Available online at www.sciencedirect.com
ScienceDirect
Journal of Systems Architecture 53 (2007) 539–549
JOURNAL OF SYSTEMS ARCHITECTURE
www.elsevier.com/locate/sysarc

Optimizing data structures at the modeling level in embedded multimedia

Marijn Temmerman [a,*], Edgar G. Daylight [b], Francky Catthoor [b], Serge Demeyer [c], Tom Dhaene [c]

[a] *Karel de Grote-Hogeschool, Salesianenlaan 30, B-2660 Antwerp, Belgium*
[b] *IMEC, Kapeldreef 75, B-3001 Leuven, Belgium*
[c] *Universiteit Antwerpen, Middelheimlaan 1, B-2020 Antwerp, Belgium*

Received 29 March 2006; received in revised form 30 November 2006; accepted 30 November 2006
Available online 12 January 2007

Abstract

Traditional design techniques for embedded systems apply transformations on the source code to optimize hardware-related cost factors. Unfortunately, such transformations cannot adequately deal with the highly dynamic nature of today's multimedia applications. Therefore, we go one step back in the design process. Starting from a conceptual UML model, we first transform the model before refining it into executable code. This paper presents: various model transformations, an estimation technique for the steering cost parameters, and three case studies that show how our model transformations result in factors improvement in memory footprint and performance with respect to the initial implementation.

Keywords: Embedded systems; UML-model transformation; Data structures; Design-space exploration; Multimedia

1. Introduction

Over the past decade, multimedia design has shifted from an image-based to an object-based approach. While this has drastically improved the software design process, it has often led to less efficient implementations. We believe the reason for this is twofold. First, conceptual software models are built today irrespectively of the hardware platform, as is encouraged in the software community. Second, embedded system specialists typically start their optimization crusade from executable code. This approach results in suboptimal implementations on the embedded platform, because at the source-code level not all the inefficiencies of the modeling level can be removed any more. To bridge this gap, our approach starts with an original conceptual model, applies platform-related transformations to it, and only then descends to the source-code level.

Multimedia applications rely on – often large – data structures that are frequently accessed. The performance and energy consumption of these *data-intensive* applications are, therefore, dominated

* Corresponding author.
E-mail address: marijn.temmerman@kdg.be (M. Temmerman).

1383-7621/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.sysarc.2006.11.008

by the number of data transfers over the memory hierarchy of the platform [1,2]. Each data transfer introduces a delay due to the memory latency. The memory-related energy consumption of a data structure can be estimated as $A \times J$ [3]. A is the overall data access count to the data structure and J, which increases with the capacity of the memory device, is the energy required for one data access.

Awareness of this knowledge at the modeling level enables us to construct *energy-conscious models* for the data-dominant part of the application. These models can then be refined, in the next stage of the design flow, into efficient *concrete data structures*, such as lists or arrays, that are economically accessed and consume a minimum amount of memory space.

We realize our goal by exploiting knowledge of the most likely dynamic behavior of the application at the modeling level (i.e. at design time). Dynamic variation and the modeled characteristics are two separate issues from a pure functional/behavioral viewpoint, but they are not always independent with respect to the cost factors that we consider, i.e. memory footprint and data accesses. The latter are as important as the behavior for the embedded systems designer.

As a first example, Fig. 1 shows three equivalent conceptual models for the well-known game Snake, which is available on many handheld devices. In the initial model (Fig. 1a) that we extracted from the original source code, the snake is modeled as a sequence of *cells* on the board. For an average game, the typical shape of the moving snake is rather smooth and composed of a few segments. Hence, we combine adjacent *cells* into a snake *segment*. This results in the second model (Fig. 1b), in which the snake contains fewer parts. By combining the segments into one object, namely a *broken line* (Fig. 1c), we systematically derive yet another conceptual model for the game Snake.

For each model, multiple refinement possibilities exist at the concrete data structure level, and hence, create a large design space [3,4]. Therefore, we propose an estimation technique for assessing the costs of a given model by counting the number of data accesses and calculating the memory footprint for a representative scenario and two particular concrete data structures.

In all our case studies, we prove that our high-level estimates serve as reliable predictions for the actual implementations with respect to the memory footprint of the objects of the application, the number of data accesses (profiled with Atomium [5]), and the run-time (measured on the Trimedia [6]).

The rest of this paper is structured as follows: in Section 2, we discuss our modeling-level transformations that we express in UML, and explain our estimation technique. Section 3 presents two additional case studies that confirm the applicability of these transformations for our application domain. In Section 4, we discuss related work and conclude in Section 5.

2. Exploration of the design space at the modeling level

Rationale: Exploration of the design space at the modeling level concerning memory footprint and number of data accesses, consists of the systematic search for equivalent models. The run-time instances of equivalent models differ (i) in the set of their primitive attributes, (ii) in the quantity of their composing objects, and (iii) in their access patterns. We reach our goals by exploiting specific geometrical and/or topological characteristics of the

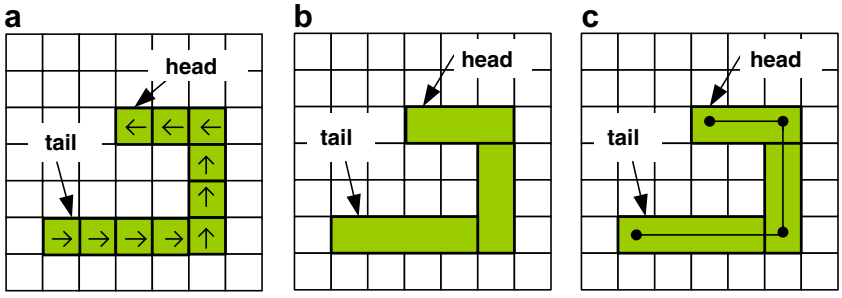


Fig. 1. Three models of the Snake game: (a) a snake modeled with cells, (b) a snake composed of segments, and (c) a snake modeled as a broken line.

objects in the multimedia application, which depend on the user-dependent input values at run-time.

2.1. Transformations at the modeling level

We selected the game Snake to demonstrate our model transformations. This simplified version of the game starts with a short snake that grows and moves, controlled by the player, on a board with a grid of 30 by 30 cells. If the snake collides with itself, the border, or if the length of the snake reaches the limit of 64 snake units, the game ends.

just one dynamic object: the moving and growing snake. All the necessary information is stored in the `SnakeBoard` concept that contains the snake that is modeled as a collection of `Cells`. The attributes `head` and `tail` of `SnakeBoard` indicate the first and the last cell of the snake. The concept `Cell` has three attributes: `position`, `color`, and `next` that is needed to model the dynamic structure of the snake and refers to the next cell in the snake.

Algorithm 1. The *moveSnake* operation: (a) application-specific pseudo-code, and (b) pseudo-code translated into generic operations on collections of objects.

```
boolean moveSnake (Direction dir){
  boolean gameOver = true
  Position p = newHeadPosition(dir)
  snake.moveTail()
  if(snake.checkCollision (p))
    return (gameOver)
  snake.moveHead(p)
  return (not gameOver)
}
```

(a)

```
boolean moveSnake (Direction dir){
  boolean gameOver = true
  Position p = newHeadPosition(dir)
  collection.removePart(tail)
  if(collection.lookUpPart(p))
    return (gameOver)
  collection.insertPart(p)
  return (not gameOver)
}
```

(b)

In Fig. 2, we present three different models of the game in the form of UML-class diagrams: (a) a cell-based model, (b) a segment-based model, and (c) a line-based model. These models conform to the three pictures of the Snake board from Fig. 1. In UML, a composition association [7] with a multiplicity value larger than one (e.g. *), represents a *collection* of objects. Note also that, at the modeling level, we do not consider the details of the concrete data structures that implement these collections.

2.1.1. The cell-based model of the snake game engine

The initial model (Fig. 2a) reflects the concepts as introduced in the rules of the game and is extracted from the original source code. This application has

We present the behavior of the snake in pseudo-code. The fact that only the head and the tail cell need to be moved to give the player the impression that the snake moves, was already exploited in the initial (and thus efficient) algorithm for the `moveSnake()` operation and the rendering process. In Algorithm 1a, the collision detection operation checks whether the head cell, on its future position, will bump into the snake's body. Finding a specific snake unit, given a position on the board, is a dominant data access operation (Algorithm 1b).

2.1.2. The segment-based model of the snake game engine

The shape of the snake is stipulated by the user input. Careful observation of an average game

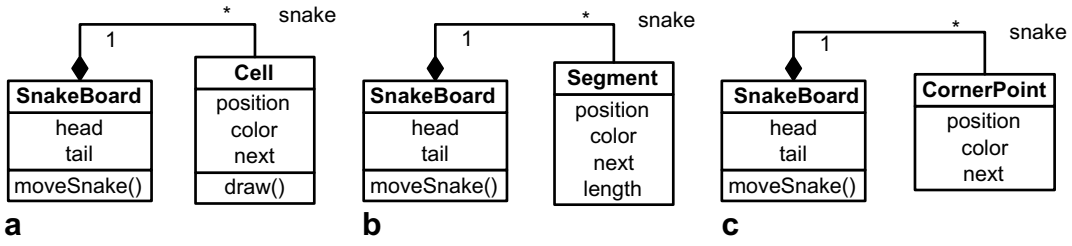


Fig. 2. (a) The UML-class diagram for the snake composed of snake cells, (b) the segment-based model, and (c) the line-based model of the snake.

shows that most of the time, the snake moves in the same direction. Hence, the shape of the snake contains only a few turns. We use the knowledge of this specific dynamic behavior – i.e. we exploit the spatial and temporal locality of adjacent snake cells – in the transformation of the cell-based model to reduce the number of elements in the snake without losing information. To obtain this second model, we compress adjacent `Cells` into one `Segment` object (Figs. 1b and 2b). The attribute `position` of `Segment` refers to the position of the first cell of the segment. The value of `length` equals the length of a `Segment` expressed in `Cell` units. The direction of a `Segment` is incorporated in the attribute `next` that we model relatively to the attribute `position`. That is, the four possible values for `next` are: left, right, up, and down.

2.1.3. The line-based model of the snake game engine

To obtain the third model, we combine the segments into one broken line (Fig. 2c). Here, the snake is modeled as a sequence of the corner points of the line. In this model, the attribute `next` has to be implemented with absolute values in order to find the next corner point in the snake. The line-based snake contains one part object more (i.e. four) compared to the segment-based model (i.e. three), but we have reduced the number of attributes by eliminating `length`.

2.2. Evaluation of the UML models

To compare the different models in terms of memory footprint and number of data accesses, we need information from the lower abstraction level of concrete data structures.

In this section, we explain the concept Pareto optimization and introduce our high-level estimation technique to cope with the complexity at the concrete data structure level.

2.2.1. Pareto optimization

To optimize the energy consumption of a given data-dominant application, we have to find the optimal data structures of the application, trading off memory footprint with data accesses to the memory. Essentially, such a trade-off investigation corresponds to a complex multi-objective optimization process where the goal is to identify so-called *Pareto points* [8].

A Pareto point represents one of the optimal solutions on a single trade-off axis when the other axes are fixed. Thus, in the context of data-dominant applications, for each Pareto point, it is possible to execute the application with less/more data accesses, but at the cost of more/less memory footprint occupied by its data structures.

Typically for embedded platforms, the *more* Pareto points a trade-off investigation identifies and the *larger their range*, the better. This represents all valid and efficient solutions the designer can choose from, out of the huge amount of non-optimal points in the overall search space. The final selection can then either happen at *design-time* or at *run-time* when the context in which the application has to run changes so significantly that it is better/needed to switch to another Pareto operating point. For more detailed information on this principle, we refer to [9] in which a run-time scheduling algorithm is presented.

We show that our model transformations enable us to enlarge the set of identified Pareto points in the objective space with new and non-trivial solutions that could hardly be obtained by solely performing local transformations of the original source code (i.e. conform to the same UML model).

2.2.2. Abstraction from the concrete data structure level

Exhaustive exploration of all the valid implementations of a collection of objects, and this for every transformed UML model, is practically infeasible. We do a light-weight exploration of the design space at the concrete data structure level (i.e. in the traditional way) to discover the optimal implementations for the collections of objects in the models. Therefore, we consider only two data structures, each located at the extreme ends of the Pareto space [3]. On the one hand, we consider a *Sequence* (Fig. 3a) that is characterized by minimal storage (i.e. it stores only the actual elements) and sequential access. On the other hand, we take an *Associative Array* (Fig. 3b) which has the largest (i.e. the worst case) memory footprint but offers direct access to its parts. Fig. 3c shows the Pareto-optimal set of the implementations of a collection of objects in the objective space as a dashed curve and the positions of the data structures *Sequence* and *Associative Array*.

Assumptions: The collections of objects that we deal with in our application domain are based on the following assumptions:

- Every object – we call it a *part* – in a collection is unique and can be identified by one of its attri-

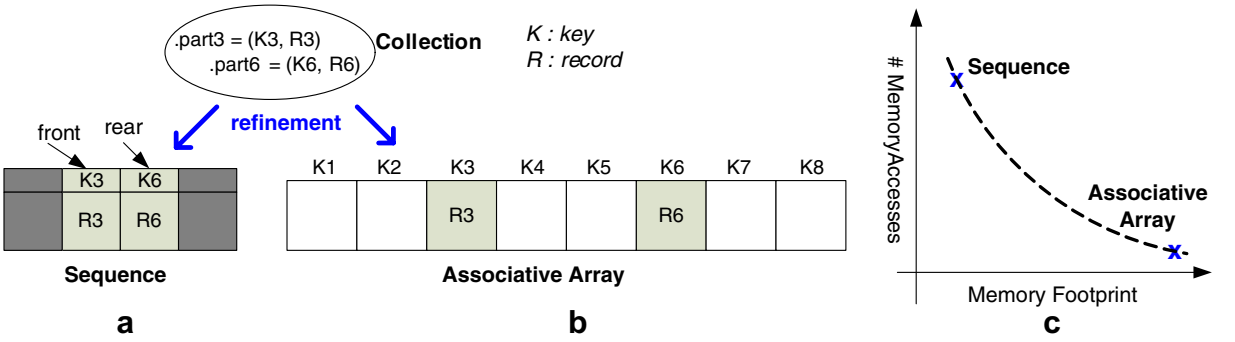


Fig. 3. Two concrete data structures to implement a collection: (a) a Sequence, and (b) an Associative Array; (c) positioning the data structures in the Pareto space.

butes, which we call the *key* (K). The remaining attributes form the *record* (R). Hence, we consider Part as a tuple (K, R).

- The key K is an integer number within a certain interval, and every number in the interval is a valid value. If this is not the case, then we apply an encoding function that translates the original values of the keys into an valid interval of integers. Kmax stands for the maximum number of different keys. We represent the actual number of objects in the collection with #*C*. The *Filling Factor* (*FF*) of a collection is calculated as #C/Kmax.
- FF is a low number (e.g. $0 \leqslant FF \leqslant 0.5$). This is always true in the applications that we target.

Memory footprint and data accesses: Fig. 3 shows a collection of two parts objects (part 3 and part 6) with Kmax = 8. This gives us the value of 0.25 for FF.

Memory footprint: The Sequence implementation of the collection is depicted in Fig. 3a. A sequence is a linear array of parts that contains no fragmentation. *Front* and *rear* are two logical links that point to the first and the last part in the sequence, respectively. Part elements are inserted at the front or rear side. An element is removed by overwriting it with the element at the rear side. The capacity of this array is a function of the actual maximum number of elements at run-time. We denote this value as #Cmax. Hence, the memory footprint (Table 1) of a Sequence equals #Cmax * sizeof(Part). In case of the Associative Array, the key K is implemented as the index of an array of records R (Fig. 3b). This implementation results in a large and sparse array. We calculate the memory footprint of the associative array (Table 1) as follows: Kmax * sizeof(R).

Table 1
High-level estimates for the memory footprint

Associative array	Sequence
Kmax * sizeof(R)	#Cmax * sizeof(K + R)

Data accesses: At the modeling level, we are only interested in the relative comparison of the memory access behavior, caused by the concrete data structures. Therefore, we make abstraction of the memory architecture of the platform by counting as one (high-level) data access to the data structure, each read or write operation to an element in the data structures. In Table 2, the access patterns to a collection of objects are placed in the first column. We consider four different access patterns: insertPart( ), removePart( ), lookUpPart( ), and traverseCollection( ). The next two columns contain the corresponding high-level estimates for the data access count for the Associative Array and the Sequence.

Table 2
High-level estimates for the data accesses

Access patterns	Associative array	Sequence
insertPart(p)	1	1 (front or rear side)
removePart(p)	1	0 (front or rear side) #C/2 + 1 (on average) #C + 1 (worst case)
lookUpPart(p)	1	1 (front or rear side) #C/2 (on average) #C (worst case)
traverseCollection( )	Kmax	#C

2.2.3. The proposed high-level estimation technique

To cope with the dynamic behavior of the application and the complexity at the concrete data-structure level, we propose the following high-level estimation technique:

(1) We tackle the dynamic behavior of the application by investigating a representative and dominant scenario, for which we consider the most likely input sequences. We assume that the maximum number of part objects in a collection at run-time is known in advance.
(2) For each collection of objects in the UML models, we consider two concrete data structures, each situated at the extreme limits of the design space of the concrete data structures: (a) a compact Sequence with sequential access, and (b) a large Associative Array with direct access. If a model contains several collections, then we consider the following two extreme implementations at the data-structure level: (a) the set of all the data structures with the smallest footprint, and (b) the set of all the data structures with the lowest number of data accesses.
(3) For every model, we (a) calculate the memory footprint of the two concrete data structures (Table 1), and (b) count the number of data accesses (Table 2) by simulating this scenario *by hand*, conform to the high-level algorithms of the application and this for the two refinements of each collection. *These numbers give us the high-level estimates for the distribution of the implementations of the various models in the Pareto space.*

2.2.4. Comparison of the snake models

We apply our technique on Snake:

(1) We investigate the *specific scenario depicted in* Fig. 4a. In this scenario, the snake moves one position up on the board and is composed of *ten* cells, *three* segments, or *four* corner points. We consider a snake with maximum 64 cells. The maximum number of segments amounts to 32 segments.
(2) For each collection of objects in the three UML models for Snake, we consider two concrete data structures: (a) a Sequence used as a circular buffer and (b) an Associative Array. We use the attribute `position` as the key to identify an element (i.e. `Cell`, `Segment`, or `CornerPoint`) in its collection. Hence, `position` is the index to access the Associative Array.
(3) The high-level estimates for this scenario are gathered in Fig. 4b for both the Sequence (S) and Array (A) implementations. This graph also shows the Pareto curves of each of the three models: Cell, Segment, and Line model. For each refinement, we present the total size of the snake object after the one movement of Fig. 4a. The calculation of the memory footprint of the snake is based on the size of `tail` and `head`, and the known maximum number of elements (i.e. 64 cells or 32 seg-

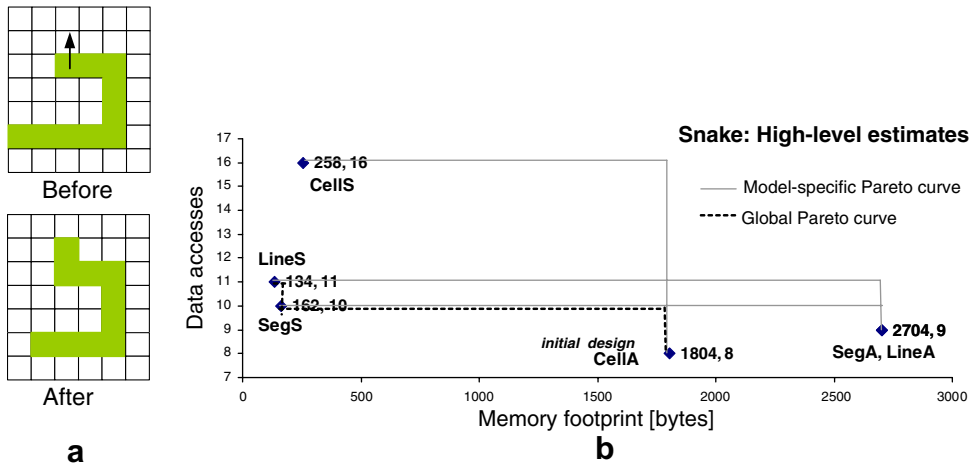


Fig. 4. (a) The specific MoveSnake scenario used to compare the models of the Snake game. (b) Pareto space with the high-level estimates for the three Snake models.

ments) and their size (bytes). The number of data accesses is obtained by counting by hand all the read and write accesses to elements of the snake object conform to this scenario, Algorithm 1, and Table 2.

Conclusion: Without exploration at the modeling level, the CellA and CellS implementations of the initial Cell-based model would be both Pareto-optimal solutions, discovered by local optimization at the concrete data structure level. After combining the local (i.e. model-dependent) Pareto curves (Fig. 4b) into one global Pareto curve for all the models and projecting the high-level estimates for a *complete* game, we expect that the initial CellA implementation remains a Pareto-optimal solution. We predict also two additional Pareto points for the Sequence versions of the Segment and the Line model. Remark that these two Pareto points enlarge the range of identified good solutions. Because of this, the Sequence version of the Cell model can become suboptimal.

2.3. Validation of the high-level estimates for snake

To check the high-level estimates, we implemented five additional versions of the Snake game in the C language: CellS, SegS, SegA, LineS, and LineA. CellA is the original version. Fig. 5 shows the profiling results of one complete, realistic game. We measured the execution time for the game engine on the Trimedia platform [6] and obtained the data accesses to the snake object from the analysis with the ATOMIUM tool [5].

With respect to the number of data accesses, the implementations CellA, SegS, and LineS are indeed Pareto-optimal solutions. The memory footprint of the Snake for LineS is a factor 13 smaller than for CellA. For this gain, LineS requires only 50% more data accesses than CellA. Trade-offs arise also between the memory size of the snake and the computing complexity of the implemented algorithms. Our experiment shows that SegS and LineS require an extra cost in run-time for the extraction of the detailed cell information. In this paper, we focus on the data modeling aspects of the transformations leaving the computing complexity to future work.

Comparing the graphs of Figs. 4b and 5, it is clear that the relative positioning of the high-level estimates is preserved for the six implementations. Hence, this experiment confirms that our high-level estimates serve as reliable predictions for the arrangement of the profiled implementations in the Pareto space with respect to the memory footprint and the number of data accesses.

We are aware that for now, our methodology is quite labor-intensive. As far as we know, only very restricted code generation tools are available. Moreover, their use greatly limits our core work because they do not offer the expressibility that we need. Therefore, we implement the UML models by hand.

3. Additional experiments

Computer game engines are especially representative for the application domain that we target because they are data dominant and user-dependent. Therefore, our second case study concerns the game Tetris. The third case study, 3D-Mesh rendering, is representative for the kind of data type processing present in the graphics rendering pipeline.

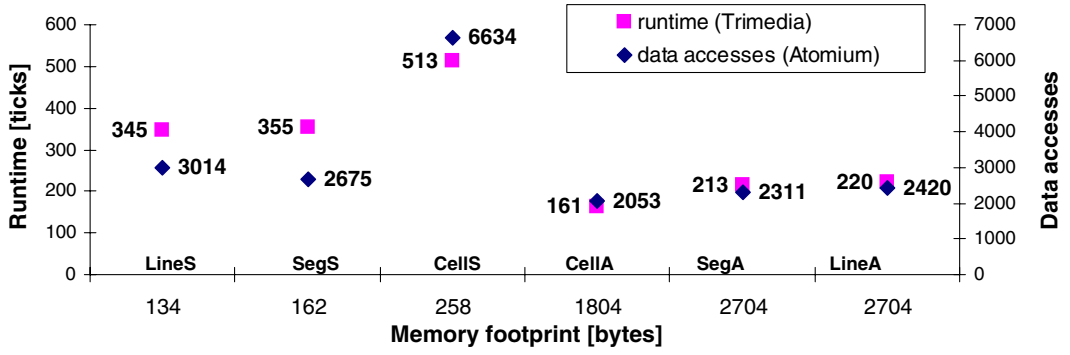


Fig. 5. Profiling results of a complete, realistic Snake game.

3.1. Tetris game

In Tetris, the player controls falling pieces that appear in seven different shapes and colors. The landed pieces are piled up at the bottom of the board and form rows of cells in the pile (Fig. 6a). Full rows are removed and the rows above move down. The game ends when the pile reaches the top of the board.

3.1.1. Model transformations

The original model is the cell-based model. Both the `Piece` and the `Pile` object on the board are modeled as collections of `Cells`. The piece object contains the four falling cells. All the cells in the bottom part of the board belong to the pile. In a similar way as for the Snake game, we transform the cell-based model to the segment-based model by compressing the `Cells` on the same row into a `Segment` and to a third model by compressing the `Cells` into rectangular `Blocks` (Fig. 6c).

3.1.2. High-level estimates and validation

Fig. 7 shows the high-level estimates for the scenario depicted in Fig. 6a when the O-shaped piece moves one position down on the board. The board has 16 columns and 20 rows. The `Piece` object is very small. We present for the `Pile` object the memory footprint and the number of data accesses, for the Sequence and the Associative Array implementations of the three UML models. Besides the Pareto point for CellA (from the initial implementation), these estimates predict a new Pareto point for BlockS.

Here again, the distribution of the profiling results (Fig. 8) in the Pareto space follows our high-level estimates: the profiling of a complete game for the six implementations gives us two Pareto points, both in run-time and data accesses, for CellA and BlockS.

3.2. 3D-mesh rendering

A 3D mesh is traditionally constructed with triangular faces. Scenes with objects such as buildings or furniture, often contain many rectangular surfaces. By exploiting the co-planarity of the vertices at the modeling level, we transform the conceptual model of the mesh.

3.2.1. UML-model transformations

For this experiment we started from an existing software 3D-rendering API, designed for triangular meshes. By adding the class `Quad` to the API, we are able to render meshes which also contain quadrangular faces. Fig. 9b shows the UML-class diagram of the Triangle-based model and the Quad-based model. A hybrid model could contain both types of faces.

3.3. High-level estimates and validation

Both meshes have the same number of vertices, but the Quad-based model needs half the number of faces compared to the original Triangle-based model. The memory size of one face equals 16 bytes for a quadrangle and amounts to 24 bytes for the two composing triangles in the Triangle-based mesh. This results in a gain of 34% for the memory footprint of the list of faces from the Quad Model. The rendering of the mesh requires a traverse of the list of faces and a lookup of the coordinates of each vertex of the face. To display one quad we need five data accesses to the mesh object, compared to eight

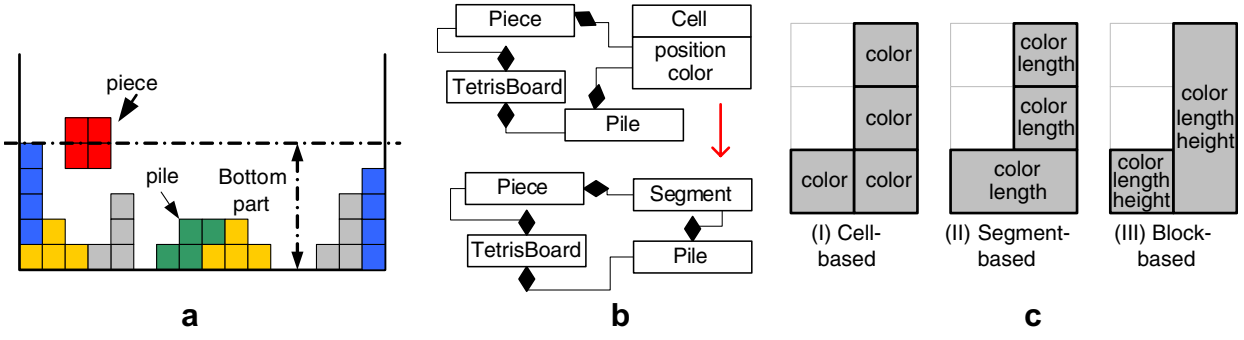


Fig. 6. (a) The scenario for the Tetris game, (b) the transformation of the Cell-based to the Segment-based UML model, and (c) a J-shaped piece modeled with (I) 4 Cells, (II) 3 Segments, and (III) 2 Blocks.

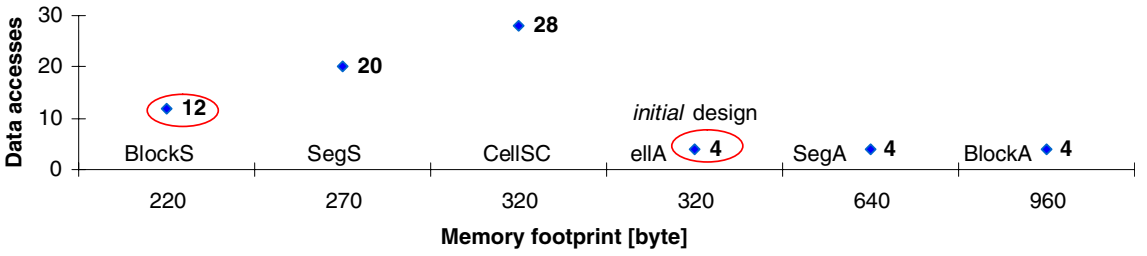


Fig. 7. High-level estimates for the pile object with the scenario of Fig. 6a.

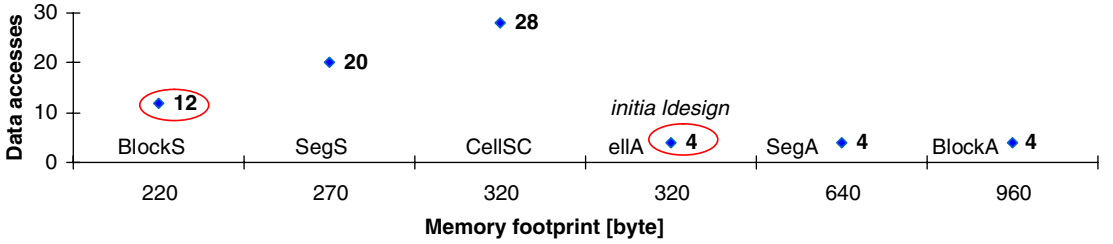


Fig. 8. Profiling results for a complete Tetris game.

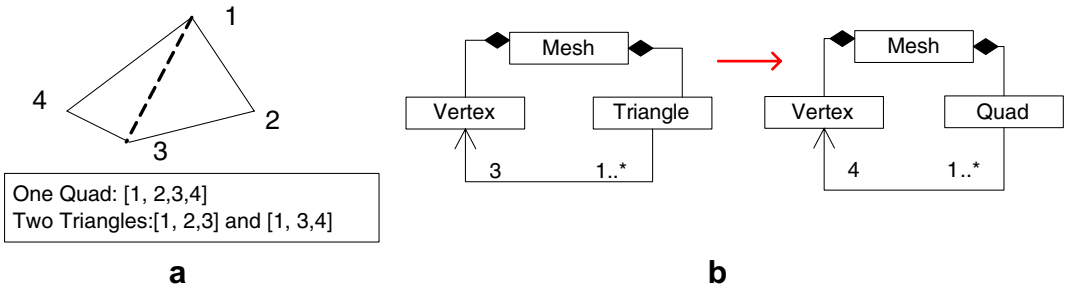


Fig. 9. (a) A planar mesh modeled with two triangular faces, and one quadrilateral face, and (b) the UML-model transformation of the mesh.

data accesses for the two composing triangles (i.e. a reduction of 38% for the Quad Model).

In this API, a mesh is implemented with two related arrays: the list of the vertices with their 3D coordinates, and the list of the faces. A face object is implemented with the references (i.e. indices) to its composing vertices.

For profiling, we created first the data for a mesh composed of 3480 quadrangular faces and generated an equivalent mesh built up with triangles by splitting up each quad from the first mesh into two triangles. The application is implemented in C++ and executed on a Pentium IV platform under Windows XP. Compared to the mesh from the triangle-based model, the Quad mesh requires 78% of the memory footprint for the whole mesh object. The profiled run-time for the rendering equals 481 ms for the triangle-based mesh and 349 ms for the quadrangle-based one (i.e. a gain of 28%).

4. Related work

The work presented in this paper is complementary and closely related to [1,3,4], which all concentrate on the optimization and exploration of concrete data structures in data-dominant embedded applications. Data management for nondynamic applications is covered in detail in the data transfer and storage exploration (DTSE) methodology [1]. A methodology for the systematic exploration and optimization of association tables in telecom network applications is presented in [4]. The use of techniques such as hashing and key splitting/merging/ordering give rise to various

multi-layered implementation alternatives. This leads to a multiplicative combination of many different aspects and hence to a high-degree polynomial design space explosion, with even some (smaller) exponential components. By taking into account the platform-related cost factors and the implementation parameters (e.g. the number of layers in the data structure) of the association table, the design space exploration method is characterized by an optimization problem. Data structure level transformations for multimedia applications are presented in [3]. The application of transformations such as ''implicit versus explicit keys'', marking, and partitioning produces customized and very efficient data structures.

In the design process, we analyze the data of the application at the next higher abstraction level namely the modeling level, which is fully complementary to the lower-level optimizations. With this additional (higher level) exploration, our model-level optimizations will result in a broad range of even better optimized implementations.

Our approach is inspired by work from the software engineering community [10–12], which all discuss behavior-preserving transformations of existing software systems. There are, however, three main differences. First, we concentrate on restructuring applications to map them efficiently onto an (embedded) platform as opposed to restructuring existing applications to improve their internal structure for readability, reuse, or maintenance. Second, we explicitly intend to explore the design space of efficient data models in contrast to the more ''reuse of good OO-practice'' approach. Third, we exploit application-specific knowledge to obtain optimal implementations, which restricts the usage of our transformations to specific application domains, as opposed to the more ''general-purpose'' techniques of software refactoring and re-engineering.

In the embedded systems community, existing research work on the exploration of the design space at the modeling level steered by platform-related cost parameters is quite recent and it is obtained parallel to our work. Moreover, it does not focus on the optimization of the costs related to the data structures like we do. For example, in [13], the mapping of models of computation and models of architectures is investigated. Methods for fast exploration of the design space at multiple levels of abstraction, is the subject of [14].

5. Conclusions and future work

We have presented UML-model transformations for multimedia applications, steered by platform-related parameters, that exploit specific characteristics of the run-time variability at design time. In this way, we obtain new, non-trivial points in a Pareto space in which data accesses are traded off with memory footprint. The proposed high-level estimation technique allows a fast comparison of the various models without having to generate executable code. The transformations have been applied in three representative case studies with experiments that confirm our UML-level estimates.

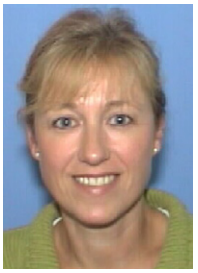
Our transformations are intended for a specific application domain, namely 2D- and 3D-object manipulation and rendering. Therefore, the exploitation of domain-specific knowledge enables us to obtain very efficient solutions. A general approach would, on the contrary, introduce inefficiency because this results in a search space that is possibly too large to represent, and hence, cannot be explored effectively and in a systematic way.

The elaboration of our rather complex UML transformations into a catalog of elementary transformations is very promising and allows us to explore systematically the design space of data structures for the multimedia domain at the modeling level.

References

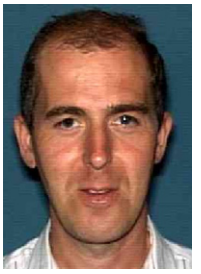
[1] F. Catthoor, E. de Greef, S. Wuytack, Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design, Kluwer Academic Publishers, Norwell, MA, USA, 1998.
[2] N. Vijaykrishnan, M. Kandemir, M.J. Irwin, H.S. Kim, W. Ye, D. Duarte, Evaluating integrated hardware-software optimizations using a unified energy estimation framework, IEEE Trans. Comput. 52 (1) (2003) 59–76.
[3] E.G. Daylight, D. Atienza, A. Vandecappelle, F. Catthoor, J.M. Mendías, Memory-access-aware data structure transformations for embedded software with dynamic data accesses, IEEE Trans. VLSI Syst. 12 (3) (2004) 269–280.
[4] C. Ykman-Couvreur, J. Lambrecht, A. van der Togt, F. Catthoor, H. De Man, System-level exploration of association table implementations in telecom network applications, Trans. Embedded Comput. Syst. 1 (1) (2002) 106–140.
[5] ATOMIUM, 2005. http://www.imec.be/design/atomium.
[6] Philips, TriMedia: TM1000 Preliminary Data Book, Philips Electronics North America Corporation, Sunnyvale, 1997.
[7] J. Rumbaugh, I. Jacobson, G. Booch, The Unified Modeling Language Reference Manual, Addison-Wesley Longman Ltd., Essex, UK, 1999.

[8] V. Pareto, Manual of Political Economy (translation of the 1927 edition), Philips Electronics North America Corporation, New York, 1971.
[9] P. Yang, F. Catthoor, Pareto-optimization-based run-time task scheduling for embedded systems, in: CODES+ISSS'03: Proceedings of the First International Conference on Hardware/Software Codesign and System Synthesis, 2003, pp. 120–125.
[10] S. Demeyer, S. Ducasse, O. Nierstrasz, Object Oriented Reengineering Patterns, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.
[11] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
[12] G. Sunyé, D. Pollet, Y.L. Traon, J.-M. Jézéquel, Refactoring UML models, in: UML'01: Proceedings of the Fourth International Conference on the Unified Modeling Language, Modeling Languages, Concepts, and Tools, 2001, pp. 134–148.
[13] A.D. Pimentel, C. Erbas, S. Polstra, A systematic approach to exploring embedded system architectures at multiple abstraction levels, IEEE Trans. Comput. 55 (2) (2006) 99–112.
[14] V. Reyes, W. Kruijtzer, T. Bautista, G. Alkadi, A. Nunez, A unified system-level modeling and simulation environment for mpsoc design: Mpeg-4 decoder case study, in: DATE '06: Proceedings of the Conference on Design, Automation and Test in Europe, 2006, pp. 474–479.



Marijn Temmerman received the M.Sc. degree in computer science from the Katholieke Universiteit (KU) Leuven, Belgium in 1980. She is currently pursuing the Ph.D. degree at the University of Antwerp. Since 1980, she is a professor at the Karel de Grote-Hogeschool, Antwerp, Belgium (Department of Electronic Engineering). Her research focuses on optimization of data structures for embedded systems and software engineering.

Edgar G. Daylight (a.k.a. Karel van Oudheusden) received the M.Sc. in computer science and his Ph.D. degree from the Katholieke Universiteit Leuven, Belgium, in 2000 and 2006, respectively. His research interests include dynamic memory management for power and performance and formal reasoning about data structures.



Francky Catthoor received the M.Sc. degree and the Ph.D. in electrical engineering from the Katholieke Universiteit Leuven, Belgium in 1982 and 1987, respectively. Since 1987, he has headed research in architectural and system-level synthesis methodologies within the DESICS division at IMEC. His research interests include architecture design methods and system-level exploration for power and memory footprint within real-time constraints.

Serge Demeyer is a professor in the Department of Mathematics and Computer Science at the University of Antwerp (UA) in Belgium where he leads a research group investigating the theme of Software Reengineering (LORE – Lab On REengineering). His main research interest concerns reengineering. Due to historical reasons he maintains a heavy interest in hypermedia systems as well.

Tom Dhaene received his M.Sc. degree in electro-technical engineering, and his Ph.D. degree from the University of Ghent (UGent), Belgium, in 1989 and 1993. Since September 2000, he is a professor at the University of Antwerp, Belgium (Department of Mathematics and Computer Science). He is head of the COMS research group (Computer Modeling and Simulation). His current research focuses on meta-modeling of complex physical systems.