OBJECT ORIENTATION PROBLEMS WHEN APPLIED TO THE EMBEDDED SYSTEMS DOMAIN

Júlio C. B. Mattos[1], Emilena Specht[1], Bruno Neves[1], Luigi Carro[1,2]
[1] *Federal University of Rio Grande do Sul, Informatics Institute, Av. Bento Gonçalves, 9500 - Campus do Vale - Porto Alegre, Brasil*
[2] *Federal University of Rio Grande do Sul, Electrical Engineering Dept. Av. Oswaldo Aranha 103 - Porto Alegre, Brasil*

Abstract: Software is more and more 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 in the same time increase the software productivity and can manipulate the embedded systems constraints like memory footprint, real-time behavior, power dissipation and so on. Object-oriented modeling and design is a widely-know methodology in software engineering. This paradigm may satisfy the software portability and maintainability requirements, but it presents an overhead in terms of memory, performance and code size. This paper presents some experimental results that shown that, for some OO applications, more than 50% of the execution time is taken just for the memory management. This is a huge overhead that cannot be paid by many embedded systems. This way, this paper shows experimental results and indicates a solution of this problem in order to reduce execution time, while maintaining memory costs as low as possible.

Keywords: Embedded Systems, Embedded Software, Object Oriented

1. INTRODUCTION

The fast technological development in the last decades exposed a new reality: the widespread use of embedded systems. Nowadays, one can find these systems everywhere, in consumer electronics, entertainment, communication systems and so on. In embedded applications, requirements

like performance, reduced power consumption and program size, among others, must be considered. Many of these products today contain software and probably in the future even more products will contain 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 [1,2].

Over the years, embedded software coding is traditionally developed in assembly language, since there are stringent memory and performance limitations. [3]. The best software technologies use large amounts of memory, layers of abstraction, elaborate algorithms, and other approaches that are not directly applicable. However, the hardware capabilities have improved, and the market demands more elaborate products, increasing software complexity. But, the existing software methodologies are not adequate for embedded software development. This development is very different from the one used in the desktop environment. Embedded software development should address constraints like memory footprint, real-time behavior, power dissipation and so on. Thus, it is necessary to adapt the available techniques and methodologies, or to create novel approaches that can manipulate the embedded systems constraints.

Object-oriented modeling and design is a widely-know methodology in software engineering. Object oriented analysis and design models include various modeling abstractions and concepts such as objects, polymorphism, and inheritance to model system structure and behavior. 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. Nevertheless, the object-oriented design paradigm presents an overhead in terms of memory, performance and code size [4].

Using object-oriented design the developers need an object-oriented language to do the implementation. Over the past few years the developers have embraced Java, because this technology can provide high portability and code reuse for their applications [5,6]. 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 provide. In any case, one must also deal with the limited resources of an embedded.

The Java language deallocates objects by using garbage collection [7]. Garbage collectors have advantages freeing programmers from the need to deallocate the objects when these objects lost their reference, and helping to

avoid memory leaks. However, garbage collectors produce an overhead in program execution and a non-deterministic execution.

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.

In this way, this work proposes a pragmatic approach, transforming, as many dynamic objects to static ones, in the goal to reduce execution time, while maintaining memory costs as low as possible. This change should deal with the objects after the programmer has coded the application, and before the execution – when the memory management and garbage collector act. This approach is also compliant with classical OO techniques and embedded systems requirements.

This paper is organized as follows. Section 2 discusses related work in the field of OO and procedural programming comparison and techniques to improve the memory management system. Section 3 describes some Java Object-Oriented applications analysis to make the problem characterization. Section 4 presents our proposed approach based on a case study. Finally, section 6 draws conclusions and introduces future work.

2. RELATED WORK

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, to be able real-time applications and so on. In [8] a hardware mechanism (co-processor) to support the runtime memory management providing real-time capability for embedded Java devices is presented. This approach guarantee predictable memory allocation time. Chen [9] presents management strategies to reduce heap footprint of embedded Java applications that execute under severe memory constraints and a new garbage collector. Another work [10] of the same author focuses on tuning the GC to reduce energy consumption in multibanked memory architecture.

In [4], 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 been shown that oriented objected programming can significantly increase both execution time and power consumption.

Shaham [11] presents a heap-profiling tool for exploring the potential for 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 to improvements in program runtime. This approach is based on three code rewriting techniques: assigning the null value to a reference that is no longer in use, remove code that has no effect on the result of the program and delay the allocation of an object until its first use.

Our proposed approach starts from a more radical point of view. Instead of trying just to improve the code written by the programmer, we tries to transform, as many dynamic objects to 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. PROBLEM CHARACTERIZATION

In this work, we have analyzed some Java Object-Oriented applications that may run on embedded systems. 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. The goal of this section is to characterize the exact amount of overhead one has to pay to effectively use the OO paradigm.

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

- MP3Player - is an MP3 decoder implementation. This algorithm reads an MP3 file and translates it in an audio signal. This code is a version based on a description available on [12].
- SymbolTest – is a simple application that displays Unicode char ranges and different fonts types [13].
- Notepad - is a text editor with simple resources [14].
- Address Book - is an application used as electronic address book that stores some data (like name, address, telephone number, etc.) [15].
- Pacman – is the well-known game with a labyrinth and rewards [16].

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.

To generate the application results representing the dynamic behavior of the application, an instrumentation tool was developed. It is based on BIT

(Bytecodes Instrumentation Tool) [17], which is a collection of Java classes that allow the construction of customized tools to instrument Java bytecodes. This instrumentation tool allows the dynamic analysis of Java Class files, generating a list of objects information (objects allocated, object time life, etc.), memory use and performance results.

Table 1 shows some object information 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. The table 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 from the other applications.

Table 1. Object Results

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 2 shows some memory results. Two results are shown: total memory allocated during the application execution and the maximum memory used during the application. Using object-oriented programming the total memory allocated should be larger, because there is an intensive memory use. 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 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.

The table 3 presents the results in terms of performance and the overhead caused by garbage collector making the allocation and deallocation of the objects. 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 [18]. 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 deallocates is about 696 instructions on average.

Table 2. Memory results

Application	Total memory allocated (bytes)	Maximum Memory utilization (bytes)
MP3	10,080,512	23,192
SymbolTest	1,509	625
Notepad	9,199	4,580
AddressBook	867	185
Pacman	216,080	456

The plot in figure 1 shows some statistics about the overhead 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 from figure 1, for some applications the memory allocation needed to support the OO paradigm can represent 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.

It is interesting to notice what happens when the cost of allocation/deallocation is increased. In some application more than 80 % of the execution time is used by memory management system. In the case of the FemtoJava processor, its Garbage collector takes 696 instructions, and the cost of each application can be easily seen to surpass 35% for most applications.

Table 3. Performance results

Application	Performance (instructions)	GC Overhead (%)
MP3	85,767,756	37.40
SymbolTest	67,342	27.91
Notepad	136,621	93.86
AddressBook	24,435	79.84
Pacman	2,091,684	84.85

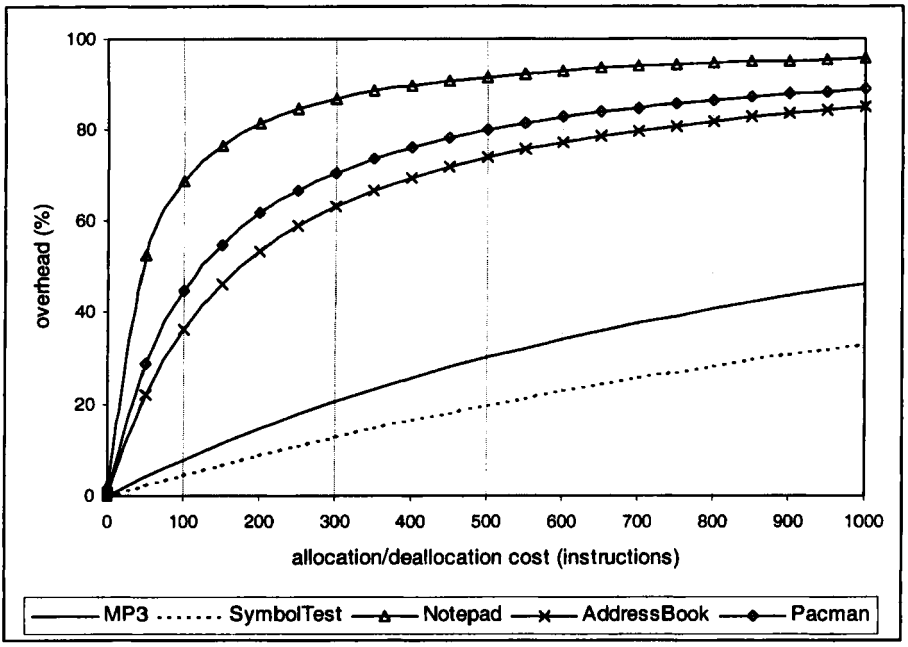

Figure 1. Object-Oriented Overhead

4. THE PROPOSED APPROACH

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 in 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 section 3 have shown that, for some OO applications the largest part of the execution time is taken just for the memory management. However, if the designer allocates memory in a static fashion, the price to be paid is a memory much larger than actually needed, with obvious problems in cost, area and power dissipation.

In section 3, table 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. The main idea of the proposed approach is transforming, as many dynamic objects to static ones.

According the table 3, there are 101 possible objects transformation in MP3 application. Table 4 presents 7 different allocation instructions 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 by the application with the number of allocated objects by the first instruction allocation (first row) shows that just one allocation instruction is responsible by 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 also presents that other allocation instructions can improve the results too. But when a static transformation occurs, this transformation implies in a memory increase.

Table 4. MP3 Allocation instruction

Allocation instruction	Number of allocated objects	Object Size (bytes)
#1	28,800	64
#2	1,728	144
#3	1,728	36
#6	1,600	72
#8	1,536	144
#11	900	2,048
#36	25	4,608

Table 5 shows the results after the static transformations with the same allocation instructions of the table 4. These results show the performance in terms of instruction, the percentage reduction in relationship on 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.

Table 5. MP3 results after static transformation

Allocation instruction	Performance (instructions)	Reduction (%)	Memory Increase (%)
#1	65,636,556	23.47	0.28
#2	84,559,884	1.41	0.62
#3	84,559,890	1.41	0.16
#6	84,649,356	1.30	0.31
#8	84,694,092	1.25	0.62
#11	53,700,828	0,73	8,83
#36	85,750,231	0.02	19.87
#1+#2	53,612,844	24.88	0.90
#1+#2+#11	53,609,244	25.61	9.73

The other allocation instructions present different results in terms of performance gain and memory increase. These values can seem

insignificantly, but these transformations can be grouped taking more advantages. The two last rows show the results of the combination of the allocation instruction 1 and 2, and the combination of the 1, 2 and 11. These combinations show, as example, that grouping different static transformations can be obtained a great number of possibilities with different characterization in terms of performance and memory overhead. Thus, the process of search the best combination according the systems requirements is a hard task.

The table 6 also presents that the total memory allocated can be reduced taking more advantages in terms power saving. Table 2 shows that the MP3 application uses 10,080,512 bytes and when the proposed approach is applied the reduction can be excellent.

Table 6. MP3 total memory allocated results

Allocation instruction	Total memory allocated (bytes)	Reduction (%)
#1	8,237,320	18.28
#2	9,831,880	2.47
#3	10,018,504	0.62
#6	9,965,512	1.14
#8	9,859,524	2.19
#11	8,237,320	18.28
#36	9,965,316	1.14
#1+#2	7,988,688	20.75
#1+#2+#11	6,145,496	39.04

5. CONCLUSIONS AND FUTURE WORK

This paper shows that in the same time oriented-object programming increases the software productivity satisfying the software reusability and maintainability requirements, it presents a critical overhead in terms of performance and memory.

The paper proposes a technique to management this problem, transforming as many as possible dynamic objects to static ones, as reducing execution time, while maintaining memory costs as low as possible. Thus, it provides a large design space exploration for a given application.

As a future work, we plan to implement a tool making possible the transformation of dynamic objects to static ones in automatic way. Furthermore, this tool can be able to allow an automatic selection of the best object organization (combinations of static and dynamically objects) for given application based on systems requirements. We also plan to evaluate the amount of power savings obtained.

REFERENCES

1. Graaf, B., Lormans, M., Toetenel, H. Embedded Software Engineering: The State of the Practice. IEEE Software, (Nov./Dec. 2003), 61-69.
2. Embedded Systems Roadmap 2002. http://www.stw.nl/progress/Esroadmap/ /ESRversion1.pdf.
3. Lee, E. What's Ahead for Embedded Software ?. IEEE Computer, New York, Sept. 2000, 18-26.
4. Chatzigeorgiou, A.; Stephanides, G. Evaluating Performance and Power of Object-Oriented Vs. Procedural Programming in Embedded Processors. In Proceedings of 7th Ada-Europe International Conference on Reliable Sofware Technologies. LNCS 2361. Springer-Verlag Berling Heidelberg, 2002.65-75.
5. Mulchandani, D. Java for Embedded Systems. Internet Computing, 31(10), May 1998, 30–39.
6. Lawton, G. Moving Java into Mobile Phones, IEEE Computer, vol. 35, n. 6, 2002, 17-20.
7. Richard. Jones; Rafael D. Lins. Garbage Collection: algorithms for automatic dynamic memory management. Chichester: John Wiley, 1996.
8. Lin, C.; Chen, T. Dynamic memory management for real-time embedded Java chips. In Proceedings of Seventh International Conference on Real-Time Computing Systems and Applications, 2000.
9. Chen, G. et al. *Heap compression for memory-constrained Java environments.* In Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Anaheim, California (2003), 282-301.
10. Chen, G. et al.Tuning Garbage Collection for Reducing Memory System Energy in an Embedded Java Environment. ACM Transactions on Embedded Computing Systems, vol. 1, n. 1, November 2002, 27–55.
11. Shaham, R.; Kolodner, E.; Sagiv, M. Heap profiling for space-efficient Java. In Proceedings SIGPLAN Conf. on Prog. Lang. Design and Impl., ACM Press, 2001. 104-113.
12. Javalayer. Java MP3 Player. Available at http://www.javazoom.net/javalayer/ /sources.html(2004).
13. Sun Microsystems. SymbolTest. Available at http://java.sun.com/j2se/1.3/docs/ /guide/awt/demos/ symboltest/actual/index.html
14. Sun Microsystems. Notepad. Available at http://java.sun.com/j2se/1.3/docs/ /relnotes/demos.html
15. Brenneman, Todd R. Java Address Book (ver. 1.1.1). Available at www.geocities.com/SiliconValley/2272.
16. Pacman Silver Edition. Available at http://www.netconplus.com/antstuff/pacman.php
17. Lee, H.B.; Zorn, B.G. BIT: A Tool for Instrumenting Java Bytecodes, USITS'97 - USENIX Symposium on Internet Technologies and Systems, Dec. 1997.
18. Ito, S. A.; Carro, L.; Jacobi, R. Making Java Work for Microcontroller Applications, IEEE Design & Test, vol. 18, no. 5, Sep-Oct, pp. 100-110.