# Memory-Access-Aware Data Structure Transformations for Embedded Software With Dynamic Data Accesses

Edgar G. Daylight, David Atienza, Arnout Vandecappelle, Francky Catthoor, and Jose M. Mendias

***Abstract*—Embedded systems are evolving from traditional, stand-alone devices to devices that participate in Internet activity. The days of simple, manifest embedded software [e.g. a simple finite-impulse response (FIR) algorithm on a digital signal processor DSP)] are over. Complex, nonmanifest code, executed on a variety of embedded platforms in a distributed manner, characterizes next generation embedded software. One dominant niche, which we concentrate on, is embedded, multimedia software. The need is present to map large scale, dynamic, multimedia software onto an embedded system in a systematic and highly optimized manner. The objective of this paper is to introduce high-level, systematically applicable, data structure transformations and to show in detail the practical feasibility of our optimizations on three real-life multimedia case studies. We derive Pareto tradeoff points in terms of accesses versus memory footprint and obtain significant gains in execution time and power consumption with respect to the initial implementation choices. Our approach is a first step to systematically applying high-level data structure transformations in the context of memory-efficient and low-power multimedia systems.**

***Index Terms*—Data structure transformations, dynamic data access, multimedia, power consumption.**

## I. Introduction

DATA structure analysis for dynamic data accesses is mainly done in the computer science community at a high level of abstraction under the implicit assumption that the platform contains one monolithical memory. Exploiting platform related knowledge such as available on-chip and off-chip memory footprint, cache size, and number of SDRAM banks, is mainly done in the system engineering community (i.e., at a lower abstraction level) when the refined data structure has already been chosen. A convergence of these two communities is desirable since this leads to large decreases in cost, as we will show.

Manuscript received February 28, 2003; revised June 21, 2003. This work was funded in part by the Spanish Government Research Grant TIC2002/0750, and in part by the E.C. Marie Curie Fellowship under Contract HPMT-CT-2000-00031.

E. G. Daylight is with the DESICS Division Inter-University Micro-Electronics Center (IMEC) vzw, Kapeldreef 75, B-3001 Heverlee, Belgium, and is also with the Department of Computer Science, Katholieke Universiteit (K.U.) Leuven, Celestijnenlaan 200A, B-3001, Heverlee, Belgium (e-mail: voudheus@imec.be).

D. Atienza and J. M. Mendias are with the Computer Architecture and Automation Department (DACYA) / Universidad Complutense of Madrid (UCM), Facultad de Informatica, 28040 Madrid, Spain.

A. Vandecappelle is with the DESICS Division, Inter-University Micro-Electronics Center (IMEC) vzw, B-3001 Heverlee, Belgium.

F. Catthoor is with the DESICS Division, Inter-University Micro-Electronics Center (IMEC) vzw, B-3001 Heverlee, Belgium, and is also a Professor in the Department of Electrical Engineering, Katholieke Universiteit (K.U.) Leuven, B-3001 Belgium.

Digital Object Identifier 10.1109/TVLSI.2004.824303

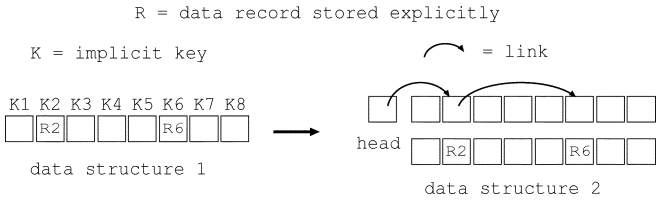



Fig. 1. Adding an array of links on top of an array of records.

In this article, we apply systematic, high-level, data-structure transformations in the context of low power, embedded software design for dynamic, multimedia applications. We present transformations that have a direct and large impact on memory footprint, execution time, and power consumption. Memory footprint and execution time are measured on the TriMedia platform and power consumption is estimated analytically based on an SDRAM memory model.

A simple example of such a high-level data structure transformation is presented in Fig. 1. The initial data structure 1 is transformed into data structure 2 by adding links to data structure 1. These links allow easy traversal through the data structure which usually results in a decrease of the average amount of data accesses. The memory footprint on the other hand has increased due to the additional links. A tradeoff is present between data accesses and memory footprint.

The traversal operation, used in the above example, is an access operation in which all stored records (R) in the data structure need to be consulted. Due to the dynamic behavior of the multimedia application, it is not *a priori* known, which records are stored in the data structure. Traversal is a very dominant access operation in multimedia applications as we demonstrate in all of our case studies. This implies that power consumption can decrease drastically for data structure 2 due to the decrease in number of data accesses, even though the memory footprint has increased.

### A. Power Consumption

The memory-related power consumption (P) of a data structure implementation can be estimated as follows. $P = A\ \text{accesses/frame} \times 20\ \text{nJ} \times 60\ \text{frames/s}$. A is the overall data access count; i.e., the sum of the accesses needed to insert, remove, and traverse records. For one data access, energy of 20

nJ is consumed based on an 8-MB off-chip SDRAM model of $0.18\mu$m-technology [22] and incorporation of bus interconnect. In order to achieve a smooth visual effect for the human eye, 60 frames are output (on the video screen) per second.

Throughout this paper, we informally use the concept of data access to denote an access of (directly related) data whose size is between 1–8 B. For instance, in Fig. 1, an access to a link or to a record (R) in data structure 2, both consume one data access even though the physical accesses may differ. This simplification allows us to compare different data structure implementations, relatively to each other, without having to use a more detailed and, hence, complicated model. It is also realistic for modern SDRAM accesses [34] where never an individual word of a few bytes will be read but always a larger group (e.g. over a wide bus of 64–128 b). It is safe to assume that directly related data will belong to such a single bus access, i.e., when enough spatial locality is present. This assumption is justified if the compiler is SDRAM-aware in the sense that it tries to map an array of records into the least possible amount of pages.

In this paper, we present the memory-related power consumption of different data-structure implementations. We use, for all implementations, the aforementioned 8-MB off-chip SDRAM. Indeed, all our data structures fit into an 8-MB memory. A more customized software/hardware design approach would benefit from the small memory footprint of some of our data structure implementations by using small (on-chip) SRAM memories [1], [24] instead. This will make the savings larger compared to the initial reference. In other words, our obtained reductions in power consumption are only made due to the reduction in data access count.

### B. Application Specific Access Behavior

In our case studies, we distinguish between three access patterns: insertion, removal, and traversal. In addition to this, we realistically assume that when a record is inserted, it is not already present in the data structure. Similarly, when a record is removed, it is indeed the case that the record is *a priori* stored in the data structure. These characteristics are specific for our case studies, and we exploit these characteristics during optimization in order to obtain large gains. For brevity, we do not introduce and analyze other similar characteristics. The proposed data-structure transformations themselves are, however, invariant to these characteristics. Therefore, we claim that our transformations are applicable on other case studies with similar behavior as well.

### C. Overview

We give a detailed exposition on related work in Section II, and present our transformations in Section III. In Section IV, we provide examples of data-structure transformations applied on a simplified case study. In Section V, we present experimental results based on three real-life case studies. In addition to this, we analyze the optimal implementations of Case Study 1 in detail and gain insight into the achieved power reductions. We present a discussion on profiling in Section VI, and explain the differences between reusable and efficient software in Section VII. We conclude and present future work in Section VIII.

## II. Related Work

The work presented in this paper is inspired by [39] and [43]–[45]. There are, however, two main differences.

First of all, we concentrate on data-structure optimizations in the context of multimedia applications as opposed to focusing on heavy data-oriented network routers. This implies that several of the data-structure transformations that we present are new with respect to the earlier work.

Second, we analyze the software implementation of power-efficient data structures as opposed to explicitly designing and using specific configured, physical memories. Redesigning the available memory organization is not feasible in our context. We assume that the embedded platform is already designed and fully functional on the Internet. Our final research goal is to transform the critical data structures, of software that migrates onto an embedded system, in conformance with the physical predefined memory architecture of the target platform.

Since the related work on data-structure management is immense and interdisciplinary, we compare our work to different research communities. By no means do we wish to categorize a research contribution into one specific community. For instance, Palem's work [32] is mentioned in the embedded system engineering domain, but it can just as well be cited in the dynamic memory management domain.

In the *embedded system engineering* community, low power is a main objective in the dynamic-power management domain (see [3] for more references) and in the dynamic voltage scheduling domain [23]. However, data management related issues are not or hardly covered.

Data management for traditional (i.e., nondynamic) embedded applications is covered in detail in the data transfer and storage exploration (DTSE) methodology [5]. However, dynamic applications, i.e., interactive, multimedia applications, are not dealt with directly in that context. Ideally, the work we present in this paper should be followed by transformations which are useful for actual platform mapping, for example, those advocated by the DTSE methodology.

Work related to dynamic applications in the context of low-cost design is presented in [25], [29], [32], and [45]. This work is situated at a much lower abstraction level (e.g. platform dependent optimizations) than the one we present in this paper. McKee [29] designs a stream–memory controller and demonstrates its applicability for fast-page mode and Rambus DRAM memory systems. Zhang [45] presents compression techniques in order to reduce heap allocated storage, execution time, and power consumption. Palem [32] remaps data based on metrics such as locality, prefetching, and regularity of memory accesses and others. He lowers the memory needed without compromising execution time. Kistler [25] captures, by using dynamic profiling, which paths through the program are taken with what frequency. This allows spatial locality to be increased and load latency to be minimized. This is done by striving to cluster and order data members that are accessed closely after one another onto the same cache line.

Our own previous work [11], [12] deals with energy-friendly data-structure transformations. Both platform-dependent and platform-independent transformations are briefly covered. The

main emphasis of the referenced work is to show the practical feasibility of the optimizations for a specific computer game scenario. In this paper, however, we extensively cover several additional platform independent optimizations and we present three different case studies to prove the practical feasibility of our methodology. We do not discuss any platform dependent optimizations.

In the general-purpose domain, *dynamic memory management (DMM)* has a long history. In [41], an extensive survey on how efficient memory allocator design has evolved over the past decades is presented. Often, in DMM for embedded systems [31], the dynamic memory is partitioned into blocks which store the data structures of the application. Current approaches include managing all the free blocks in a single linked list [31] and simulating the application with different general-purpose memory managers [4]. In addition, Haggander [20] defines and exploits the object-oriented characteristics of an application in order to obtain less physical memory accesses by using application specific structure pools. Furthermore, part of Chilimbi's work in [7] is devoted to analyzing garbage collection based programming languages in terms of memory performance. Moreover, he presents three data placement design principles: clustering, coloring, and compression. These can be used to improve the spatial and temporal locality of pointer-centric applications.

More attention is paid to power consumption and other embedded systems criteria in [6], [10], and [28]. A fast, stepwise, cost-driven, and automated exploration and refinement is proposed by Leeman [28] at the system level for multimedia applications, which operate on large and irregular data structures. Also, in the telecom network domain, da Silva [10] presents a methodology which allows a system specification to be systematically mapped onto an optimized embedded single-chip hardware/software implementation. Chang [6] designs a high-performance memory allocator in hardware for object-oriented systems for full-customized architectures.

Dynamic memory allocation addresses the problem of when and how to allocate extra (or deallocate "garbage") memory. DMM schemes deal with multiple data structures of various shapes and access patterns in order to obtain an overall good mapping of the data onto physical memory. Extra, but necessary management, is added in order to cope with malloc requests from different parts of the application. Clearly, this problem is complementary to the problem we focus on, namely choosing a good representation of a specific data structure. Some of our driver implementations contain malloc requests but we have respected this as is and focused on the problem of data-structure representation instead. Potentially, more gains can be made, once the complementary problem of DMM has been addressed and the corresponding optimizations are applied to our drivers as well.

In the *software performance engineering* community, mostly unified modelling language (UML) based designs [16] are analyzed in terms of performance. Application specific scenarios are investigated and transformed into more economical solutions. Patterns [14] and antipatterns [36] are developed. For instance, one of the antipatterns presented in [36] is directly related to data-structure management. However, power consumption in particular and embedded software in general are not considered. The reason why this community is relevant to consider for our own work, is because the applications we want to embed are of the nature presented in this community, e.g. distributed gaming, web browsers, etc. (see [9]). Dealing with such large scale and dynamic software implies that higher abstraction levels than pure *C* code are considered (e.g. UML or equivalent software design languages) during optimization. These high-level abstractions are situated on top of the transformations that we propose in this paper. Our proposed method chooses an efficient application-specific implementation of an 1-to-n relationship described in UML.

In the *theoretical computer science* community a lot of effort has been spent on formalizing abstract data types [2], [13], corresponding optimizations [40], and verification [21]. Also, data structure and algorithmic design for specific problems have been a main research topic in the past [8], [26]. Attempts to formalizing data-structure transformations have been around for quite some time [15], but obtaining success outside academic institutes remains a challenge.

In the *database* community [17]–[19], [35], people have systematically explored various data structures expressed in formal (e.g. query) languages. Using a well-designed language for a specific problem domain (e.g., querying a spatial database) results in explicit information being expressed by the user. This information can then easily be exploited by optimization tools. In our multimedia application context, such explicit information is not present and hence our problem is quite different and requires other approaches to solve.

## III. Transformations

We present various data-structure transformations and informally motivate the usefulness of each transformation in terms of memory footprint and/or data accesses for specific examples. Analytical formulas are omitted for brevity.

### A. Adding a Linked Structure

In Fig. 1, a data structure consisting of one array (i.e. data structure 1) is transformed into a data structure consisting of two arrays and a head pointer (i.e., data structure 2). The head pointer is a logical pointer or link to the first entry of the data structure. It merely contains the index (and not the physical address) of the array element to which it is "pointing" to.

Data structure 1 is a sparse array of records: it contains fragmentation. Each record is denoted by "R." The index of the corresponding array element is denoted by "K." This index value is equivalent to an implicit key, i.e., the index or key value is not stored explicitly. Every record R corresponds to one unique position in the array. This for instance means that record R2 can only be stored in the array element that is characterized by K2 and nowhere else. Data structure 2 is the transformed data structure: a linked layer (i.e., the top array with the head pointer) is introduced.

The top array of data structure 2 contains links which are merely index values (or logical pointers). Every array element of both the first and the second array (of data structure 2) corresponds to a unique record R. In other words, fragmentation is present in both arrays. Since an array is added and fragmentation

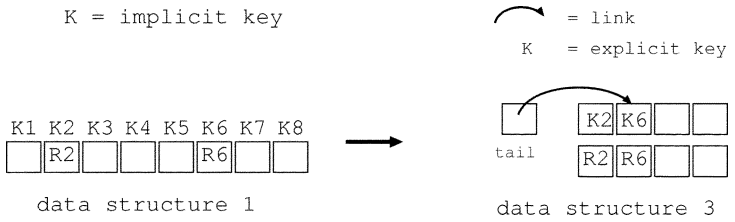



Fig. 2. Adding an array of explicit keys on top of an array of records. The assumption is made that maximum four records are stored in the data structure (at any moment in time).

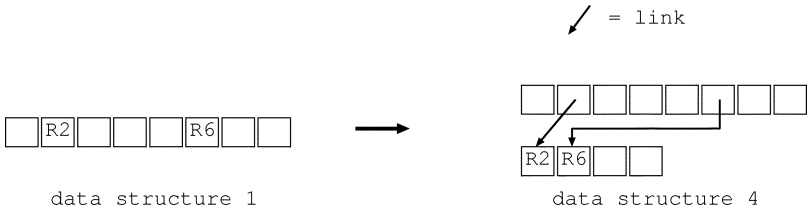



Fig. 3. Adding an array of links on top of an array of records. Each link points to a record (R). The assumption is made that maximum four records are stored in the data structure (at any moment in time).

is not removed, data structure 2 has a larger memory footprint in comparison to data structure 1.

An implicit correlation is present between the array elements of the top array and the array elements of the bottom array of data structure 2. For instance, if the second array element of the top array is consulted, then the associated record (i.e., R2) in the bottom array can be consulted directly as well with a second data access (i.e., no overhead accesses are present). Implicit correlations are not depicted explicitly in Fig. 1 and other figures.

Intuitively, finding a specific record is as cheap for data structure 2 as it is for data structure 1: just one data access is needed. The link layer is not used for this operation. Traversing all records is cheaper for data structure 2 than data structure 1 due to the links that can be used to quickly traverse the whole data structure.

To summarize, transforming data structure 1 without a linked layer into data structure 2 with a linked layer has the disadvantage that memory footprint increases. However, the data accesses needed to traverse all records decrease, thus, a tradeoff is involved. The number of data accesses needed to find a specific record remains unchanged.

### *B. The Use of Implicit Versus Explicit Keys*

In Fig. 2, a data structure consisting of one array (i.e., data structure 1) is transformed into a data structure consisting of two arrays and a tail pointer (i.e., data structure 3). The tail pointer is a logical pointer or link to the last entry of the data structure.

Data structure 1 is the initial sparse array of records. The top array of data structure 3 contains the key values of the array of data structure 1. In other words, the keys are stored explicitly. The bottom array of data structure 3 is similar to the array of data structure 1. However, a record R can potentially be placed anywhere in this bottom array.

An implicit correlation is present between the array elements of the top array and the array elements of the bottom array of data structure 3. For instance, if the second array element of the top array (i.e., K6) is consulted, then the associated record (i.e., R6) in the bottom array can be consulted in a second data access without additional accesses.

The transformation presented in Fig. 2 results in a nonfragmented representation (cfr. data structure 3). In this specific example, it is *a priori* known that maximum four (out of eight) records are to be stored in the data structure at any moment in time. Based on Fig. 2, data structure 3 consumes slightly more memory than data structure 1.

Based intuitively on Fig. 2, finding a specific record is more costly, in terms of data accesses, for data structure 3 than it is for data structure 1. This is because in data structure 3 the record to be found can be stored anywhere in the data structure. On the other hand, traversing all records is cheaper, in terms of data accesses, for data structure 3 than it is for data structure 1. Five data accesses are needed for data structure 3 while eight are needed for data structure 1 (since it is not known *a priori* which two records are stored in the array).

To summarize, transforming data structure 1 with implicit keys into data structure 3 with explicit keys has the advantage that traversal becomes less expensive in terms of data accesses. On the other hand, the memory footprint and the data accesses needed to find a specific record increase.

### *C. Introducing Indirection*

In Fig. 3, a data structure consisting of one array (i.e., data structure 1) is transformed into a data structure consisting of two arrays (i.e., data structure 4).

Data structure 1 is once again the initial sparse array of records. Data structure 4 is the transformed data structure: an indirection layer (i.e., the top array) is introduced.

The top array, of data structure 4, contains the index values (or logical pointers) into the bottom array of data structure 4. Every array element of the top array corresponds to a unique record R. On the other hand, in the bottom array, every array element can potentially contain any record R.

Data structure 4 has a larger memory footprint than data structure 1. Analyzing Fig. 3 in terms of data accesses, we conclude


Fig. 4. Adding an array of markings on top of an array of records.

that looking up a specific record R is more expensive for data structure 4 than it is for data structure 1 due to the indirection. The same holds for traversal: eight accesses are needed for data structure 1 while ten are needed for data structure 4 (i.e., eight accesses for the top array and two accesses for the bottom array).

To summarize, transforming data structure 1 without indirection into data structure 4 with indirection has the disadvantage that memory footprint and the data accesses to find and traverse increase. We will, however, frequently apply this transformation together with other transformations, and obtain an overall decrease in access count.

### D. Marking

In Fig. 4, a data structure consisting of one array (i.e., data structure 1) is transformed into a data structure consisting of two arrays (i.e., data structure 5).

Data structure 1 is the initial sparse array of records. Data structure 5 is the transformed data structure: an array of elements containing 0 or 1 is introduced. This top array is called a bit vector. An implicit correlation is present between the top array and the bottom array. For instance, since the value stored in the second array element of the top array contains a 1, the second array element of the bottom array contains a record (i.e., R2).

In this specific example, the bit vector is only eight bits or one byte long. This implies that traversing the bit vector only consumes one data access as opposed to eight data accesses.

The total memory footprint of data structure 5 is only slightly larger than that of data structure 1.

Analyzing data structure 5 in terms of data accesses, we observe that, to insert a record, a total of three data accesses is needed: a) one data access to store the record in the bottom array; b) one data access to retrieve all the bits in the bit vector; and c) one data access to store the updated bit vector. Note that for step b) the entire byte of the bit vector needs to be retrieved.

Looking up a specific record in data structure 5 is achieved in one data access. Only the bottom array needs to be consulted.

Traversing data structure 5 is cheap. One data access is needed to retrieve the bit vector and consequently, two data accesses are needed to retrieve the two records R2 and R6 from the bottom array.

To summarize, transforming data structure 1 without marking into data structure 5 with marking, has the disadvantage that memory footprint increases. However, the data accesses needed to traverse all records decreases. The number of data accesses needed to find a specific record remains unchanged.

### E. Key Splitting

Our fifth data-structure transformation is key splitting [12], [45]. Since this is a more involved transformation, we defer the explanation to Section IV-E, in which we directly apply key splitting on a realistic example.

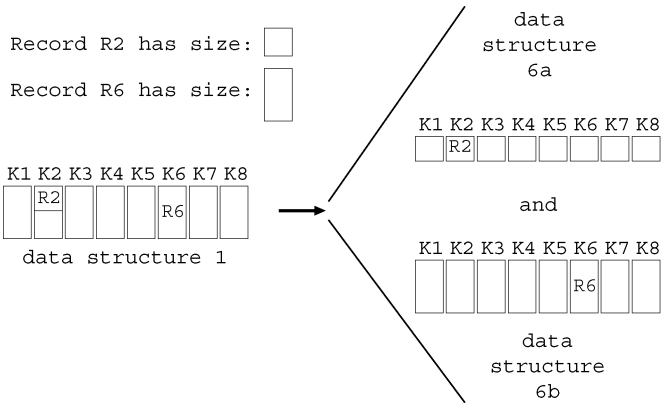



Fig. 5. Partitioning a data structure (e.g., an array) into two data structures (e.g., two arrays).

### F. Partitioning

In Fig. 5, an array (i.e., data structure 1) is partitioned into two arrays (i.e., data structures 6a and 6b). This transformation allows a set of different sized records to be decomposed into two (or more) sets of similar sized records. As a result, after applying the transformations presented previously, on each set of similar sized records individually, we increase the search space of low-cost data-structure implementations considerably.

## IV. Simplified Case Study: Tetris Game

In this section, we present a simplified version of Case Study 1 in Section V-A. We apply most of the previously presented transformations to the pixels buffer of a Tetris game and give insights into the effects of each transformation.

Tetris is a popular computer game in which the user controls falling objects of varying shapes and sizes. During game play, multiple objects are piled at the bottom of the screen and typically this pile grows until it reaches the top of the screen. When the latter occurs, the game ends. Multiple variants of the game exist, but we concentrate in particular on a version of the game in which multiple objects can fall down simultaneously while the user is interactively controlling the movement of these objects.

Each object has a specific shape which is decomposed into rectangles. A rectangle has one color and can be associated with a specific position on the screen by storing its $x$ and $y$ coordinates of the lower left and upper right corners. In this version of Tetris, the maximum number of rectangles is 100 (i.e., at the end of the game). We exploit this knowledge when transforming the pixels buffer.

### A. The Pixels Buffer

The pixels buffer is a data structure which manages the rectangles of the Tetris game. This includes the movement of the rectangles and the rendering of the rectangles from red, green, blue (RGB) format to video output Y (lminance) and U (chrominance blue) and V (chrominance red) format [30].

We present a typical implementation in Section IV-B and apply the transformations of Section III to obtain more economical implementations in terms of data accesses and/or memory footprint in Sections IV-C and IV-D. In Section IV-E we introduce and directly apply key splitting to the pixels buffer. We compare the different implementations in Section IV-F.

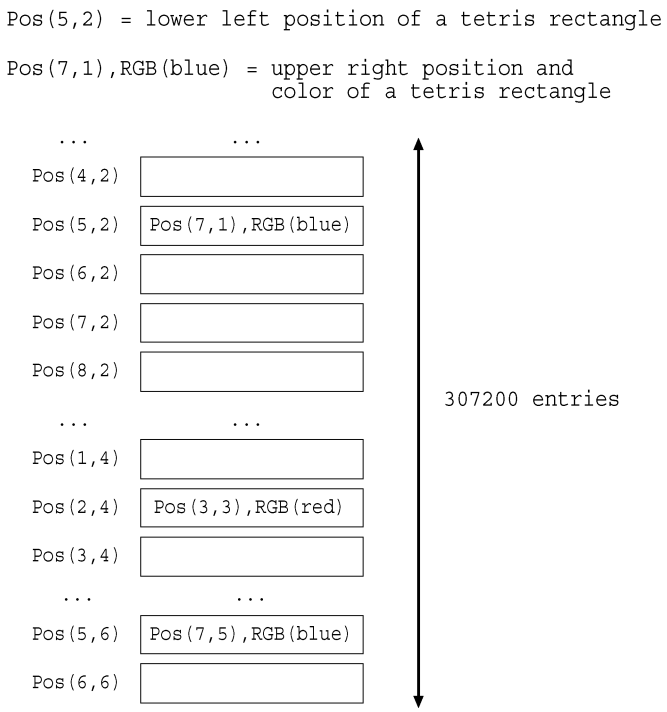



Fig. 6. Implem1: a sparse array of records. A record represents the position (Pos) of the upper right corner and the RGB color of a rectangle in the Tetris game. Each array element is indexed by a position value which represents the lower left corner of a rectangle. The total number of array elements is equal to the resolution of the screen, i.e., $640 \times 480 = 307\,200$. Only 100 of these contain a Pos-RGB value since there are only 100 rectangles in our version of the Tetris game. Note that for simplicity only three of the 100 rectangles are shown.

### B. The Initial Pixels Buffer: A Sparse Array

A typical implementation, Implem1, of the pixels buffer is a sparse array of records. This data structure is presented in Fig. 6, in which only three out of the maximum 100 rectangles are shown (to simplify the figure).

The position of the lower left corner of a rectangle corresponds to the index value of an array element. The record stored in the array element contains the position of the upper right corner and the RGB color of the rectangle.

The most important access operations are the insertion of a rectangle, removal of a rectangle, and traversal through all rectangles (in order to render them onto the screen). Note that this knowledge can easily be obtained by profiling the Tetris game or by asking the programmer of the game.

In Table I, we present the memory footprint of the array and the number of data accesses that are needed for the access operations for 100 rectangles. Since a Pos value and an RGB value both consume 3 B, we obtain a total memory footprint of $640 \times 480 \times (3+3)$ B or 1800 KB.

One data access is needed to insert a rectangle into the array. Since 100 rectangles need to be inserted, a total of 100 data accesses are needed. To remove a rectangle from the array, one data access is needed. This too amounts to 100 data accesses for the removal of 100 rectangles. To traverse through the array (i.e., to consult all 100 rectangles) a total of $640 \times 480 = 307\,200$ accesses are needed.

TABLE I
PIXELS BUFFER IMPLEMENTATIONS MEMORY FOOTPRINT [KB], DATA ACCESSES TO INSERT 100 TETRIS RECTANGLES, REMOVE 100 RECTANGLES AND TRAVERSE ONCE THROUGH ALL 100 RECTANGLES, AND ESTIMATED POWER CONSUMPTION [mW]

| | mem.ftprt. [KB] | insert | remove | traverse | power [mW] |
|---|---|---|---|---|---|
| Implem1 | 1800 | 100 | 100 | 307200 | 368.9 |
| Implem2 | 0.88 | 400 | 3100 | 201 | 4.4 |
| Implem3 | 300.9 | 500 | 900 | 201 | 1.9 |
| Implem4 | 11.1 | 1200 | 750 | 201 | 2.6 |

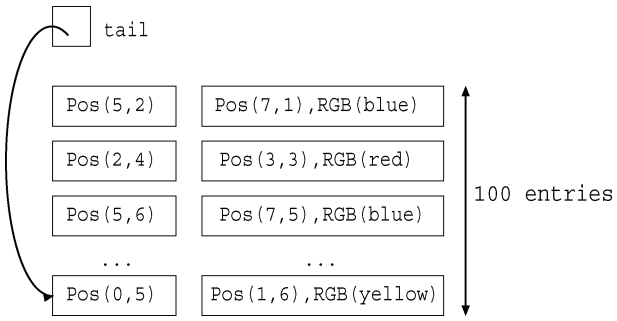



Fig. 7. Implem2: a more compact storage representation: the keys are explicit. Only four of the 100 rectangles are shown for simplicity.

### C. Implem2: Explicit Keys

A second implementation of the pixels buffer, Implem2, is obtained by making the keys (or the indices) of the array of Fig. 6 explicit. This is in correspondence to Section III-B and results in the data structure of Fig. 7. The two arrays in this data structure are implicitly correlated. This means that when for instance Pos(2,4) is retrieved from the first array, the associated Pos(3,3)-RGB(red) pair can be retrieved in the second array in a second data access.

The memory footprint of this data structure is only $100 \times (3+6)+1$ B or 0.88 KB. In this calculation, we realistically assume that the tail pointer consumes one byte.

Inserting a rectangle only takes four data accesses: a) one data access to retrieve the tail pointer's value; b) two data accesses to store the Pos and Pos-RGB values, respectively; and c) one data access to store the incremented tail pointer. Thus, for the insertion of 100 rectangles, a total of 400 data accesses are needed.

Removing a record is a more involved operation. It includes: a) finding the to-be-deleted Pos and Pos-RGB entries (which takes a nonconstant amount of accesses). Assuming that in the average case 50 rectangles are stored in the data structure, a total of $1 + (50/2) = 26$ data accesses are needed. One access is needed for the tail pointer and 25 accesses are needed in the average case to find a random rectangle in the data structure. In addition to this, the following needs to be done. b) Copying the last Pos and Pos-RGB values into the previously found entries in four data accesses; and c) storing the decremented tail pointer in one data access. In total, this amounts to 31 data accesses. For 100 rectangles, a total of 3100 accesses are needed (to implement the removal of those 100 rectangles). This is about a factor of three more accesses than in Implem1.

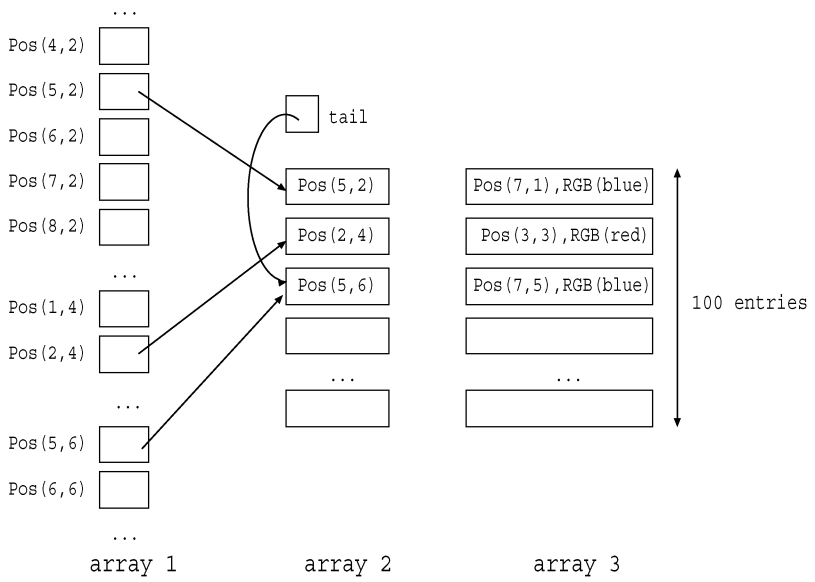



Fig. 8. Implem3: a storage representation where the key values are stored both implicitly (array 1) and explicitly (array 2). The drawing corresponds to the scenario where only three of the 100 rectangles have been inserted into the data structure. This is done for simplicity. Other 97 rectangles still need to be inserted.

However, traversing the data structure (when all 100 rectangles are stored) is achieved by a) retrieving the tail pointer's value in one data access and b) accessing each Pos and Pos-RGB entry in 200 data accesses. This results in a total of 201 data accesses for traversal. This is about 1500 times less accesses than in Implem1.

### *D. Implem3: Explicit Keys and Indirection*

A third implementation of the pixels buffer is obtained by adding a layer of indirection to Implem2. This is in correspondence to Section III-C. The data structure is shown in Fig. 8. In this implementation, the key values (i.e., the positions of the lower left corner of the rectangles) are stored both implicitly (in array 1) and explicitly (in array 2). This form of redundancy results in a) a large memory footprint but b) a small number of data accesses for the removal of a rectangle as is shown in Table I.

To insert a record, an additional data access (for array 1) is needed in comparison to Implem2. This results in a total of five data accesses for one rectangle and thus 500 data accesses for 100 rectangles.

Removing a record consists of the following steps: a) the to-be-deleted Pos and Pos-RGB entries are found via array 1 in one data access; b) the indirection (i.e., the pointer from array 1 to array 2) is deleted in one data access; c) the tail pointer is consulted in one data access; d) the last Pos and Pos-RGB values are copied into the previously found entries (of arrays 2 and 3) in four data accesses; e) the indirection from array 1 to these moved Pos and Pos-RGB values is updated in one data access; and f) the decremented tail pointer's value is stored. This takes nine data accesses in total. For 100 rectangles, a total of 900 data accesses are needed. This is a factor of three less accesses in comparison to Implem2.

Traversing through the data structure is similar to Implem2: only arrays 2 and 3 are used. Array 1 in Fig. 8 is not used during traversal. Thus, a total of 201 data accesses are needed.

### *E. Implem4: Key Splitting*

In this section, we apply key splitting to array 1 of Implem3 (see Fig. 9). Key splitting implies that the key bits, that make up a key value, are split into two (or more) groups. For instance, the Pos(5,2) value is a key value present in array 1 of Fig. 9. In binary form this corresponds to 000 0000 0101 0000 0101. A possible key splitting, is to split these 19 key bits into one group of 13 b and another group of 6 b, i.e., 000 0000 0101 00 and 00 0101, respectively. The newly obtained 13 key bits represent the key value of an entry in array 1a in Fig. 9. Similarly, the newly obtained 6 b represent the key value of an entry in array 1b. The reason why we have chosen to split the 19 key bits into, respectively, 13 and 6 key bits is given below.

Let $y$ represent the number of key bits of an array element of array 1b; e.g. $y = 6$ in the above example. Let $N$ represent the total number of array elements of array 1a. The resolution of the video screen is 307 200 pixels. Based on these definitions, we know that $N \times 2^y = 307\,200$. Choosing a specific value for $y$ (e.g. $y = 6$), results in a value for $N$ (e.g. $N = 4800$). The total number of key bits for an array element of array 1 is 19. Therefore, $19 - y$ represents the number of key bits for an array element of array 1a. In our particular example, $19 - y = 13$ key bits.

The objective of key splitting is to reduce the total memory footprint. In other words, the total memory footprint of arrays 1a and 1b in Fig. 9 is much less than the memory footprint of array 1. We calculate the memory footprint of arrays 1a and 1b as follows: $\mathrm{MemFootpr} = N \times 1\,\mathrm{B} + 100 \times (2^y \times 1\,\mathrm{B} + 1\,\mathrm{B})$. The factor 100 is present in the equation because we know that a maximum of 100 Tetris rectangles are present in the Tetris game. Each large array element of array 1b has a counter C whose size is 1 B. It denotes the number of stored entries. Since we want to minimize the memory footprint, we apply the previous two equations for different values of $y$ and select the value which corresponds to the minimum memory footprint. In this example, the best value of $y$

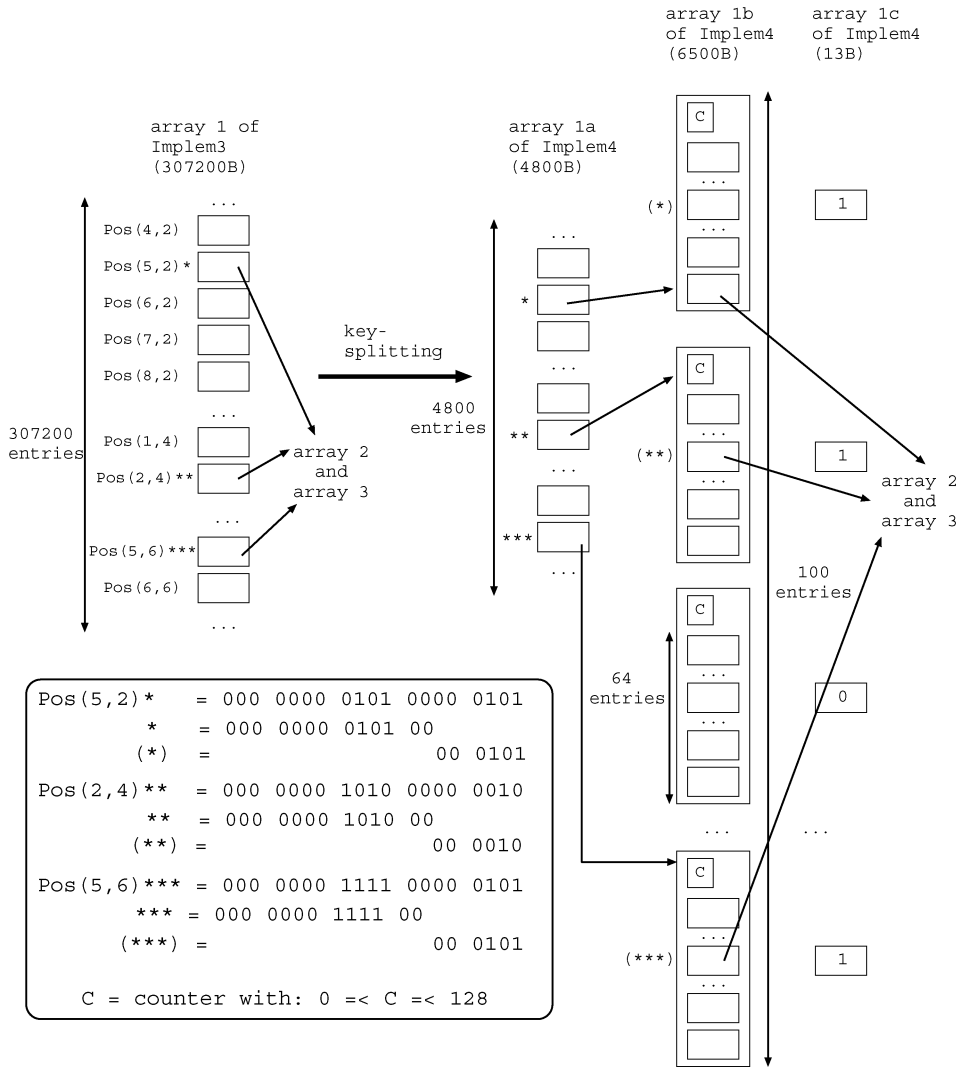



Fig. 9. Key splitting of array 1 of Implem3 into three arrays that belong to Implem4. The arrows leaving the small array elements in array 1 b point to specific entries of array 2 (cfr. Fig. 8).

is $y = 6$ and consequently $N = 4800$. The memory footprint of arrays 1a and 1b together is 11.1 KB. Compared to the memory footprint of array 1, which is 300 KB, this is a significant reduction. In Fig. 9, these optimal values are used.

In addition to arrays 1a and 1b, we add an additional array 1c as well. Each array element of this array marks (cfr. Section III-D) a large array element of array 1b. Recall that $2^6 = 64$ small array elements are present in each large array element of array 1b; i.e., array 1b is a two-dimensional (2–D) array in programming jargon. If no small array elements are used in a specific large array element of array 1b, then the corresponding array element of array 1c contains a 0, else it contains a 1. Since there are 100 large array elements of array 1b, the same amount of array elements of array 1c are present as well. This implies that the memory footprint of array 1c equals $(100/8) = 13$ B.

Even though the total memory footprint of arrays 1a, 1b, and 1c is significantly smaller than the memory footprint of array 1, the data accesses needed, to find, insert, or retrieve have increased. For instance, finding a specific record implies that array 1a is consulted first by specifying the thirteen most significant key bits, then array 2a is consulted by specifying the six least significant key bits, and finally the record is retrieved in arrays 2 and 3. Even more access overhead is present for insertion but we omit further explanation for brevity. The numerical results are shown in Table I.

### *F. Comparing the Different Implementations*

Comparing Implem1, 2, 3, and 4 in Table I we observe that the least power consuming implementation is Implem3. On the other hand, Implem2 consumes the least amount of memory footprint. Note that in Implem2, Implem3, and Implem4 a total of 201 data accesses are needed for traversal. This is because all three data-structure implementations use the same arrays (i.e., arrays 2 and 3 and the tail pointer in Fig. 8) for traversal.

We also observe that Implem4 lies in between two extremes, namely Implem2 and Implem3. Implem4 has a relatively small access count and a relatively small memory footprint in comparison to the other implementations. This makes Implem4 a very interesting implementation point for certain situations. It also

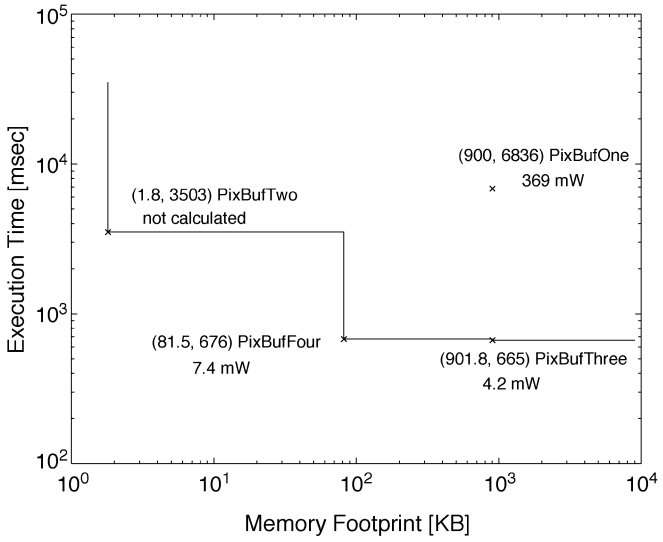



Fig. 10. Pareto curve based on experiments on the TriMedia of the Pixels Buffer's data structure for 100 Tetris rectangles. Memory footprint [KB] is measured and execution time [msec] as well for 100 consecutive output frames. Power consumption [mW] is estimated based on profiled data accesses.

forms another Pareto point in the overall tradeoff space. Furthermore, recall that many key splitted variants of Implem4 exist (including implementations in which key splitting is applied more than once) which are potentially additional Pareto points in the Pareto space. We have only presented one possibility.

## V. Experimental Results

We present experimental results of three real-life case studies in which we apply data-structure transformations of the kind described in Section III. The power consumption of our optimal data structures are only analyzed in detail for Case Study 1. Similar reasonings have led to the results for the other cases.

### A. Case Study1: Pixels Buffer

In this case study, we examine the Pixels Buffer of a more involved Tetris game (in comparison to Section IV). For instance, we use double buffering [30] and a more complicated conversion algorithm from RGB to YUV (chrominance and luminance) format [38]. Our first set of experiments result in Fig. 10 in which we present three Pareto optimal and one suboptimal data-structure implementation of the pixels buffer.

Execution time and memory footprint in Fig. 10 are measured on the TriMedia [38] platform. Power consumption is estimated analytically as discussed in Section I-A. The 16 KB cache of the TriMedia has been configured as fully hardware controlled with a standard Least Recently Used (LRU) policy and is enabled.

PixBufOne represents the initial implementation of the pixels buffer. PixBufTwo represents an implementation in which the keys are stored explicitly. In PixBufThree, the keys are stored implicitly resulting in a much larger memory footprint. PixBufFour is obtained by key splitting PixBufThree. This results in a small increase in execution time and a relatively large decrease in memory footprint (see Fig. 10). For simplicity PixBufOne can be compared to Implem1 (cfr. Table I), PixBufTwo can be compared to Implem2, etc.

### B. Detailed Discussion on Power Consumption

To derive the power formula in Section I-A, we have taken into account the data accesses and memory size of the SDRAM. However, the accesses that are done in the cache of the TriMedia are not incorporated in the formula. In this section, we justify why this is so. We do this by exploiting the fact that the TriMedia processor allows a maximum of 8 KB of the 16-KB cache to be software controlled.

In a second set of experiments, we reexecute all data structures presented in Fig. 10 with 8 KB of the cache containing useless data throughout the whole execution run. This amounts to having a useful cache of only 8 KB as opposed to 16 KB. We have compared the execution times to those presented in Fig. 10 and observe no or negligibly small changes in execution time.

In a third set of experiments, we lock as much of the optimal data structure under investigation in the cache (and keep the rest in the SDRAM). This results in having the equivalent of an 8 KB foreground memory and an 8 KB cache. In these experiments, the difference in execution times are once again negligibly small (i.e., 1.67%) from those presented in Fig. 10.

Based on these experiments, we conclude that, in our initial set of experiments, the cache is continuously reading and writing data from the SDRAM. In other words, temporal locality is not present in the application under investigation. Indeed, the same conclusion is made when analyzing our game from a functional point of view. All moving or changing rectangles in the game are sent to the pixels buffer for video output (while all static rectangles are only sent once). This explains the lack of temporal locality in our application and, hence, the uselessness of having a (relatively) big cache in this context. In this case study, the useful size of the cache is relatively small since only spatial locality can still motivate its use. We, therefore, conclude that an 8-KB cache (and also a 16-KB cache) is too big for our application. For a future embedded system, we assume to be able to specify the exact amount of cache space we need for our application. Therefore, we have not incorporated the redundant accesses that are made to the TriMedia cache in Fig. 10.

Unfortunately, due to the limitations of the TriMedia processor, we are not able to apply more detailed experiments on this topic. The above discussion is not quantitative in nature. It merely motivates why we consider a useful cache for this application not to be of major concern when estimating power consumption.

### C. Case Study 2: Real-Time Strategy Game Engine

In our second case study, we have simulated and profiled part of a real-time strategy game engine. A typical scenario in this game is that an army of soldiers travels across different regions toward an enemy army. We implement different data structures for a specific region and compare the implementations in Table II for a varying number of soldiers that enter the region under investigation.

The management of a region containing soldiers can be summarized as follows. First, soldiers are inserted into the region data structure (when the soldiers enter the region in the game). Second, all soldiers are traversed in order to a) coordinate their

TABLE II
GAME ENGINE IMPLEMENTATIONS. MEMORY FOOTPRINT [KB]/DATA ACCESSES FOR ONE OUTPUT FRAME AND VARYING NUMBER OF SOLDIERS

| soldiers | data struc 1 | data struc 2 | data struc 3 | data struc 4 |
|---|---|---|---|---|
| 40 | 8.59/ 1600 | 9.375/ 563 | *0.5/ 743* | *1.21/ 484* |
| 200 | 8.59/ 3200 | 9.375/ 2801 | *2.54/ 2600* | *2.93/ 2404* |
| 400 | *8.59/ 5200* | 9.375/ 5603 | 5.08/ 27073 | *5.07/ 5404* |

movements and interactions amongst each other and b) to inform the video output functionality of the game what the new position is of each soldier. This second step is applied repeatedly until the soldiers reach the border of the region. Finally, in a third step, all soldiers are removed from the region data structure (when the soldiers enter an adjacent region in the game).

In Table II, memory footprint and data accesses are traded off. The data structures correspond exactly to those presented in Section III and the Pareto points are printed in italics.

### D. Case Study 3: Meshing

Our third case study is related to three-dimensional (3-D) rendering in general [41] and meshing of a 3-D object [27] in particular. The 3-D object under investigation is represented by 6713 vertices and 13 406 faces (or triangles). These vertices and faces need to be managed in a meshing algorithm and corresponding data structure. It is this data structure that we analyze and transform into more optimal data structures (cfr. Table III).

The data structure under investigation contains faces and vertices that need to be traversed repeatedly in order to render them onto the screen. Insertion and removal are less profound access operations and we omit them in our discussion for clarity.

Our first solution, Solution 1, stores the faces and the vertices in separate arrays. Indirection is present between these two arrays (cfr. Fig. 3). Since many faces of the 3-D object share common vertices, this solution results in a small memory footprint since each vertex is only stored once. However, relatively many data accesses are needed to traverse all faces and vertices due to the indirection overhead (cfr. Section III-C).

On the other hand, Solution 2 uses one large array to store all vertices of each face. This results in an increased memory footprint since each vertex that is shared by two or more faces is stored multiple times in the data structure. Due to the lack of indirection however, traversal is much cheaper in terms of data accesses.

Table III shows the tradeoff between Solution 1 and Solution 2. The original solution is suboptimal both in memory footprint and in data accesses, and is not a Pareto point.

## VI. IS PROFILING APPROPRIATE?

In our work, the use of profiling (e.g. for a game application) seems inappropriate as the profiled information should vary significantly based on actual inputs. Indeed, we have, for instance, exploited the fact that the 3-D object in Case Study 3 has exactly 6713 vertices and 13 406 faces. Exploiting this results in huge gains, but how can we cope with an application in which multiple 3-D objects of various sizes have to be rendered after each other? A possible approach to solving this problem is to design a data structure which is, on average, optimal for a range of 3-D objects. For instance, we could have analyzed and designed an optimal data structure for 3-D objects whose vertices range between 5000 and 7000 and whose faces ranges between 13 000 and 15 000. This is exactly the approach we followed in the Tetris game in which we designed optimal data structures that could cope with maximally 100 Tetris rectangles.

Playing games or profiling in general is thus needed to some extent in order to characterize the relevant contexts of the game, but the instances that occur are clustered in ranges (or "scenarios") so the profiling does not need to distinguish between every possible individual case. Viewed from a practical stance, we are inclined to say that it is the application designer who will—while developing and testing his software—do the profiling and characterize his application in such a way that the system engineer can exploit this powerful information during optimization.

TABLE III
MESH IMPLEMENTATIONS. MEMORY FOOTPRINT [KB], DATA ACCESSES FOR ONE OUTPUT FRAME, AND POWER CONSUMPTION [mW] FOR A 3-D IMAGE OF 6713 VERTICES AND 13 406 FACES

| | mem.ftprt. [KB] | data accesses | power [mW] |
|---|---|---|---|
| Solution1 | 281.8 | 113961 | 136.8 |
| Solution2 | 693.9 | 40219 | 48.6 |
| Original Solution | 1669.6 | 194397 | 233.3 |

## VII. REUSABLE VERSUS. EFFICIENT SOFTWARE

The transformations presented in Section III seem to be ones that would normally be done in a good software implementation. We explain why this is a misconception.

First, all initial implementations (of all our drivers) are obtained from other sources. We did not write these ourselves. Second, a "good" software implementation can be interpreted in contradictory ways as we demonstrate below.

An initial software implementation, written by a software engineer, will consider his implementation to be good if it is 1) reusable; 2) extendible; 3) easy to test; and 4) if it has sufficient clarity. In this setting and at this stage of the overall design trajectory, the software engineer is not interested in efficient execution (and certainly not in low power solutions); he is dealing with a problem and wants to express his solution (in code) in a readable manner; or he is specifying his solution in executable code.

In the system engineering community, it is more likely that good software is interpreted as being efficient software. It is indeed realistic to assume that the initial implementation contains a well-tuned data structure as opposed to reusing one off the shelf. However, as we demonstrate in Section IV, applying a sequence of transformations—including the time consuming

and error prone key splitting transformation—results in a very low-power solution and it is doubtful whether every system engineer has a) the time to derive and especially implement such a solution and b) the skill to derive such a solution in the first place.

The reason why large gains are made in our case studies is because the structure in the access pattern of the data structure is exploited to its full extent. For instance, for the Tetris game, the access pattern is exactly as follows in chronological order:

- Tetris rectangles are inserted into the data structure;
- all rectangles are traversed, and finally;
- all rectangles are removed.

This structure has been exploited fully in Case Study 1 (and partly, but sufficiently, in our simplified case study) by choosing appropriate data structures. In common current-day practice, software engineers just (re)use an easy-to-implement data structure. A system engineer will optimize the chosen data structure but he will typically do this at a lower abstraction level, i.e., at a level where the semantics of the data accesses to the data structure are lost.

## VIII. Conclusion and Future Work

We have presented various high-level and systematically applicable data-structure transformations for dynamic multimedia applications. They have been applied on one conceptually simplified and three real-life case studies. Applying various transformations in sequence results in nontrivial data-structure implementations. In this way, we can obtain optimal implementation points in a Pareto space in which data accesses are traded off with memory footprint. Execution time and memory-related power consumption are reduced significantly in comparison to the initial implementations.

In future work, we will examine different types of user behavior, that characterize one specific application, in more detail. Also, experiments will be done on a simulator, in order to evaluate the different data-structure implementations in more detail. In addition to this, leakage energy, processor power, and also the small cache or foreground memories used, need to be incorporated in our analysis in order to obtain more precise power figures. In this paper, we make no absolute claims in terms of total power reduction, but express the potential usefulness of our data-structure transformations in the context of embedded multimedia applications, based on relative comparisons of data memory-related power.

## References


[1] B. S. Amrutur and M. A. Horowitz, "Speed and power scaling of SRAM's," *IEEE Trans. Solid-State Circuits.*, vol. 35, Feb. 2000.

[2] E. Astesiano and M. Bidoit *et al.*, "CASL: the common algebraic specification language," *Theoretical Comput. Sci.*, vol. 2001.

[3] L. Benini and G. DeMicheli, *Dynamic Power Management Design Techniques and CAD Tools*. Norwell, MA: Kluwer, 1998.

[4] E. D. Berger and B. G. Zorn *et al.*, "Composing high-performance memory allocators," in *Proc. ACM SIGPLAN Conf. Programming Language Design Implementation (PLDI)*, Utah, 2001, pp. 114–124.

[5] F. Catthoor and S. Wuytack *et al.*, *Custom Memory Management Methodology: Exploration of Memory Organization for Embedded Multimedia System Design*. Norwell, MA: Kluwer, 1998.

[6] J. M. Chang and C. D. Lo, "OMX: Object management extension," in *Proc. 2nd Int. Workshop on Compiler Architiecture Support for Embedded Systems (CASES)*, Washington, DC, Oct. 1–3, 1999.

[7] T. M. Chilimbi, M. D. Hill, and J. R. Larus, "Making pointer-based data structures cache conscious," *IEEE Computer*, vol. 33, pp. 67–74, Dec. 2000.

[8] T. H. Cormen and C. E. Leiserson *et al.*, *Algorithms*. Englewood Cliffs, NJ: Prentice-Hall, 1998.

[9] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems Concepts and Design*, 2nd ed. Reading, MA: Addison-Wesley, 1999.

[10] J. L. da Silva Jr *et al.*, "Efficient system exploration and synthesis of applications with dynamic data storage and intensive data transfer," in *Proc. 35th ACM/IEEE Design Automation Conf.*, San Fransisco, CA, June 1998, pp. 76–81.

[11] E. G. Daylight and T. Fermentel *et al.*, "Incorporating energy efficient data structures into modular software implementations for Internet-based embedded systems," in *Proc. Workshop on Software Performance (WOSP)*, Rome, Italy, July 23–26, 2002.

[12] E. G. Daylight and S. Wuytack *et al.*, "Analyzing energy friendly steady state phases of dynamic application execution in terms of sparse data structures," in *Proc. ISLPED*, Monterey, CA, Aug. 12–14, 2002, pp. 76–79.

[13] W. Dosch and S. Magnussen, "The Lbeck transformation system: a transformation system for equational higher order algebraic specifications," in *Proc. Workshop on Algebraic Development Techniques (WADT)*, 2001, pp. 85–108.

[14] B. P. Douglass, *Real-Time UML, Developing Efficient Objects for Embedded Systems*. Reading, MA: Addison-Wesley, 1998.

[15] K. Frenkel, "An interview with the 1986 A. M. Turing award recipients—John E. Hopcroft and Robert E. Tarjan," *CACM*, vol. 30, no. 3, pp. 214–223, Mar. 1987.

[16] E. Gamma and R. Helm *et al.*, *Design Patterns, Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[17] Y. J. Garcia and M. A. Lopez *et al.*, "Post-optimization and incremental refinement of R-trees," in *Proc. 7th ACM Symp. Advances Geographic Information Systems (ACM GIS'99)*, pp. 91–96.

[18] G. Graefe and P. Larson, *B-Tree Indexes and CPU Caches*: Microsoft.

[19] A. Guttman, "R-trees: a dynamic index structure for spatial searching," in *Proc. ACM SIGMOD Conf.*, 1984, pp. 47–57.

[20] D. Haggander and P. Liden *et al.*, "A method for automatic optimization of dynamic memory management in C++," in *Proc. 30th Int. Conf. Parallel Processing (ICPP)*, Valencia, Spain, Sept. 2001, pp. 489–498.

[21] C. A. R. Hoare, "Proof of correctness of data representations," in *Acta Informatica 1*. New York: Springer-Verlag, 1972, pp. 271–281.

[22] [Online] www.infineon.com/cgi/ecrm.dll/ecrm/scripts/prod_cat.jsp

[23] N. K. Jha, "Low power system scheduling and synthesis," in *Proc. IEEE Int. Conf. Computer-Aided Design*, Nov. 2001, pp. 259–263.

[24] J. Kin and M. Gupta *et al.*, "Filtering memory references to increase energy efficiency," *IEEE Trans. Comput.*, vol. 49, Jan. 2000.

[25] T. Kistler and M. Franz, "Automated data-member layout of heap objects to improve memory-hierarchy performance," *ACM Trans. Programming Languages Syst.*, vol. 22, no. 3, pp. 490–505, May 2000.

[26] D. E. Knuth, *The Art of Computer Programming*. Reading, MA: Addison-Wesley, 1973, vol. 3.

[27] P. Kuismanen. View-Dependent Polygonal Simplification. [Online] www.tml.hut.fi/Opinnot/Tik-111.500/2000/kuismanen_final.pdf

[28] M. Leeman and D. Atienza *et al.*, "Methodology for refinement and optimization of dynamic memory management for embedded systems in multimedia applications," in *Proc. Signal Processing Symp.*, Seoul, Korea, Aug. 2003, pp. 369–374.

[29] S. A. McKee and W. A. Wulf *et al.*, "Dynamic access ordering for streamed computations," *IEEE Trans. Comput.*, vol. 49, pp. 1255–1270, Nov. 2000.

[30] T. Moller and E. Haines, *Real-Time Rendering*. Natick, MA: A K Peters, Ltd., 1999.

[31] N. Murphy, "Safe memory usage with dynamic memory allocation," *Embedded Systems*, pp. 49–57, May 2000.

[32] K. Palem *et al.*, "Design space optimization of embedded memory systems via data remapping," in *Proc. Languages, Compilers, and Tools for Embedded Systems Software and Compilers for Embedded Systems (LCTES-SCOPES)*, June 2002, pp. 28–37.

[33] P. R. Panda and L. Semeria *et al.*, "Cache-efficient memory layout of aggregate data structures," in *ISSS'01*, Montreal, QB, Canada, Oct. 2001.

[34] B. Prince, *High Performance Memories: New Architecture DRAMS's and SRAM's Evolution and Function*. New York: Wiley., 1999.

[35] P. Seshadri and M. Livny *et al.*, "E-ADTs: turbo-charging complex data," *Data Engineering Bulletin*, vol. 19, no. 4, pp. 11–18, 1996.
[36] C. U. Smith and L. G. Williams, "Software performance antipatterns," in *Proc. 2nd Int. Workshop Software Performance*, Ottawa, ON, Canada, Sept. 2000.
[37] *TriMedia TM1000 Preliminary Data Book*, Philips Electronics North America Corporation, Sunnyvale, CA, 1997.
[38] D. Verkest and J. da Silva Jr. *et al.*, "Matisse: a system-on-chip design methodology emphasizing dynamic memory management," *J. VLSI Signal Processing*, vol. 21, pp. 277–291, July 1999.
[39] D. Walker and G. Morrisett, "Alias types for recursive data structures," in *Proc. Workshop on Types in Compilation*, Montreal, Canada, Sept. 2000.
[40] P. R. Wilson and M. S. Johnstone *et al.*, "Dynamic storage allocation: a survey and critical review," in *Proc. 1995 Int. Workshop Memory Management*, Kinross, Scotland, UK, Sept. 27–29, 1995.
[41] M. Woo and J. Neider *et al.*, *OpenGL Programming Guide*, 2nd ed: Silicon Graphics, Inc., 1997.
[42] S. Wuytack and F. Catthoor *et al.*, "Transforming set data types to power optimal data structures," in *Proc. IEEE Int. Workshop Low-Power Design*, Laguna Beach, CA, Apr. 1995, pp. 51–56.
[43] S. Wuytack and J. L. da Silva Jr *et al.*, "Memory management for embedded network applications," *IEEE Trans. on Computer-Aided Design*, vol. 18, no. 5, May 1999.
[44] C. Ykman-Couvreur and J. Lambrecht *et al.*, "Exploration and synthesis of dynamic data sets in telecom network applications," in *Proc.12th ACM/IEEE Int. Symp. System-Level Synthesis (ISSS)*, San Jose, CA, Dec. 1999, pp. 125–130.
[45] Y. Zhang and R. Gupta, "Data compression transformations for dynamically allocated data structures," in *Proc. Int. Conf. Compiler Construction*, Grenoble, France, Apr. 2002, pp. 14–28.

**Edgar G. Daylight** (a.k.a. Karel Van Oudheusden) received the M.Sc. in computer science from the Katholieke Universiteit (K.U.) Leuven, Belgium. in 2000. He is currently pursuing the Ph.D. degree at the Inter-University Micro-Electronics Center (IMEC), Heverlee, Belgium.

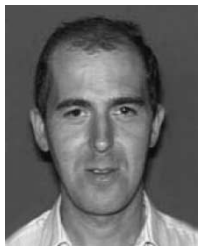
His research interests include dynamic memory management for power and performance and formal reasoning about data structures. Currently, he is trying to formalize the data structure transformations presented in this paper in order to systematically explore the search space of low power data structures.

**David Atienza** received the M.Sc. degree in computer science from the Complutense University of Madrid, Spain, in 2001. He is currently pursuing the Ph.D. degree in the Department of Computer Architecture and Automation, at the same university.

His research interests include optimization of dynamic memory management on multimedia applications for low power and high performance, computer architecture and high-level design automation.

**Arnout Vandecappelle** received the M.Sc. degree in computer science from the Katholieke Universiteit (K.U.) Leuven, Belgium, in 1997.

In 1997, he joined the Inter-University Micro-Electronics Center (IMEC), Heverlee, Belgium as a researcher and software architect. His research interests include optimization of memory architecture and memory management units for power and performance. He is currently working on algebraic optimization of addressing arithmetic.



**Francky Catthoor** received the engineering degree and the Ph.D. in electrical engineering from the Katholieke Universiteit (K.U.) Leuven, Belgium in 1982 and 1987 respectively.

Since 1987, he has headed research domains in the area of architectural and system-level synthesis methodologies, within the DESICS (formerly VSDM) division at IMEC. His current research interests are in the field of architecture design methods and system-level exploration for power and memory footprint within real-time constraints, oriented toward data storage management, global data transfer optimization, and concurrency exploitation. Platforms that contain both customizable/configurable architectures and (parallel) programmable instruction-set processors are targeted.

Dr. Catthoor is a Fellow at IMEC, Heverlee, Belgium.

**Jose M. Mendias** received the M.Sc. and Ph.D. degrees in physics from the Complutense University of Madrid, Madrid, Spain, in 1992 and 1998, respectively.

In 1992, he joined the Department of Computer Architecture and Systems Engineering, Complutense University of Madrid as a Lecturer and became an Associate Professor in 2001. Since 2002, he has been Vice Dean of the Computer Science Faculty at the same University. His current research interests include design automation, computer architecture and formal methods.