space design exploration has several options to provide a final solution using a different combination of SW IPs and HW IPs. Using only a single core and different algorithmic versions of the same function, the designer has a good set of alternatives. However, when multiple cores are used, the range of solutions is hugely increased.

Figure 1 shows the design flow of this embedded SW exploration. After coding the application using de IP library, the designer submits the application to design exploration tool. The design exploration tool maps the routines of an embedded program to an implementation using instances of the software IP library, so as to fulfill given system requirements. The user program is modeled as a graph, where the nodes represent the routines, while the arcs determine the program sequence. The weight of the arcs represents the number of times a certain routine is instantiated. In our approach, different threads are modeled as parallel structures and can be mapped to different processor cores.

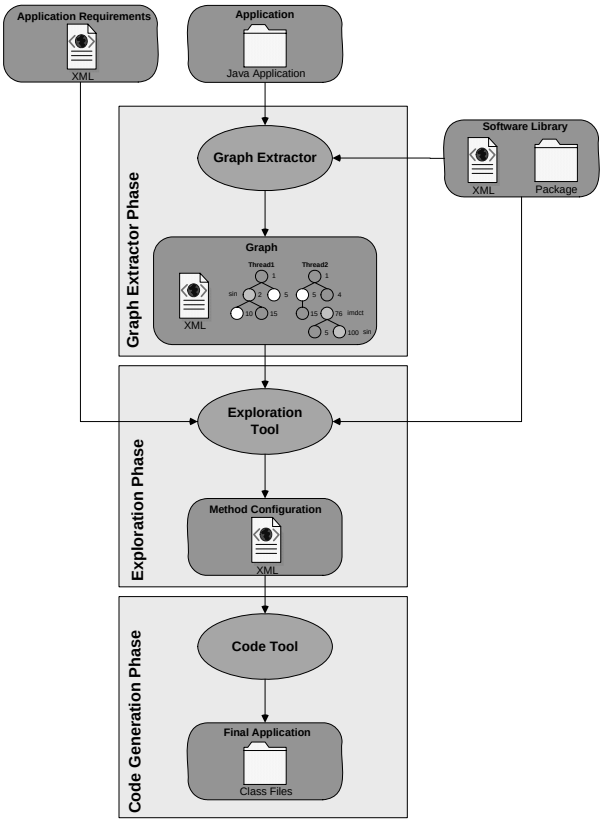

Figure 1. Method Exploration Level Design Flow.

In the exploration tool, before the search begins, the user may determine the application requirements (weights for power, delay and memory optimization). The tool automatically explores the design space and finds the optimal or near optimal mapping for that configuration. In the final step (code generation phase), the tool links the algorithm calls to their implementation according to the results obtained in exploration phase.

Problem Complexity

The design space to be explored can be large. The number of solutions is function of the number of the processor cores (HP IPs) and the number of the routines (SW IPs) that are available.

The equation of the problem complexity can be obtained as follows. There is a program with m different methods that use the IP library. Each method can be implemented by r different algorithmic solutions. The IP library was characterized into p processor cores. Thus, each method can be implemented by one of the r*p options. Finally, if there are m different methods in the program, the exploration tool should evaluate several options given by equation:

$$(r * p)^m$$

where r is the number of different implementations of the same routine, p is the number of the available processor cores and m is the number of different method that uses the IP library in the program.

3.2 Object exploration level

This section shows the software exploration approach to improve the dynamic memory management. The main idea in this level is to explore the organization of the objects of the application according the requirements. Our approach is composed by a design space exploration tool that allows an automatic selection of the best object organization. When a programmer uses an object-oriented design paradigm, the application objects can be statically or dynamically allocated. When the programmer uses static allocation the memory footprint is known at compilation time. Hence, in this approach, normally, the memory size is big, but there is a lower execution overhead while dealing with the dynamic allocation (produced by the memory manager). On the other hand, when the programmer uses a dynamic allocation, there is an overhead in terms of performance, but the memory size decreases because the garbage collector removes the unreachable objects.

The related work and our experimental results [24] shown that, for some OO applications, the largest part of the execution time is taken just by memory management. However, if the designer allocates memory in a static fashion, the price to be paid is a much larger memory than it is actually needed, with obvious problems in cost, area and static and dynamic power dissipation.

This tool is divided into three main parts: analysis phase, transformation phase and design space exploration phase. Figure 2 shows the design flow. The methodology starts with the original application analysis and stores the results on a database. Afterwards, the tool transforms, in an automatic way, each allocation instruction that allocated objects dynamically to a static way memory reservation. Afterwards, the tool analyzes each modified application and stores the results on the database. This

task is done to the whole allocation instructions. The final step does the design space exploration. Based on application requirements provided by the designer, the tool tries to search the best object organization (objects allocated dynamically or statically).

In analysis phase, the tool extracts the results from one application. The results are based on: object results: these results show the number of allocated objects for some instance execution, and the number of allocation instructions, object time life, number of accesses per object and object size, etc; memory results: these results show the total dynamic memory allocated during the application execution, the maximum memory utilization and a memory usage histogram; performance results: these show the performance results in terms of executed instructions and the overhead caused by GC.

The analysis phase provides the object, memory and performance results and stores these results in a database that will be used by the exploration phase. This task analysis is done to the whole allocation instructions, one by one, after the transformation step.

This phase aims to transform the object that has been previously dynamically allocated into statically allocated objects. The idea assumes that objects allocated dynamically produce more overhead in terms of performance, because of the memory management, while objects allocated statically produce less overhead in terms of performance, but cause memory overhead, since there must be extra memory space while the application is executing. The tool transforms the dynamically allocated objects into static ones by manipulating the Java bytecodes.

In the exploration phase, the tool does the design space exploration. Based on the application requirement provided by the designer, the tool tries to search for the best object organization (objects allocated dynamically or statically). The main goal of this phase is to search, based on the original application, which objects will be changed into static ones. One of the inputs of the tool is a set of results (the database generated during the analysis phase) that shows: the allocation instruction number (identification number), the performance improvement (instructions) and the memory overhead (bytes). The other input is the application requirements.

Problem Complexity

This problem has a large design space to be explored. The number of combinations that the tool should search is function of the number of allocation instructions that can be changed to static allocation. Thus, the exploration tool should evaluate several options given by equation:

$$2^n$$

where n is the number of allocation instructions that can be changed. This equation can be obtained as follows. For example, in OO program, there are n allocation instructions. All of these instructions can be converted to static ones. Each allocation instruction can be dynamic or static, thus there are 2^n possible combinations. Each combination is composed by a set of allocation instructions that allocates in dynamic or static way.

The solution to the proposed problem needs a heuristic algorithm because of its complexity. Based on application requirements and the database results, the exploration tool searches for the best object organization that fulfills the requirements. The output is a list of allocation instructions that should be transformed into static allocation. This list is used as input of the transformation step to make the final application.

Our problem is very similar to the 0-1 Knapsack Problem [25]. The problem consists in searching the best combination of performance and memory of a set of objects transformations. We implemented our algorithm using dynamic programming [26].

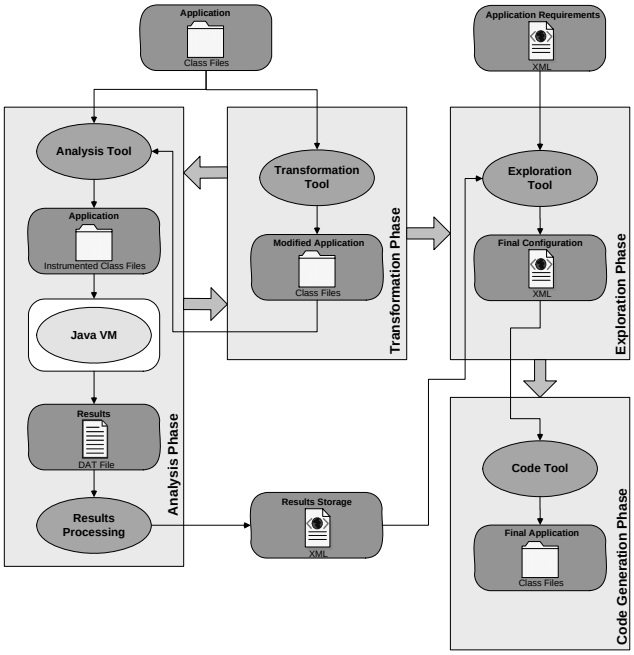

Figure 2. Object Exploration Level Design Flow.

4. EXPERIMENTAL RESULTS

This section presents the results of our approach to explore object-oriented embedded software. The case study application, target platform and results are presented in the following.

4.1 Application

MPEG-Audio is an international standard for digital high quality sound compression. Generally speaking, the standard takes a digital audio file and reduces its size, while maintaining the quality of the recording. Our application code is based on a description freely available on the Internet [27]. All the MP3 code was written in Java but obeying certain constraints of our architecture. An example constraint is the use of integer arrays instead of floating-point arrays because there is no such unit available in the processor.

4.2 Target Platform

We use a platform composed by different core implementations of the same ISA. The platform is based on different implementations of a Java processor, called FemtoJava [28, 29]. This processor implements an execution engine for Java in hardware through a stack machine compatible with Java Virtual Machine (JVM) specification. In this work, we use the multicycle version and the pipeline version.

4.3 Results

This section shows the results of our approach. First, we summarize the results concerning method exploration level and object exploration level on a MP3 application.

The method exploration level is based on software IP library and a set of cores (Java processors) that uses a tool for automatic design space exploration and SW and HW IP selection. The complete explanation of the library and its characterization was presented in [30]. This software IP library contains different algorithmic versions of the same function, like sine, table search, square root, IMDCT, thus supporting design space exploration. Considering a certain core (HW IP) and for each algorithmic implementation of the library functions, it measures performance, memory usage (for data and instruction memories), and energy and power dissipation.

In the object exploration level, the tool tries to transform, in an automatic way, as many dynamic objects to static ones, in the goal to reduce execution time, while maintaining memory costs as low as possible. This idea is based on fact that a small part of the code creates most part of the objects. For example, during the MP3 execution 46,068 objects were created by only 101 allocation instructions, and hence some allocation instructions created more than one object.

Figures 3, 4 and 5 show the design space exploration of the MP3 application that can be explored by the tools. There is a large design space exploration. The number of solutions is defined by the number of method that can explored by the tool and the number of allocation instructions that can be transformed. The plots in the figures show only a small part of possible solutions. The figures show different points: the black points represent solutions running in the pipeline version of the processor. On the other hand, the gray points represent the solutions running in the multicycle one.

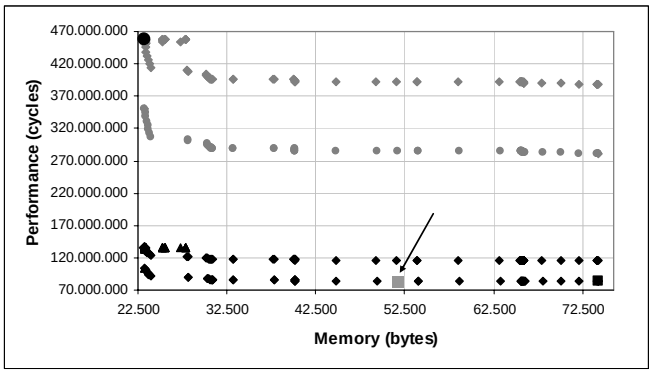

Figure 3. MP3 Performance vs. Memory Design Space.

In the figure 3, the performance versus the increase of memory usage caused by the transformation of the dynamic objects to static ones is presented. The figure shows the corner cases (the circle and the square). The black circle is the solution with worst performance and minimum memory overhead and the black square is the application with best performance and maximum memory overhead. The black solutions present better results in terms of performance because uses the pipeline processor.

The figure 4 shows the results in terms of power versus the increase of memory usage. As one can see, the different processors dissipate different amounts of power (about 23-24mW in the pipeline processor and about 6-7mW in the multicycle version). However, solutions based in the same processor implementation present similar results in terms of dissipated power.

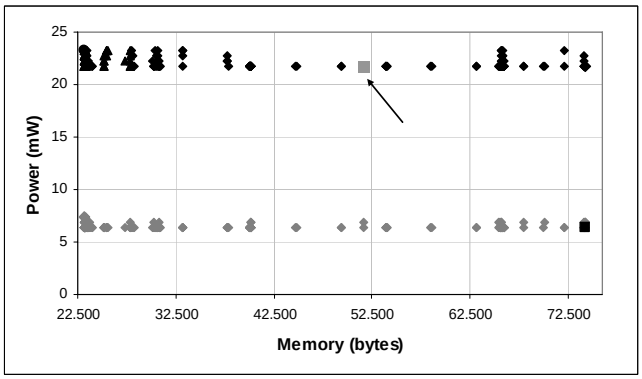

Figure 4. MP3 Power vs. Memory Design Space.

Finally, figure 5 shows the results in terms of energy. Based on these solutions the exploration tool tries to find the best method implementation and the best object organization for certain application and certain application requirements. For example, the square solution presented in the three figures (pointed by an arrow) represents a good solution in terms of performance. But it has an overhead in terms of power.

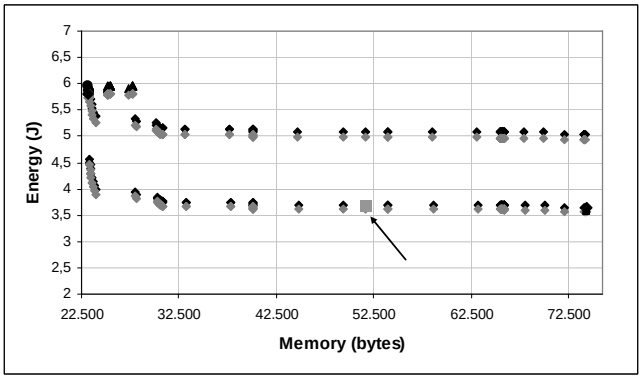

Figure 5. MP3 Energy vs. Memory Design Space.

5. CONCLUSIONS AND FUTURE WORK

This paper introduces a methodology to explore object-oriented embedded software, improving different tasks in the system design. This design space exploration is divided into different phases: the method exploration level and the object exploration level.

Experimental results have confirmed the hypothesis that there is a large space to be explored based on decisions taken at higher levels of abstraction, much before compiler intervention. Selecting the right algorithm, right architecture or right object organization might give orders of magnitude of gain in terms of

physical characteristics like memory usage, performance, and power dissipation. Moreover, not only is this approach very simple, but it can also lead to substantial gains.

As a future work, we plan to evaluate the whole methodology with more applications.

6. REFERENCES

[1]Balarin, Felice; et al. Synthesis of Software Programs for Embedded Control Applications. IEEE Transactions on CAD of Integrated Circuits and Systems, New York, June 1999.

[2]Shandle, Jack; Martin, Grant. Making Embedded Software reusable for SoCs. March, 01 2002. http://www.eetimes.com/news/design/features/showArticle.jhtml?articleID=16504598.

[3]Graaf, B., Lormans, M., Toetenel, H. Embedded Software Engineering: The State of the Practice. IEEE Software, Nov./Dec. 2003, 61-69.

[4]Embedded Systems Roadmap 2002. http://www.stw.nl/progress/ESroadmap/index.html.

[5]Lee, Edward. What's Ahead for Embedded Software ?. IEEE Computer, New York, p.18-26, Sept. 2000.

[6]Detlefs, David L.; Dosser, Al; Zorn, Ben. Memory Allocation Costs in Large C and C++ Programs. Software Practice and Experience, 24(6):527--542, June 1994.

[7]Chatzigeorgiou, A.; Stephanides, G. Evaluating Performance and Power of Object-Oriented vs. Procedural Programming in Embedded Processors. In Proceedings of 7th Ada-Europe International Conference on Reliable Software Technologies. LNCS 2361. Springer-Verlag, 2002. 65-75.

[8]Bhakthavatsalam, Sumithra; Edwards, Stephen H. Applying object-oriented techniques in embedded software design. CPES 2002 Power Electronics Seminar and NSF/Industry Annual Review, April, 2002.

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

[10] Strom, Oyvind; Svarstad, Kjetil; Aas, Einar J. On the Utilization of Java Technology in Embedded Systems, Design Automation for Embedded Systems, vol. 8, n. 1, March 2003, 87-106.

[11] Lanfear, Christopher; Ballaco, Sthepen. The Embedded Software Strategic Market Intelligence: Java in Embedded Systems. July, 2003. http://www.vdc-corp.com/embedded//white/03/03esdtvol4.pdf.

[12] Reyneri, L.M.; et al. A Hardware/Software Co-design Flow and IP Library Based on Simulink. DAC'01 - Design Automation Conference, Proceedings, ACM, 2001.

[13] Dutt, N., et al. New Directions in Compiler Technology for Embedded Systems. In: Asia-Pacific Design Automation Conference, Jan. 2001. Proceedings, IEEE Computer Society Press (2001).

[14] Tiwari, V.; Malik, S.; Wolfe, A. Power Analysis of Embedded Software: a First Step Towards Software Power Minimization. In: IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 2, n. 4, Dec. 1994.

[15] Panda, P. R.; et al. Data and memory optimization techniques for embedded systems. ACM Trans. Des. Autom. Electron. Syst. 6, 2 (Apr. 2001), 149-206.

[16] El-Kharashi, M. W.; Elguibaly, F.; Li, K. F. A quantitative study for Java microprocessor architectural requirements. Part II: High-level language support," Microprocessors and Microsystems, vol. 24, no. 5, pp. 237-250, Sept. 1, 2000.

[17] Jones, Richard; Lins, Rafael D. Garbage Collection: algorithms for automatic dynamic memory management. Chichester: John Wiley, 1996.

[18] Lin, C.; Chen, T. Dynamic memory management for real-time embedded Java chips. In Proceedings of Seventh International Conference on Real-Time Computing Systems and Applications, 2000.

[19] Ritzau, Tobias. Hard Real-Time Reference Counting without External Fragmentation In Proc. of the JOSES Workshop Genoa, Italy, 2001.

[20] Bollella, G.; et al. The Real-TimeSpecification for Java.Addison-Wesley, June 2000.

[21] Kistler, T.; Franz, M. Continuous program optimization: A case study. ACM Trans. Program. Lang. Syst. 25, 4 (Jul. 2003), 500-548.

[22] Cherem, S.; Rugina, R. Region analysis and transformation for Java programs. In Proceedings of the 4th international Symposium on Memory Management, October, 2004. ISMM '04. ACM Press, New York, NY, 85-96.

[23] Shaham, R.; Kolodner, E.; Sagiv, M. Heap profiling for space-efficient Java. In Proceedings SIGPLAN Conf. on Prog. Lang. Design and Impl., ACM Press, 2001. 104-113.

[24] Mattos, Júlio C. B.et al. Making Object Oriented Efficient for Embedded System Applications.. In 18th Brazilian Symp. Integrated Circuit Design (SBCCI 2005), Sep. 2005. New York: ACM Press, 2005. p.104-109.

[25] Martello, Silvano; Toth, Paolo. Knapsack Problems: Algorithms and Computer Implementations. Chichester: John Wiley & Sons, 1990.

[26] Horowitz, Ellis; Sahni; Sartaj. Fundamentals of Computer Algorithms. Computer Science Press, 1978.

[27] MP3, http://www.mp3-tech.org/, 2006.

[28] Ito, S. A.; Carro, L.; Jacobi, R. Making Java Work for Microcontroller Applications, IEEE Design & Test, vol. 18, no. 5, Sep-Oct, pp. 100-110.

[29] 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.

[30] Mattos, Júlio C. B.et al. Design Space Exploration with Automatic Generation of IP-Based Embedded Software. In: IFIP Conference on Distributed and Parallel Embedded Systems, Boston: Kluwer Publishers, 2004. p.237-246.