Evaluating the Cost of Object Oriented Software in Embedded System Applications

Emilena Specht
emilenas@inf.ufrgs.br

Júlio de Mattos
julius@inf.ufrgs.br

Luigi Carro
carro@inf.ufrgs.br

Instituto de Informática - Departamento de Engenharia Elétrica - UFRGS

Abstract

By the use of popular programming languages such as Java and C++, nowadays embedded systems can be developed in a very short time, by the reuse of legacy code. However, developers should be free to use any object oriented coding style and the whole package of advantages that these languages usually provide. However, one must also deal with the limited resources of an embedded system - small amount of memory and low power processor.

This article studies the cost of dynamic object allocation, compared with the availability of common embedded systems resources. The analysis includes information about the memory used and an estimation of time consumed when objects are created in typical applications. Results show that an overhead ranging from 10 to 50% can be found in different meaningful applications like text editors, graphic programs and MP3 players. This overhead is also related to the cost of the memory allocation procedure.

1. Introduction

Embedded systems are not only required to provide solutions only for simple cases as they used to. Nowadays, with the growing complexity of portable equipment like cell phones, games and MP3 players, developers have to care about memory, time and power, finding solutions that guarantee the correct response of a real-time system without significant extra cost. Object oriented paradigm may satisfy the software portability and maintainability requirements, but its impact on the other attributes involved must be considered.

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

2. Object-Oriented Applications Analysis

Some of these applications used as benchmarks are common in portable devices, although they were profiled in j2sdk1.4.0 environment: MP3Player [1], DCT (algorithm that implements the Discrete Cosine Transform) [2,3], Address Book [4], Notepad (text editor) [5], DrawTest (allows drawing in a box) [5], SymbolTest (displays Unicode char ranges) [5].

It is important to mention that none of the above applications has been code by one of the authors. We made a blind analysis, in order to avoid influence on a particular code style.

The Bytecode Instrumentation Tool (BIT) [6] provided a first view of oriented object paradigm presence in the benchmarks. A class that imports BIT packages divides the application classes in blocks of bytecodes and identifies instructions, counting them as they appear in the source code of each application.

BIT deals with classes of instructions. Comparing instructions types on each application gives an idea of their features. On the other hand, a dynamic analysis is fundamental: it traces an application execution in order to count method calls and instructions in real-time. For this reason, BIT classes also have methods which are called along a class execution, allowing a measurement of memory allocated besides the instructions counting. Table 1 shows the percentage of allocation instructions on the source code and on total executed. As seen in the table, memory related instructions are responsible for a small percentage of the total dynamic instruction count. However, each time the memory allocation procedure is called, a large number of microinstructions might be called, thank to the complex memory model adopted in Java.

BIT was indirectly used to count the size of memory occupied by the application thread as the instructions were executed. At some instant, a maximum size was reached and this number is taken as the amount of memory needed to execute that application. However, this number depends on the specific garbage collection implementation of that virtual machine.

Another tool, **Jprofiler** [7], was used to count the memory occupied by live and garbage collected objects. This represents the amount of memory that would be used without the garbage collector action. Table 1 shows memory occupation statistics calculated on tools information.

Table 1. Percentage of object instructions on total and memory occupation

Application	Allocation instructions		Memory occupation
	Static	Dynamic	
MP3	7.2%	<0.01%	66.34%
DCT	10.6%	0.03%	17.09%
AddBook	15.7%	5.0%	17.65%
Notepad	13.3%	1.0%	41.54%
DrawTest	14.9%	1.9%	19.74%
SymbTest	12.0%	0.04%	26.38%

3. Results

In cases such as Notepad, SymbolTest and DrawTest - the amount of instructions manipulating objects is larger after the start-up phase, while memory occupation remains almost constant during the execution of the whole application. This happens because the major structures are allocated once and stay active until the end of execution. Although there is a very small number of allocation instructions in DCT and SymbolTest, these applications demand a large memory usage. The MP3Player was analysed when playing a music with more than 3 minutes. This caused, in the beginning of execution, the creation of objects for the MP3 standard decoding process. After that, the audio decoding was just changing values from already allocated objects. The AddressBook application takes the information that is wanted to be stored from a simple file and place it in a dynamic hash table. This means that the memory occupation is directly linked with the table number of entries.

Considering that regular instructions take one cycle to be executed, in a pipeline machine like femtoJava [8], the plot in figure 1 shows statistics about the overhead that might be expected by dynamic allocation.

As it can be seen form figure 1, for some application the memory allocation needed to support the OO paradigm can represent more than 50% of the total cycle cost, indicating that the CPU spends more time and energy just managing memory, instead of actually executing the target application.

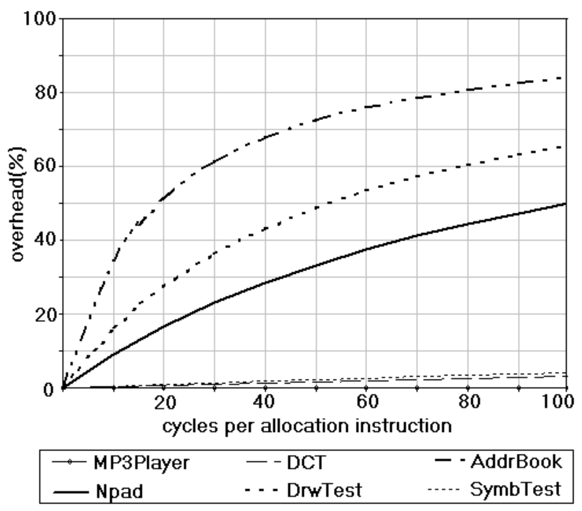

Figure 1. Overhead caused by allocation

4. Conclusions and Future work

As the overhead depends on the number of allocation instructions, it also depends on the application features, the kind of data it manipulates and its use of memory – what takes an allocation instruction more or less expensive. There are cases that an application performance is still better using only static instructions.

The future work includes building a tool able to analyze an application and determine the best way to implement it in an embedded system with specified resources.

References

[1] Javalayer. *Java MP3 Player.* Available at http://www.javazoom.net/javalayer/sources.html(2004).

[2] Salomon, David. *Data Compression: the complete reference.* New York:Springer, 2000.

[3] Agostini, L et al. *Pipeline fast 2-D DCT architecture for image compression.* Symposium on Integrated Circuits and Systems Design, Pirenopolis, IEEE Computer Society, 2001, p.226-231.

[4] Brenneman, Todd R. *Java Address Book* (ver. 1.1.1). Available at www.geocities.com/SiliconValley/2272.

[5] J2SDK1.4.0 demo applications. Available at http://java.sun.com (2004).

[6] Lee, Han Bok; Zorn, Benjamin G. *BIT: A Tool for Instrumenting Java Bytecodes.* USENIX Symposium on Internet Technologies and Systems, Monterey, California, USENIX Association, 1997. p.73-82.

[7] EJ-TECHNOLOGIES. *Jprofiler 3.0.1.* Available at http://www.jprofiler.com (2004).

[8] Ito, Sérgio; Carro, Luigi; Jacobi, Ricardo. *Designing a JAVA Microcontroller to Specific Applications.* XII Brazilian Symposium on Integrated Circuits and Systems Design, IEEE Computer Society, 1999, p.12