Towards Energy-Conscious Class Transformations for Data-Dominant Applications: a Case Study

Marijn Temmerman[§], Edgar G. Daylight[†], Serge Demeyer[*], Francky Catthoor[†], Tom Dhaene[*]

[§]KdG dep. IWT, Salesianenlaan 30, 2660 Hoboken, Belgium.Email:{name.surname}@kdg.be

[*]UA, Middelheimlaan 1, 2020 Antwerpen, Belgium. Email:{name.surname}@ ua.ac.be

[†]IMEC vzw, Kapeldreef 75, 3001 Heverlee, Belgium. Email:{name.surname}@imec.be

***Abstract*— For data-dominant applications running on embedded systems, the energy consumed by the data memory organisation represents a very large cost. We present a method to explore the design space of an object-oriented application in order to optimise the energy consumption and the footprint of the memory. Because we are working at the class design level, we can exploit the functionality of the system and focus on optimising the data related properties of the classes. We demonstrate our technique with a case study in which we achieve a gain of at least 35% in the energy cost factor.**

I. Introduction

Embedded systems have limited hardware resources, such as a small memory capacity, and are often powered by batteries. To meet these non-functional requirements, the hardware aspects of the system must be taken into account at the software design level. The increasing complexity and the decreasing time-to-market of embedded software systems force the designers to raise the abstraction level of the system design language. We focus on dynamic and data-dominant applications, like modern network protocol and multimedia applications. It is known that up to 50% of the energy is consumed by data related memory accesses and transfers [3,4]. The energy consumption is strongly dependent on the memory architecture of the hardware platform and the mapping of the data structures onto the memory devices, according to their sizes and number of memory accesses.

The conventional OO-design methodology leads to small, relatively independent modules or classes. Classes typically encapsulate the data structures where the state of the objects is stored. This results in a dispersed view on the data of the whole application, which makes it quite difficult to globally optimise the memory footprint and the power consumption for the data of the whole design. When designing a new application, the choices to be made for the construction of the classes are nearly endless, creating a large design space. To guide the designer quickly to a cost-efficient implementation, a fast exploration technique at a high abstraction level is needed. To achieve this, we systematically transform an original class design into more optimised ones. The following factors drive the class transformations: (a) we try to reduce the number of data accesses by exploiting the functionality of the application; (b) we try to minimize the memory footprint by merging classes, eliminating the storage of redundant information.

This contribution is organized as follows. First, we discuss the hardware (memory) related aspects. Next, we propose our method. Finally, we introduce the case study to demonstrate our technique and present some results [1].

II. The memory model

The performance gap between the external main memory and the processor has a great impact on the execution speed of the application. However, also a difference exists concerning energy: an off-chip memory access consumes more energy (i.e. two orders of magnitude) than an access to an on-chip memory due to the high capacitive communication bus, and it increases with the memory size. The Cacti memory model [5] produces an energy estimation value consumed for one data access, which is dependent on the technology and the size of the memory. Knowing the access behaviour of the application, we calculate the energy cost (i.e. only for the data) of the application as the multiplication of the energy value from the CACTI model and the number of data accesses. Note that the main memory used in our case study has a 32-bit bus and a size of 512 KB with two subbanks. This gives an energy value of *0.84 nJ* for a data access in a 0.13µ technology.

III. The exploration method

The main characteristics of the domain of applications we want to investigate are: data-dominant, dynamic, non-deterministic and running on an embedded platform. The code transformations [6] must preserve the I/O behaviour of the application. Therefore, the method we propose consists of a number of consecutive steps.

1. Start with the original application, designed conform to the OO-paradigm [1,2]. Complete source code must be available.

2. Generate and store input and output values for the application in realistic situations.

UML diagrams illustrate the design concepts[1]

3. Instrument the application to count and to store the number of data accesses. Calculate the total memory footprint of the objects.

4. Profile the application with the data from step 2.

5. From the results of step 4, calculate the energy cost and locate the bottleneck class.

6. Eliminate unnecessary data accesses, if present.

7. Go through steps 4 and 5. Check the I/O behaviour.

8. Eliminate redundant storage of data, if present.

9. Go through steps 4 and 5. Check the I/O behaviour.

10. Repeat from step 6 until all the involved classes are adapted.

IV. THE CASE STUDY

The design has not to be too complex for a first experiment, so we decide to design a rather simple multimedia game. We select the well-known Tetris game. The main activity of the game consists in the rendering of the board and the piece on the screen.

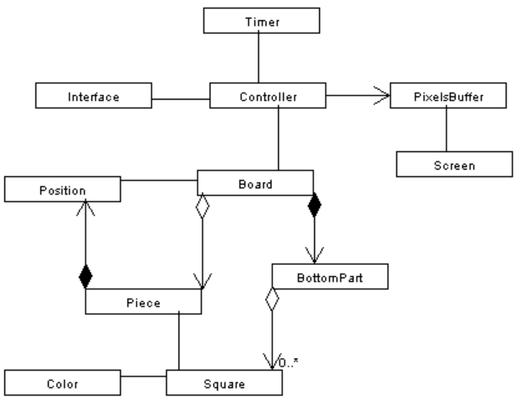
Starting from the requirements, we develop a class model (and implement) it in a pure OO designer style, without any concern about power consumption. Two optimisation steps are worked out: the first one focuses on the reduction of the number of data accesses, the second one on reducing the memory footprint of the largest objects.

The results are obtained with the MinGW port of gcc on a Pentium III at 864 Mhz with 265 MB RAM and running under Windows 2000. We use the Allegro library [7] for the rendering.

A. *The original design (Version1)*

To manage the complexity of the system, we partition the problem domain in modules: the user interface, the logic (intelligence) of the game and the rendering part. A sequential pipe processes the data: after the user input is interpreted by the user interface, the game logic calculates the next state of the game.

Figure 2. The original design



Then the rendering module generates and displays the next frame on the screen. We consider the graphical processing as one of the main functionalities of the application and incorporate it at the conceptual level. We divide the board in two separate parts: an upper part of empty rows (which will not be stored explicitly) and the rest of the board that we call the BottomPart. A Piece is a collection of connected squares. The BottomPart is a collection of non-empty rows (array of Squares). Rows can be added when a piece lands in the bottom part and the colours of the squares are set. Filled rows are removed. We use a dynamic data structure to implement [8] this dynamic behaviour. Depending on the implementation of the BottomPart class, a Square has to store at least its colour (RGB value). The data for the graphical part of the game is stored in the PixelsBuffer. This buffer is a two dimensional array of pixels and captures the information of the next image of the board that will be displayed on the screen. The Screen class represents

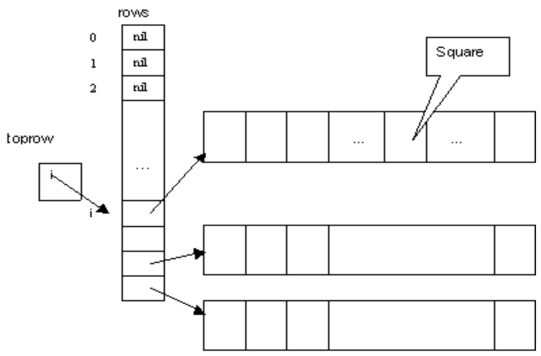


Figure 1. The BottomPart implementation

the video memory of the screen (640x480 pixels).

Note that the pixelsBuffer is cleared and repainted for every frame, due to the separation of the logic and the rendering part (fig. 3).

1) *Profiling results*

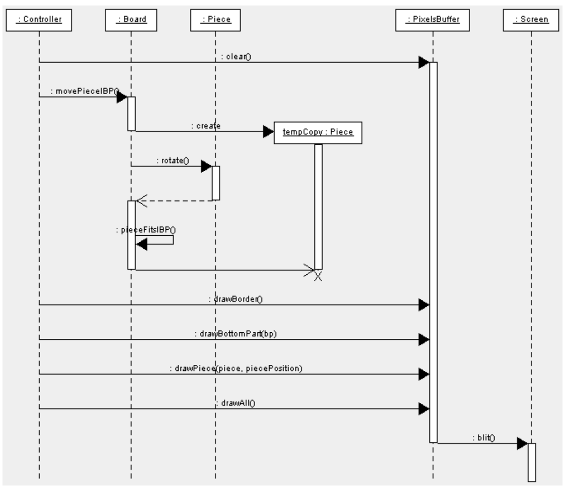
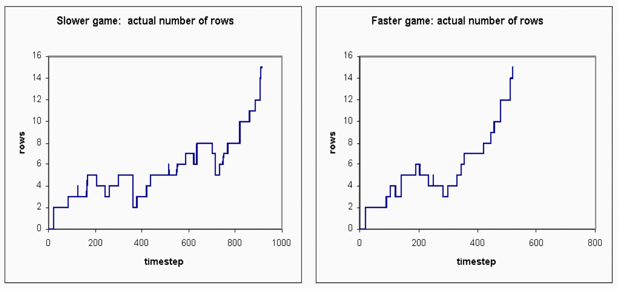


Figure 3. Sequence diagram: Move Piece in BottomPart (v1)

We use two different input sets (fig. 4) for the game: a slow game (input1) and a faster game (input2). We run the application for three different resolutions of the squares (SW10 (10x10 pixels), SW20, SW25). The board has 16 rows and 10 columns.

Fig. 5. demonstrates that the pixelsBuffer and screen objects are the most accessed for all the different test cases so they are the most dominant data types.

Figure 4. Flow of a slower and faster game



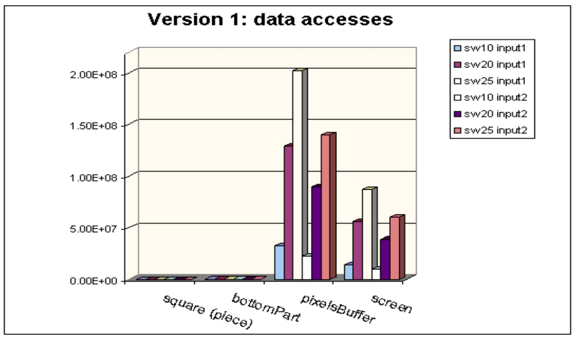


Figure 5. Profiling results for Version1

B. Reducing data accesses: Incremental rendering (Version2)

Because the pixelsBuffer is also an object with a large memory size, we focus the optimisation on the pixelsBuffer by reducing its number of accesses. The data members of the classes are not changed. The memory size of the objects remains the same as for Version1. We adapt the methods of the classes and repaint only those parts of the board that have changed.

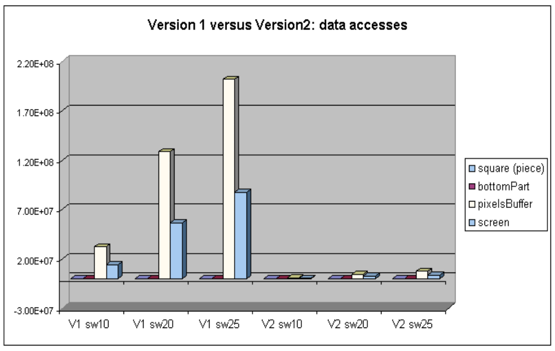


Figure 6 Version1 versus Version2

This results in a large decrease of the number of accesses for all the data structures (Fig. 6).

C. Reducing memory area: Merging classes (Version 3)

The pixelsBuffer object needs the most memory space. In this optimising step we try to reduce the memory area. In Version2, we use a static array for the pixels of the pixelsBuffer. This bitmap holds a copy of the display and is redundant to the information in the video memory. Also the content of the bottomPart object is redundant. We eliminate the PixelsBuffer class and merge its data with Piece and BottomPart.The data members are distributed over the other classes. The methods are moved to the corresponding classes, close to the data. The Piece class gets a small bitmap (size SWxSW pixels) for the rendering. The BottomPart class keeps mainly the same structure as in the earlier versions, but each square in the rows has now its own bitmap with a capacity of SWxSW pixels. These bitmaps are dynamically created and deleted together with the rows on the board. The control of the rendering process is completely removed from the Controller class and distributed over other classes.

1) Profiling results

In comparison to Version2, the total number of data accesses (Table 1.) is reduced to 65%, which clearly demonstrates the large potential impact of our approach.

Data Accesses	Version 2	Version 3	ver3/ver2
piece (Square)	3684	1065312	
toprow	919	981	
rows	1138	1122	
squares	2307	286333	
bottomPart Total	4364	288436	
pixBuf Global	131136	0	
pixBuf UP	4191264	0	
pixBuf BP	328032	0	
pixBuf Total	4650432	0	
screen	2339668	2239732	
blackSquare	0	1014120	
Total	**6998148**	**4607600**	**65.84%**

Table 1. Data accesses Version 2 versus Version 3 (sw20, input 1)

Table 2 shows the values of the energy consumption for the three versions for input1.

Energy (Joule)	sw10 input1	sw20 input1	sw25 input1
Version1	3.89E-02	1.55E-01	2.42E-01
Version2	1.19E-03	5.87E-03	9.55E-03
Version3	7.74E-04	3.86E-03	6.30E-03

Table 2 Energy consumption for input1

REFERENCES

[1] E.Gamma et al., "Design Patterns", Addison-Wesley, 1995.

[2] B.P.Douglass, "Real Time UML", (sec. ed.). Addison-Wesley, 2000.

[3] N.Vijaykrishnan et al., "Evaluating integrated hardware-software optimisations using a unified energy estimation framework", IEEE Transactions on Computers, Vol.52, No.1, pp.59-75, Jan. 2003.

[4] F.Catthoor et al., "Custom Memory Management Methodology -- Exploration of Memory Organisation for Embedded Multimedia System Design", ISBN 0-7923-8288-9, Kluwer Acad. Publ., Boston, 1998.

[5] Shivakumari et al., CACTI 3.0: "An Integrated Cache Timing, Power and Area Model", http://research.compaq.com/wrl/people/jouppi/cacti3.pdf

[6] M.Fowler, "Refactoring", Addison-Wesley, 2000.

[7] The Allegro library; http://www.allegro.cc

[8] E.G. Daylight et al., "Analyzing energy friendly states of dyn. app. in terms of sparse data struct.", Proc. of ISLPED, USA, 2002.