UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

JÚLIO CARLOS BALZANO DE MATTOS

Design Space Exploration of SW and HW IP based on Object Oriented Methodology for Embedded System Applications

Thesis presented in partial fulfillment
of the requirements for the degree of
Doctor of Computer Science

Prof. Dr. Luigi Carro
Advisor

Porto Alegre, December 2007

CIP – CATALOGING-IN-PUBLICATION

Mattos, Júlio Carlos Balzano de

Design Space Exploration of SW and HW IP based on Object Oriented Methodology for Embedded System Applications / Júlio Carlos Balzano de Mattos. – Porto Alegre: PPGC da UFRGS, 2007.

91 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2007. Advisor: Luigi Carro.

1. Embedded systems. 2. Embedded software. 3. Object orientation. 4. Design space exploration. I. Carro, Luigi. II. Title.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. José Carlos Ferraz Hennemann
Vice-Reitor: Prof. Pedro Cesar Dutra Fonseca
Pró-Reitora de Pós-Graduação: Prof^a. Valquíria Linck Bassani
Diretor do Instituto de Informática: Prof. Flávio Rech Wagner
Coordenadora do PPGC: Prof^a. Luciana Porcher Nedel
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

"Which direction the wind is blowing ?"

ACKNOWLEDGEMENTS

During my PhD studies (a "long journey") I have worked with several people and I would like to thank all of them. I had to write this text by myself, however this work was not done just by myself.

First of all, I would like to thank Dr. Luigi Carro. He is not just my advisor but an example of professor and researcher. Through him I learned to know the "the direction from which the wind is blowing" and I learned to be more questioning. Thanks indeed for the discussions, support and his friendship.

From home, I want to express my gratitude to my fiancée, Martinha, for her love and understanding during all moments. There are no words to describe you and I am so happy living with you. I am also grateful to all my family, especially my parents, Carlos and Regina, for confidence in me along my whole life.

There are lots of people from UFRGS, both LSE and GME, that I worked together and others that we just had a "churrasco" together. I am grateful for their support and friendship: Márcio Oyamada, Lisane Brisolara, Renato Hentschke, Edgard Correa, Alexandre Amory, Andre Borin, Erika Cota, Mateus Beck, Victor Gomes, Renato Barcelos, Carlos Lisboa, Márcio Oliveira, Marco Wehrmeister, Gustavo Neuberger, Felipe Marques, Leomar Rosa, Fernando Paixão, Rodrigo Motta, Ricardo Redin, Bruno Neves, Sandro Sawicki, José Carlos Santanna, Alexandre Gervini, Emerson Hernandez, Eduardo Rhod, Eduardo Brião, Crístofer Kremer and several others that I may have forgotten.

There are two special colleagues that I am deeply grateful: Antônio (Caco) and Emilena. Emilena worked with me since her undergraduate studies and she help me a lot. There are no words to describe Caco and how thankful I am. He is not a good colleague but also a good friend.

From the period that I was an exchange student at TUDelft, I would like to express my gratitude to Dr. Stephan Wong and Dr. Stamatis Vassiliadis (in memoriam) for the opportunity to work with them. To all the people from the Computer Engineering Lab, especially Felipa and Mahomod that used to share the office with me. To all new international friends for the dinners, parties and the experience. I also want to thank to new Brazilian friends Arthur and Evandro for their support and friendship, especially Arthur, a really good friend. I miss our talks and dinners. That time was so nice !

Thanks to the professors of the PPGC, especially to Dr. Flávio Wagner, for discussions, suggestions and knowledge. I also want to thank to all Informatics Institute staff for their support and help during this period of studies. I should give special thanks to Luis Otávio, Beatriz, Ida, Elisiane and Eliane.

From ULBRA, I want to express my gratitude to all colleagues and the students, especially to Patrícia. They contribute indirectly for the successful of this work.

I want to express my gratitude to all of my friends, especially Dr. Rafael for his friendship and support with English classes. Also to the "Cambada" friends (we are still really good and close friends since the undergraduate years) I swear: we will meet more frequently.

This research is partially supported by Conselho Nacional Cientifico e Tecnológico (CNPq) and Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) for scholarship during this PhD and the "sandwich" internship.

I want to thank to Carlos Eduardo Pereira, Felipe França and Ricardo Ferreira for having accepted to be a member of my dissertation board.

I would like to thank all people that usually ask: "When will you finish the dissertation?" and certainly, I have forgotten several people. Thanks to all those who have contributed to this work.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS 9

LIST OF FIGURES 11

LIST OF TABLES 13

ABSTRACT 15

RESUMO 17

1 INTRODUCTION 19
1.1 Main Goal 20
1.2 Text Organization 21

2 BACKGROUND AND RELATED WORK 23
2.1 Embedded Systems 23
2.2 Embedded Software 24
2.3 Object Orientation 26
2.3.1 Object Orientation and Embedded Systems 27
2.4 Java Language 28
2.4.1 Java and Embedded Systems 29
2.5 Related Work 30
2.5.1 Discussion 34

3 DESIGN SPACE EXPLORATION OF OBJECT ORIENTED EMBEDDED SOFTWARE 37
3.1 Introduction 37
3.2 Method exploration level 39
3.2.1 The problem 39
3.2.2 The proposed approach 40
3.3 Object exploration level 44
3.3.1 The problem 44
3.3.2 The problem characterization 45
3.3.3 The proposed approach 48
3.3.4 Target Platform 53

4 RESULTS 55
4.1 Library Characterization 55
4.2 MP3 Case Study Results 57
4.2.1 Method exploration on MP3 58
4.2.2 Object exploration level results 59

5 CONCLUSIONS AND FUTURE WORK 65

REFERENCES 67

LIST OF PUBLICATIONS 75

APPENDIX A EXPLORAÇÃO DO ESPAÇO DE PROJETO DE COMPONENTES DE SW E HW IP BASEADA EM UMA METODOLOGIA ORIENTADA A OBJETOS PARA SISTEMAS EMBARCADOS 77

APPENDIX B CLASS DIAGRAM - STATIC 83

APPENDIX C CLASS DIAGRAM - ORIENTED OBJECT 85

APPENDIX D SEQUENCE DIAGRAM - STATIC 87

APPENDIX E SEQUENCE DIAGRAM - ORIENTED OBJECT . . . 89

APPENDIX F CDROM DESCRIPTION 91

LIST OF ABBREVIATIONS AND ACRONYMS

A/D	Analog/Digital
API	Application Program Interface
ASIC	Application Specific Integrated Circuits
ASIP	Application Specific Instruction set Processors
BIT	Bytecode Instrumentation Tool
CAD	Computer Aided Design
CBSE	Component-Based Software Engineering
CORDIC	Coordinate Rotation Digital Computer
CPU	Central Processing Unit
D/A	Digital/Analog
DSP	Digital Signal Processing
GC	Garbage Collection
HW	Hardware
I/O	Input/Output
ILP	Instruction Level Parallelism
IMDCT	Inverse Modified Discrete Cosine Transform
IP	Intellectual Propriety
ISA	Instruction Set Architecture
JIT	Just-in-time
JVM	Java Virtual Machine
KVM	Kilobyte Virtual Machine
MP3	MPEG 1 Layer 3
MPEG	Moving Picture Experts Group
OO	Object Orientation
OS	Operating System
PDA	Personal Digital Assistant

RISC	Reduced Instruction Set Computer
RTSJ	Real-Time Specification for Java
SPE	Software Performance Engineering
SW	Software
VLIW	Very Long Instruction Word

LIST OF FIGURES

Figure 2.1: Embedded Sofware 25
Figure 2.2: Different Java execution ways (KAZI et al., 2000) 29

Figure 3.1: Thesis Design Flow. 38
Figure 3.2: Method Exploration Level Design Flow. 42
Figure 3.3: Object-Oriented Overhead. 45
Figure 3.4: Object Exploration Level Design Flow. 49
Figure 3.5: Original Code. 51
Figure 3.6: The code after the transformation. 51

Figure 4.1: MP3 Decoding steps. 58
Figure 4.2: MP3 Performance vs. Memory Design Space. 63
Figure 4.3: MP3 Power vs. Memory Design Space. 63
Figure 4.4: MP3 Energy vs. Memory Design Space. 64

LIST OF TABLES

Table 3.1: Object data (original applications). 46
Table 3.2: Memory data (original applications). 47
Table 3.3: Performance data (original applications). 47

Table 4.1: Sine Characterization. 56
Table 4.2: Sine Characterization (hardware dependable). 56
Table 4.3: IMDCT Characterization. 56
Table 4.4: IMDCT Characterization (hardware dependable). 57
Table 4.5: IMDCT Characterization (hardware dependable)(cont.). 57
Table 4.6: MP3 profiling results using IMDCT1 and Cosine Cordic. 59
Table 4.7: MP3 profiling results using IMDCT4 and Cosine Table. 59
Table 4.8: MP3 profiling results using IMDCT1 and Cosine Table. 60
Table 4.9: MP3 Allocation instruction. 61
Table 4.10: MP3 results after static transformation. 61
Table 4.11: MP3 combinations of allocation instructions. 62

ABSTRACT

Software is increasingly becoming the major cost factor for embedded devices. Nowadays, with the growing complexity of embedded systems, it is necessary to use techniques and methodologies that can, at the same time, increase software productivity and manipulate embedded systems constraints - like memory footprint, real-time behavior, performance and energy. Object-oriented modeling and design is a widely known methodology in software engineering. This paradigm may satisfy software portability and maintainability requirements, but it presents overhead in terms of memory, performance and code size. This thesis introduces a methodology and a set of tools that can deal, at the same time, with object orientation and different embedded systems requirements. To achieve this goal, the thesis presents a methodology to explore object-oriented embedded software improving different levels in the software design based on different implementations with the same processor. The results of the methodology are presented based on an MP3 player application.

Keywords: Embedded systems, embedded software, object orientation, design space exploration.

Exploração do Espaço de Projeto de IPs de SW e HW em uma Metodologia Orientada a Objetos para Aplicações Embarcadas

RESUMO

O software vem se tornando cada vez mais o principal fator de custo no desenvolvimento de dispositivos embarcados. Atualmente, com o aumento aumentando da complexidade dos sistemas embarcados, se faz necessário o uso de técnicas e metodologias que, ao mesmo tempo, permitam o aumento da produtividade do desenvolvimento de software e permitam manipular as restrições dos sistemas embarcados como tamanho de memória, comportamento de tempo real, desempenho e energia. A análise e projeto orientado a objetos são altamente conhecidos e utilizados na comunidade de engenharia de software. Este paradigma auxilia no desenvolvimento e manutenção do software, porém apresenta uma significativa sobrecarga em termos de memória, desempenho e tamanho do código. Esta tese introduz uma metodologia e um conjunto de ferramentas que permitem o uso concomitante de orientação a objetos e os diferentes requisitos dos sistemas embarcados. Para atingir este objetivo, esta tese apresenta uma metodologia para exploração de software embarcado orientado a objetos que permite melhoria em diferentes níveis do processo de desenvolvimento do software baseado em diferentes implementações do mesmo processador. Os resultados da metodologia são apresentados baseados na aplicação de um tocador de MP3.

Palavras-chave: Sistemas Embarcados, Software Embarcado, Orientação a Objetos, Exploração do Espaço de Projeto.

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 energy consumption and program size, among others, must be considered. Moreover, the complexity of embedded systems is increasing in a considerable way.

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 (BALARIN et al., 1999) (SHANDLE; MARTIN, 2001). Probably in the future even more products will have most of their characteristics developed in software. Hence, software is more and more becoming the major cost factor for embedded devices (GRAAF; LORMANS; TOETENEL, 2003; EGGERMONT, 2002).

More recently, platform-based design was introduced (SANGIOVANNI VINCENTELLI; MARTIN, 2001). These platforms are composed by a set of resources and services that can implement an entire systems. In platform-based design, design derivatives are mainly configured by software using a fixed hardware platform, and software development is where most of the design time is spent. But the quality of the software development also impacts embedded systems requirements in a direct way.

Presently, there is wide variety of Intellectual Property (IP) blocks such of processor cores with several architecture styles, like RISC, DSP, VLIW. Also, there is an increasing number of software IPs that can be used in a complex embedded system design. Thus, with an wide range of SW and HW IP solutions, the designer has several possibilities, and need methodologies and tools to make an efficient design exploration to achieve a short design cycle due to stringent time-to-market requirements.

Over the years, embedded software coding has been traditionally developed in assembly language, since there are stringent memory and performance limitations (LEE, 2000). 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 to embedded systems. 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 (SOMMERVILLE, 2000). 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, its acceptance in the embedded world has been slow, since embedded software designers are reluctant to employ these techniques due the memory and performance overhead (DETLEFS; DOSSER; ZORN, 1994; CHATZIGEORGIOU; STEPHANIDES, 2002; BHAKTHAVATSALAM; EDWARDS, 2002).

Moreover, over the past few years embedded developers have embraced Java, because this technology can provide high portability and code reuse for their applications (MULCHANDANI, 1998; LAWTON, 2002). 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 they should address different constraints from desktop software. Moreover, embedded software design needs to deal with the increasing complexity of applications. In this way, this thesis introduces a methodology to design space exploration of SW and HW IPs based on a platform. The methodology uses object-oriented embedded software to improve different tasks in the system design.

1.1 Main Goal

The main goal of this thesis is to provide a methodology and a set of tools that can deal, at the same time, with well-known software development methodologies (platform-base design, object orientation, component-based engineering and Java language) and different embedded systems requirements (energy, memory area and performance). A methodology to explore object-oriented embedded software improving different tasks in the system design is introduced. 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 implementation of methods (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 object organization to improve the dynamic memory management. This level uses 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.

1.2 Text Organization

This thesis is organized as follows. Chapter 2 provides background on embedded systems and discusses related work in the field of embedded software optimization. Chapter 3 presents our approach to design space exploration of object oriented embedded software. Chapter 4 shows case study results of our methodology, and, finally, Chapter 5 draws conclusions and future work.

2 BACKGROUND AND RELATED WORK

This chapter provides background on embedded systems describing their main characteristics related to this thesis: embedded software, object orientation and the Java language. Moreover, this chapter discusses related work in the field of thesis. The works are related to embedded software optimization. It also presents a review in the field of object orientation use in embedded systems, garbage collection and memory management system optimizations.

2.1 Embedded Systems

The fast technological development in the last decades exposed a new reality: the widespread use of embedded systems. These systems are dedicated systems that perform a specific function and include a programmable computer but this is not itself intended to be a general purpose computer (WOLF, 2001). Nowadays, one can find these systems everywhere, in consumer electronics, entertainment, communication systems and so on.

There are a large number of different applications where embedded systems are involved. Moreover, there can be a wide range of functions in single system with different computation requirements of each function. For example, the latest cellular phones present several other functions like Internet access, digital camera, MP3 player, infrared communication and others (NOKIA, 2007). Thus, embedded systems requirements are very diverse and one can say that embedded systems are naturally heterogeneous, because they are composed by different models of computation, analog and digital, hardware and software.

Different embedded applications require computational systems to control them. Dedicated algorithms implemented in hardware or software define these control tasks. The embedded systems implementation can be done in an wide range of hardware configurations, like application specific integrated circuits (ASICs), microcontrollers, microprocessors, application specific instruction set processors (ASIPs), etc. On the other hand, these tasks can be implemented like software routines that execute in standard components like a microprocessor or a digital signal processor. Moreover, there are other components that can be found in embedded applications, like A/D and D/A converters, displays and other real world interfaces.

Embedded systems are different from other systems because they must address some requirements and restrictions. There are some items that are very important

in embedded system design:

- Performance: with the growing complexity of embedded systems, the speed of the system is often a major requirement that must be achieved;

- Real-time: in general, embedded systems are involved in tasks where they are submitted to strict time requirements, and must respond to external events with a precise evaluation of execution time;

- Physical size and weight: these systems are located in bigger systems where the portability requirement is very important, thus embedded processors must be small and light;

- Reliability: some of these systems are involved in critical applications where errors can have dramatic consequences, involving human lives and huge amounts of money;

- Design time and cost: nowadays, time-to-market is very important. Thus, the design time and cost must be reduced as much as possible;

- Low energy consumption: energy consumption presents a critical issue, particularly in portable and mobile platforms.

2.2 Embedded Software

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 (BALARIN et al., 1999; SHANDLE; MARTIN, 2001). 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. Software is more and more becoming the major cost factor for embedded devices (GRAAF; LORMANS; TOETENEL, 2003; EGGERMONT, 2002).

Embedded software can be found on different embedded system levels covering from application level to operating system level. Figure 2.1 shows these embedded software levels. Embedded software programmers can use an API (application program interface) and operating systems depending the application requirement and their availability. An API and an operating system make software development easier and faster.

Over the years, embedded software coding has been traditionally developed in assembly language, since there are stringent memory and performance limitations (LEE, 2000). In early days, embedded software was written exclusively in the assembly language of the target processor. This gave programmers complete control of the processor and other hardware, but at a huge price. Assembly languages have many disadvantages like higher software development costs and a lack of code portability.

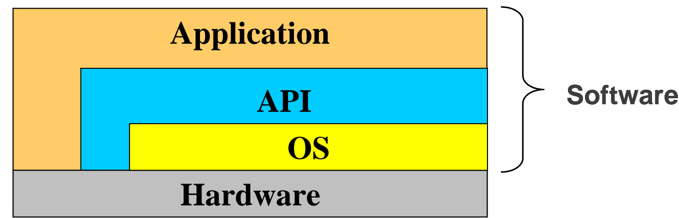

Figure 2.1: Embedded Sofware

The best software technologies use large amounts of memory, layers of abstraction, elaborate algorithms, and these approaches are not directly applicable in embedded systems (LEE, 2000). However, hardware capabilities have been improved, and the market demands more elaborate products, increasing software complexity. Thus, the use of better methodologies is clearly required, for example object orientation and components-based development. In spite of these advanced resources, the problems of embedded software development still exist.

The use of software paradigm allows the flexibility and the portability offered by software implementations. When designers have greater focus on software functionality in detriment of hardware, the design time and cost is reduced. However, some aspects like power dissipation can be affected since, for example, there is more memory usage.

The bottleneck for the implementation of embedded systems has been considered the software development, its debugging, and its integration with the hardware components (BALARIN et al., 1999). However, contrasting the maturity of hardware design methodologies with the software development methodologies, these software techniques are very immature.

Nowadays, embedded software designers are using languages like C and Assembly to deal with strict restrictions like performance and cost. However, these languages present limitations on abstraction, validation and maintenance. Thus, it is necessary the use of design methodologies and techniques that allow suitable languages with higher abstraction levels to make easier the specification and design of embedded systems.

However, existing software methodologies are not adequate for embedded software development. This development is very different from the one used in desktop environment. Embedded software development should address constraints like memory footprint, real-time behavior, power dissipation and so on. Moreover, traditional methodologies need more memory resources and more performance.

The traditional ad hoc approaches used on embedded systems design, more specifically embedded software design, are not able to deal with new applications complexity. Thus, it is necessary to adapt the available techniques and methodologies, or to create novel approaches that can manipulate the embedded systems constraints, while still maintaining good software characteristics.

2.3 Object Orientation

The software engineering area offers several methodologies to use in traditional software development. One of the main methodologies is object-oriented paradigm. The use of object-oriented programming starts in the 1960s. However, it was not commonly used in mainstream software development until the early 1990s. From recent decades object-oriented paradigm has become the dominant programming paradigm (BUDDY, 2001). Object-oriented programming scales very well, from the most trivial problems to complex ones.

Object-oriented technique allows more abstraction. Abstraction is a key to design and manage complex systems. In this respect, Budd (2001) says that abstraction is the purposeful suppression, or hiding, of some details of a process or artifact, in order to bring out more clearly other aspects, details, or structures.

A typical program written in the object-oriented style introduces several levels of abstraction. Higher abstraction levels are part of the object-oriented concepts and make an object-oriented program. Using higher modeling abstractions that are closer to the problem space make the design process and implementation easier, and besides, these abstractions provide a very short development time and a lot of code reuse.

An object-oriented program is structured as a collection of objects that interact among them. Each object is responsible for specific tasks. An object is an encapsulation of state (data values) and behavior (operations). The computation is performed by object actions and the communication with each other. This communication is done by sending and receiving messages. An object will perform its behavior by invoking a method in response to a message. Each object is an instance of a class and a class represents a grouping of similar data.

The most common technique people use to deal with complex systems is to combine abstraction with a division into software components. The object orientation benefits other software engineering areas, like Component-Based Software Engineering (CBSE). The CBSE (HEINEMAN; COUNCILL, 2001) is a widely used methodology that is primarily concerned with three functions: developed software from pre-produced parts, the ability to reuse those parts in other applications and easily maintaining and customizing those parts to produce new functions and features. It provides a methodology and a process for building and providing reliable components that maintain software functioning.

Software components permit the programmer to deal with the problems on a higher level of abstraction. The programmer can define and manipulate objects on a simple way by just knowing the object tasks and the messages, ignoring implementation details. Reducing the dependence among software components, object orientation permits the development of reusable software systems. Such components can be created and tested as independent units, isolated from other portions of the software application.

The abstraction is achieved because object orientation introduces several concepts like:

- Responsibilities: object orientation provides a concept that is to describe behavior in terms of responsibilities. Responsibilities increase the level of abstraction and permit greater independence between objects, a critical factor in solving complex problems;

- Inheritance: Using inheritance, classes can be organized into a hierarchical tree. Data and behavior associated with higher classes (parent classes) in the tree can be accessed by lower classes in the tree. Thus, inheritance allows different data types to share the same code, leading to a reduction in code size and increase in functionality;

- Polymorphism: allows this shared code to be tailored to fit the specific circumstances of individual data types;

- Method binding and overriding: allows information in a subclass to override the information inherited from a parent class. In general, this is implemented by using a method in a subclass having the same name as the method in a parent class;

- Interface and implementation: the terms 'interface' and 'implementation' describe the distinction between the view of the user and how the tasks are implemented. The interface says nothing about how the assigned task is being performed. The division between interface and implementation makes the design and the interchange of the components easier;

- Encapsulation: an important feature of interchangeability is encapsulation. The components and objects can encapsulate certain functionalities and interact with other components through a simple and well defined interface. An object encapsulates both data and behavior (methods).

In the field of Component-Based Software Engineering (CBSE), there are approaches on applying this technique in embedded systems to improve embedded software productivity (GENSSLER et al., 2002; YEN et al., 2002; CRNKOVIC, 2005).

2.3.1 Object Orientation and Embedded Systems

As mentioned, object-oriented methodology introduces several features and concepts that increase software productivity. 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 overhead. Object-oriented design paradigm presents an overhead in terms of memory, performance and code size (DETLEFS; DOSSER; ZORN, 1994; CHATZIGEORGIOU; STEPHANIDES, 2002; BHAKTHAVATSALAM; EDWARDS, 2002).

There are several features on object orientation that produce overhead. We highlight the main features:

- Dynamic Allocation problems: one of the main problems when using object orientation is the overhead produced by memory management. This overhead is imposed due to the extensive use of dynamic memory allocation and dynamic creation and destruction of objects. This causes overhead in memory management in terms of performance and memory usage. Moreover, pointer manipulation increases memory traffic and, due to lack of data locality, causes cache inefficiency;

- Memory problems: memory leak takes place when memory is allocated but never deallocated. Also, there is memory fragmentation produced by intensive memory allocation and deallocation. Other problem is when memory is deallocated: there may still be active pointers in use to the deallocated memory because of cyclic structures;

- Dynamic binding problems: using dynamic binding allows information in a subclass to override the information inherited from a parent class. The search for a method to invoke in respose to a given message begins with the class of the receiver. If no appropriate method is found, the search is conducted in a parent class of this class until the method is found. Dynamic binding is usually implemented using indirection (BHAKTHAVATSALAM; EDWARDS, 2002). The actual entry point is looked up at run-time in a dispatch table. The cost of performing the lookup search and using the extra level of indirection to execute operations significantly increases the overhead of calling methods;

- Method call overhead: when using object orientation, classes typically have a greater number of smaller methods arranged in deeper call trees. This pattern leads to additional overhead in relation to non object-oriented programming (BHAKTHAVATSALAM; EDWARDS, 2002). Larger methods reduce the number of method calls, but it produces a conflict with the goal of software reusability.

2.4 Java Language

The Java programming language is one of the most interesting languages developed in recent years. Java presents several features like platform independence for portability, an object-oriented model, multithreading support, support for distributed programming and an automatic garbage collector. Java obeys the "write-once, run everywhere" philosophy. Over the past few years, developers have embraced Java, because this technology can provide high portability, flexibility, robustness, code reuse and security for their applications. However, Java has an overhead in terms of performance due the hardware abstraction provided (KAZI et al., 2000).

Java source code is translated into Java bytecodes to allow portability. These bytecodes can be executed on any platform that supports an implementation of Java Virtual Machine (JVM) (LINDHOLM; YELLIN, 1999). Most JVMs execute Java bytecodes through interpretation or Just-in-time (JIT) compilation. These execution options are relatively slow, because they translate the bytecodes on the fly. However, there are solutions that improve this execution like compiling Java directly to native machine code (but the portability will be lost) and using a Java

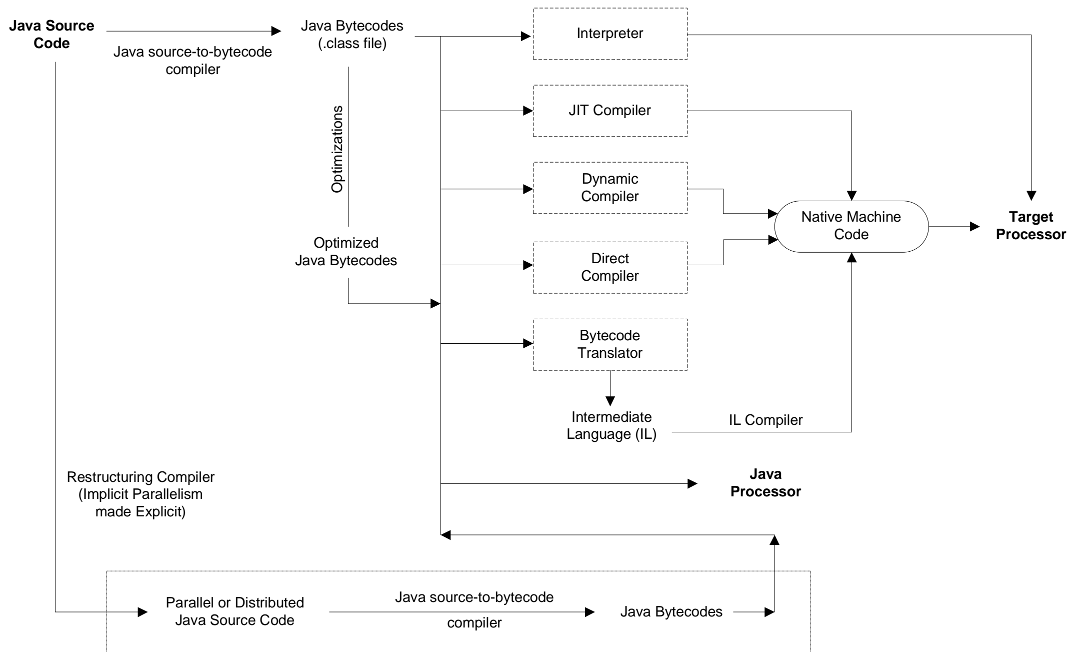

Figure 2.2: Different Java execution ways (KAZI et al., 2000)

Processor. Figure 2.2 summarizes the ways that Java can be translated to machine code.

2.4.1 Java and Embedded Systems

Over the past few years embedded developers have embraced Java, because this technology can provide high portability and code reuse for their applications (MULCHANDANI, 1998; LAWTON, 2002; STROM; SVARSTAD; AAS, 2003; LANFER; BALLACO, 2003). 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.

The Java language deallocates objects by using garbage collection (JONES; LINS, 1996). Garbage collectors have advantages freeing programmers from the need to deallocate the objects when these objects lose their reference, and helping to avoid memory leaks. However, garbage collectors produce an overhead in program execution and a non-deterministic execution. Automatic garbage collection is usually not as efficient as programmer managed allocation and deallocation.

There are several problems when using Java in embedded systems. They are summarized as follow:

- Slow execution: Java presents a slow execution (performance overhead). This overhead is produced because Java is interpreted on JVM, thus to improve the execution it is necessary to use faster processors;

- Large amounts of memory: Java requires large amounts of memory, since this is necessary for the application memory and for the JVM memory;

- Power dissipation: when the memory usage and performance overhead increase, more power dissipated as a consequence;

- Hardware abstraction: Java does not provide hardware access, thus it is necessary to provide mechanisms to access the hardware, since embedded software often connects to the real world;

- Garbage Collector: Java has a reduced control with memory allocation and deallocation. Garbage collector is responsible for this control and it increases the memory management overhead. Also, the garbage collector interferes in real-time tasks.

Nowadays, Java is used in a variety of embedded systems. There are several hardware solutions to improve the Java execution. These solutions are based on hardware accelerators like (ARM, 2007; NAZOMI COMMUNICATIONS, 2007) where critical parts of the code are executed in hardware. There are other solutions based on Java processors. These machines execute Java natively like (MCGHAN; O'CONNOR, 1998; ITO; CARRO; JACOBI, 2001).

During recent years, Java has been used in the mobile phone market (LAWTON, 2002; TAKAHASHI, 2001). Java platform portability enables Java phone users to download software like games from third party providers.

According to this movement, Sun released two environments dedicated to embedded Java: Java 2 Micro Edition (SUN MICROSYSTEMS, 2007a) specification and Java Card (SUN MICROSYSTEMS, 2007b) specification. Java 2 Micro Edition defines full support for the Java language and the Java Virtual Machine specification with a limited number of support libraries classes in order to execute in environments with limited amount of memory. JavaCard defines an environment to develop application into smart cards. Nowadays, there is a significant number of smart cards that run Java.

Java has several problems concerning real-time. Thus, a research group developed the Real-Time Specification for Java (RTSJ) (BOLLELLA, 2000; BOLLELLA; GOSLING, 2000). This specification makes JVM deterministic and provides full real-time capabilities for Java technology.

2.5 Related Work

In (PEYMANDOUST; MICHELI; SIMUNIC, 2002), an energy profiler is used to identify critical arithmetic functions and replace them by using polynomial approximations and floating-point to fixed-point conversions. For the MP3 software, power improvements by a factor of 400 are reported with regard to original code from the standard body, but code transformations are restricted to arithmetic functions and most of the improvement comes from the floating-point to fixed-point conversion.

In (REYNERI et al., 2001), a library of alternative hardware and software parameterized implementations for Simulink blocks that present different performance, power, and area figures are characterized. An analysis tool quickly evaluates algorithmic and parameter choices performed by the designer.

Most approaches on embedded software optimization are based on compilation optimization. In spite of compilation optimizations, both approaches presented above present higher level of optimization handling with functions and libraries.

Compiler code optimizations for embedded systems have been traditionally oriented towards improving performance or reducing memory space (DUTT et al., 2001), for instance targeting code to specialized architectures, reducing cache misses, or compressing code. Although some of these code optimizations may eventually also reduce power consumption, for instance by reducing the number of memory accesses, the energy issue has been often neglected, and it has been shown (DALAL; RAVIKUMAR, 2001; KANDEMIR et al., 2001) that many compiler optimizations may even increase power consumption.

Power-aware software optimization has gained attention in recent years, mainly in embedded systems. It has been shown (TIWARI; MALIK; WOLFE, 1994) that each instruction of a processor has a different power cost. Instruction power costs may vary in a very wide range and are also strongly affected by addressing modes. By taking these costs in consideration, a 40% power improvement obtained by code optimizations is reported.

Reordering of instructions in the source code has been also proposed (CHOI; CHATTERJEE, 2001), considering that power consumption depends on the switching activity and thus also on the particular sequence of instructions, and improvements of up to 30% are reported.

A survey of data and memory optimization techniques for embedded systems are presented in (PANDA et al., 2001). This paper discusses platform-independent memory optimizations and techniques applicable to memory structures. The platform-independent memory optimizations are done by compiler, for example loop transformations. Another survey (WOLF; KANDEMIR, 2003) concentrates just on software techniques to improve the memory system. However, the software techniques presented in both papers are based on compilation techniques.

On the other hand, when using the object-oriented languages there are several problems. In (CHATZIGEORGIOU; STEPHANIDES, 2002), the object oriented programming style is evaluated in terms of both performance and power for embedded applications. A set of benchmark kernels, written in C and C++, is compiled and executed on an embedded processor simulator. The paper has shown that oriented objected programming could significantly increase both execution time and power consumption.

Another work, (DETLEFS; DOSSER; ZORN, 1994), presents a detailed measurements of the cost of dynamic storage allocation of C and C++ using five very dif-

ferent dynamic storage allocation implementations, including a conservative garbage collection algorithm. This work shows that overhead in terms of instructions ranges, depending on the allocator, from 6.17% to 36.40% on average. For certain applications the overhead achieves more than 60%.

The use of Java in embedded systems presents problems concerning performance, power and real-time. There are some works that prove this huge overhead. El-Kharashi (2000) shows that object manipulation instructions are the most time consuming. This work presents a dynamic instruction analysis of CaffeineMark Benchmark. The object manipulation instructions are responsible for 23.10% of the total instruction execution. However, these same instructions are responsible for 88.86% of total execution time. Another work (LUN; FONG; HAU, 2003) also shows similar results using other benchmarks. In this work the object manipulation instructions are responsible for 15% of the total instruction execution.

Radhakrishnan (1999) reports that invoking methods in Java is expensive, because they need an execution environment and a new stack for each new method. This work shows that the most common dynamically invoked methods have 1, 10 or 27 bytecodes long and that 45% of all dynamic methods have less than 9 bytecodes or 16 bytes long. This behavior is produced by object-oriented characteristics like the use of private and public methods and interfaces.

A memory system behavior of Java Programs was studied in (KIM; HSU, 2000). This paper analyzes the SPECjvm98 applications running with a Just-In-Time (JIT) compiler. The results show that the overall cache miss ratio is increased due to garbage collection and that the Java programs generate a substantial amount of short-lived objects. However, the size of frequently referenced long-lived objects is similar to the application working set size.

There are several works that present some optimizations and techniques to produce better results in memory management. These works present optimizations to reduce the memory and performance overhead, make real-time applications possible and so on. There are optimized and reduced versions of virtual machines, like the KVM (Kilobyte Java Virtual Machine) (SUN MICROSYSTEMS, 2007c).

There is a rich work on garbage collector architecture and algorithms. A complete discussion can be found in (JONES; LINS, 1996). Much of this work is aimed at avoiding the most severe pitfalls of garbage collection, mainly their performance loss and unpredictability. There are solutions using garbage collection implemented in software and hardware.

In (LIN; CHEN, 2000), a hardware mechanism to support the runtime memory management providing real-time capability for embedded Java devices is presented. This approach guarantees predictable memory allocation time.

Chen (2003) proposes a set of memory management strategies to reduce heap footprint of embedded Java applications that execute under severe memory constraints. It presents a new garbage collector that allows an application to run with

a heap smaller than its footprint using a technique that is based on memory compression. Another work (CHEN et al., 2002) of the same author focuses on tuning the GC to reduce energy consumption in multibanked memory architecture.

Pfeffer (2004) presents a garbage collector implementation for multithreaded processors. This GC runs in a thread slot in parallel to real-time applications. Other software and hardware solution can be found in (BERLEA et al., 2000; RITZAU, 2001; SRISA-AN; LO; CHANG, 2003; DETERS et al., 2004; BACON; CHENG; GROVE, 2004).

The Real-Time Specification for Java (RTSJ) introduces some mechanisms to solve the unpredictability. RTSJ provides two other kinds of memory (besides the normal memory): immortal and scoped memories (BOLLELLA, 2000; LOCKE; DIBBLE, July 2003). With the immortal memory, the objects should be created once for the lifetime of the application (during initialization phase). Objects in immortal memory ca be created and accessed without GC delay, but there is no mechanism for freeing those objects during of the application execution. There is another memory model, the scoped memory. This memory area has limited lifetime (the memory area is valid as long as one or more threads refer to it). Differently from immortal memory, the objects allocated in this memory can be removed when all reference to this memory is removed.

The approach of pre-allocate objects instead of creating them dynamically is concerned with high level modeling (SMITH; WILLIAMS, 2003). This strategy is used on software performance engineering (SPE) approach to save this unnecessary overhead. Each call object is used over and over again, rather than creating a new one for each execution.

In (SHUF et al., 2002), techniques aimed at improving the memory behavior of pointer-intensive applications with dynamic memory allocation in Java is presented. The technique relies on identification of frequently instantiated types of the given program, and tries to co-locate objects at allocation time. This way, the related objects are placed close to each other in memory, improving the data locality.

Kistler (2003) presents an optimization technique that improves the object organization during the program execution. Object layout adaptation improves the storage layout of dynamically allocated data structures, maximizing data cache locality. This strategy assigns the fields to cache lines and then optimizes the order of fields within individual cache lines.

In (CHEREM; RUGINA, 2004), a region analysis and transformation framework for Java programs is presented. Given an input Java program, the compiler automatically translates it into an equivalent output program with region-based memory management. The generated program contains statements for creating regions, allocating objects in regions and removing regions.

Shaham (2001) presents a heap-profiling tool for exploring the potential for memory space savings in Java programs. The output of the tool is used to direct rewriting

of application source code in a way that allows more timely garbage collection of objects, thus saving space. The rewriting can also avoid allocating some objects that are never used, making space savings and, in some cases also improving program runtimes. This approach is based on three code rewriting techniques: assigning the null value to a reference that is no longer in use, removing code that has no effect on the result of the program and delaing the allocation of an object until its first use.

In (ARNOLD et al., 2005), a rich survey of optimization in virtual machines is provided. It provides a complete review of adaptive optimization techniques: techniques for determining when, and on what parts of the program, to apply a runtime optimizing compiler; techniques for collecting fine-grained profiling information; and techniques for using profiling information to improve the quality of the code generated by an optimizing compiler.

2.5.1 Discussion

Our method approach is similar to (REYNERI et al., 2001), but instead of aiming at a partitioning between software and hardware functions, 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 (DUTT et al., 2001; DALAL; RAVIKUMAR, 2001; KANDEMIR et al., 2001; CHOI; CHATTERJEE, 2001; TIWARI; MALIK; WOLFE, 1994; PANDA et al., 2001; WOLF; KANDEMIR, 2003). 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 overheads (CHATZIGEORGIOU; STEPHANIDES, 2002; DETLEFS; DOSSER; ZORN, 1994; EL-KHARASHI; ELGUIBALY; LI, 2000; LUN; FONG; HAU, 2003; RADHAKRISHNAN; RUBIO; JOHN, 1999; KIM; HSU, 2000). However, these papers only measure this overhead in terms of performance.

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 (JONES; LINS, 1996; LIN; CHEN, 2000; CHEN et al., 2003, 2002; PFEFFER et al., 2004; BERLEA et al., 2000; RITZAU, 2001; SRISA-AN; LO; CHANG, 2003; DETERS et al., 2004; BACON; CHENG; GROVE, 2004). 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) (BOLLELLA, 2000) 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 (SHUF et al., 2002; KISTLER; FRANZ, 2003). However, these techniques improve the data locality during the execution time. Another paper (CHEREM; RUGINA, 2004) 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.

On the other hand, Shaham (2001) 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 OBJECT ORIENTED EMBEDDED SOFTWARE

This chapter shows the thesis concerning embedded software design space exploration. Our approach is divided into two main parts where the embedded software exploration methodology can improve.

The first part, called method exploration level, aims to improve method implementation (the algorithms that implement these methods). The second part, called object exploration level, aims to explore object organization in order to improve the dynamic memory management.

3.1 Introduction

The first chapter showed that embedded systems must address some different requirements and restrictions. Thus, the embedded software development is different from the traditional development (desktop development) and available techniques and methodologies should be adapted to manipulate embedded systems constraints.

New technologies (methodologies, programming languages, tools, etc.) present several problems. The main problem is related to the learning cost of these new technologies by designers. Thus, lots of companies avoid the adoption of new methodologies or programming languages. In this way, the aim of our proposed approach is to avoid making changes on the traditional software design flow.

The proposed approach uses well-known technologies and the idea is to introduce a set of tools to improve the original software by generating an optimized code. These tools intend to be easy to learn and use. Also, the tools will do a design space exploration, allowing the automatic configuration of an optimized software solution for a specific application according to the embedded software requirements.

Our approach of embedded software exploration is divided into two main exploration parts. Figure 3.1 shows a simple design flow of the thesis that contemplates these phases.

The fist exploration phase introduces a mechanism for the automatic selection of software IP components for embedded applications, which is based on a software IP library and a design space exploration tool. The software IP library has different

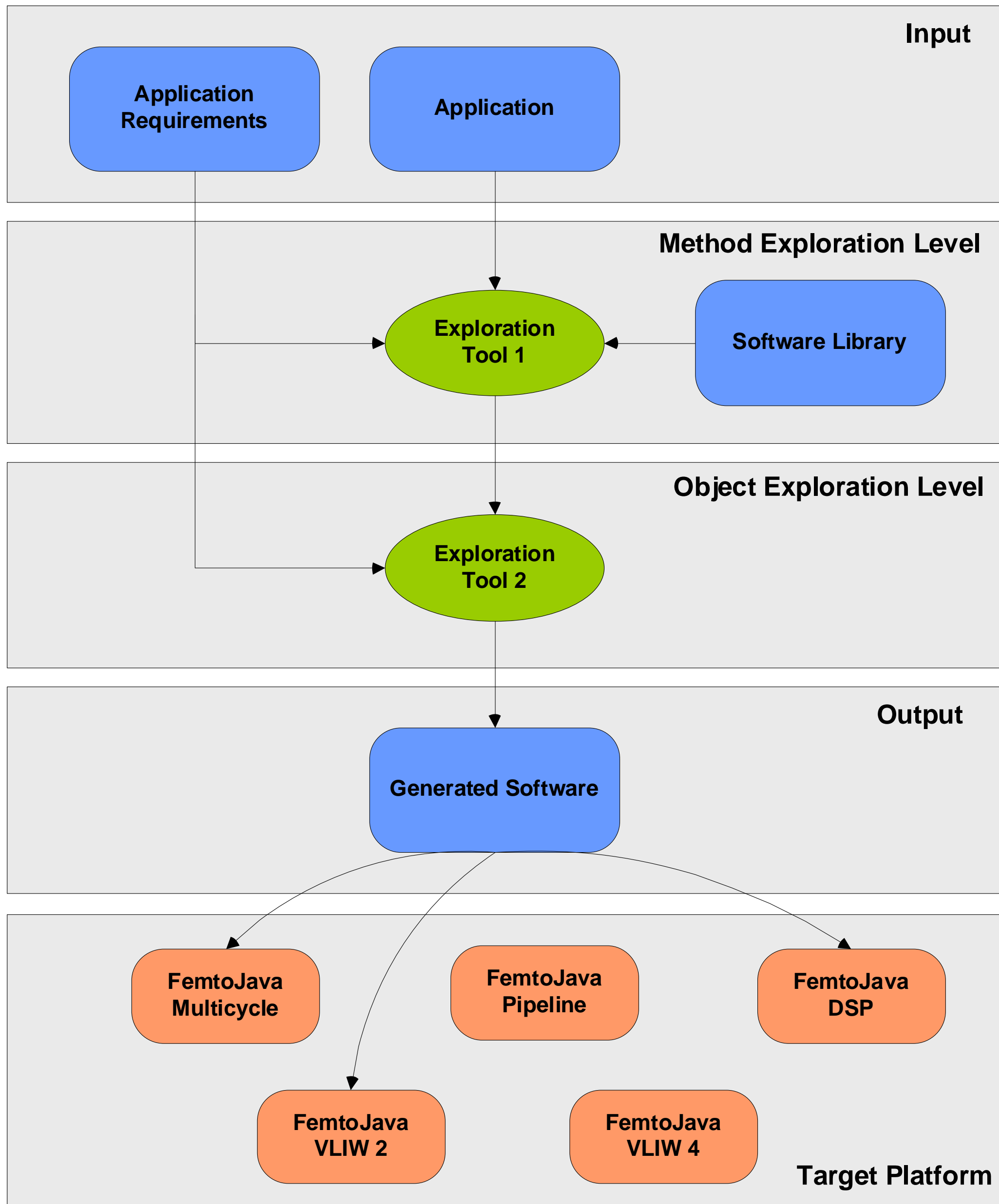

Figure 3.1: Thesis Design Flow.

algorithmic implementations of several routines commonly found in different application domains.

The second exploration phase consists in the use of a design space exploration tool to allow an automatic selection of the best object organization. This tool tries to transform, in an automatic way, as many dynamic objects into static ones, in the goal to reduce execution time, while maintaining memory costs as low as possible.

Moreover, these two phases (one based on method and another based on object exploration) are orthogonal, that is, their execution are 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.

The target platform is composed of a set of Java processors. There are different versions of FemtoJava processor (ITO; CARRO; JACOBI, 2001). Section 3.4 describes in more detail the target platform.

3.2 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. First, we show the problem and, afterwards, the proposed solution.

3.2.1 The problem

It is widely known that design decisions taken at higher abstraction levels can lead to substantially superior improvements. Software engineers involved with software configuration of embedded platforms, however, do not have enough experience to measure the impact of their algorithmic decisions on issues such as performance and power.

Moreover, different applications have different resource requirements during their execution. Some applications may have a large amount of instruction-level parallelism (ILP), which can be exploited by a processor that can issue many instructions per cycle. Other applications have a little amount of ILP, which can be executed by a simple processor.

Presently, as it was shown in Chapter 2, the software designer writes the application code and relies on a compiler to optimize it. However, these compiler optimizations can improve the final code, but they are very limited. Some design decisions taken at higher abstraction levels, like algorithm level, can produce better or worse improvements. For example, when a software designer writes a code that it is a sort algorithm and he/she uses a bubble sort algorithm, there is no compiler that can improve this decision.

3.2.2 The proposed approach

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.

This one has different levels of optimization, one on the software level, and other on the hardware level, selecting the best core providing different performance levels and consuming different levels of power. By exploring design alternatives at the algorithmic level and the architectural level, that offer a much wider range of power and performance, the designer is able to automatically find, through the exploration tool, corner cases that result in optimizations far better that those reported by later code optimizations. This exploration tool can take to the choice of an algorithm and an architecture that exactly fit the requirements of the application, without unnecessarily wasting resources.

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 3.2 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 represent 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.

To generate the application graph representing the dynamic behavior of the application, an instrumentation tool was developed. It is based on BIT (Bytecodes Instrumentation Tool) (LEE; ZORN, 1997), which is a collection of Java classes that allow the construction of customized tools to instrument Java byte-codes. This instrumentation tool allows the dynamic analysis of Java Class files, generating a list of invoked methods with its corresponding number of calls, which can be mapped to the application graph.

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.

It is important to notice that different threads can be mapped to different implementation cores available (different versions of FemtoJava: multicyle, pipeline, VLIW). However, in our approach, the routines that belong to the same thread are mapped to the same processor core.

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:

$$\prod_{i=1}^{m} r_i * p$$

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.

Example (using numbers):

- One program has 4 different methods that use the IP library;

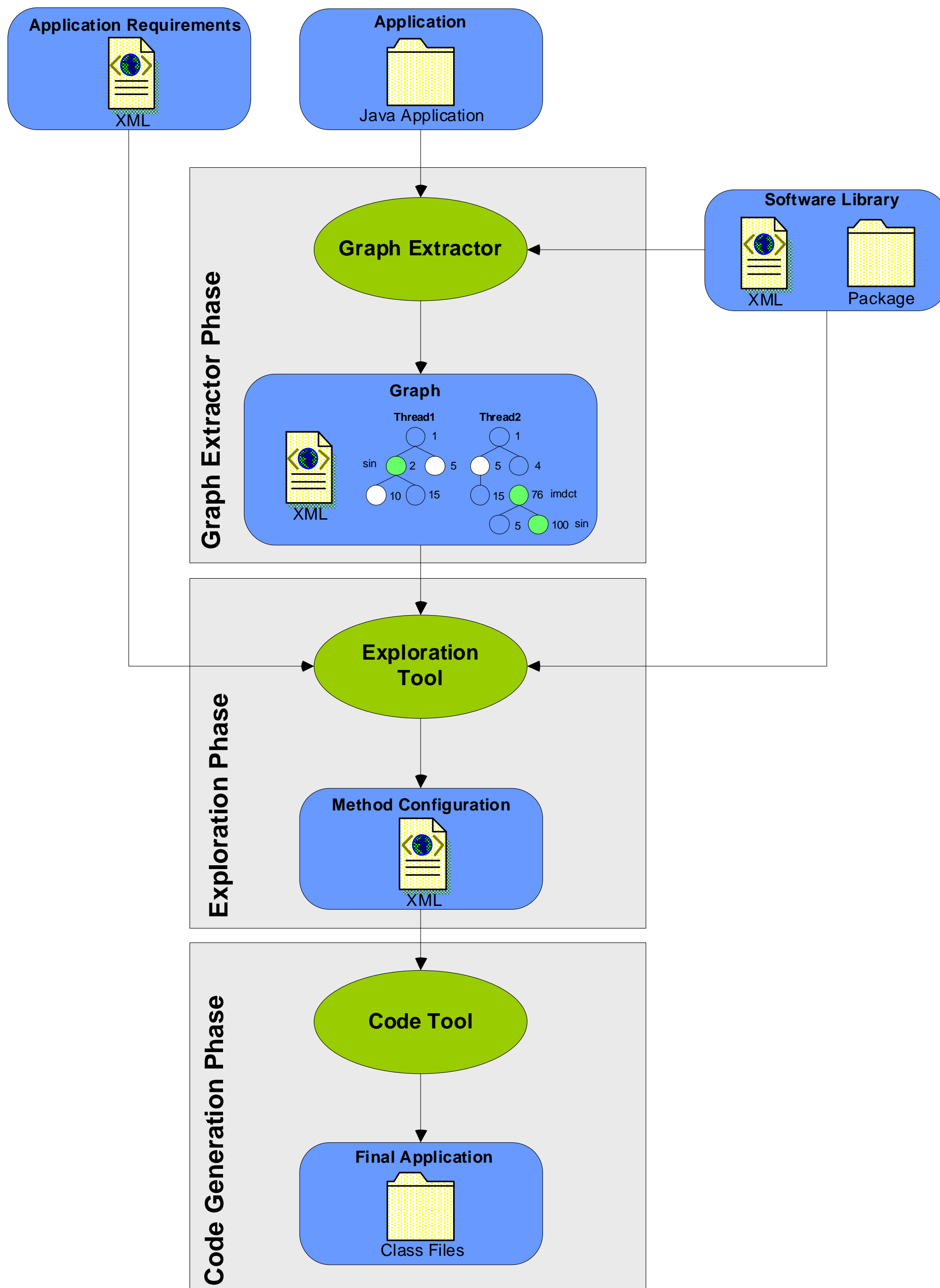

Figure 3.2: Method Exploration Level Design Flow.

- Each routine has 2 alternative implementations and these implementations are characterized into 4 processor cores;
- Thus, there are $8^4 = 2^{12}$ possible solutions.

Software IP library

As it has been already mentioned, the library contains different algorithmic versions of the same function, thus supporting design space exploration. Each algorithmic implementation of the library functions is measured in terms of performance (in cycles), the memory usage (for data and instruction memories), and energy and power dissipation.

Since embedded systems are found in many different application domains, this investigation has been started using classical functions:

- Sine - Two ways to compute the sine of an angle are provided. One is a simple table search, and the other one uses the CORDIC (Coordinate Rotation Digital Computer) algorithm (OMONDI, 1994). The sine function is representative of many other functions with an arithmetic-like behavior. The first alternative has three different variations, regarding the precision of the input angle. The precisions used were 1, 0.5, and 0.1 degrees;
- Table Search - We wanted to have a clearer understanding of the impact of different approaches to the same problem, checking whether the best algorithm for a large data set is also the best one for a smaller data set or not. Not only the search, but also the insertion function of each algorithm, was taken into account. Four approaches were tried: a) Unordered Table, with exhaustive search but very fast insertion (at the end of the vector); b) Ordered Table, with a faster search but slower insertion than the previous alternative; c) Binary Search on an ordered table; and d) Hash Table;
- Square Root - As for the sine function, there are many ways a square root can be computed, with different trade-offs. Two versions are implemented in the library;
- Sort - Four algorithms for sorting a vector were implemented: Bubble Sort, Insert Sort, Select Sort, and Quick Sort;
- IMDCT - The Inverse Modified Discrete Cosine Transform is a critical step in decompression algorithms like those found in MP3 players. Together with windowing, it takes roughly 70% of the processing time (SALOMONSEN, 1997). Four versions of the IMDCT have been implemented.

Design Space Exploration

In our approach, we are using the Dragon Lemon tool (HENTSCHKE, 2007). This tool maps the routines of an embedded program to an implementation using

instances of the software IP library, so as to fulfil given system requirements.

In the exploration tool, before the search begins, the user may determine weights for power, delay and memory optimization. It is also possible to set maximum values for each of these variables. The tool automatically explores the design space and finds the optimal or near optimal mapping for that configuration.

The cost function of the search is based on a trade-off between power, timing, and area. Each library option is characterized by these three factors. The exploration tool normalizes these parameters by the maximum power, timing and area found in the library. The user can then select weights for the three variables. This way, the search can be directed according to the application requirements. If area cost, for example, must be prioritized because of small memory space, the user may increase the area weight. Although one characteristic might be prioritized, the others are still considered in the search mechanisms.

There are two search mechanisms. The first one is an exhaustive search. For small programs, this search runs in acceptable times. For larger problems, since we are dealing with an optimization problem of multiple variables, we implemented a genetic algorithm.

As output, Dragon Lemon also provides 2D and 3D Paretto curves. For both curves, the user may select which variables (power, delay, or memory) will be used in each axis (x, y and z). While running the exhaustive search, the tool finds the exact Paretto points as it explores all possibilities. In genetic search, only the searched nodes are considered. However, as the genetic heuristic searches for the most promising solutions while avoiding the bad ones, it is adequate to find the Paretto points. By running several experiments it can be observed that, in fact, the Paretto curves found by genetic search are very much similar to those found by exhaustive search.

3.3 Object exploration level

This section shows the software exploration approach to improve the operating system level. The main idea in this level is to explore the organization of the objects of the application according the requirements. First, it shows the problem and its characterization. Finally, it shows the proposed solution.

3.3.1 The problem

Object-oriented programming increases the software productivity. As mentioned in chapter 1, it is well known that the object-oriented programming paradigm significantly increases the dynamic memory used, producing considerably overhead in terms of performance, memory and power.

This is a serious problem. On the one hand, the OO methodology can help embedded designers making the design process easier and faster. On the other hand, the OO methodology introduces several types of overhead that cause problem to embedded systems. Previous works (EL-KHARASHI; ELGUIBALY; LI, 2000; LUN;

FONG; HAU, 2003) show that this overhead can reach about 88% of total execution time.

Our results agree with these works, the plot in figure 3.3 shows our statistics about the overhead in terms of performance that might be expected by dynamic allocation and deallocation. The figure shows the overhead caused in different applications considering a cost of 1 to 1000 instructions per allocation/deallocation.

As it can be seen, for some applications the memory allocation needed to support the OO paradigm means that more than 50% of the execution time is taken just for the memory management, thus the CPU spends more time and energy just managing memory, instead of actually executing the target application. This is a huge overhead that cannot be paid by many embedded systems, mainly those in mobile devices.

It is interesting to notice what happens when the cost of allocation/deallocation is increased. In some applications more than 80% of the execution time is used by memory management system.

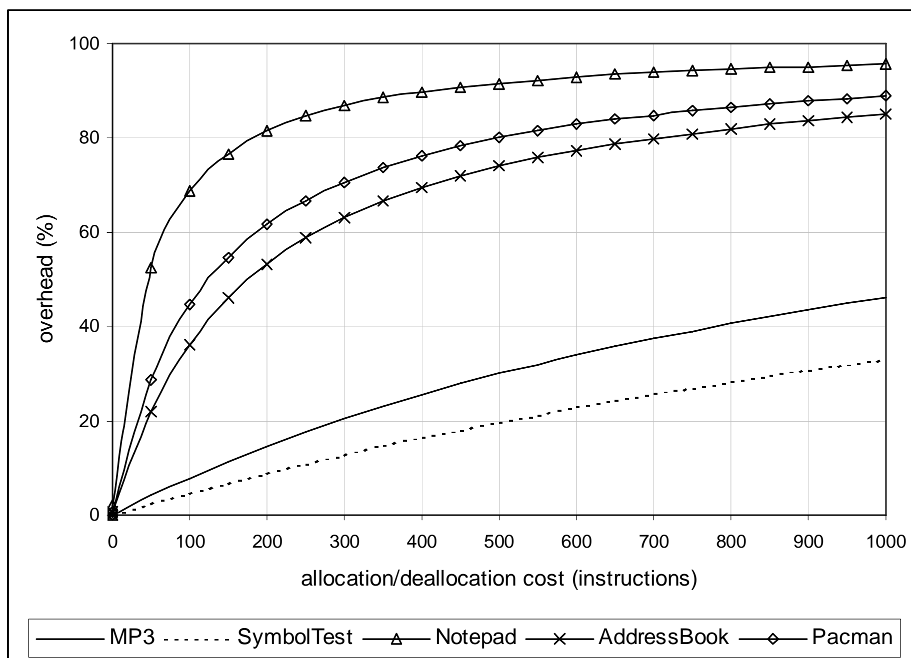

Figure 3.3: Object-Oriented Overhead.

3.3.2 The problem characterization

As it has been already mentioned, OO programming produces an overhead. The goal of this section is to characterize the exact amount of overhead one has to pay to effectively use the OO paradigm. This problem characterization was published in a conference (MATTOS et al., 2005b).

In this work we analyzed some Java applications that can be found in embedded systems. The applications we used as benchmarks are:

- MP3Player - it is an MP3 decoder implementation. This algorithm reads an MP3 file and translates it into an audio signal. This code is a version based on a description available on (JAVALAYER, 2007);
- SymbolTest - it is a simple application that displays Unicode char ranges and different fonts types (SUN MICROSYSTEMS, 2007d);
- Notepad - it is a text editor with simple resources (SUN MICROSYSTEMS, 2007e);
- Address Book - it is an application used as an electronic address book that stores some data (like name, address, telephone number, etc.) (BRENNEMAN, 2007);
- Pacman - it is the well-known game with a labyrinth and rewards (PILON, 2007).

It is important to mention that except for the MP3 application, none of the above applications has been coded by the authors. A completely blind analysis has been performed, in order to avoid influence of a particular code style.

Table 3.1 shows some object information about original applications like the total allocated objects for some instance execution, and the number of allocation instructions. This number of allocation instructions shows the instructions that perform the memory allocation task. Each one of these instructions can create several objects (objects with the same type) because it can be located in a method that is called several times, or can be located in a loop, for example. Table 3.1 shows that during MP3 execution 46,068 objects were created by only 101 allocation instructions, and hence some allocation instructions create more than one object. During the execution, the Garbage Collector collects, from the memory, the objects that have lost their reference. The table also presents the results for the other applications.

Table 3.1: Object data (original applications).

Application	**Total allocated objects**	**Number of allocation instructions**
MP3	46,068	101
SymbolTest	27	16
Notepad	184	66
AddressBook	28	14
Pacman	2,547	30

Table 3.2 shows some memory data. Two results are shown: total memory allocated during the application execution and the maximum memory used during the application. Using object-oriented programming, there is an intensive memory use (allocations and deallocations). However, the memory necessary to run the application should be enough to store just the objects used in the moment (it depends on GC implementation, considering a GC implementation that all objects that lose

their reference are colleted immediately). It is clear from Table 3.2 that there is a huge waste on memory resources, since only a fraction of the allocated memory is effectively used in a certain point of the algorithm.

Table 3.2: Memory data (original applications).

Application	**Total memory allocated (bytes)**	**Maximum Memory utilization (bytes)**
MP3	10,080,512	23,192
SymbolTest	1,509	625
Notepad	9,199	4,185
AddressBook	867	185
Pacman	216,080	456

Table 3.3 presents data in terms of performance and the overhead caused by garbage collector making the allocation and deallocation of the objects assuming that GC takes about 696 instructions. The performance results are shown as the number of executed instruction. The overhead caused by GC was calculated based on a GC implementation in software targeting the FemtoJava processor and Sashimi Tool (NEVES, 2005).

This implementation is based on the Reference Counting algorithm that has a low memory overhead. At each object manipulation the garbage collector needs to make some changes in the respective object counter, and as soon as a counter reaches zero, the corresponding memory block becomes available to a new object. The cost of allocation and deallocation is about 696 instructions on average. In this case the cost of each application can be easily seen to surpass 35% for most applications.

Table 3.3: Performance data (original applications).

Application	**Number of executed instructions**	**GC Overhead (%)**
MP3	85,767,756	37.40
SymbolTest	73,364	27.91
Notepad	136,621	93.86
AddressBook	24,435	79.84
Pacman	2,091,684	84.85

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 2n possible combinations. Each combination is composed by a set of allocation instructions that allocates in dynamic or static way.

Example (using numbers):

- One program has 3 allocation instructions that allocate the objects in a dynamic way;
- Each allocation instruction can be transformed to allocate the objects in a static way;
- There are 23 = 8 possible combinations: DDD, DDS, DSD, DSS, SDD, SDS, SSD and SSS (D stands for dynamic and S stands for static).

3.3.3 The proposed approach

Our approach is composed by a design space exploration tool (DESEJOS Tool: DEsign of Software for Embedded Java with Object Support) 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 experimental results in the previous section have 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.

The DESEJOS Tool was implemented in Java language and uses the BIT Library, the Bytecodes Instrumentation Tool (LEE; ZORN, 1997), which is a collection of Java classes that allow the construction of customized tools to instrument Java bytecodes.

This tool is divided into three main parts: analysis phase, transformation phase and design space exploration phase. Figure 3.4 shows the DESEJOS Tool 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). These main DESEJOS phases are presented next.

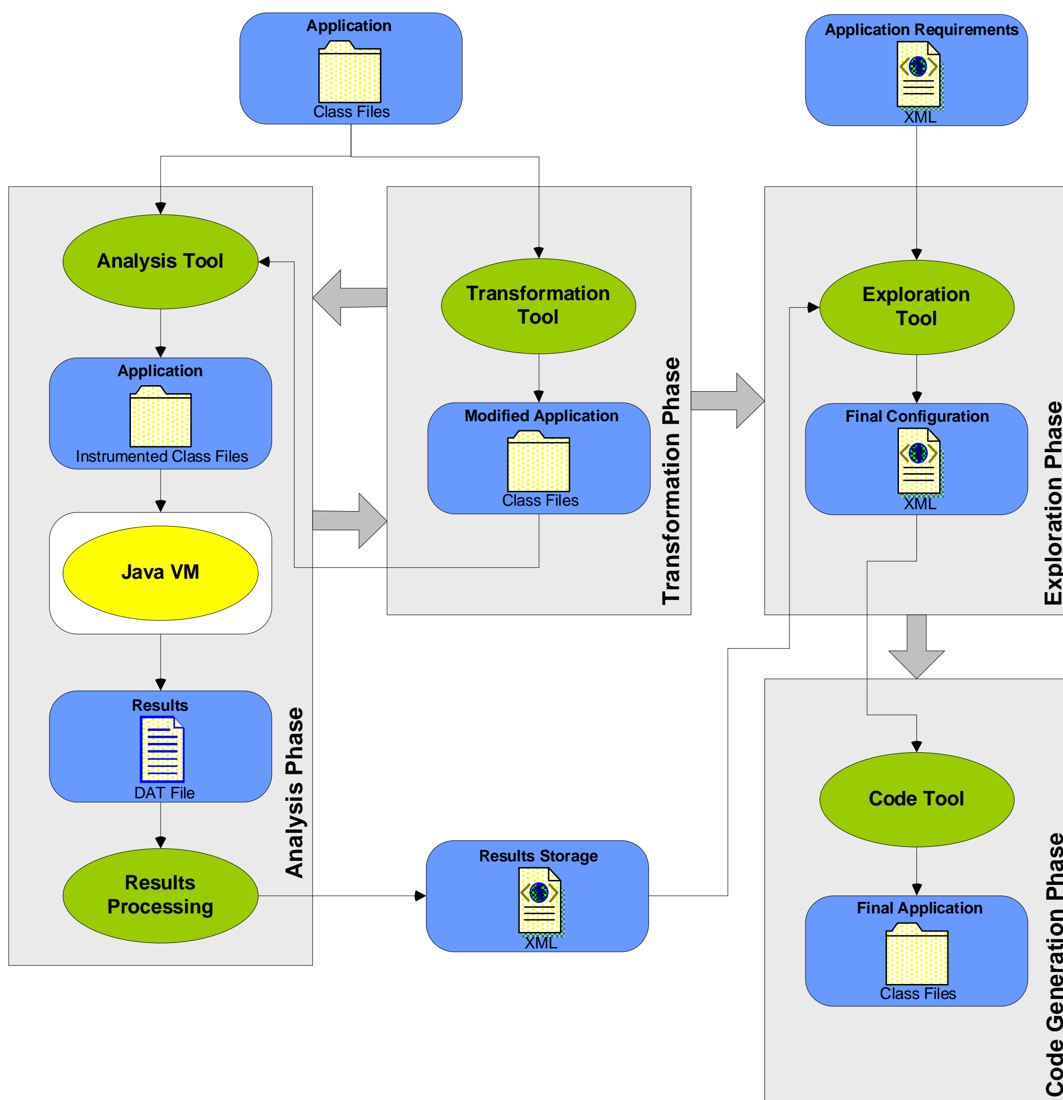

Figure 3.4: Object Exploration Level Design Flow.

Analysis Phase

In analysis phase, the DESEJOS 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. The tool presets for each allocation instruction the number of allocated objects. Moreover, the tool provides other results like: object description, object time life, number of accesses per object and object size;
- Memory results: these results show the total dynamic memory allocated during the application execution, the maximum memory utilization (the memory necessary to run the application should be enough to store just the objects used in the moment - considering a GC implementation that all objects that lose their reference are colleted immediately), and a memory usage histogram;
- Performance results: these show the performance results in terms of executed instructions and the overhead caused by GC. The overhead caused by GC was calculated based on a GC implementation in software targeting the FemtoJava processor and Sashimi Tool (ITO; CARRO; JACOBI, 2001).

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.

Transformation Phase

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. However, to better explain how this transformation was done, we show this change in a Java code (high level code) and not in bytecodes. Figure 3.5 shows the part of original sample code (before changes) and figure 3.6 shows the code after the transformation.

```
// inside the method
for (int i=0; i<5;i++){
   Obj1 refObj = new Obj1();
   refObj.set(i);
   refObj.doTask();
   . . .
}
```

Figure 3.5: Original Code.

```
public class Test {
private static Obj1 refObj = new Obj1();
. . .
// inside the method
for (int i=0; i<5;i++){
   refObj.set(i);
   refObj.doTask();
   . . .
}
```

Figure 3.6: The code after the transformation.

The second code (after changes) is more efficient because the object is allocated just one time. Nevertheless, codes written in this way are very common, because programmers are concerned with the code legacy and intelligibility and not with performance and memory usage. Furthermore, these programmers develop code for desktop machines that have high performance and lots of available memory. The embedded software developers have constraints in terms of performance, power, memory and others, but they want to use the code legacy provided by desktop developers.

Exploration Phase

In the exploration phase, the DESEJOS 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.

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.

The designer can set the following conditions:

1. Set the maximum memory overhead: the tool returns the list of allocation instructions that provides the best performance gain and fulfills the memory requirement;
2. Set the minimum performance gain: the tool returns the list of allocation instructions that provides the smaller memory overhead and fulfills the performance requirement;
3. Set the maximum memory overhead and the minimum performance gain: the tool returns the list of allocation instructions that provides the better results in terms of performance and memory and fulfills the both requirements.

Our problem is very similar to the 0-1 Knapsack Problem (MARTELLO; TOTH, 1990). The problem consists in searching the best combination of performance and memory of a set of objects transformations.

The first designer option (number 1), where the designer sets the maximum memory overhead, is modeled as: given a set of n items (object transformations) with

$$p_j = \text{performance gain of item } j,$$

$$m_j = \text{memory overhead of item j},$$

$$t = \text{maximum memory overhead},$$

select a subset of the items so as to

$$\text{maximize } z = \sum_{j=1}^{n} p_j x_j$$

$$\text{subject to } \sum_{j=1}^{n} m_j x_j \leq t$$

$$x_j = 0 \text{ or } 1,\ j \in N = \{1, ..., n\}$$

where $x_i = 1$ if item j is selected and $x_i = 1$ otherwise.

The second designer option (number 2), where the designer sets the minimum performance gain, is modeled as: given a set of n items (object transformations) with

$$p_j = \text{performance gain of item } j,$$

$$m_j = \text{memory overhead of item j},$$

$$t = \text{minimal target performance},$$

select a subset of the items so as to

$$\text{minimize } z = \sum_{j=1}^{n} m_j y_j$$

$$\text{subject to } \sum_{j=1}^{n} p_j y_j \geq t$$

$$y_j = 0 \text{ or } 1,\ j \in N = \{1, ..., n\}$$

where $y_i = 1$ if item j is selected and $y_i = 1$ otherwise.

We implemented our algorithm using dynamic programming (HOROWITZ; SAHNI, 1978). The essence of dynamic programming is to build large tables with all known previous results. The tables are constructed iteratively. Each entry is computed from a combination of other entries above it or on the left of it in the matrix.

This algorithm has the pseudo-polynomial time and its complexity is $O(t * n)$. The runtime depends on the size of matrix $(t * n)$, where t is the size of memory and n is the number of allocation instructions. The main problem is to organize the construction of the matrix in the most efficient way. In our case, the size of the memory can be a problem, because it increases the size of the matrix. But, we can reduce the size of the memory using Kbytes instead of bytes.

3.3.4 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 microcontroller, called FemtoJava (ITO; CARRO; JACOBI, 2001; BECK FILHO; CARRO, 2003, 2004). The FemtoJava Microcontroller implements an execution engine for Java in hardware through a stack machine compatible with Java Virtual Machine (JVM) specification. A CAD environment (Sashimi Tool) that automatically synthesizes an Application Specific Instruction-Set Processor (ASIP) version of the Java microcontroller for a target application (ITO; CARRO; JACOBI, 2001) is available, using only a subset of critical instructions to the specific application.

Our platform uses three different versions of the FemtoJava processor: multicycle, pipeline and a VLIW one. The multicycle version supports stack operations through stack emulation on their register files. This approach reduces the memory access bottleneck of the stack machine, improving performance.

The second architecture is the pipelined version (BECK FILHO; CARRO, 2003), which has five stages: instruction fetch, instruction decoding, operand fetch, execution, and write back. Thanks to the forwarding technique in the stack architecture, the write back stage is not always executed, and hence there are meaningful energy savings when compared to a regular 5 state pipeline of a RISC CPU.

The VLIW processor is an extension of the pipelined one (BECK FILHO; CARRO, 2004). Basically, it has its functional units and the instruction decoders replicated.

The VLIW packet has a variable size, avoiding unnecessary memory accesses. A header in the first instruction of the word informs to the instruction fetch controller how many instructions the current packet has. The search for ILP in the Java program is done at the bytecode level. The algorithm works as follows: all the instructions that depend on the result of the previous one are grouped in an operand block. The entire Java program is divided into these groups and they can be parallelized respecting the functional unit constraints.

This platform is chosen because it is available in our research group. Furthermore, the native execution of Java bytecodes can improve performance and solve several problems concerning embedded systems, mainly the overhead produced by JVM.

4 RESULTS

This chapter shows the results the design methodology into one case study: an MP3 Player (an audio encoding format). First, we summarize the results concerning de software IP library. After, we show the design space exploration of MP3 player application results concerning method and object exploration level.

4.1 Library Characterization

This section shows the results of the software IP library. The software IP library contains different algorithmic versions of the same function, like sine, table search, square root, IMDCT (described in section 3.2.2), 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.

To illustrate the results of library characterization using different algorithmic versions of the same function and different cores (multicycle, pipeline and a VLIW version with 2 words), there are three routines selected: sine, table search and Inverse Modified Discrete Cosine Transform. The results in terms of performance, power and energy are obtained using the CACO-PS simulator (BECK et al., 2003).

Table 4.1 and 4.2 illustrates the characterization of the alternative implementations of the sine function. Table reftab:SineCharacterization shows the software results that do not depend on hardware and Table 4.2 shows the results that depend on hardware. Since Cordic is a more complex algorithm, program memory size is larger than it is with Table Look-up (Table 4.1), as well as the number of cycles required for computation for all the cores (Table 4.2). It is interesting to notice, however, that when the resolution increases, the amount of data memory increases exponentially for the Table Look-up algorithm, but only sublinearly for the Cordic algorithm. The increase in memory reflects not only in the required amount of memory, but also in the power dissipation of a larger memory.

Table 4.2 presents the results in terms of performance, power and energy, using a frequency equal to 50 MHz and Vdd equal to 3.3v. The pipeline and VLIW architectures provide better results in terms of cycles. The best results in terms of performance came from the combination of sine calculation as simple table search and VLIW architecture, but the worst results in terms of power. The best combination in terms of power, but worst in terms of energy, is the sine routine using Cordic

and the multicycle core.

Table 4.3, 4.4 and 4.5 shows the main results of the characterization of the four different implementations of the IMDCT function. The IMDCT4 implementation has the best results in terms of performance in all architectures, but the size of program memory significantly increases. The opposite happens with the IMDCT1 implementation, which has far better results in terms of program memory, but consumes more cycles. In terms of power (using a frequency equal to 50 MHz and Vdd equal to 3.3v) the best combination is the IMDCT2 and IMDCT3 with multicycle core, but this combination does not have good results in terms of energy. Table 4.4 and 4.5 show that the best results in terms of performance are the results that execute in VLIW core, since the IMDCT routine has lots of parallelism.

Table 4.1: Sine Characterization.

Characteristic		**Cordic**	**Table**
Program size (bytes)		206	88
	1°	184	220
Data mem (bytes)	0.5^o	184	400
	0.1^o	184	1840

Table 4.2: Sine Characterization (hardware dependable).

Characteristic	**Cordic**			**Table**		
	Multi	Pipeline	VLIW2	Multi	Pipeline	VLIW2
Performance (cycles)	2447	755	599	136	65	55
Power (mW)	11.8092	16.1626	22.9019	13.4431	17.8235	20.1606
Energy (μJ)	577.9421	244.0559	274.4414	36.5652	23.1705	22.1779

Table 4.3: IMDCT Characterization.

Characteristic	**IMDCT1**	**IMDCT2**	**IMDCT3**	**IMDCT4**
Program size (bytes)	344	2,137	4,260	15,294
Data mem (bytes)	3546	3546	3546	3546

Table 4.4: IMDCT Characterization (hardware dependable).

Characteristic	**IMDCT1**			**IMDCT2**		
	Multi	Pipeline	VLIW2	Multi	Pipeline	VLIW2
Performance (cycles)	140300	40306	33051	97354	31500	19325
Power (mW)	8.8533	20.0533	24.9227	8.6595	17.8944	25.3609
Energy (μJ)	24.8424	16.1654	16.4744	16.8607	11.2735	9.8021

Table 4.5: IMDCT Characterization (hardware dependable)(cont.).

Characteristic	**IMDCT3**			**IMDCT4**		
	Multi	Pipeline	VLIW2	Multi	Pipeline	VLIW2
Performance (cycles)	92882	30369	17329	51345	18858	9306
Power (mW)	8.6483	17.7355	27.2193	9.1435	17.3849	34.5890
Energy (μJ)	16.0654	10.7722	9.4334	9.3894	6.5569	6.4380

4.2 MP3 Case Study Results

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. Figure 4.1 shows the decodification steps of the MP3 standard.

Our application code is based on a description freely available on the Internet (JAVALAYER, 2007). All the MP3 code was written in Java but obeying certain constraints of our tools and architecture:

- It is not possible to deal with API codes. Many of the standard library classes use native code (when an application cannot be written entirely in the Java and should be written in another programming language) to provide functionality to the developer and the user, e.g., I/O file reading and sound capabilities. Thus, these tools can not handle this native code;
- Several instructions are not supported by the processors. An example constraint is the use of integer data instead of floating-point ones, because there is no such unit available in the processor;
- There is a set of instructions that are supported by software. Several instructions are not implemented in hardware, consequently they produce an overhead. Most of the instructions (bytecodes) that manipulate memory management are implemented in software (e.g. new and newarray bytecodes).

The MP3 applications are coded on two different styles: a static and dynamic version. The first version (static) does not use resources like create objects, use interfaces and son. The dynamic version creates objects, so it is necessary to use a garbage collector. The appendix B and D present the class and sequence diagram, respectively, of the static version while the appendix C and E present the class and

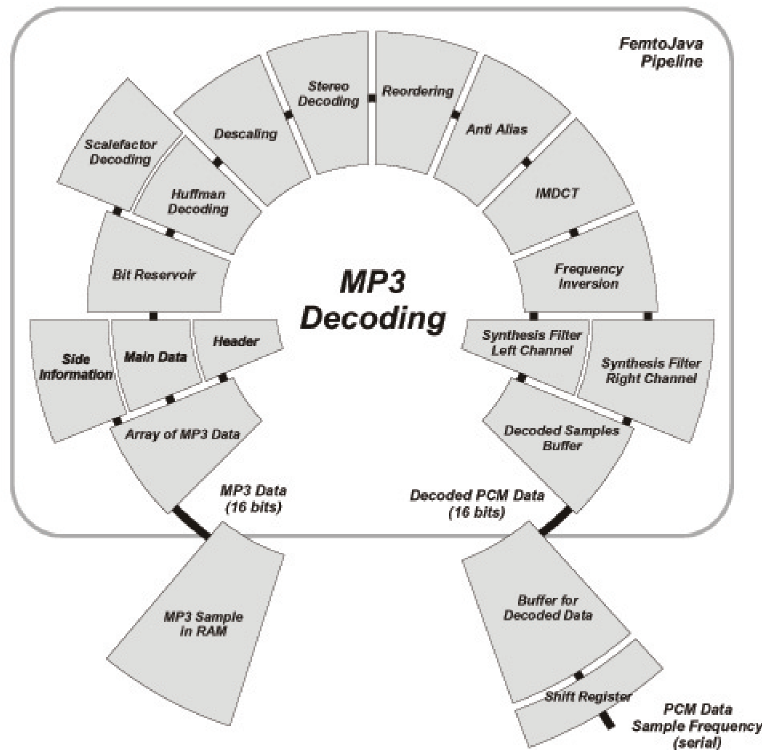

Figure 4.1: MP3 Decoding steps.

sequence diagram of the dynamic version.

4.2.1 Method exploration on MP3

This section shows the results of the methodology based on software IP library and a set of cores with the same ISA that uses a tool for automatic design space exploration and SW and HW IP selection. These preliminary results were published in two conferences (MATTOS et al., 2004)(MATTOS et al., 2004).

To show that this approach can be applied to a real application Table 4.6 presents the profiling results of the MP3 static application implement with a traditional of IMDC routine running in the pipeline version of the processor. The table shows for each method: the name, number of calls, number of cycles and percentage of the cycles of each method in relation of total cycles (the table shows the results for only four methods - the most executed ones).

Table 4.6 shows that just two methods are responsible for more than 90% of the total application execution. The method IMDCT is called 864 times and cos_cordic method is called 504,576 times. It is important to notice that cos_cordic method is called by IMDCT method. This implementation uses IMDCT1 and Cosine calculation (Cordic algorithm).

The availability of different alternatives of the same routine is just a first step in the design space exploration of the application software. One must notice that embedded applications are seldom implemented with a single routine. There is an-

Table 4.6: MP3 profiling results using IMDCT1 and Cosine Cordic.

Method name	**Number of calls**	**Number of cycles**	**Percentage (%)**
Imdct1.IMDCT	864	34,824,384	7.69
CosIMDCT.cos_cordic	504,576	380,954,880	84.15
SynFilter1.calculatePCMSamples	450	5,811,497	1.28
SynFilter1.calculatePCMSamples	450	5,811,497	1.28
Other methods		25,271,821	5.58
Total cycles:		452,674,079	

other level of optimization, which concerns finding the best mix of routines among all possible combinations that may exist in an embedded application.

In order to better illustrate the concept, let us take as an example the IMDCT function (characterized for pipeline core). In its kernel there is a cosine function that has smaller impact in terms of overall performance. If one is aiming only at global performance, one could pick the IMDCT4 core that takes 18,858 cycles. Adding a cosine function based on the table look-up method the total cycles of the application takes 49,090,752 reducing roughly 89% the number of total cycles. However, the memory size increases 15,383 program words. Table 4.7 shows these results.

Table 4.7: MP3 profiling results using IMDCT4 and Cosine Table.

Method name	**Number of calls**	**Number of cycles**	**Percentage (%)**
Imdct4.IMDCT	864	16,293,312	18.95
CosIMDCT.cos_table	504,576	32,797,440	38.11
SyntFilter1.calculatePCMSamples	450	5,811,497	6.76
SyntFilter1.calculatePCMSamples	450	5,811,497	6.76
Other methods		25,271,821	29.39
Total cycles:		85,985,567	

Table 4.8 shows an intermediate solution. If one is aiming only at global performance, it is not possible, however have to pay the memory overhead (one could pick the IMDCT1 core and the cosine function based on the table look-up method). Table shows the results of this solution where it increases the total performance of the application 4.33 times. This is just a simple example considering only performance and memory aspects. Other requirements (power and energy) can be analyzed depending on application requirements.

4.2.2 Object exploration level results

This section shows the results of the methodology based on exploring the organization of the objects of the application. These results were published in (MATTOS et al., 2005a) and (MATTOS; CARRO, 2007).

Table 4.8: MP3 profiling results using IMDCT1 and Cosine Table.

Method name	Number of calls	Number of cycles	Percentage (%)
Imdct4.IMDCT	864	34,824,384	33.32
CosIMDCT.cos_table	504,576	32,797,440	31.31
SyntFilter1.calculatePCMSamples	450	5,811,497	5.56
SyntFilter1.calculatePCMSamples	450	5,811,497	5.56
Other methods		25,271,821	24.18
Total cycles:		104,516,639	

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.

In section 3.3.2, Table 3.1 shows that during MP3 execution 46,068 objects were created by only 101 allocation instructions, and hence some allocation instructions created more than one object. Thus, there are 101 possible objects transformation in MP3 application.

Table 4.9 presents just ten different allocation instructions and their results in terms of the number of objects that each instruction allocates and the size of the object. The comparison between the number of total allocated objects (46,068) by the application with the number of allocated objects by the first instruction allocation (first row) shows that just one allocation instruction is responsible for 62.51% of allocations. Transforming this allocation instruction in a static way, the results in terms of GC overhead can be extremely improved. The Table 4.9 also presents that other allocation instructions can improve the results too. But when a static transformation occurs, this transformation obviously implies in memory increase.

Table 4.10 shows the results after the static transformations with the same allocation instructions of the Table 4.9. These results show the performance in terms of cycles (in the pipeline version), the percentage reduction in respect to the original code (total OO code) and the memory increase necessary to make the static transformation. It is interesting to notice that the static transformation in only one allocation instruction can improve the performance results in 23.47% paying only 0.28% of memory increase.

The other allocation instructions present different results in terms of performance gain and memory increase. These values seem insignificant, but these transformations can be grouped taking more advantages. The Table 4.10 shows the results of the combination of different allocation instructions. For example, the third row shows the results of the combination of the allocation instruction 1, 2 and 3. These combinations show, as example, that grouping different static transformations can be obtained a great number of possibilities with different characterization in terms

Table 4.9: MP3 Allocation instruction.

Allocation instruction	**Number of allocated objects**	**Object Size**
#1	28,800	64
#2	1,728	144
#3	1,728	144
#4	1,728	36
#5	1,728	36
#6	1,600	144
#7	1,600	72
#8	1,536	144
#9	943	4,096
#10	900	128

Table 4.10: MP3 results after static transformation.

Allocation instruction	**Number of cycles**	**Reduction (%)**	**Memory Increase (%)**
#1	104,306,114	23.47	0.28
#2	134,378,057	1.41	0.62
#3	134,378,067	1.41	0.62
#4	134,378,057	1.41	0.16
#5	134,378,057	1.41	0.16
#6	134,520,241	1.30	0.62
#7	134,520,241	1.30	0.31
#8	134,591,333	1.25	0.62
#9	135,250,046	0.77	17.66
#10	135,297,811	0.73	0.55

of performance and memory overhead.

When different static transformations are grouped, there are a great number of possibilities with different characterization in terms of performance and memory overhead. The MP3 application has 101 allocation instructions that can be transformed in static allocation, thus there are 2^{101} combinations that the exploration tool can evaluate.

Figures 4.2, 4.3 and 4.4 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.

Table 4.11: MP3 combinations of allocation instructions.

Allocation instruction	**Number of cycles**	**Reduction (%)**	**Memory Increase (%)**
#1	104,306,114	23.47	0.28
#1+#2	102,386,628	24.88	0.90
#1+#2+#3	100,467,152	26.29	1.06
#1+#2+#3+#4	98,547,666	27.70	1.22

In the figure 4.2, 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 solution present better results in terms of performance because uses the pipeline processor.

The figure 4.3 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.

Finally, figure 4.4 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. To show the use of the tool, we selected one example with different application requirements. Set the maximum memory overhead at 51,700 bytes as application requirements. The results are:

- The performance gain is 84,670,324 instructions that represent 37.87% of gain in relation the original application;

- The memory overhead is 51,688 bytes (fulfill the requirements).

This solution is represented by the square solution presented in the three figures (pointed by an arrow). The solution fulfill the requirements, however it has an overhead in terms of power because the pipeline version is selected.

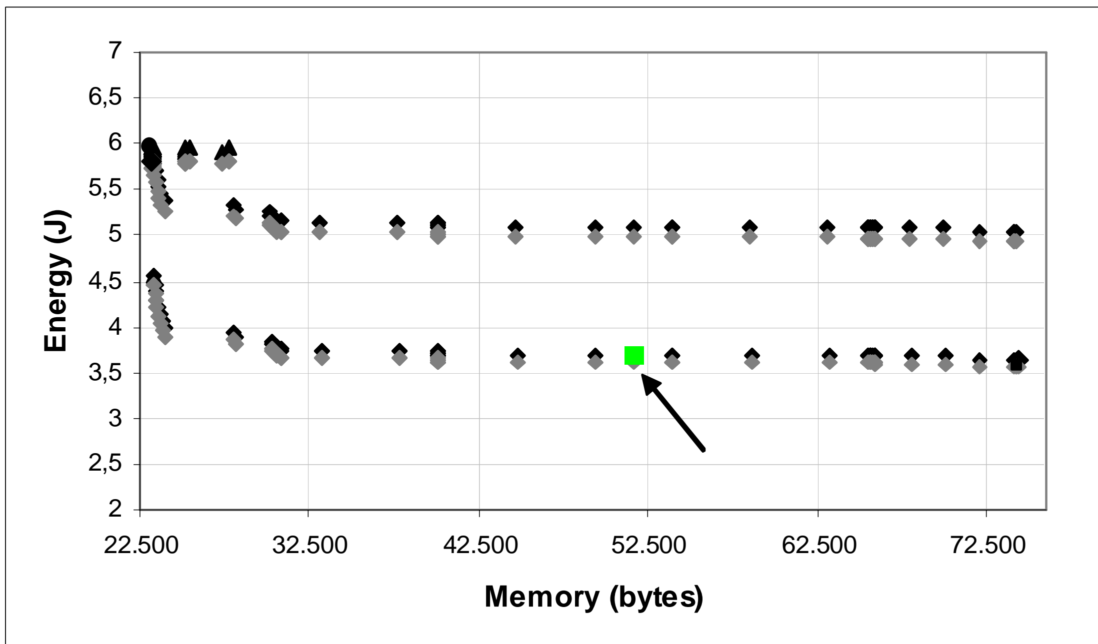

Figure 4.2: MP3 Performance vs. Memory Design Space.

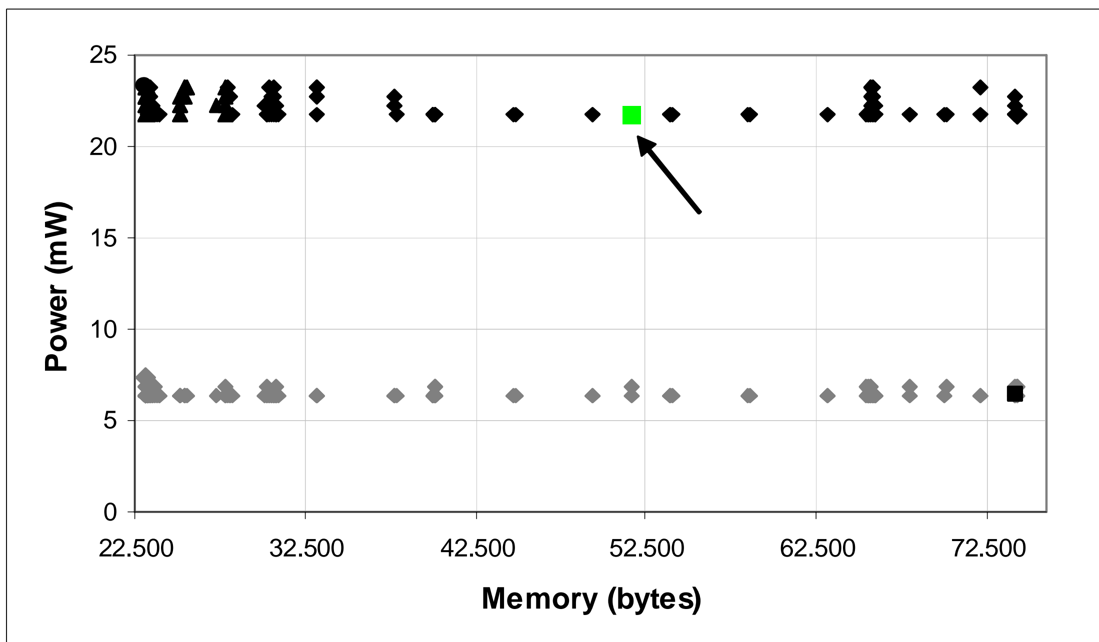

Figure 4.3: MP3 Power vs. Memory Design Space.

7
6,5
6
5,5
5
4,5
4
3,5
3
2,5
2
Energy (J)
22.500
32.500
42.500
52.500
62.500
72.500
Memory (bytes)

Figure 4.4: MP3 Energy vs. Memory Design Space.

5 CONCLUSIONS AND FUTURE WORK

This thesis has presented a methodology to explore object-oriented embedded software improving different tasks during the system design. This exploration is divided into different phases: the method exploration level and the object exploration level.

On the first exploration level, we propose a methodology for design space exploration using automatic selection of software and hardware IP components for embedded applications. It is based on a software IP library and a set of cores with the same ISA, the library is previously characterized for each core, and we use a tool for automatic design space exploration and SW and HW IP selection.

Experimental results have confirmed the hypothesis that there is a large space to be explored based on algorithmic decisions taken at higher levels of abstraction, much before compiler intervention. Selecting the right algorithm/right architecture might give orders of magnitude of gain in terms of physical characteristics like memory usage, performance, and power dissipation.

On the second exploration level, we propose a methodology to improve the execution of OO embedded application, transforming as many dynamic objects as possible to static ones, reducing execution time, while maintaining memory costs as low as possible. Not only is this approach very simple, but it can also lead to substantial gains. The experimental results have confirmed the expect gains, and the methodology provides a large design space exploration for a given application.

There are some limitations of the proposed approach. First, on the method exploration level, the exploration tool can map different threads (with a set of routines) to different processors cores. Nowadays, the hardware infrastructure does not support this type of application, because a communication mechanism is necessary. Moreover, the communication overhead must be considered during the mapping.

Second, there are also some limitations on the object exploration level. The main restriction is the impossibility to deal with APIs, because these use native code. Thus, it is not possible to use common applications. Furthermore, not all objects can be transformed into static ones. There are some cases when this not possible, for example when some data from the user input is used to set the size of an array of objects.

Some technical contributions of this work can be mentioned:

1. The methodology based on software IP library and a set of cores with the same ISA that uses a tool for automatic design space exploration and SW and HW IP selection was published in two conferences. The first work presents the results based only on SW IP exploration (MATTOS et al., 2004). The second work presents the idea using design space exploration of SW and HW IP (MATTOS et al., 2004);
2. The methodology based on exploring the organization of the objects of the application was published in two conferences. First, the problem characterization was published in (MATTOS et al., 2005b). After, the methodology has been evaluated with a complex example (an MP3 player) and these results were published in (MATTOS et al., 2005a);
3. The whole methodology using the both approaches, method and object exploration levels was published in (MATTOS; CARRO, 2007). Moreover, the whole methodology was present at DATE06 EDAA PhD Forum (MATTOS; CARRO, 2006).
4. During this thesis some work related to reconfigurable architectures have been investigated. As a result of the "Sandwich" PhD, one paper was published in (MATTOS; WONG; CARRO, 2006). A methodology based on the reconfiguration of the most created object was published in (MATTOS; BECK FILHO; CARRO, 2006).

Considering this research subject, there are several general issues to be investigated, like code optimization, design space exploration, use of object orientation in embedded system. However, considering specifically the topics of this thesis, there are some points that can be studied as a future work:

1. Nowadays, the main constraint is the impossibility to deal with APIs. To solve this problem an API must be implemented. Thus, it will be possible to use applications available on the community;
2. Evaluate the whole methodology with more applications considering the API. The API code can produce a huge overhead hiding the benefits of the methodology on the application code;
3. During this work, a large set of tools have been used to obtain the results. These tools must be integrated, making their interface more easy to developers;
4. The study to provide mechanism to support threads allowing the use of the multiple cores in the same application;
5. Integration of the methodology presented in this thesis with other works in our research group. Mainly the use of this technique with the strategy for embedded software development based on high-level models (BRISOLARA, 2007);
6. Study the impact of object organization in micro-architecture, like caches. We intend to analyze the memory traffic, the behavior of the objects and how we can modify the code to improve the performance and power.

REFERENCES

ARM. **ARM Jazelle Technology**. 2007. Available at: <http://www.arm.com/products/solutions/Jazelle.html>. Visited on: September 2007.

ARNOLD, M.; FINK, S.; GROVE, D.; HIND, M.; SWEENEY, P. A survey of adaptive optimization in virtual machines. **Proceedings of the IEEE**, Los Alamitos, CA, USA, v.93, n.2, p.449–466, 2005.

BACON, D. F.; CHENG, P.; GROVE, D. Garbage collection for embedded systems. In: ACM INTERNATIONAL CONFERENCE ON EMBEDDED SOFTWARE, EMSOFT, 4., 2004, Pisa, Italy. **Proceedings...** New York: ACM, 2004. p.125–136.

BALARIN, F.; CHIODO, M.; GIUSTO, P.; HSIEH, H.; JURECSKA, A.; LAVAGNO, L.; SANGIOVANNI-VINCENTELLI, A.; SENTOVICH, E.; SUZUKI, K. Synthesis of software programs for embedded control applications. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, Los Alamitos, CA, USA, v.18, n.6, p.834–849, June 1999.

BECK FILHO, A. C.; CARRO, L. Low Power Java Processor for Embedded Applications. In: IFIP WG 10.5 INTERNATIONAL CONFERENCE ON VERY LARGE SCALE INTEGRATION OF SYSTEM-ON-CHIP, VLSI-SOC, 2003, Darmstadt, Germany. **Proceedings...** Boston: Springer, 2003. p.239–244.

BECK FILHO, A. C.; CARRO, L. A VLIW low power Java processor for embedded applications. In: INTEGRATED CIRCUITS AND SYSTEM DESIGN, SBCCI, 17., 2004, Pernambuco, Brazil. **Proceedings...** New York: ACM, 2004. p.157–162.

BECK FILHO, A. C.; MATTOS, J. C. B.; WAGNER, F. R.; CARRO, L. CACO-PS: a general purpose cycle-accurate configurable power simulator. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 16., 2003, São Paulo, Brazil. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 2003. p.349–354.

BERLEA, A.; COTOFANA, S. D.; ATHANASIU, I.; GLOSSNER, C. J.; VASSILIADIS, S. Garbage collection for the Delft Java Processor. In: IASTED INTERNATIONAL CONFERENCE ON APPLIED INFORMATICS, AI, 8., 2000, Innsbruck, Austria. **Proceedings...** [S.l.: s.n.], 2000. p.232–238.

BHAKTHAVATSALAM, S.; EDWARDS, S. H. Applying object-oriented techniques in embedded software design. In: CPES 2002 POWER ELECTRONICS SEMINAR AND NSF/INDUSTRY ANNUAL REVIEW, 2002. **Proceedings. . .** [S.l.: s.n.], 2002. Available at: <http://web-cat.cs.vt.edu/PEBB/CPES02-Bhakthavatsalam.pdf>. Visited on: September 2007.

BOLLELLA, G. (Ed.). **The Real-Time Specification for Java**. Boston: Addison Wesley, 2000.

BOLLELLA, G.; GOSLING, J. The Real-Time Specification for Java. **IEEE Computer**, Los Alamitos, CA, USA, v.33, n.6, p.47–54, June 2000.

BRENNEMAN, T. R. **Java Address Book (ver. 1.1.1)**. 2007. Available at: <http://www.geocities.com/SiliconValley/2272>. Visited on: September 2007.

BRISOLARA, L. B. de. **Strategies for embedded software development based on high-level models**. 2007. Tese (Doutorado em Ciência da Computação) — Programa de Pós-Graduação em Computação, UFRGS, Porto Alegre.

BUDDY, T. **An Introduction to Object-Oriented Programming**. Boston: Addison Wesley, 2001.

CHATZIGEORGIOU, A.; STEPHANIDES, G. Evaluating Performance and Power of Object-Oriented Vs. Procedural Programming in Embedded Processors. In: ADA-EUROPE INTERNATIONAL CONFERENCE ON RELIABLE SOFTWARE TECHNOLOGIES, 7., 2002, Vienna, Austria. **Proceedings. . .** Berlin: Springer, 2002. p.65–75. (Lecture Notes in Computer Science, v.2361).

CHEN, G.; KANDEMIR, M.; VIJAYKRISHNAN, N.; IRWIN, M. J.; MATHISKE, B.; WOLCZKO, M. Heap compression for memory-constrained Java environments. In: ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMING, SYSTEMS, LANGUAGES, AND APPLICATIONS, OOPSLA, 18., 2003, Anaheim, California. **Proceedings. . .** New York: ACM, 2003. p.282–301.

CHEN, G.; SHETTY, R.; KANDEMIR, M.; VIJAYKRISHNAN, N.; IRWIN, M. J.; WOLCZKO, M. Tuning garbage collection for reducing memory system energy in an embedded java environment. **ACM Transactions on Embedded Computing Systems (TECS)**, New York, NY, USA, v.1, n.1, p.27–55, 2002.

CHEREM, S.; RUGINA, R. Region analysis and transformation for Java programs. In: INTERNATIONAL SYMPOSIUM ON MEMORY MANAGEMENT, ISMM, 4., 2004, Vancouver, BC, Canada. **Proceedings. . .** New York: ACM, 2004. p.85–96.

CHOI, K. won; CHATTERJEE, A. Efficient instruction-level optimization methodology for low-power embedded systems. In: INTERNATIONAL SYMPOSIUM ON SYSTEMS SYNTHESIS, ISSS, 14., 2001, Montreal, Canada. **Proceedings. . .** New York: ACM, 2001. p.147–152.

CRNKOVIC, I. Component-based software engineering for embedded systems. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE, 27., 2005, St. Louis, MO. **Proceedings. . .** New York: ACM Press, 2005. p.712–713.

DALAL, V.; RAVIKUMAR, C. Software power optimizations in an embedded system. In: INTERNATIONAL CONFERENCE ON VLSI DESIGN, 14., 2001, Bangalore, India. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 2001. p.254–259.

DETERS, M.; LEIDENFROST, N.; HAMPTON, M.; BRODMAN, J.; CYTRON, R. Automated reference-counted object recycling for real-time Java. In: IEEE REAL-TIME AND EMBEDDED TECHNOLOGY AND APPLICATIONS SYMPOSIUM,RTAS, 10., 2004, Toronto, Canada. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 2004. p.424–433.

DETLEFS, D.; DOSSER, A.; ZORN, B. Memory allocation costs in large C and C++ programs. **SoftwarePractice & Experience**, New York, NY, USA, v.24, n.6, p.527–542, June 1994.

DUTT, N.; NICOLAU, A.; TOMIYAMA, H.; HALAMBI, A. New directions in compiler technology for embedded systems. In: ASIA AND SOUTH PACIFIC DESIGN AUTOMATION CONFERENCE, ASP-DAC, 2001, Yokohama, Japan. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 2001. p.409–414.

EGGERMONT, L. D. **Embedded Systems Roadmap**. 2002. Available at: <http://www.stw.nl/progress/ESroadmap/index.html>. Visited on: September 2007.

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**, New York, v.24, n.5, p.237–250, Sept. 2000.

GENSSLER, T.; CHRISTOPH, A.; WINTER, M.; NIERSTRASZ, O.; DUCASSE, S.; WUYTS, R.; AREVALO, G.; SCHONHAGE, B.; MULLER, P.; STICH, C. Components for embedded software: the pecos approach. In: INTERNATIONAL CONFERENCE ON COMPILERS, ARCHITECTURE, AND SYNTHESIS FOR EMBEDDED SYSTEMS, CASES, 2002, Grenoble, France. **Proceedings...** New York: ACM Press, 2002. p.19–26.

GRAAF, B.; LORMANS, M.; TOETENEL, H. Embedded software engineering: the state of the practice. **IEEE Software**, Los Alamitos, CA, USA, v.20, n.6, p.61–69, Nov./Dec. 2003.

HEINEMAN, G. T.; COUNCILL, W. T. **Component-Based Software Engineering**: putting the pieces together. Boston: Addison Wesley, 2001.

HENTSCHKE, R. **Dragon Lemon**. 2007. Available at: <http://www.inf.ufrgs.br/ renato/dragonlemon>. Visited on: September 2007.

HOROWITZ, E.; SAHNI, S. **Fundamentals of Computer Algorithms**. Maryland: Computer Science Press, 1978.

ITO, S. A.; CARRO, L.; JACOBI, R. P. Making Java Work for Microcontroller Applications. **IEEE Design and Test**, Los Alamitos, CA, USA, v.18, n.5, p.100–110, Sept./Oct. 2001.

JAVALAYER. **Java MP3 Player**. 2007. Available at: <http://www.javazoom.net/javalayer/sources.html>. Visited on: September 2007.

JONES, R.; LINS, R. D. **Garbage Collection**: algorithms for automatic dynamic memory management. Chichester: John Wiley, 1996.

KANDEMIR, M.; VIJAYKRISHNAN, N.; IRWIN, M.; YE, W. Influence of compiler optimizations on system power. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, Los Alamitos, CA, USA, v.9, n.6, p.801–804, Dec. 2001.

KAZI, I. H.; CHEN, H. H.; STANLEY, B.; LILJA, D. J. Techniques for obtaining high performance in Java programs. **ACM Computing Surveys**, New York, NY, USA, v.32, n.3, p.213–240, Sept. 2000.

KIM, J.-S.; HSU, Y. Memory system behavior of Java programs: methodology and analysis. **ACM SIGMETRICS Performance Evaluation Review**, New York, NY, USA, v.28, n.1, p.264–274, June 2000.

KISTLER, T.; FRANZ, M. Continuous program optimization: a case study. **ACM Trans. Program. Lang. Syst.**, New York, NY, USA, v.25, n.4, p.500–548, 2003.

LANFER, C.; BALLACO, S. **The Embedded Software Strategic Market Intelligence**: java in embedded systems. 2003. Available at: <http://www.vdc-corp.com/embedded/white/03/03esdtvol4.pdf>. Visited on: September 2007.

LAWTON, G. Moving Java into Mobile Phones. **Computer**, Los Alamitos, CA, USA, v.35, n.6, p.17–20, 2002.

LEE, E. What's ahead for embedded software? **Computer**, Los Alamitos, CA, USA, v.33, n.9, p.18–26, 2000.

LEE, H. B.; ZORN, B. G. BIT: a tool for instrumenting java bytecodes. In: USENIX SYMPOSIUM ON INTERNET TECHNOLOGIES AND SYSTEMS, USITS, 1997, Monterey, California. **Proceedings...** Berkeley: USENIX Association, 1997. p.7–17.

LIN, C.-M.; CHEN, T.-F. Dynamic memory management for real-time embedded Java chips. In: INTERNATIONAL CONFERENCE ON REAL-TIME COMPUTING SYSTEMS AND APPLICATIONS, RTCSA, 17., 2000, Cheju Island, South Korea. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 2000. p.49–56.

LINDHOLM, T.; YELLIN, F. **The Java Virtual Machine Specification**. Reading: Prentice Hall, 1999.

LOCKE, C.; DIBBLE, P. Java technology comes to real-time applications. **Proceedings of the IEEE**, [S.l.], v.91, n.7, p.1105–1113, July 2003.

LUN, M. P.; FONG, A.; HAU, G. K. W. Object-oriented processor requirements with instruction analysis of Java programs. **SIGARCH Computer Architecture News**, New York, v.31, n.5, p.10–15, 2003.

MARTELLO, S.; TOTH, P. **Knapsack Problems**: algorithms and computer implementations. New York: John Wiley & Sons, 1990.

MATTOS, J. C. B.; BECK FILHO, A. C.; CARRO, L. Object-Oriented Reconfiguration. In: IEEE/IFIP INTERNATIONAL WORKSHOP ON RAPID SYSTEM PROTOTYPING, RSP, 18., 2006, Porto Alegre, Brazil. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 2006. p.69–72.

MATTOS, J. C. B.; BECK FILHO, A. C.; CARRO, L.; WAGNER, F. R. Design Space Exploration with Automatic Generation of IP-Based Embedded Software. In: INTERNATIONAL WORKSHOP ON SYSTEMS, ARCHITECTURES, MODELING, AND SIMULATION, SAMOS, 2004, Samos, Greece. **Proceedings...** Berlin: Springer-Verlag, 2004. p.303–312. (Lecture Notes in Computer Science, v.3133).

MATTOS, J. C. B.; BRISOLARA, L.; HENTSCHKE, R.; CARRO, L.; WAGNER, F. R. Design Space Exploration with Automatic Generation of IP-Based Embedded Software. In: IFIP WORKING CONFERENCE ON DISTRIBUTED AND PARALLEL EMBEDDED SYSTEMS, DIPES, 2004, Toulouse, France. **Proceedings...** Boston: Kluwer Academic Publishers, 2004. p.237–246.

MATTOS, J. C. B.; CARRO, L. **Design Space Exploration in the Use of Object Oriented Software for Embedded System Applications**. 2006. Poster Presentation in EDAA PhD Forum at Design, Automation and Test in Europe, DATE, Munich, Germany.

MATTOS, J. C. B.; CARRO, L. Object and Method Exploration for Embedded Systems Applications. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 20., 2007, Rio de Janeiro, Brazil. **Proceedings...** New York: ACM Press, 2007. p.318–323.

MATTOS, J. C. B.; SPECHT, E.; NEVES, B.; CARRO, L. Object Orientation Problems when Applied to the Embedded Systems Domain. In: INTERNATIONAL EMBEDDED SYSTEMS SYMPOSIUM, IESS, 2005, Manaus, Brazil. **Proceedings...** New York: Springer, 2005. p.147–156.

MATTOS, J. C. B.; SPECHT, E.; NEVES, B.; CARRO, L. Making Object Oriented Efficient for Embedded System Applications. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 18., 2005, Florianópolis, Brazil. **Proceedings...** New York: ACM Press, 2005. p.104–109.

MATTOS, J. C. B.; WONG, S.; CARRO, L. The Molen FemtoJava Engine. In: IEEE INTERNATIONAL CONFERENCE ON APPLICATION-SPECIFIC SYSTEMS, ARCHITECTURES AND PROCESSORS, ASAP, 17., 2006, Steamboat Springs, Colorado. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 2006. p.19–22.

MCGHAN, H.; O'CONNOR, M. PicoJava: a direct execution engine for java bytecode. **Computer**, Los Alamitos, CA, USA, v.31, n.10, p.22–30, Oct. 1998.

MULCHANDANI, D. Java for Embedded Systems. **IEEE Internet Computing**, Piscataway, NJ, USA, v.2, n.3, p.30–39, 1998.

NAZOMI COMMUNICATIONS. **Nazomi Multimedia Application Processor**. 2007. Available at: <http://www.nazomi.com/images/ja108_pb.pdf>. Visited on: September 2007.

NEVES, B. S. **Gerência dinâmica de memória em aplicações Java embarcadas**. 2005. Dissertação (Mestrado em Ciência da Computação) — Programa de Pós-Graduação em Computação, UFRGS, Porto Alegre.

NOKIA. **Nokia Home Page**. 2007. Available at: <http://www.nokia.com>. Visited on: September 2007.

OMONDI, A. R. **Computer Arithmetic Systems**: algorithms, architecture and implementation. Upper Saddle River: Prentice Hall, 1994.

PANDA, P. R.; CATTHOOR, F.; DUTT, N. D.; DANCKAERT, K.; BROCKMEYER, E.; KULKARNI, C.; VANDERCAPPELLE, A.; KJELDSBERG, P. G. Data and memory optimization techniques for embedded systems. **ACM Transactions on Design Automation of Electronic Systems (TODAES)**, New York, NY, USA, v.6, n.2, p.149–206, Apr. 2001.

PEYMANDOUST, A.; MICHELI, G. D.; SIMUNIC, T. Complex library mapping for embedded software using symbolic algebra. In: DESIGN AUTOMATION CONFERENCE, DAC, 39., 2002, New Orleans, Louisiana. **Proceedings...** New York: ACM, 2002. p.325–330.

PFEFFER, M.; UNGERER, T.; FUHRMANN, S.; KREUZINGER, J.; BRINKSCHULTE, U. Real-Time Garbage Collection for a Multithreaded Java Microcontroller. **Real-Time Systems**, Norwell, MA, v.26, n.1, p.89–106, 2004.

PILON, A. **Pacman Silver Edition**. 2007. Available at: <http://www.netconplus.com/antstuff/pacman.php>. Visited on: September 2007.

RADHAKRISHNAN, R.; RUBIO, J.; JOHN, L. K. Characterization of Java Applications at Bytecode and Ultra-SPARC Machine Code Levels. In: IEEE INTERNATIONAL CONFERENCE ON COMPUTER DESIGN, ICCD, 1999, Austin, Texas. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 1999. p.281–284.

REYNERI, L. M.; CUCINOTTA, F.; SERRA, A.; LAVAGNO, L. A hardware/software co-design flow and IP library based on simulink. In: DESIGN AUTOMATION CONFERENCE, DAC, 38., 2001, Las Vegas, Nevada. **Proceedings...** New York: ACM, 2001. p.593–598.

RITZAU, T. Hard Real-Time Reference Counting without External Fragmentation. In: JAVA OPTIMIZATION STRATEGIES FOR EMBEDDED SYSTEMS, JOSES, 2001, Genoa, Italy. **Proceedings...** [S.l.: s.n.], 2001.

SALOMONSEN, K. **Design and Implementation of an MPEG/Audio Layer III Bitstream Processor**. 1997. Master Thesis — Aalborg University, Denmark.

SANGIOVANNI VINCENTELLI, A.; MARTIN, G. Platform-based design and software design methodology for embedded systems. **IEEE Design & Test of Computers**, Los Alamitos, CA, USA, v.18, n.6, p.23–33, Nov./Dez. 2001.

SHAHAM, R.; KOLODNER, E. K.; SAGIV, M. Heap profiling for space-efficient Java. In: ACM SIGPLAN CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION, PLDI, 2001, Snowbird, Utah, USA. **Proceedings...** New York: ACM, 2001. p.104–113.

SHANDLE, J.; MARTIN, G. **Making Embedded Software reusable for SoCs**. 2001. Available at: <http://www.eetimes.com/news/design/features//showArticle.jhtml?articleID=16504598>. Visited on: September 2007.

SHUF, Y.; GUPTA, M.; FRANKE, H.; APPEL, A.; SINGH, J. P. Creating and preserving locality of java applications at allocation and garbage collection times. In: ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS, OOPSLA, 17., 2002, Seattle, Washington, USA. **Proceedings...** New York: ACM, 2002. p.13–25.

SMITH, C. U.; WILLIAMS, L. G. Software performance engineering. In: LAVAGNO, L.; MARTIN, G.; SELIC, B. V. (Ed.). **UML for real**: design of embedded real-time systems. Norwell, MA, USA: Klumer Academic Publishers, 2003. p.343–365.

SOMMERVILLE, I. **Software Engineering**. Reading: Addison Wesley, 2000.

SRISA-AN, W.; LO, C.-T.; CHANG, J.-M. Active memory processor: a hardware garbage collector for real-time java embedded devices. **Transactions on Mobile Computing**, Los Alamitos, CA, USA, v.2, n.2, p.89–101, 2003.

STROM, O.; SVARSTAD, K.; AAS, E. J. On the Utilization of Java Technology in Embedded Systems. **Design Automation for Embedded Systems**, New York, v.8, n.1, p.87–106, Mar. 2003.

SUN MICROSYSTEMS. **Java ME Technology**. 2007. Available at: <http://java.sun.com/javame/technology/index.jsp>. Visited on: September 2007.

SUN MICROSYSTEMS. **Java Card Technology**. 2007. Available at: <http://java.sun.com/products/javacard/>. Visited on: September 2007.

SUN MICROSYSTEMS. **White Paper on KVM and the Connected, Limited Device Configuration (CLDC)**. 2007. Available at: <http://java.sun.com/products/cldc/wp/KVMwp.pdf>. Visited on: September 2007.

SUN MICROSYSTEMS. **SymbolTest**. 2007. Available at: <http://java.sun.com/j2se/1.3/docs/guide/awt/demos/symboltest/actual/index.html>. Visited on: September 2007.

SUN MICROSYSTEMS. **Notepad**. 2007. Available at: <http://java.sun.com/j2se/1.3/docs/relnotes/demos.html>. Visited on: September 2007.

TAKAHASHI, D. **Java Chips Make a Comeback**. 2001. Red Herring, 2001.

TIWARI, V.; MALIK, S.; WOLFE, A. Power analysis of embedded software: a first step towards software power minimization. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, Los Alamitos, CA, USA, v.2, n.4, p.437–445, Dec. 1994.

WOLF, W. **Computer as Components**: principles of embedded computer systems design. San Francisco: Morgan Kaufmann Publishers, 2001.

WOLF, W.; KANDEMIR, M. Memory system optimization of embedded software. **Proceedings of the IEEE**, Los Alamitos, CA, USA, v.91, n.1, p.165–182, Jan. 2003.

YEN, I.-L.; GOLUGURI, J.; BASTANI, F.; KHAN, L.; LINN, J. A component-based approach for embedded software development. In: IEEE INTERNATIONAL SYMPOSIUM ON OBJECT-ORIENTED REAL-TIME DISTRIBUTED COMPUTING, ISORC, 5., 2002, Washington, DC. **Proceedings. . .** Los Alamitos: IEEE Computer Society Press, 2002. p.402–410.

LIST OF PUBLICATIONS

MATTOS, J. C. B.; CARRO, L. Object and Method Exploration for Embedded Systems Applications. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 20., 2007, Rio de Janeiro, Brazil. **Proceedings...** New York: ACM Press, 2007. p.318–323.

MATTOS, J. C. B.; BECK FILHO, A. C.; CARRO, L. Object-Oriented Reconfiguration. In: IEEE/IFIP INTERNATIONAL WORKSHOP ON RAPID SYSTEM PROTOTYPING, RSP, 18., 2006, Porto Alegre, Brazil. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 2006. p.69–72.

MATTOS, J. C. B.; WONG, S.; CARRO, L. The Molen FemtoJava Engine. In: IEEE INTERNATIONAL CONFERENCE ON APPLICATION-SPECIFIC SYSTEMS, ARCHITECTURES AND PROCESSORS, ASAP, 17., 2006, Steamboat Springs, Colorado. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 2006. p.19–22.

MATTOS, J. C. B.; CARRO, L. **Design Space Exploration in the Use of Object Oriented Software for Embedded System Applications**. 2006. Poster Presentation in EDAA PhD Forum at Design, Automation and Test in Europe, DATE, Munich, Germany.

MATTOS, J. C. B.; SPECHT, E.; NEVES, B.; CARRO, L. Making Object Oriented Efficient for Embedded System Applications. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 18., 2005, Florianópolis, Brazil. **Proceedings...** New York: ACM Press, 2005. p.104–109.

MATTOS, J. C. B.; SPECHT, E.; NEVES, B.; CARRO, L. Object Orientation Problems when Applied to the Embedded Systems Domain. In: INTERNATIONAL EMBEDDED SYSTEMS SYMPOSIUM, IESS, 2005, Manaus, Brazil. **Proceedings...** New York: Springer, 2005. p.147–156.

BECK, A. C.; HENTSCHKE, R.; MATTOS, J. C. B.; REIS, R.; CARRO, L. Fast and Efficient Test Generation for Embedded Stack Processors. In: IEEE LATIN AMERICAN TEST WORKSHOP, LATW, 6., 2005, Salvador, Brazil. **Proceedings...** [S.l.: s.n.], 2005. p.331–336.

BECK, A. C.; MATTOS, J. C. B.; CARRO, L. Applying JAVA on Single-Chip Multiprocessors. In: XI INTERNATIONAL WORKSHOP IBERCHIP, 2005, Salvador, Brazil. **Proceedings...** Iberchip, 2005. p.19–22.

GOMES, V. F.; BECK, A. C.; MATTOS, J. C. B.; BARCELOS, R. H.; CARRO, L. Automatic Generation of an MP3 Player. In: XI INTERNATIONAL WORKSHOP IBERCHIP, 2005, Salvador, Brazil. **Proceedings...** Iberchip, 2005. p.31–34.

MATTOS, J. C. B.; BRISOLARA, L.; HENTSCHKE, R.; CARRO, L.; WAGNER, F. R. Design Space Exploration with Automatic Generation of IP-Based Embedded Software. In: IFIP WORKING CONFERENCE ON DISTRIBUTED AND PARALLEL EMBEDDED SYSTEMS, DIPES, 2004, Toulouse, France. **Proceedings...** Boston: Kluwer Academic Publishers, 2004. p.237–246.

MATTOS, J. C. B.; BECK, A. C.; CARRO, L.; WAGNER, F. R. Design Space Exploration with Automatic Generation of IP-Based Embedded Software. In: INTERNATIONAL WORKSHOP ON SYSTEMS, ARCHITECTURES, MODELING, AND SIMULATION, SAMOS, 2004, Samos, Greece. **Proceedings...** Berlin: Springer-Verlag, 2004. p.303–312. (Lecture Notes in Computer Science - LNCS 3133).

HENTSCHKE, R.; BECK, A. C.; MATTOS, J. C. B.; CARRO, L.; LUBASZEWSKI, M.; REIS, R. Using Genetic Algorithms to Accelerate Automatic Software Generation for Microprocessor Functional Testing. In: IEEE LATIN AMERICAN TEST WORKSHOP, LATW, 5., 2004, Cartagena, Colômbia. **Proceedings...** [S.l.: s.n.], 2004. p.37–42.

BECK, A. C.; MATTOS, J. C. B.; WAGNER, F. R.; CARRO, L. CACO-PS: a general purpose cycle-accurate configurable power simulator. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 16., 2003, São Paulo, Brazil. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 2003. p.349–354.

APPENDIX A EXPLORAÇÃO DO ESPAÇO DE PROJETO DE COMPONENTES DE SW E HW IP BASEADA EM UMA METODOLOGIA ORIENTADA A OBJETOS PARA SISTEMAS EMBARCADOS

Este anexo apresenta um resumo expandido da tese com introdução e motivação ao tema, os objetivos, metodologia e principais contribuições geradas pelo trabalho.

Introdução

Atualmente, o mercado de sistemas embarcados cresce em um ritmo acelerado disponibilizando novos produtos com diferentes aplicações. Estes sistemas encontram-se por toda a parte, como por exemplo, em telefones celulares, carros, vídeo games, entre outros. Nas aplicações embarcadas, os requisitos como desempenho, redução do consumo de energia e tamanho de memória, entre outros, devem ser considerados. Além do mais, a complexidade dos sistemas embarcados está aumentando de maneira considerável.

Os sistemas embarcados são heterogêneos e eles contêm uma grande variedade de algoritmos implementados por diversas tecnologias de software ou de hardware. No passado, as configurações de hardware dominavam a maioria as implementações, enquanto que, atualmente, a maioria das aplicações são implementadas em configurações mistas onde o software constitui a principal parte (BALARIN et al., 1999)(SHANDLE; MARTIN, 2001). Provavelmente no futuro cada vez mais produtos possuirão as suas principais características desenvolvidas em software. Assim, o software está se tornando cada vez mais o principal fator de custo no desenvolvimento de dispositivos embarcados (GRAAF; LORMANS; TOETENEL, 2003; EGGERMONT, 2002).

Mais recentemente o projeto baseado em plataformas (Platform-Based Design) foi introduzido (SANGIOVANNI VINCENTELLI; MARTIN, 2001). Essas plataformas podem ser vistas como um conjunto de recursos e serviços oferecidos para a implementação do sistema, permitindo ao projetista configurar o sistema alvo para melhor suportar uma determinada aplicação. Na realidade, estas metodologias baseiam-se em um hardware único (plataforma) que disponibiliza um conjunto de recursos e serviços que são configurados através da geração de software para esta

plataforma. Porém, o desenvolvimento de software para esta plataforma consome a maior parte do tempo, além do produto gerado por este desenvolvimento impactar diretamente nos requisitos dos sistemas embarcados.

Atualmente, existe uma grande variedade de blocos IP (Intellectual Property) como cores de processadores com diversos estilos de arquitetura, como RISC, DSP, VLIW. Também, existe um crescente número de IP de software que podem ser utilizados no projeto de sistemas embarcados complexos. Assim, com um grande número de soluções de IPs de software e hardware, o projetista possui diversas possibilidades e necessita de metodologias e ferramentas que permitam uma eficiente exploração do espaço de projeto para atingir um curto tempo de projeto devido ao prazo exíguo tempo imposto pelo mercado.

Através dos anos, a codificação de software embarcado foi tradicionalmente desenvolvida em linguagem assembly, devido à existência de tamanho restrito de memória e desempenho limitados (LEE, 2000). As melhores tecnologias de software utilizam grandes quantidades de memória, níveis de abstração, algoritmos elaborados, e outras abordagens que não são diretamente aplicáveis nos sistemas embarcados. Contudo, as capacidades do hardware foram melhoradas, e o mercado demanda produtos mais elaborados, aumentando a complexidade do software. Assim, o uso de melhores metodologias de software é claramente necessário, como por exemplo, a orientação a objetos. No entanto, estas técnicas abstratas de software necessitam um alto custo no domínio embarcado, e o problema do desenvolvimento de software embarcado para o mercado de embarcados persiste.

Uma das principais metodologias de software é o paradigma orientado a objetos (SOMMERVILLE, 2000). Nas últimas décadas, a técnica de orientação a objetos se tornou o paradigma de programação dominante. A orientação a objetos pode ser utilizada desde problemas triviais até problemas complexos. Apesar das vantagens da orientação a objetos, a sua aceitação no mundo de embarcados tem sido lenta. Os projetistas de software embarcado são relutantes em adotar esta metodologia devido a sobrecarga em termos de memória e desempenho (DETLEFS; DOSSER; ZORN, 1994; CHATZIGEORGIOU; STEPHANIDES, 2002; BHAKTHAVATSALAM; EDWARDS, 2002).

Utilizando o paradigma orientado a objetos os desenvolvedores necessitam de uma linguagem orientada a objetos para realizar a implementação. Ao logo de alguns anos atrás, os desenvolvedores de embarcados adotaram a linguagem Java, devido esta tecnologia fornecer uma alta portabilidade e reuso de código para suas aplicações (MULCHANDANI, 1998; LAWTON, 2002). Além, Java possui diversas características como tamanho de código eficiente e menor necessidade de memória em relação às outras linguagens de programação, tornando Java uma alternativa interessante como linguagem para especificar e implementar sistemas embarcados. Assim, os desenvolvedores estão livres para o uso da orientação a objetos e todo o conjunto de vantagens que esta linguagem fornece. Neste caso, deve-se também tratar com os recursos limitados de um sistema embarcado.

Como mencionado, as metodologias de software existentes utilizadas no desen-

volvimento de sistemas embarcados, especificamente software embarcado, não tem sido suficiente para tratar a crescimento da complexidade das novas aplicações. Além disso, estas metodologias são voltadas a resolver apenas os problemas para o desenvolvimento de software tradicional. Desta maneira, este trabalho apresenta uma metodologia para exploração de componentes IP de software e hardware baseados em uma plataforma. Esta metodologia utiliza um software embarcado orientado a objetos para melhorar diferentes tarefas no projeto do sistema.

Objetivos

O principal objetivo da tese é fornecer uma metodologia e um conjunto de ferramentas que permita, ao mesmo tempo, manipular as metodologias tradicionais de desenvolvimento de software (projeto baseado em plataformas, engenharia baseada em componentes, orientação a objetos e linguagem Java) com os diferentes requisitos do projeto para sistemas embarcados (energia, desempenho e tamanho de memória).

A tese introduz uma metodologia para exploração do software orientado a objetos que procura melhorar diferentes pontos no projeto do sistema. A abordagem é dividida em duas principais partes onde a exploração do software embarcado pode ser melhorada.

A primeira parte, chamada de nível de exploração de método, possui como objetivo melhorar a implementação dos métodos (dos algoritmos que implementam os métodos). Esta fase de exploração apresenta um mecanismo de seleção automática de componentes IP de software e de hardware para aplicações embarcadas, que é baseada em uma biblioteca de IPs de software e uma ferramentas de exploração de projeto.

A segunda parte, chamada de nível de exploração de objetos, procura explorar a organização dos objetos e com isso melhorar o gerenciamento dinâmico de memória. Este nível utiliza uma ferramenta de exploração de projeto que permite a seleção automática da melhor organização dos objetos. Esta abordagem está de acordo com as técnicas clássicas de orientação a objetos e com os requisitos físicos dos sistemas embarcados. O objetivo geral desta fase é fornecer um suporte em alto nível que permita ao mesmo tempo otimizar a memória, energia e desempenho dos sistemas embarcados.

Metodologia

Os sistemas embarcados trabalham como diversos requisitos e restrições. Assim, o desenvolvimento do software embarcado difere do desenvolvimento tradicional (desenvolvimento para desktop e aplicações coorporativas) e as técnicas e metodologias disponíveis devem ser adaptadas para manipular as restrições dos sistemas embarcados.

Novas tecnologias (metodologias, linguagens, etc.) apresentam diversos problemas. O principal problema é relacionado ao tempo de aprendizado de novas tecnologias pelos projetistas. Assim, muitas companhias evitam adotar novas metodologias

ou linguagens de programação. Desta maneira, um dos objetivos da abordagem apresentada neste trabalho é evitar alterações no fluxo de projeto tradicional de desenvolvimento de software.

A abordagem proposta utiliza tecnologias utilizadas no dia a dia e a idéia é introduzir um conjunto de ferramentas que permitam a melhoria do software original através da geração de um novo código otimizado. Estas ferramentas devem ser de fácil aprendizado e uso. Também, as ferramentas fazem a exploração do espaço de projeto, permitindo a configuração automática de uma solução de software otimizada para uma aplicação específica de acordo com os requisitos do software embarcado.

A abordagem de software embarcado apresentada é dividida em duas principais partes de exploração. A Figura A.1 mostra o fluxo simplificado de projeto da tese contemplando estas fases.

A primeira fase introduz um mecanismo de seleção automática de componentes de software para aplicações embarcadas, que é baseado em uma biblioteca de software e uma ferramenta de exploração de projeto. A biblioteca de software possui diferentes implementações algorítmicas de diversas rotinas comumente encontradas no domínio de embarcados.

A segunda fase consiste no uso de uma ferramenta de exploração de projeto que permite uma seleção automática da melhor organização dos objetos. Esta ferramenta tenta transformar, de maneira automática, os objetos dinâmicos em objetos estáticos, com o objetivo de reduzir o tempo total de execução da aplicação mantendo o custo (tamanho) de memória o mais baixo possível.

Além disso, estas duas fases (baseada na exploração de métodos e a outra baseada na exploração de objetos) são ortogonais, isto é, a suas execução são independentes. O projetista pode utilizar primeiro a ferramenta de exploração de métodos e após a ferramenta de exploração de objetos, ou vice-versa. Assim, as otimizações provocadas por cada ferramenta são ortogonais, tornando a complexidade da exploração simples.

A plataforma alvo é composta de um conjunto de processadores Java. Estes processadores implementam diferentes versões do processador FemtoJava (ITO; CARRO; JACOBI, 2001).

Contribuições do Trabalho

Durante o desenvolvimento do trabalho foram produzidas algumas contribuições técnicas:

1. Uma metodologia baseada em uma biblioteca de IPs de sofware e um conjunto de núcleos de processadores com o mesmo conjunto de instruções que utilizam uma ferramenta de exploração do espaço de projeto de IPs de SW e HW foi publicada em duas conferências. O primeiro trabalho apresentou os resultados baseados somente na exploração de IPs de SW (MATTOS et al., 2004). O segundo trabalho apresentou a exploração combinada de IPs de SW e HW

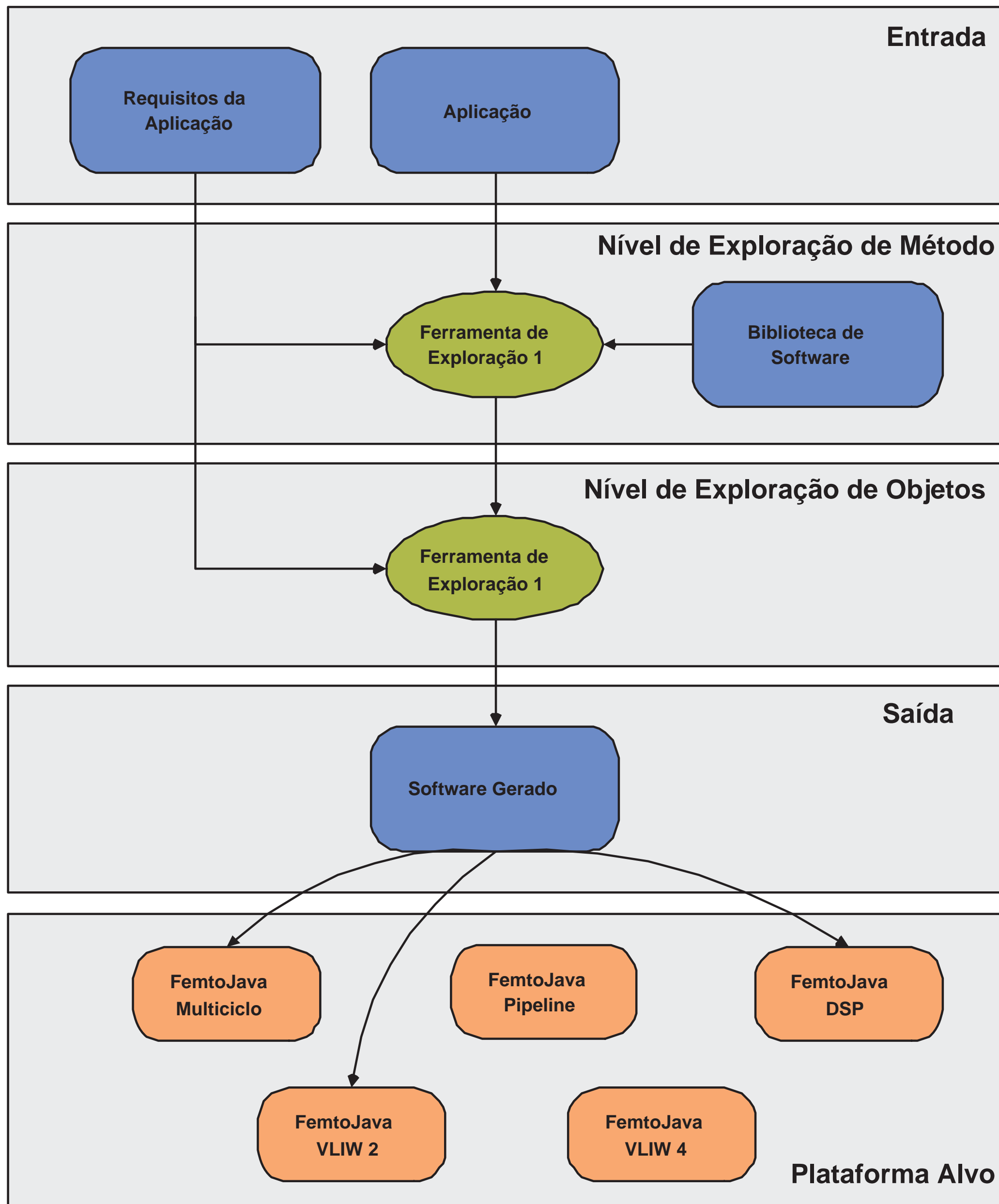

Figure A.1: Fluxo de Projeto da Metodologia.

(MATTOS et al., 2004);

2. A metodologia baseada na exploração da organização dos objetos da aplicação foi publicado em duas conferências. Primeiro, a caracterização do problema foi publicado em (MATTOS et al., 2005b). Após, a metodologia foi avaliada com um exemplo complexo (um tocador de MP3) a estes resultados foram publicados em (MATTOS et al., 2005a);

3. A metodologia completa utilizado as duas abordagens (exploração de métodos e objetos) foi publicada em (MATTOS; CARRO, 2007). Além disso, a metodologia completa foi apresentada no DATE06 EDAA PhD Forum (MATTOS; CARRO, 2006);

4. Durante o desenvolvimento da tese alguns trabalhos relacionados a arquiteturas reconfiguráveis foi investigado. Como resultado do estágio no exterior, um artigo foi publicado em (MATTOS; WONG; CARRO, 2006). Uma metodologia baseada na reconfiguração dos mais criados objetos foi publicado em (MATTOS; BECK FILHO; CARRO, 2006).

APPENDIX B CLASS DIAGRAM - STATIC

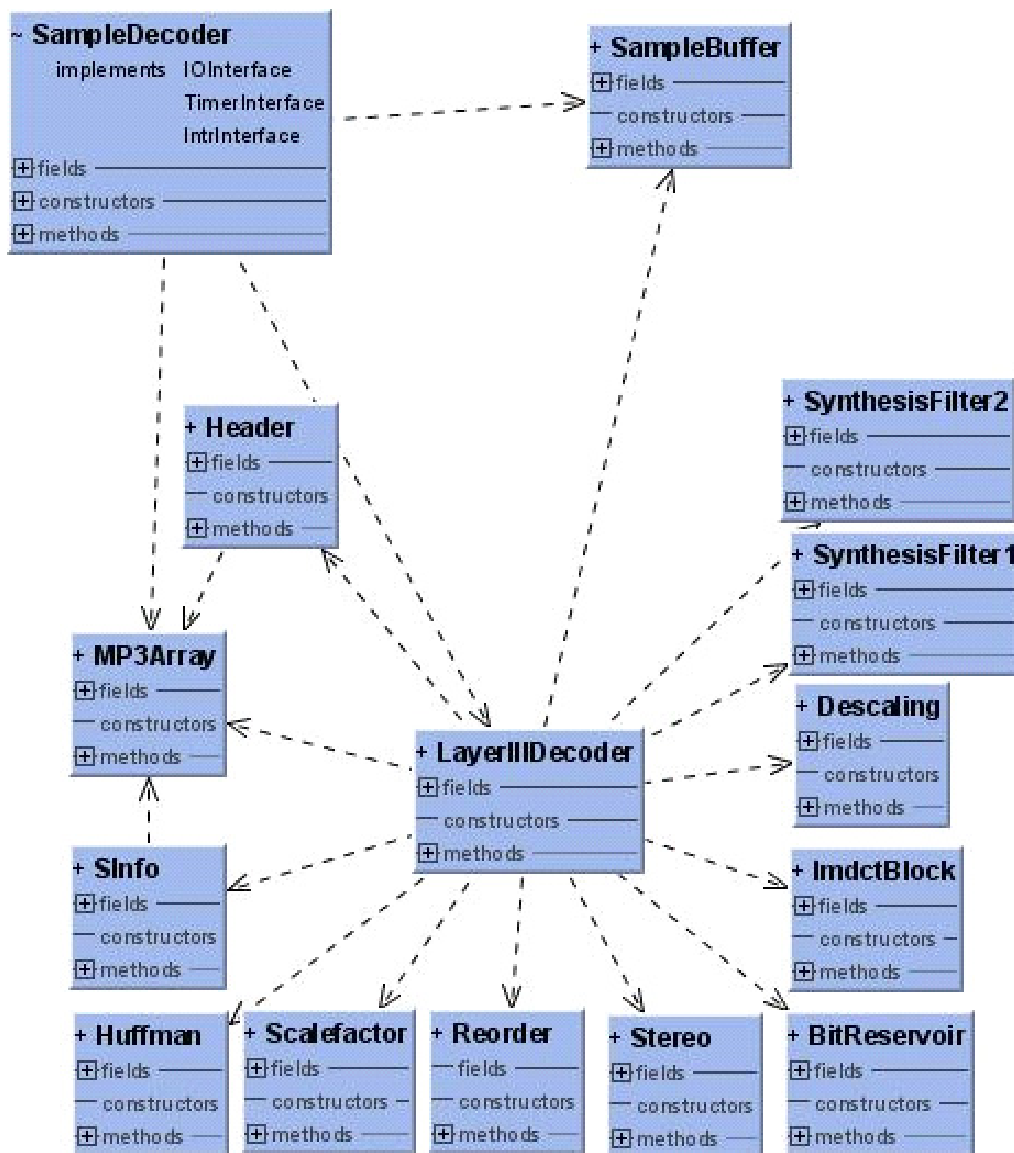

Figure B.1: Class Diagram - Static.

APPENDIX C CLASS DIAGRAM - ORIENTED OBJECT

Figure C.1: Class Diagram - Oriented Object.

APPENDIX D SEQUENCE DIAGRAM - STATIC

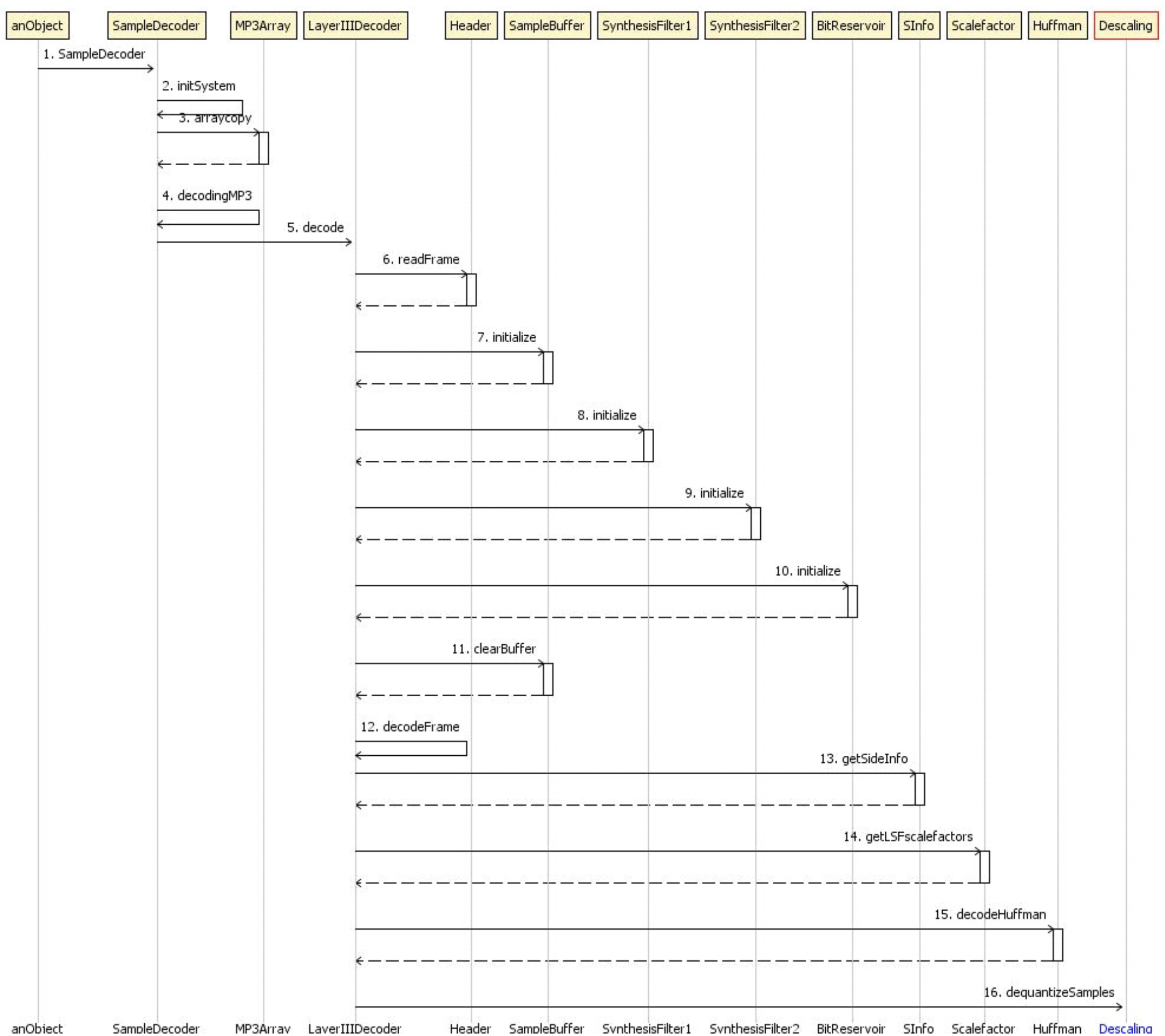

Figure D.1: Sequence Diagram - Static.

APPENDIX E SEQUENCE DIAGRAM - ORIENTED OBJECT

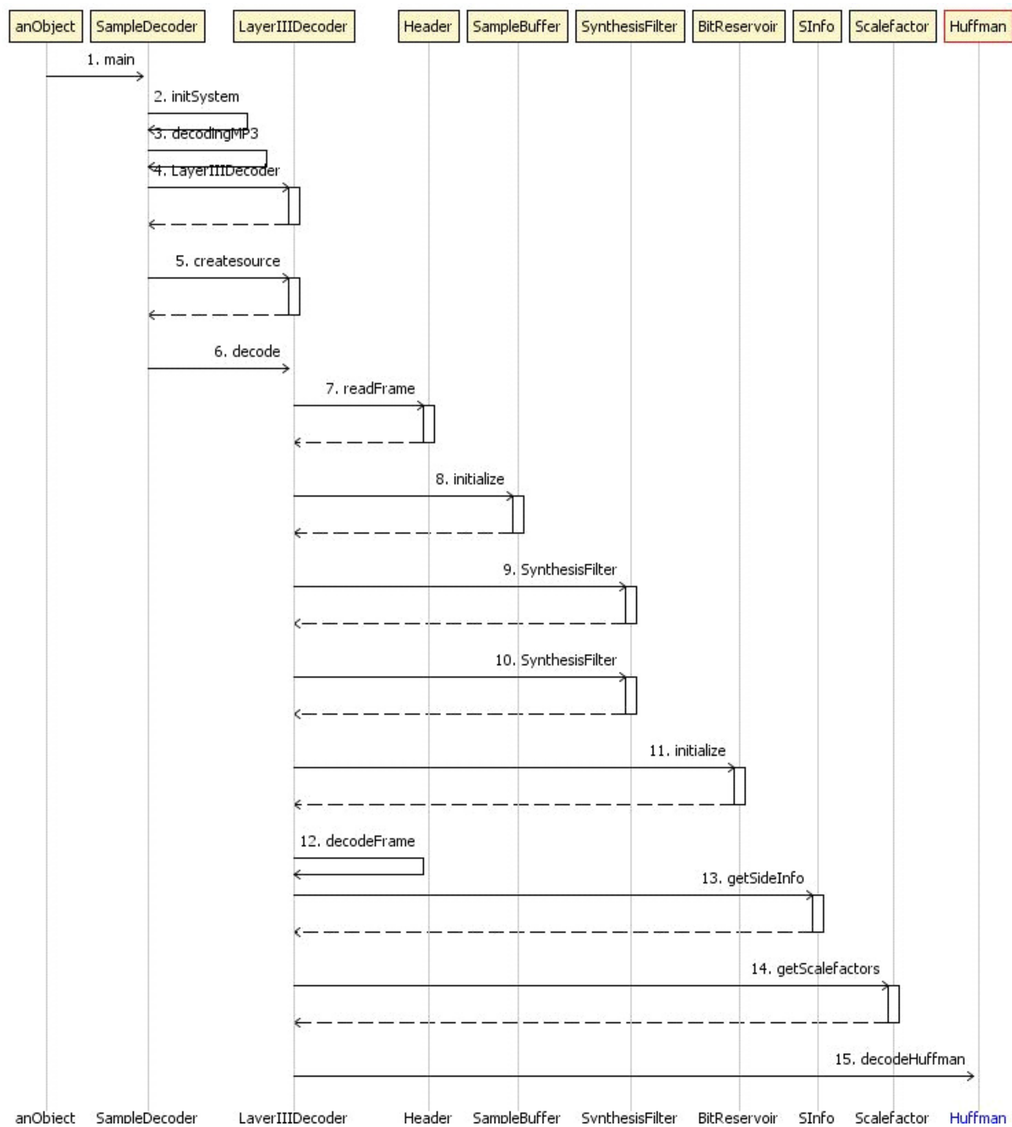

Figure E.1: Sequence Diagram - Oriented Object.

APPENDIX F CDROM DESCRIPTION

The CDROM contains the following directory structure:

- **DESEJOS Tool**: this directory contains the set of a tools developed (the source and binaries codes);

- **Library Characterization**: contains the source code and the results of the library characterization;

- **MP3 Case Study**: this directory contains the source code of different versions of the MP3 player application and all of the results concerning method and object exploration;

- **Publications**: contains the PDF files of the author's publications;

- **Thesis**: contains the PDF file of the thesis and the PowerPoint presentation.