Journal of VLSI Signal Processing 40, 383–396, 2005

Methodology for Refinement and Optimisation of Dynamic Memory Management for Embedded Systems in Multimedia Applications*

MARC LEEMAN
ESAT/K.U. Leuven, Kasteelpark Arenberg 10, B-3001 Leuven, Belgium

DAVID ATIENZA
DACYA/UCM, Avda. Complutense s/n, 28040 Madrid, Spain

GEERT DECONINCK AND VINCENZO DE FLORIO
ESAT/K.U. Leuven, Kasteelpark Arenberg 10, B-3001 Leuven, Belgium

JOSÉ M. MENDÍAS
DACYA/UCM, Avda. Complutense s/n, 28040 Madrid, Spain

CHANTAL YKMAN-COUVREUR, FRANCKY CATTHOOR AND RUDY LAUWEREINS
IMEC vzw, Kapeldreef 71, B-3001 Leuven, Belgium

Received September 18, 2003; Revised November 12, 2003; Accepted January 9, 2004

Abstract. In multimedia applications, run-time memory management support has to allow real-time memory de/allocation, retrieving and processing of data. Thus, its implementation must be designed to combine high speed, low power, large data storage capacity and a high memory bandwidth. In this paper, we assess the performance of our new system-level exploration methodology to optimise the memory management of typical multimedia applications in an extensively used 3D reconstruction image system [1, 2]. This methodology is based on an analysis of the number of memory accesses, normalised memory footprint[1] and energy estimations for the system studied. This results in an improvement of normalised memory footprint up to 44.2% and the estimated energy dissipation up to 22.6% over conventional static memory implementations in an optimised version of the driver application. Finally, our final version is able to scale perfectly the memory consumed in the system for a wide range of input parameters whereas the statically optimised version is unable to do this.

Keywords: multimedia, memory management, memory bandwidth, low power, memory footprint, dynamic memory management, memory hierarchy, dynamic data types, system-level exploration

1. Introduction

The fast growth in the variety and complexity of multimedia applications and platforms has created the need for optimal algorithms on the one hand, and the development of high storage capacity and efficient memory systems on the other hand. Good examples where they become necessary are archaeological site recording and reconstruction, architectural planning, augmented reality and film industry [2–4]. These systems depend upon dynamic data management, which constitutes one

*The original version of this paper first appeared in the Proceedings of Signal Processing Systems 2003.

of the most difficult design challenges when mapping them on low-power and high-speed processors that are often not equipped with extensive hardware and system support for dynamic memory. This dynamic memory management (DMM) must provide an efficient memory de/allocation, retrieving and processing of the data involved in the multimedia algorithms by combining speed, low power, large data storage and an optimal management of multiple Dynamic Data Types (DDTs). It has to take into account the fact that these DDTs have a limited lifetime and a variable behaviour while the application is running. As a consequence, three factors influence the overall performance of the memory system:

1. The access pattern over time of the algorithm implemented (temporal locality). If some (dynamic) data is reused throughout the entire application, e.g. in the form of a small dynamic buffer, it will occupy precious internal memory.
2. The amount of memory accesses. If the data is present in the processor registers, the elements have no access penalty. Since the number of registers is limited, access will be needed from lower, larger and slower levels of the memory hierarchy and each access to such a lower memory hierarchy can result in CPU stalls if this is not properly addressed.
3. The size of the required data. Just as in statically dominated applications, the size of the data has an important impact on the use of the memory hierarchy. Contrary to the static data types, the dynamic data types can be refined to exploit this memory hierarchy.
4. The mechanisms to access the data (as defined by the data structures of the system). The retrieval of a particular data element can incur several other accesses in complex data types. For example, accessing a random element in a double linked list (an often used dynamic data type) requires on average $\frac{N}{4}$ pointer dereferences.

Taking all these factors into account, it is clear that a systematic exploration at the system-level of the possible choices for memory management in multimedia applications is a necessity. Thus, detailed power consumption, memory footprint and performance profiling must be available at this system-level.

For statically allocated data and operations, the aforementioned information is available from modern analysis and profiling tools (e.g. [5, 6]). However, for dynamically (de)allocated data types implementations, the situation is much worse. At the most, summarised information can be available for the pools of data, but the details about individual DDTs are lost. Also, simulation data can be generated at a much closer level to the final implementation on a certain platform, e.g. at instruction or cycle accurate hardware level. This is however very CPU-time consuming and requires the complete mapping trajectory, thus it is not acceptable for system-level exploration purposes.

In this paper, we demonstrate the efficiency of our new system-level exploration methodology, which optimises the DMM for typical 3D multimedia applications with the aforementioned behaviour [1, 2, 7]. After applying the methodology, the results improve to a great extent the memory footprint, memory accesses and estimated energy dissipation compared to manually optimised implementations.

The remainder of this paper is organised as follows. In Section 2, the foundations of the methodology and design features of DMM for multimedia systems are presented. In Section 3, the specification of the 3D multimedia application used to apply our methodology is illustrated. In Section 4, we characterise the behaviour of the relevant DDTs of the system. After that, in Section 5, we explain all the different steps to optimise the DMM of the system. In Section 6, the experimental results are presented. Finally, in Section 7, we draw our conclusions and present future extensions.

2. Related Work

Conceptually, the basis of a good DMM is already well established for general-purpose systems [8]. Also, several implementations for dynamic memory managers exist in a general-purpose context to allow large data storage, real-time de/allocation and frequent updates of the data structures [8, 9].

Due to the behaviour of multimedia applications (with a high demand of data retrieval and storage), the access to the data must be highly optimised. Presently, research has been started to propose suitable access methods to DDT implementations [10].

For manual and automatic memory management in embedded systems, research is performed [11, 12]. In manual memory management work, dynamic memory is partitioned into blocks and tracked by single linked lists [12]. For general purpose manual memory managers, Berger et al. [9] describes a fast `C++` template infrastructure to improve performance of general purpose dynamic memory allocators. However, from our point of view, its main definition for performance

exploration and enhancement of general-purpose memory managers limits its extensibility for other metrics (e.g. power consumption, memory accesses), as embedded systems require.

In a different field, telecom network applications, an approach that performs an exploration methodology driven by the number of memory accesses has been outlined in [13]. These applications have very specific behaviour dominated by key accesses, lookups and sparse data structures. Furthermore, they use one or few independent DDTs. These and other characteristics restrict the work to the telecom domain (see [13] for details).

System-level optimisations and techniques for general-purpose design to reduce power consumption are explained in [14]. Nevertheless, optimising the memory management at the system-level in multimedia applications with complex dynamic behaviour has not been given much attention.

Regarding power consumption estimation and profiling in general, a lot of work based on software instead of at a circuit level can be found. Most of this work is done using an instruction level analysis (amongst others [15–17]). Work to obtain accurate figures on a higher level is more recent [6, 18, 19]. Also, several analytical and abstract power estimation models at the architecture-level have received more attention recently [20] since they are needed for high-level power analysis in very large scale integration systems. However, such estimations are based on an analysis and design space without run-time analysis. This is not sufficient to deal with algorithms governed by their dynamic memory accesses and storage (such as multimedia applications). In this kind of applications, the control flow and accesses to the DDTs are unknown at compile-time, and run-time analysis and exploration becomes necessary.

In this paper, we propose to use a fast, stepwise, cost-driven exploration and refinement for the DDTs in multimedia applications at the system-level, where the impact on memory performance is the most important part.

3. Demonstrator

The 3D image reconstruction algorithm used as case study in this paper (to show in detail the applicability and flow of our approach) is heavily characterised by intensive internal dynamic memory use. This metric 3D-reconstruction from video algorithm [1] allows the reconstruction of 3D scenes from images and requires no other information apart from multiple frames. This makes the code especially useful for situations where extensive 3D setup with sensitive equipment is extremely difficult, e.g., crowded streets or remote locations, or impossible as when the scene is no longer available [2, 3].

For quick on-site visualisation and processing of more frames for a more detailed reconstruction, speeding up the application is necessary and demands extensive code transformations and optimisations. Moreover, energy consumption is paramount for hand-held visualisation devices.

Within the application framework of our methodology, we depict a typical dedicated system for intensive numeric processing (see Fig. 1). An instance of this can be a representative example of the platforms used for multimedia applications, where the data is immediately transferred into memory via DMA and the execution is triggered by a software interrupt. When processing is finished, another software interrupt is fired to handle the processed data.

The software module used as our driver application is one of the basic building blocks in many current 3D vision algorithms: *feature selection and matching*.

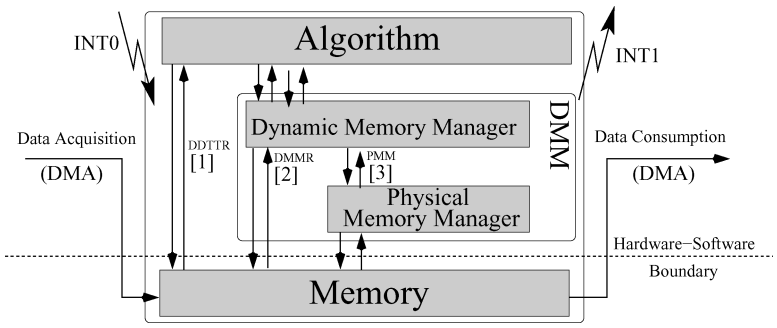


Figure 1. Overview of the system design in the optimisations.

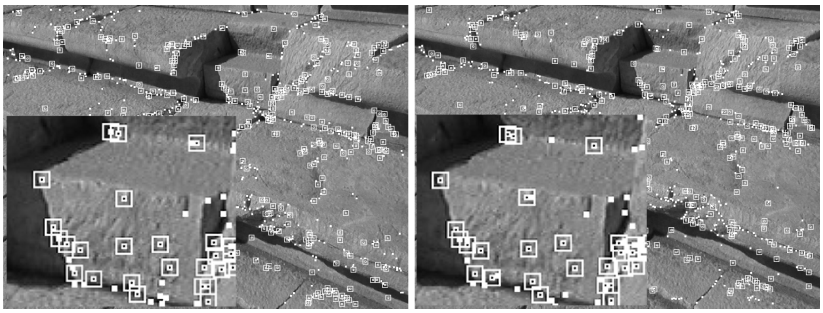


Figure 2. Initialisation of the matching of corners on two images of the steps of amphitheatre (archaeological site): based on neighbourhood search. Already most matches seem to be correct (partially due to the minor difference between the images, which can be seen at the right hand bottom corner). Part of the centre is enlarged.

The algorithm selects and matches features (corners) on different images and the relative offsets of these features define their spatial location (see Fig. 2).

The number of generated candidate matches is highly dependent on a number of factors. Firstly, the *image properties* affect the generation of the matching candidates. Images with highly irregular or repetitive areas will generate a large number of localised candidates, while a low number will be detected in other parts of the image. Secondly, the *corner detection parameters* have a profound influence on the results (and, consequently, on the input to the subsequent matching algorithm) because they affect the sensitivity of the algorithm used to identify the interesting features in the images/frames [21]. Finally, the *corner matching parameters* that determine the matching phase have a profound influence and are changed at runtime (e.g. the acceptance and rejection criterion changes over time as more 3D information is retrieved from the scene).

Taking all the previous factors into account, the possible combinations of parameters in the system make an accurate estimation of the memory cost, memory accesses and energy dissipation at compile time very hard or nearly impossible. This unpredictable memory behaviour can be observed in many state-of-the-art 3D vision algorithms [22] and multimedia algorithms because they use some sort of *candidate selection* followed by *a criterion evaluation*.

4. Memory Performance

In this paper, we will not focus on the static data (images and detected points), but on the internal DDTs of the algorithm. The optimisation of the transfers and accesses to the static data can be done by other techniques [23, 24]. The memory performance and behaviour of the 3D module is characterised by the following DDTs:

1. `ImageMatches` is the list of pairs where, for every point in the first image, (new) matches on a second image are considered based on neighbourhood or epipolar distance [1].
2. `CandidateMatches` is the list of candidates that needs evaluation, e.g. normalised cross correlation of a a window around the points [22].
3. `MultiMatches` is the list that stores the match pairs that pass the evaluation criterion. In this list, one point can have still multiple counterparts on the other image.
4. `BestMatches` is a subset of `MultiMatches` and retains only the best match for a point, according to the criterion already mentioned (if the evaluation is satisfied).

All these DDTs were originally implemented using a double linked list and exhibit typical *data consumption* and *generation behaviour*. Figure 3 shows the interaction of the DDTs in the algorithm code. From the images, corners are selected. On each corner of one image, a neighbourhood search is done and the pairs that pass a threshold are stored in `ImageMatches`. Based on information of previous frames, these are accepted into `CandidateMatches`. Every pair in `CandidateMatches` is checked by a *normalised cross correlation* and stored in `MultiMatches` and `BestMatches`. These results are passed to the next software module involved in the 3D reconstruction algorithm. It is important to mention that, although in this phase of the algorithm the image is still being accessed, these accesses are randomised. As such, classic optimisations

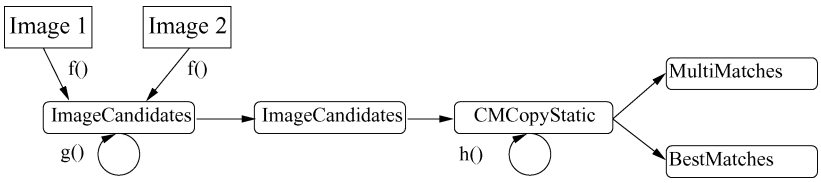


Figure 3. The interaction of the dynamic variables.

like row dominated accesses versus column wise accesses and other image access optimisations are not relevant. Furthermore, this algorithm variant uses the variant that re-uses the intermediate data in later steps.

5. Memory Management

In order to optimise the dynamic memory performance, our methodology uses a layered approach. It starts from a high level specification and adds more system-specific information in each step. This new information is used to refine previously made estimations. This approach allows to obtain an early idea of representative implementation costs and requirements without going through the entire expensive design process.

Figure 1 shows the three most important steps to optimise the aforementioned dynamic memory in the methodology. Firstly, the approach starts with the premise that the algorithm interacts directly with the memory, this is the Dynamic Data Type Transformation and Refinement step (DDTTR). Secondly, memory allocators that handle and optimise the dynamic memory de/allocations are added to the design space evaluation in the Dynamic Memory Management Refinement step (DMMR). Finally, a physical memory manager layer is added to optimise further, e.g. solve bank conflicts and introduce paging; this is the Physical Memory Management step (PMM). Because the last step tackles specific system and hardware information with all their complexities, it falls outside the scope of this paper.

5.1. *Dynamic Data Type Transformation and Refinement*

DDTs define the way in which the memory is allocated, accessed and free. Initially, simple DDTs are modelled by Data Types (DTs), e.g. lists, arrays or graphs, in combination with operations like `add`, `remove` and `get`. Then, these simple DDTs can be combined in layered structures that offer a compromise between flexibility, memory use and access time. For instance, a linked list solution is very flexible, but slow in access, while an array offers fast access, but it is hard to maintain and rigid in size. Highly optimised programs combine these simple data types in some sort, but these decisions are rarely taken in a methodological way and are left to the *experience* and *inspiration* of the programmer. Our methodology makes these decisions in a systematic way. In order to do this and identify the optimal DDT implementation for each variable, relatively detailed information of the dynamic memory behaviour at run-time is required.

The results described in this paper are based on DDTs that are modelled in "low level" `C++`, which means that it provides some object oriented conceptual benefits, while the code is still easy to convert to `C` (the code size overhead we have obtained due to our "low level" `C++` compared to `C` is really negligible, i.e. 2% or 3% on average), the target language of many SoC integration flows. The choice also allows minimal changes in the `C` algorithm code to integrate the DDTs.

The `C++` code used is actually only a subset of the language and is still relatively close to what is expected further down the optimisation flow and more to the hardware level: `C`. The main constructs that are used are:

1. *simple template classes* allow to write program code, making abstraction from the type. This is especially useful for writing *containers*: there is little difference from a coding perspective between the access of an array of `int`s or `double`s or even more complex data structures. Template code allows to postpone this decision until it is known what types are going to be used in the code by the program: at compile time. At that point, the template instantiation mechanism embedded in the `C++` compiler will specialise and instantiate the code and create intermediate, type specific code [25].

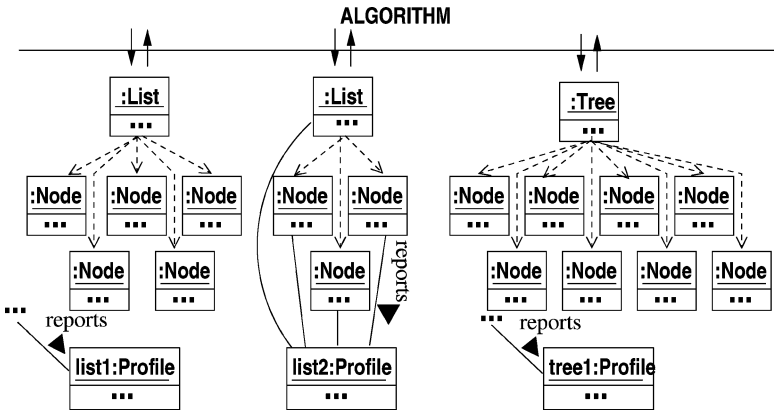


Figure 4. An algorithm uses multiple classes. Each object reports to profile object as defined while creating the algorithm DDT. For clarity purposes, not all associations are shown in the left and right DDT.

2. *simple derived classes* allow to postpone the decision of the precise implementation of a particular base class until at runtime. For exploration of the optimal DDT implementation this is useful because the program can be run and re-initialised with other derived sibling classes without expensive recompilation and relinking of the program.

The tool developed to support this step has two main building blocks. First, *profile objects* are added and supported by an object oriented profiling framework. This way, the use of combined layered DDTs is made transparent to the algorithm since all memory behaviour (allocations, accesses, ...) is contributed to the complex DDT, and not to the composing simple DDTs. With the creation of a DDT, a *profile object* is created. From that point onward, the *DDT master* object passes a reference down to each object it creates, and so forth (see Fig. 4).

For example, one possible DDT implementation for `CandidateMatches` can be seen in Fig. 5. It is an array to do the first lookup. Then, each position in the array points to a tree, which points to an array that finally stores the data (pairs of related points in the frames). When the memory behaviour of a DDT is evaluated, the developer is not only interested in the contribution of the array type in the DDT, but rather what the combined cost of this complex structure is. Finally, this profile framework, together with all the required code transformations and instrumentation are added automatically to the code.

The combination of multiple DTs in multiple layers and the use of multiple DDTs in an application results

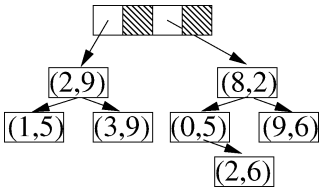


Figure 5. Example of a simple layered DDT. The organisation can be more complex as the DDT organisation is adjusted to the data access requirements of the application at hand.

in an exponential search space. Currently, some simple heuristics (e.g. limited ranges of values for size of the basic blocks some of the DDTs) in are employed to automatically explore the search space, using a representative input data set. The evaluation of the solutions is based on multiple objectives. As such, an *optimal solution* in the classical sense can normally not be identified. A better approach is to use Pareto optimal points. A point is said to be Pareto optimal if it is not longer possible to improve upon one cost factor without worsening any other. As a result of the exploration process, Pareto optimal solutions are located, based on normalised memory use, memory accesses and energy estimates (see Figs. 6 and 7) and the final solution depends on the restrictions of the designer and system. If constraints change (e.g., the available cycle budget), a new optimal Pareto point can be indicated. By modifying the heuristics, more details about (sub-optimal) DDTs can be obtained, at the cost of an increased exploration time.

Finally, the automated post-processing generates a list of Pareto solutions and each DDT is linked with a

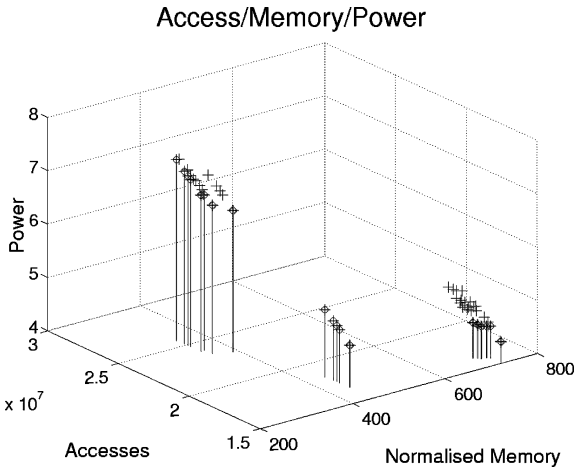


Figure 6. This figure shows a typical set of global Pareto solutions (Power in mW, Normalised memory in Bytes) as combinations of Pareto optimal solutions for every DDT in the application under study. They form a Pareto curve.

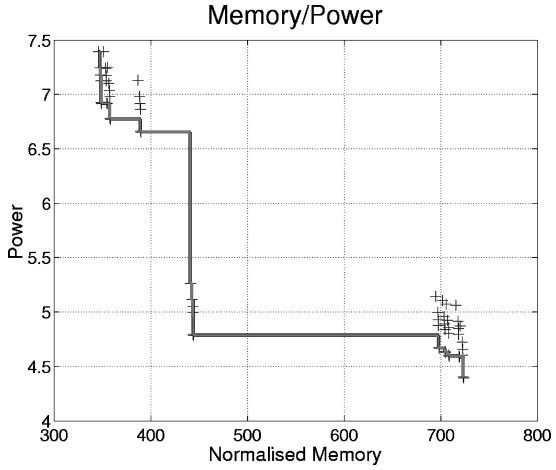


Figure 7. This figure shows a projection of the 3D space of Pareto optimal solutions in the Power (mW) / Memory (Bytes) plane.

particular DDT implementation according to the system constraints.

5.2. *Dynamic Memory Management Refinement*

The purpose of DMMR is an efficient use of the the dynamic memory available in the system for certain constraints. An inattentive management of the DDTs of the application can lessen severely the performance of the whole system and increase the memory accesses and the energy dissipation. Current general-purpose dynamic memory managers include inside them a very broad range of mechanisms and policies [8]. This fact allows them to accomplish a relatively good trade-off between performance and memory footprint in general-purpose systems (e.g., desktops), where the applications that are going to be executed are unknown at design time. Hence, these dynamic memory managers are very complex in their internal organisation and design (e.g., fit algorithms, several data structures for the free blocks, etc. [8]) and not really optimised for a particular dynamic memory behaviour, consuming thus a lot of available resources in the system [26]. However, for embedded systems, the dynamic memory managers must be implemented inside their constrained operating system making use of the knowledge of the application (or set of applications) that will run on the platform (e.g. 3D games, video algorithms, etc.). Then, a well-adjusted custom dynamic manager can be designed for the multimedia application under study taking really into account the limited resources available in the system.

In order to select efficiently the dynamic memory manager, first we analyse the run-time dynamic memory behaviour of the application under study. Then, we design a custom dynamic memory manager for this specific behaviour taking into account a wide range of high-level strategic issues [8, 27, 28]. For example, the organisation overhead and internal data structures of the dynamic memory manager, fit algorithms (e.g., best fit, first fit, etc.), additional mechanisms to prevent and eliminate fragmentation (e.g., coalescing or splitting mechanisms, etc. The final dynamic memory manager has to allow certain trade-offs between performance of the application and other constraints (e.g., memory footprint, power consumption, etc.). Within this context, the profiling information obtained in the previous steps of the methodology is used to characterise the evolution in time of the DDTs involved in the application and its de/allocation pattern (see Fig. 8).

The information about the DDTs is characterised in this set of high-level strategic issues, which tackles different parts of any possible allocator [8] for manual memory management. The choices from this set of high-level strategic issues are combined to eventually constitute global dynamic memory managers according to the constraints of the system.

After that, we explore the different candidates of suitable dynamic memory managers using a fast infrastructure of `C++` mixin layers [26], which allows to compose and create these managers with very complex

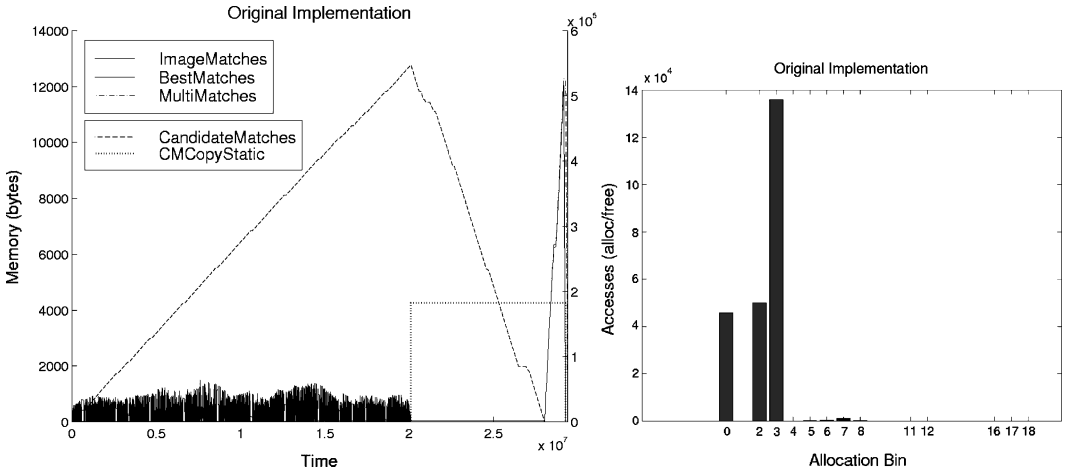


Figure 8. On the left, memory footprint over time (micro-secs) in the original implementation. All plots mapped on the left axis, except `CandidateMatches` and `CMCopyStatic` (right one). On the right, the use of different block-size allocation for the original algorithm.

de/allocation strategies [26]. For example, first fit with address ordered for the free blocks and deferred coalescing allowed [8]. This allows fast changes in the features of dynamic memory managers (e.g. replacement policies or sizes of the pools). Due to mixin based template instantiation, localised changes for detailed profiling can be made by inserting additional layers. An example is shown in Fig. 11, where a simplified definition of our *Loglayer* is added to the Kingsley allocator [8] to obtain information of what memory is requested from the system and recycled (fragmentation). The infrastructure of layers and profiling objects is used to heuristically explore the values in many characteristics of the DMM candidates and select the ideal one for the application.

6. Results

As we have already explained, optimising the memory management at system-level in multimedia applications with complex dynamic behaviour has not been given much attention. Therefore, our methodology will be used to refine the 3D image reconstruction system used as our case study.

In the first step, DDTTR, the representation of the four main DDTs described in Section 4 is optimised. After running the tool to get the profiling information from the original code, memory use graphs are generated (see left chart in Fig. 8). In order to account for the varying memory use during the program run, normalised memory is used to get an estimation of the overall contribution of each DDT to the energy dissipation (see Table 1). This attributes more accurate contributions to energy cost estimates and avoid that e.g. DDTs with very short and high memory usage distort and hide the memory contribution from other DDTs. The model used in the results to obtain energy estimations is described in [29] for large SRAMs with .25 μm technology. This model depends on memory footprint factors (i.e. memory size, internal structure of banks and sub-banks, memory leaks, working time of the memory and technology) and energy consumption factors created by memory accesses (i.e. number of memory accesses, energy consumption in active mode, size of the memory and technology). In the tools, the memory model can be replaced by others in a modular way.

A first analysis of the run-time profiling information of Fig. 1 shows the existence of a dynamic array

Table 1. Dynamic memory use from DDTs in the original implementation.

Variable	Memory accesses	Memory footprint (B)	Energy (nJ)
`ImageMatches`	1.201×10^6	0.340×10^3	2.162×10^4
`CandidateMatches`	8.442×10^5	2.486×10^5	3.039×10^5
`CMCStatic`	9.140×10^7	6.247×10^4	1.695×10^7
`MultiMatches`	1.849×10^4	0.678×10^3	3.328×10^3
`BestMatches`	1.664×10^4	0.623×10^3	2.996×10^3
`Total`	9.348×10^7	3.127×10^5	1.728×10^7

copy of `CandidateMatches` in the original code to optimise the vast amount of accesses to this DDT. This copy is the DDT `CMCopyStatic (CMCStatic)`. Also three dynamic sets form clear bottlenecks. First of all, `CandidateMatches` is the largest dynamic data structure in the application. Secondly, `ImageMatches` has a low normalised memory use, but it is accessed extensively. Finally, the `CMCStatic` *speed optimisation dynamic array* accounts for the most important part of the memory accesses and consumes most of the energy in the application.

After the subsequent exploration step, our tool suggested an ideal solution for the different DDTs according to the constraints entered. We used minimal energy dissipation and two layered DDT structures (pointer-arrays to arrays) with array sizes of 756, 1024 and 16384 Bytes (B) were selected. As Fig. 9 depicts, the DDT refinement already attains a significant influence on performance and dynamic memory footprint.

Finally, because of the optimised DDTs found in the exploration, it is possible to remove `CMCStatic` and refine even more the algorithm combining it with `CandidateMatches` (for more details, see [30]). After this, the figures in Table 2 and Fig. 10 are generated. They show that the accesses to `ImageMatches` are less than half of the original ones and the normalised memory use of `CandidateMatches` is 43.9% less. The removal of `CMCStatic` influences the normalised memory footprint (and removed the short memory peak used while copying), but has little effect on the performance (speed) of the overall program. This shows the importance of DDTTR to speed up the application comparing the values of the X-axes from Figs.8–10 to match two images (time values of $\times 10^7$ microseconds in the first plot, in the refined ones with our methodology only $\times 10^5$ microseconds). Furthermore, DDTTR reduces memory footprint and power of the DDTs, and allow further global refinements.

Table 2. Dynamic memory use from relevant DDTs after DDTTR.

Variable	Memory accesses	Memory footprint (B)	Energy (nJ)
`ImageMatches`	4.020×10^5	0.624×10^3	7.243×10^4
`CandidateMatches`	3.898×10^5	1.174×10^5	7.017×10^4
`MultiMatches`	7.684×10^3	1.391×10^3	1.383×10^3
`BestMatches`	7.161×10^3	1.368×10^3	1.289×10^3
`Total`	8.066×10^5	1.208×10^5	1.452×10^5

In the next DMMR phase, an optimised dynamic memory manager is selected. After the exploration has been performed using the profiling information from the previous steps and our search space, the dynamic memory manager selected discerns the different behaviours of the DDTs in the case study, which are just a few now as the right plot of Fig. 10 shows. It partitions the dynamic memory of the system in three different regions or pools with different sizes that suit the DDTs in the application. Inside every block allocated, the object size is recorded and, for every region, a double linked list is used to lessen the time required to traverse and

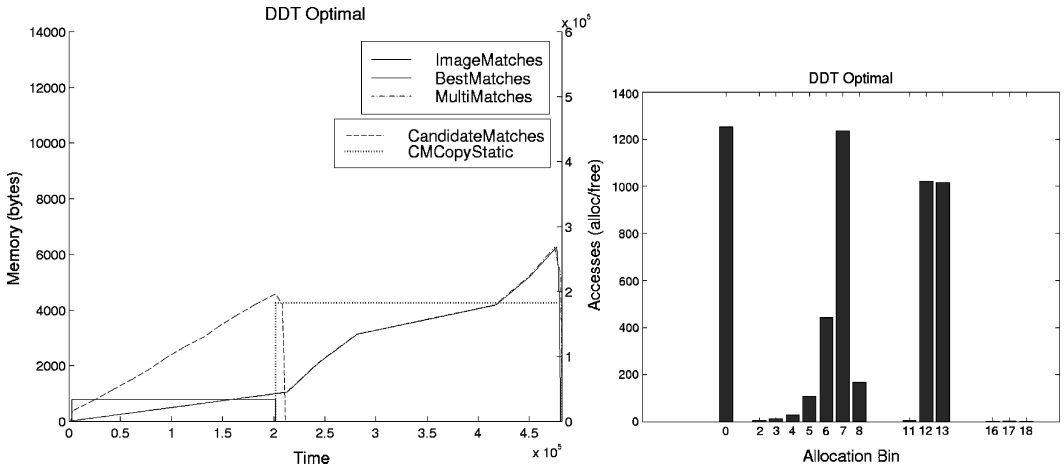


Figure 9. On the left, memory footprint over time (micro-secs) in the Pareto implementation with `CMCopyStatic`. All plots mapped on the left axis, except `CandidateMatches` and `CMCopyStatic` (right one). On the right, the use of different block-size allocation for this version.

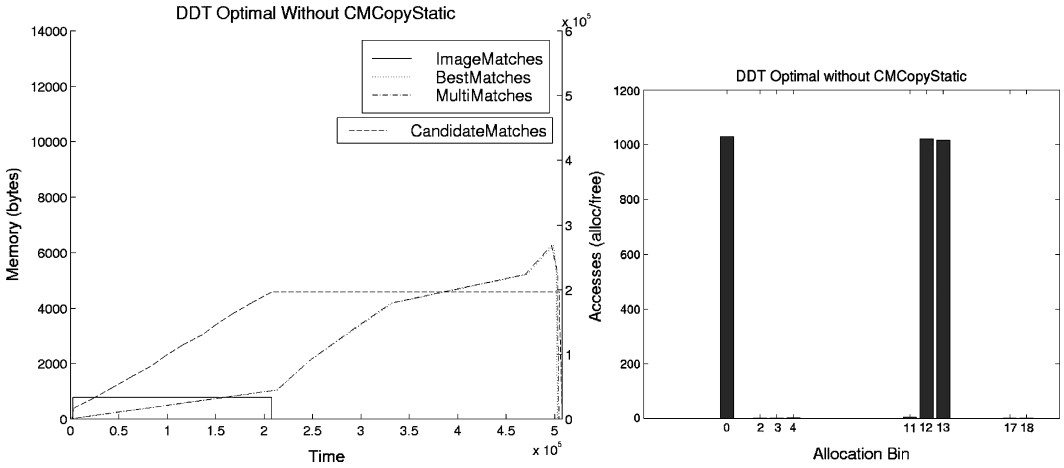


Figure 10. On the left, memory footprint over time (micro-secs) in the Pareto implementation after removing `CMCopyStatic`. All plots mapped on the left axis, except `CandidateMatches`. On the right, the use of different block-size allocation for this final version.

```
template <class SHeap> class LogLayer:\
     public SHeap, public _profile{
 public:
  inline void* malloc(size_t sz){
   EPRINT("(%d) malloc %d bytes\n", id, sz);
   return SupHeap::malloc(sz); }
  inline void free(void* ptr){
   EPRINT("(%d) free %d bytes\n",id,SHeap::getSize(ptr));
   SuperHeap::free(ptr); } };

class Kingsley KingsleyNC<SbrkHeap>{};
class KingsleyLogged KingsleyNC<LogLayer<SbrkHeap> >{};
```

Figure 11. Definition and use of a LogLayer

access the data. Furthermore, the blocks are stored inside each sublist of sizes using a LIFO order. When a block is freed, the manager returns the deallocated memory to the appropriate region and it becomes available for block requests inside the specific range of sizes allowed in this pool. It provides an excellent performance and fragmentation is minimised. It only adds approximately a 10% overhead in normalised memory footprint and 18% in energy consumption compared to the results of the bare DDTs shown in Table 2 due to internal management and fragmentation. These final values (e.g. memory footprint, energy consumption) substantially improve the results obtained in our experiments with state-of-the-art general-purpose dynamic memory managers. For example, the Lea Allocator [28] (typical in Linux-based systems) added 30% overhead in memory footprint and 78% overhead in energy consumption to the bare DDTs, or the Kingsley manager [8, 9] (from Windows-based systems) added 90% overhead in memory footprint and 28% in energy consumption.

Finally, in addition to the original implementation with dynamic memory, we have created a manually optimised version of the algorithm. We consider that the designer, after manually profiling and extracting the necessary information from the original code (for the full code of the entire 3D algorithm with 1.75 million lines of high level `C++`, see [22]), was able to select an optimal static DT representation of the relevant DDTs. This static version achieves very good performance and the energy consumed is much lower than the original version, as Table 3 shows. However, since it is really optimised for a specific configuration of parameters, it does not scale for larger resolutions, different parameters or abnormal images features.

After both steps (DDTTR and DMMR), the total memory used in the application is greatly improved compared to the manually optimised version and the original one, as shown in Table 2. The DDTs require less dynamic memory than the original version (see

Table 3. Dynamic memory use from DDTs in the manually optimised version.

Variable	Memory accesses	Memory footprint (B)	Energy (nJ)
`ImageMatches`	4.020×10^5	0.857×10^3	7.243×10^4
`CandidateMatches`	3.151×10^5	2.021×10^5	1.134×10^5
`MultiMatches`	5.856×10^3	6.876×10^3	1.054×10^3
`BestMatches`	4.913×10^3	6.876×10^3	0.884×10^3
`Total`	7.282×10^5	2.167×10^5	1.878×10^5

Table 1, Figs. 8 and 10) and the manually optimised version (see Table 3). The energy consumed is also reduced significantly comparing with the best version, i.e. the manually optimised version (see Tables 2 and 3). Finally, compared to the original version the memory accesses are reduced enormously, as Tables 1 and 2 show. Summing up, with the whole methodology applied, compared to the manually optimised version, the memory footprint improves up to 44.2% and estimated energy dissipation up to 22.6%. In addition, the system can scale with extreme-cases of input parameters whereas the manually optimised version cannot do so.

7. Conclusions

One of the crucial and most difficult parts in multimedia applications is DMM. In this paper, we prove the effectiveness of a new system-level exploration methodology to optimise the aforementioned DMM for typical multimedia applications applying it to a relatively new 3D reconstruction algorithm. This methodology allows a structured analysis of the memory access patterns hidden in algorithms with complex dynamic memory use and can also help to solve fundamental algorithmic problems. It allows the system integrator to obtain a detailed and clear view of the dynamic memory behaviour and optimise it. In a first phase, the way in which the data is stored for the algorithm studied is optimised. By doing this, the access patterns to memory are transformed and optimised, reducing power consumption and memory footprint. Since most embedded systems do not have complex memory management provided by a combination of hardware and system software, an approach is proposed to compose efficient custom dynamic memory managers in a modular way, using high-level `C++` code. Even though this paper only presents one driver application in detail for simplicity and accuracy purposes, similar results have been obtained in game engine algorithms [31].

Acknowledgments

This work is partially supported by the Fund for Scientific Research - Flanders (Belgium, F.W.O.) through project G.0036.99 and a Postdoctoral Fellowship for Geert Deconinck. Furthermore, this work is partially supported by the Spanish Government Research Grant TIC2002/0750.

Note

1. The sum of the memory used at a time slice, multiplied by the time. This amount is then divided over one run of the algorithm

References

1. M. Pollefeys, R. Koch, M. Vergauwen, and L. Van Gool, "Metric 3D Surface Reconstruction from Uncalibrated Image Sequences," in *Lecture Notes in Computer Science*, vol. 1506, 1998, pp. 139–153.
2. J. Cosmas, I. Taki, D. Green, O. Zalesny, L. Van Gool, M. Pollefeys, R. Degeest, M. Waelkens, K. Hraby, M. Kampel, and R. Sablatnig, "3D Murale," 2002. `http://www.brunel.ac.uk/project/murale/home.html`.
3. Eyetronics, "Eyetronics 3D Scanning Solutions," `http://www.eyetronics.com`.
4. R. Rowe, "Industrial Light and Magic," *Linux Journal*, vol. 99, 2002, pp. 32–36.
5. F. Catthoor, K. Danckaert, C. Kulkarni, E. Brockmeyer, P. Kjeldsberg, T. Van Achteren, and T. Omnes, *Data Access and Storage Management for Embedded Programmable Processors*, Boston, USA: Kluwer Academic Publishers, 2002.
6. N. Vijaykrishnan, M. Kandemir, M.J. Irwin, H.S. Kim, W. Yw, and D. Duarte, "Evaluating Integrated Hardware-Software Optimizations Using a Unified Energy Estimation Framework," *IEEE Transactions on Computers*, vol. 52, no. 1, 2003, pp. 59–75.
7. id, "id Software Inc." 2002. `http://www.idsoftware.com`.
8. P.R. Wilson, M.S. Johnstone, M. Neely, and D. Bowles, "Dynamic Storage Allocation, A Survey and Critical Review," in *Internation Workshop on Memory Management*. Kincross, Scotland, UK, 1995.
9. E.D. Berger, B.G. Zorn, and K.S. McKinley, "Composing High-Performance Memory Allocators," in *Proceedings ACM SIGPLAN Conference on Programming Language Design and Implemenatation (PLDI)*, Snowbird, Utah, 2001.
10. E.G. Daylight, S. Wuytack, F. Catthoor, and C. Ykman-Couvreur, "Analyzing Energy Friendly Steady State Phases of Dynamic Application Execution in Terms of Sparse Data Structures," in *Proceedings of ISLPED 2002*, Monterrey, California, USA, 2002.
11. R. Henriksson, "Scheduling Garbage Collection in Embedded Systems," Ph.D. thesis, Lund Institute of Technology, 1998.
12. N. Murphy, "Safe Memory Usage with Dynamic Memory Allocation," *Embedded Systems*, pp. 49–57, 2000.
13. S. Wuytack, J.L. da Silva Jr., F. Catthoor, G. de Jong, and C. Ykman, "Memory Management for Embedded Network Applications," *IEEE Transactions on Computer-Aided Design*, vol. 18, no. 5, 1999, pp. 533–544.
14. L. Benini and G. De Micheli, "System Level Power Optimization Techniques and Tools," in *ACM Transaction on Design Automation for Embedded Systems (TODAES)*, 2000.
15. V. Tiwari, S. Malik, and A. Wolfe, "Power Analysis of Embedded Software: A First Step Towards Software Power Minimization," in *Proceedings of ICCAD*, San Jose, California, USA, 1994.
16. D. Sarta, D. Trifone, and G. Ascia, "A Data Dependent Approach to Instruction Level Power Estimation," in *Proceedings of IEEE*

Alessandro Volta Memorial Workshop on Low-Power Design, Como, Italy, 1999, pp. 182–190.
17. M.T.-C. Lee, V. Tiwari, S. Malic, and M. Fujita, "Power Analysis and Minimization Techniques for Embedded DSP Software," *IEEE Transactions on Very Large Scale Integration Systems*, 1997, pp. 123–135.
18. T. Tan, A.K. Raghunathan, G. Lakishminarayana, and N.K. Jha, "High-Level Software Energy Macro-Modeling," in *Proceedings of the 38th Design Automation Conference (DAC)*, Las Vegas, NV, USA, 2001, pp. 605–610.
19. I. Kadayif, N. Vijaykrishnan, M.J. Irwin, and A. Sivasubramaniam, "EAC: A Compiler Framework for High-Level Energy Estimation and Optimization," in *Proceedings of Design, Automation and Test in Europe*, Paris, France, 2002, pp. 436–442.
20. R.Y. Chen and M.J. Irwin, "Architecture-Level Power Estimation and Design Experiments," *ACM Transactions on Design Automation of Electronic Systems*, vol. 6, no. 1, 2001.
21. C.J. Harris and M. Stephens, "A Combined Corner and Edge Detector," in *4th Alvey Vision Conference*. Manchester, 1988, pp. 147–151.
22. Target Jr, 'Target Jr'. 2002. `http://www.targetjr.org`.
23. T.M. Chilimbi, M.D. Hill, and J.R. Larus, "Cache-Conscious Structure Layout," in *SIGPLAN Conference on Programming Language Design and Implementation*, 1999, pp. 1–12.
24. P.R. Panda, N. Dutt, and A. Nicolau, "Memory Organization for Improved Data Cache Performance in Embedded Processors," in *1996 International Symposium on System Synthesis*, La Jolla CA, 1996, pp. 90–95.
25. D. Vandevoorde and N.M. Josuttis, *C++ Templates, The Complete Guide*, London, UK: Addison Wesley, 2003.
26. D. Atienza, S. Mamagkakis, M. Leeman, F. Catthoor, J.M. Mendías, D. Soudris, and G. Deconinck, "Fast System-Level Prototyping of Power-Aware Dynamic Memory Managers for Embedded Systems," in *Proceedings of Workshop on Compilers and Operating Systems for Low Power*, New Orleans, LA, USA, 2003.
27. M.S. Johnstone and P.R. Wilson, "The Memory Fragmentation Problem: Solved?" in *Proceedings of the International Symposium on Memory Management*, Vancouver, British Columbia, 1998.
28. D. Lea, "The Lea 2.7.2 Dynamic Memory Allocator," 2002. `http://gee.cs.oswego.edu/dl/`.
29. B.S. Amrutur and M.A. Horowitz, "Speed and Power Scaling of SRAM's," *IEEE Transactions on Solid-State Circuits*, vol. 35, no. 2, 2000.
30. M. Leeman, D. Atienza, F. Catthoor, G. Deconinck, J. Mendias, V. De Florio, and R. Lauwereins, "Intermediate Variable Elimination in a Global Context for a 3D Multimedia Application," in *Proceedings of International Conference on Multimedia and Expo*, Baltimore, MD, 2003a.
31. M. Leeman, D. Atienza, F. Catthoor, G. Deconinck, J.M. Mendias, V. De Florio, and R. Lauwereins, "Power Estimation Approach of Dynamic Data Storage on a Hardware Software Boundary Level," in *Integrated Circuits and System Design—Power and Timing Modelling, Optimization and Simulation, 13th International Workshop*, vol. 2799 of *Lecture Notes in Computer Science*, Turin, Italy, 2003b, pp. 289–298.

Marc Leeman has as professional research interests hardware/software co-design, code optimisation in general and optimisation of dynamic data types and dynamic memory management for low power embedded systems in particular. Personal interests include Open and Free software development, software configuration and GNU/Debian package maintenance. He received an engineering degree, a master in artificial intelligence and a Ph.D. in electrical engineering in 1997, 1998 and 2004 respectively, all at the K.U. Leuven. He is a member of the IEEE Computer Society. Currently, he works as an R&D Engineer for Barco Control-rooms Division (BCD) on hardware/software co-design for streaming video products.
marc.leeman@ieee.org

David Atienza received the M.Sc. degree in Computer Sciences from the Complutense University of Madrid (UCM), Spain in 2001. Since then he has joined the Department of Computer Architecture and Automation of Complutense University of Madrid as a sandwich Ph.D. student half-time at the Inter-university Micro-Electronics Centre (IMEC), Heverlee, Belgium. His research interests include optimisation of dynamic memory management on multimedia and wireless network applications for low power and high performance embedded systems, computer architecture and high-level design automation.
datienza@dacya.ucm.es

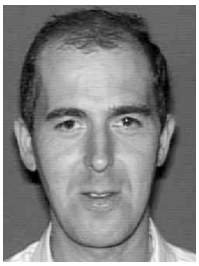
Geert Deconinck is Associate Professor (hoofddocent) at the K.U. Leuven (Belgium) since 2003 and staff member of the research group

ELECTA (Electrical Energy and Computing Architectures). His research interests include the design and assessment of software-based solutions to meet dependability, real-time, and cost constraints for embedded systems. In this field, he has authored and co-authored more than 120 publications in international journals and conference proceedings. He received his M.Sc. in Electrical Engineering and his Ph.D. in Applied Sciences from the K.U. Leuven, Belgium in 1991 and 1996 respectively. He was a visiting professor (bijzonder gastdocent) at the K.U. Leuven in 1999–2003. - Flanders (Belgium) in the period 1997–2003.
geert.deconinck@esat.kuleuven.ac.be

Vincenzo De Florio received his MSc degree in computer science in 1987 and his PhD degree in engineering in 2000, respectively from the University of Bari, Italy, and the University of Leuven, Belgium. He is currently post-doctoral researcher at the University of Antwerp, where he is doing research on adaptive and dependable mobile applications. Previously he had been researcher and lecturer with Tecnopolis/SASIAM (ECMI School for Advanced Studies in Industrial and Applied Mathematics) and member of Tecnopolis/Robotic lab, where he was responsible for design of parallel robotic vision applications. Currently he is also a reviewer for several conferences and for the Journal of System Architectures.
vincenzo.deflorio@ua.ac.be

José M. Mendías received the M.Sc. and Ph.D. degrees in physics from the Complutense University of Madrid in 1992 and 1998, respectively. He joined the Department of Computer Architecture and Systems Engineering, Complutense University in 1992 as a lecturer, and became an associate professor in 2001. Since 2002, he is Vicedean of the Computer Science Faculty at the same University. His current research interests include design automation, computer architecture and formal methods.
mendias@dacya.ucm.es

Chantal Ykman-Couvreur is born in 1956. She received the mathematics degree from the "Facultes Universitaires Notre-Dame de la Paix" of Namur in 1979. She first worked at PHILIPS Research Laboratory of Belgium, from 1979 until 1991. Her main activities were concentrated on information theory and coding, cryptography and multi-level logic synthesis for VLSI circuits. Then, she joined IMEC, where she was responsible at IMEC for the dynamic memory management and the system-level design flow in the Matisse compiler for network protocol components (ATM, Internet Protocol, etc). Currently, she works on the task concurrency management design flow in the Matador project.
ykman@imec.be



Francky Catthoor received the engineering degree and a Ph.D. in electrical engineering from the Katholieke Universiteit Leuven, Belgium in 1982 and 1987 respectively. Since 1987, he has headed several research domains in the area of high-level and system synthesis techniques and architectural methodologies, all within the Design Technology for Integrated Information and Telecom Systems (DESICS—formerly VSDM) division at the Inter-university Micro-Electronics Centre (IMEC), Heverlee, Belgium. Currently he is an IMEC fellow. He is part-time full professor at the EE department of the K.U. Leuven.

In 1986 he received the Young Scientist Award from the Marconi International Fellowship Council. He has been associate editor for several IEEE and ACM journals, like Transactions on VLSI Signal Processing, Transactions on Multi-media, and ACM TODAES. He was the program chair of several conferences including ISSS'97 and SIPS'01.
catthoor@imec.be

Rudy Lauwereins is vice-president of IMEC, Belgium's Interuniversity Micro-Electronic Centre, which performs research and development, ahead of industrial needs by 3 to 10 years, in microelectronics, nano-technology, enabling design methods and technologies for ICT systems. He leads the DESICS division of 185 researchers, currently focused on the development of re-configurable architectures, design methods and tools for wireless and multimedia applications. He is also a part-time Professor at the Katholieke Universiteit Leuven, Belgium. He had obtained a Ph.D. in Electrical Engineering in 1989. Rudy Lauwereins served in numerous international program committees and organisational committees, and gave many invited and keynote speeches. He is vice-chair of the board of DSP Valley and member of the board of several spin-off companies. He is a senior member of the IEEE.
lauwerei@imec.be