The Journal of Systems and Software 82 (2009) 590–602

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Optimization methodology of dynamic data structures based on genetic algorithms for multimedia embedded systems

Christos Baloukas [a,*], Jose L. Risco-Martin [d], David Atienza [d,e], Christophe Poucet [b,c], Lazaros Papadopoulos [a], Stylianos Mamagkakis [b], Dimitrios Soudris [a], J. Ignacio Hidalgo [d], Francky Catthoor [b,c], Juan Lanchares [d]

[a] *VLSI Design and Testing Center, Democritus University of Thrace, 12 Vas. Sofias Street, 67100 Xanthi, Greece*
[b] *IMEC vzw, Kapeldreef 75, 3001 Heverlee, Belgium*
[c] *ESAT, K.U. Leuven, 3001 Heverlee, Belgium*
[d] *Department of Computer Architecture and Automation (DACYA), Complutense University of Madrid, 28040 Madrid, Spain*
[e] *Embedded Systems Laboratory (ESL), Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland*

ARTICLE INFO

Article history:
Received 18 January 2008
Received in revised form 15 August 2008
Accepted 15 August 2008
Available online 5 September 2008

Keywords:
Data structures
Dynamic memory
Pareto-front
Multi-objective optimization

ABSTRACT

Modern multimedia application exhibit high resource utilization. In order to efficiently run this kind of applications in embedded systems, the dynamic memory subsystem needs to be optimized. A key role in this optimization is played by the dynamic data structures that reside in every real-life application. This paper presents a novel and automated way to optimize dynamic data structures. The search space is pruned using genetic algorithms that converge to the best multilayered data structure implementation for the targeted applications.

1. Introduction

In future technologies of embedded systems an increasing amount of applications (e.g. 3D games, video-players) coming from the general-purpose domain, having large run-time memory management requirements, need to be mapped onto an extremely compact device. However, embedded systems struggle to execute these complex applications because they come from desktop systems, holding very different restrictions regarding memory usage features, and more concretely not concerned with an efficient use of the dynamic memory. In fact, a desktop computer typically includes today between 512 and 1024 MB of RAM memory at least, as opposed to the 32 or 64 MB present in modern embedded systems. Therefore, one of the main tasks of the porting process of multimedia applications onto embedded multimedia systems is the optimization of the dynamic memory subsystem.

In modern dynamic applications, dynamic data is stored in entities called containers, like arrays, lists or trees, which can adapt dynamically to the amount of memory used by each application (Wood, 1993). In multimedia applications sequences are the most used containers. This category includes arrays and lists. Since there are several implementations (called dynamic data types – DDTs) of these containers, as we show in Section 3.1, choosing an improper DDT will have significant negative impact on the dynamic memory subsystem of the embedded system (Bartzas et al., 2006) as in STL (SGI, 2006).

As a single application can host a number of different containers, to optimize the use of dynamic memory, the designer must choose the best among a number of possible DDT implementations (James and Mansfield, 1999; Wood, 1993), according to the specific restrictions of typical embedded design metrics, such as, performance, memory footprint and energy consumption. This task is typically performed using a pseudo-exhaustive evaluation of the design space of DDT implementations (i.e. multiple executions) for the application to attain the Pareto front (Daylight et al., 2004). That search would try to cover all the possible optimal implementation points for the aforementioned required design metrics. This exhaustive construction of the Pareto front is a very time-consuming, sometimes even unaffordable, process. Moreover, due to the inter-dependencies between DDTs, namely, that one DDT implementation behavior may affect the performance or memory footprint of another one (Daylight et al., 2004), the refinement process must explore the whole range of possible combinations of the different DDT implementations. Thus, the number of

* Corresponding author. Address: Salaminos 44, 67100, Xanthi, Greece. Tel.: +30 2541077459.
E-mail addresses: cmpalouk@ee.duth.gr, cmpalouk@gmail.com (C. Baloukas), jlrisco@dacya.ucm.es (J.L. Risco-Martin), datienza@dacya.ucm.es (D. Atienza), poucetc@imec.be (C. Poucet), lpapadop@ee.duth.gr (L. Papadopoulos), mamagka@imec.be (S. Mamagkakis), dsoudris@ee.duth.gr (D. Soudris), hidalgo@fis. ucm.es (J. Ignacio Hidalgo), catthoor@imec.be (F. Catthoor), julandan@fis.ucm.es (J. Lanchares).

experiments to be carried out typically becomes unaffordable even for a small number of DDTs. For instance, in the case of an embedded application including 10 different DDTs that need to be explored for 10 basic relevant implementations of DDTs for multimedia applications (as proposed in Atienza et al. (2004), Bartzas et al. (2006), Leeman (2003)), the number of experiments (i.e. multiple runs of the application) that need to be performed is 10^{10}; testing all these combinations manually is not feasible. This paper presents a novel, automated, optimization approach for the DDTs of multimedia applications. It relies on the definition and the analytical pre-characterization of the possible elementary DDT blocks, which are subsequently used in a *genetic algorithm (GA)* of type *vector evaluated genetic algorithm (VEGA)* (David Schaffer, 1985) to model the existing inter-dependencies of using different DDTs implementations. Then, this modeling of inter-dependencies can be seen as a constraint set. The latter can be used to prune the design space. This paper is organized as follows: in Section 2, we review related work on DDTs design and optimization. In Section 3, we present our multi-objective optimization framework. In Section 4, we present our experimental results with real-life multimedia embedded applications and compare with state-of-the-art optimization heuristics to optimize DDT applications. Finally, in Section 5, we summarize the contributions of the paper and present future research directions.

2. Related work

It is widely accepted that forthcoming multimedia applications will require dynamic memory in embedded systems due to their dynamic behavior (e.g. the number of objects rendered on the screen while playing can significantly vary). Therefore, important research work has been started already through the optimization of dynamic data storage for embedded systems (Daylight et al., 2004; Jerraya and Wolf, 2005).

Regarding DDT refinement, the Standard Template C++ Library (STL) (SGI, 2006) or other proposed template libraries (C++ Standardisation Committee, 1998) provide many basic data structures to help designers develop new algorithms without being worried about complex DDT implementation issues. However, these libraries usually provide interfaces to simple DDT implementations and the construction of complex ones is a responsibility of the developer. Furthermore, these libraries focus exclusively on performance. They can be considered as acceptable general-purpose solutions, but are not suitable for new generation embedded devices, where performance, energy consumption and memory footprint must be optimized together.

For embedded software, suitable access methods, power-aware DDT transformations and pruning strategies based on heuristics have been proposed for multimedia systems (Daylight et al., 2004; Wuytack et al., 1996). However, these approaches require the development of efficient pruning cost functions and fully manual optimizations. Otherwise, they are not able to capture the evaluation of inter-dependencies of multiple DDTs implementations operating together, as the proposed methodology using evolutionary computation achieves. Also, several transformations have been proposed that optimize local loops in embedded programs at compile time (Muchnick, 1997). Nevertheless, they are not suitable for exploration of complex DDTs employed in modern multimedia applications, because they handle only very simple data structures (e.g. arrays or pointer arrays), and mostly focus on performance.

In addition, according to the characteristics of certain parts of multimedia applications, several transformations for DDTs and design methodologies (Smailagic et al., 1995; Benini and De Micheli, 2000; Catthoor et al., 2002) have been proposed for static data profiling and optimization considering static memory access patterns to physical memories. In this context, the use of GA-based optimization has been applied to solve linear and non-linear problems by exploring all regions of the state space in parallel (Coello et al., 2002). Thus, it is possible to perform optimizations in non-convex regular functions, and also to select the order of algorithmic transformations in concrete types of source codes (Michalewicz, 1996; Houck et al., 1995; Osyczka, 1985). However, such techniques are not applicable in DDT implementations, due to the initially unpredictable nature of the data to be stored at compile-time, as does the optimization methodology that we present in this work.

Furthermore, in the available literature there has been an exhaustive cover of data structures characterization in terms of complexity (Cormen et al., 2001) of their operations. Although this approach is useful for a high level estimation of the DDT's performance, it cannot be used for a fine tuning process. For instance, random access for both singly and doubly linked lists is of O(n) complexity. For large number of elements, however, a doubly linked list gives half the accesses in comparison to singly linked list. The latter makes clear the need for an analytical approach, rather than using standard complexity analysis. Furthermore, none of these approaches studies DDTs in regard to their behavior. For instance, no discern of sequential vs random access has been made. In that sense, the author of Leeman (2003) presents an analytical characterization of a set of basic DDTs for random access. Although these models are close to our approach, the models used in our work are more realistic and accurate (especially in implementation variations that use roving pointers). Furthermore, the formal aspect of the extraction process for these models is presented here in a more complete way.

Recently, in the workshop paper (Atienza and Baloukas, 2007) we have shown the possible advantage of a simple implementation of a GA to perform DDT exploration. However, the application of our approach was limited to two applications and the complete automation flow was not proposed as we do in this paper. In this paper, we include another major application, which is a physics engine for elastic and deformable bodies. That said, we have two major case studies and one minor to demonstrate that both can benefit from our approach. Furthermore, we enhanced our automation tools. The genetic exploration now runs in seconds, simplifying the exploration process. Moreover, the exploration speed is now compared to additional heuristic-based optimization methods. Finally, the whole framework is presented here in much more detail than in Atienza and Baloukas (2007), to show all the innovative aspects of this work.

3. DDTs global optimization flow

The proposed optimization framework uses three different phases to perform the automatic exploration of DDT implementations using evolutionary computation:

- *Phase 1: pre-characterization*. All DDTs are modeled (Section 3.1) in terms of average number of random and sequential accesses, and average size for a given number of elements. This phase happens only once. The models are required by the multi-objective evolutionary algorithm to analytically calculate the performance of each DDT implementation, for various cost factors, ie execution time, memory footprint and energy consumption.
- *Phase 2: profiling*. Here, the initial profiling of the iterator-based access methods to the different DDTs used in the application (Section 3.2) takes place. A detailed report is produced, which comprises all the accesses to the DDTs done by the application. This report, along with the analytical models and the platform description are fed to our multi-objective evolutionary algorithm during the third phase.

- *Phase 3: evaluation*. Exploiting the characteristics of the final platform, we perform an exploration of the design space of DDTs implementation using multi-objective evolutionary computation (Section 3.3).

Fig. 1 shows an overview of the different phases (in light gray) and the inputs (in dark gray) required to perform the overall DDTs optimization.

3.1. Phase 1 – analytical modeling of DDT implementations

A DDT is a software abstraction by means of which we can manipulate and access data (Wood, 1993). The implementation of a DDT has two main components: First, it has storage aspects that determine how data memory is allocated and freed at run-time and how this memory is tracked; Second, it includes an access component, which can refer to two different basic access patterns: sequential or iterator-based and random access. In our case, after studying current state of the art multimedia applications like 3D video games and physics engines, we have classified the DDT implementations in basic and multi-layer implementations relevant for embedded multimedia applications, as proposed in Daylight et al. (2004), and Atienza et al. (2004). Trees, hash tables, graphs, all utilize these structures and build upon them a more complex access pattern. That said, if we can analytically characterize the basic DDTs then multi-layer implementations consisting of various combinations of these DDTs can be analytically characterized too. For example, a singly linked list of arrays combines a singly linked list (SLL) and a array (AR) DDT. The analytical model of singly linked list with arrays [SLL(AR)] is a combination of the analytical models of the two basic DDTs. This allows our tools to synthesize and test analytically all possible combinations of basic DDTs in multi-layered implementations providing more solutions to the designer.

The basic DDTs are the following ones:

- *Array* (*AR*): is a set of sequentially indexed elements of size s_T. Each element of the array is a record of the application.
- *Single linked list (SLL)*: is a single linked list of pointers to objects of type T. Each element of the list is connected with the next element through a pointer of size s_w.
- *Double linked list (DLL)*: is a double linked list of pointers to objects of type T. Each element of the list is connected with the next and the previous element with two separate pointers (of size s_w).

In addition, we have included in our exploration the fundamental variations of these basic DDTs regarding their key value, for embedded multimedia applications (Daylight et al., 2004; Wuytack et al., 1996), namely:

- *Pointer* (*P*): in the pointer variation of each basic DDT, the record of the application is stored outside the DDT and is accessed via a pointer. This leads to a smaller DDT size, but also to an extra memory access to reach the actual data. All DDTs used in our exploration comply to this variation except the simple array.
- *Roving pointer* (*O*): the roving pointer is an auxiliary pointer (of size s_w) useful to access a particular element of a list with less accesses in case of iterator-based access patterns. For instance, for a single linked list, if you access element $n + 1$ immediately after element n, your average access count is $1 + 1$ instead of $n/2 + 1$ (see Fig. 2).

In the rest of the paper the fundamental variations to basic DDTs are represented as in the following examples:

- *SLLO*: a singly linked list with roving pointer.
- *ARP*: a simple array of pointers to the actual data.
- *SLL(AR)*: a two-level DDT comprising a singly linked list of arrays. Each element of the SLL is an array, which holds the actual data.

Additionally, all variables used in this section are presented in Table 1.

In the following sections, we describe the extraction of the analytical models for both sequential and random access patterns.

3.1.1. Sequential access models extraction

Sequential access is the case where several consecutive elements of a DDT are traversed one after another. In modern applications this traversal is done using iterator structures. Thus, it is logical to assume a loop like the one below to represent sequential access.

```
for (Iterator& i = DDT. NewIterator(); i. IsDone(); i++) {*i;}
```

Using our profiling tool we logged the accesses made by each operation (NewIterator, IsDone and operator++) running a single iteration. In Table 2, we show the number of accesses for each operation individually. A distinction is made between read and write accesses. This separation is necessary for the energy model, as reads and writes have different energy consumption.

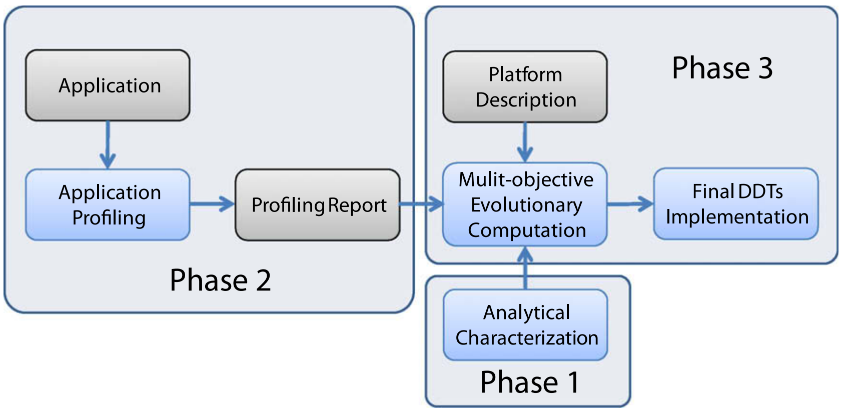
Application
Application Profiling
Profiling Report
Phase 2
Platform Description
Phase 3
Mulit-objective Evolutionary Computation
Final DDTs Implementation
Analytical Characterization
Phase 1

Fig. 1. Overview of the DDTs optimization flow.

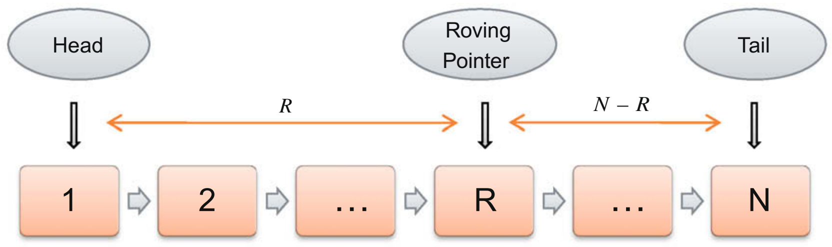

Fig. 2. A singly linked list with roving pointer – SLLO. R is the last accessed element and the one pointed by the roving pointer.

Table 1
Definition of variables used in this section

Variable	Definition
NA_r	Number of accesses required to retrieve one value with a random access pattern
NA_s	Number of accesses required to access all the values in a DDT with a sequential access pattern
S_{av}	Average memory footprint used by the DDT
N_e	Number of valid or initialized elements in the DDT
N_a	Number of reserved or allocated positions to store elements in the DDT
S_w	Width of a word on the architecture
S_T	The size of one element of type T

Table 2
Sequential access models

Implementation	Operator*	IsDone()	Operator++
AR	$4 \times N_e$(4 read)	$3 \times N_e$(3 read)	$2 \times N_e$(1 read, 1 write)
ARP	$5 \times N_e$(5 read)	$3 \times N_e$(3 read)	$2 \times N_e$(1 read, 1 write)
SLL	$3 \times N_e$(3 read)	$1 \times N_e$(1 read)	$3 \times N_e$(2 read, 1 write)
DLL	$3 \times N_e$(3 read)	$1 \times N_e$(1 read)	$3 \times N_e$(2 read, 1 write)
SLLO	$3 \times N_e$(3 read)	$4 \times N_e$(4 read)	$3 \times N_e$(2 read, 1 write)
DLLO	$3 \times N_e$(3 read)	$4 \times N_e$(4 read)	$3 \times N_e$(2 read, 1 write)

The table shows the number of accesses needed by each of the three operations considered in a sequential access mode. The accesses are separated in read and write accesses.

3.1.2. Random access models extraction

In this section, the mathematical formula of the random access model for one of the DDTs used in our exploration is presented, as an example of the model extraction process. We are interested in the average number of memory accesses to reach an arbitrary element n, which in general is defined as

$$\text{average} = \sum_{n=1}^{N} P(n)n, \text{ where } N \text{ is the number of elements in the list} \tag{1}$$

where $P(n)$ is the probability to access element n. In our case, the $P(n)$ is considered uniform, meaning that each element has the same probability of being requested by the application.

R is the element pointed by the roving pointer and also the last accessed element during a previous search. The presence of the roving pointer changes the way elements are accessed in comparison to simple SLL. Here, if the requested element is after the roving pointer, then the traversal will begin from element R. Otherwise, the list will be traversed from the beginning. Thus, our list is split in two parts, one with size R and another with size $N-R$. Eq. (1) becomes

$$\begin{aligned}\text{SLLO}_{\text{mean}} &= \sum_{n=1}^{R}\left(\frac{1}{N}\right)n + \sum_{n=1}^{N-R}\left(\frac{1}{N}\right)n \\ &= \frac{(N-R)(1+N-R)}{2N} + \frac{R(1+R)}{2N}\end{aligned} \tag{2}$$

Because of the fact that the roving pointer can point to any element, we can extract the mean for $R = 1$ to N

$$\text{SLLO}_{\text{mean}} = \sum_{R=1}^{N}\left[\frac{(N-R)(1+N-R)}{2N} + \frac{R(1+R)}{2N}\right] = \frac{1}{6N} + \frac{1}{2} + \frac{N}{3} \tag{3}$$

$$\text{SLLO}_{\text{mean}} = \frac{N}{3} \text{ for large } N \tag{4}$$

Different DDT implementations offer different trade-offs between memory use, performance and energy consumption. These trade-offs are shown in the analytical characterization of the basic DDTs used in our exploration, presented in Table 3. The analytical models of multi-layered DDT implementations are a combination of these models presented in Table 3 for basic implementations.

3.2. Phase 2 – profiling of iterator-based access methods

To enable the exploration of different DDT implementations, it is first necessary to understand how the different DDTs are being used in each studied application. Since the target applications are dynamic, hence the use of DDTs, it is necessary to profile them. It is also necessary that this profiling happens not at the memory level, but at the interface level to get an accurate view of the behavior of each DDT implementation. Available profilers do not provide information on the behavior of DDTs. That said, current solutions cannot discern a sequential kind of access to the elements of DDTs, from a random one. Knowing the behavior of each DDT is critical in choosing the right implementation for it. Some DDTs may have large variations in the number of hosted objects, while others focus more on accessing those objects in different ways. In the context of this work, we have expanded our profiling library (Poucet et al., 2006) with several higher level profiling information to identify the accesses at this level.

As a first extension to our profiling tool, we have re-implemented a sequence type, `vector`, fully compatible to the one defined by STL (SGI, 2006). The reason to stick to a commonly used interface, is that limited changes are required in the source code to profile an application and furthermore, they can be performed automatically. Most importantly, this is done without requiring a

Table 3
Analytical characterization of basic DDT implementations in the exploration

DDT implem.	Sequential accesses (NA_s)	Random accesses (NA_r)	Average aize (S_{av})
AR	$9N_a$	2	$19s_w + N_a \times s_T$
ARP	$10 \times N_a$	3	$19s_w + N_a(s_T + s_w)$
SLL	$7 \times N_e$	$\frac{N_e}{2} + 1$	$19s_w + N_e(2s_w + s_T)$
DLL	$7 \times N_e$	$\frac{N_e}{4} + 1$	$19s_w + N_e(3s_w + s_T)$
SLLO	$10 \times N_e$	$\frac{N_e}{3} + 1$	$20s_w + N_e(2s_w + s_T)$
DLLO	$10 \times N_e$	$\frac{N_e}{6} + 1$	$20s_w + N_e(3s_w + s_T)$

N_e is the number of valid or initialized elements in the DDT, N_a is the total number of reserved or allocated positions that can be used to store elements in the DDT, s_w is the width of a word on the architecture and s_T the size of one element of type T.

modification of the remainder of the application where the DDT is actually being accessed. The new `vector` includes directives that log all the different semantical operations. In particular, each function call is logged along with all accesses made to any internal variables that the type uses. That way we can have a clear view of each DDT's behavior at runtime.

Furthermore, a careful analysis of the sequence interface indicates that not only operations of the DDT, but also the iterator operations used to access the stored elements (Catthoor et al., 2002; James and Mansfield, 1999) must be logged. To enable us to couple the logging of memory accesses to specific DDTs, it is necessary to know at each point in time, from the profiling information, which container uses which memory segments. Therefore, the constructor, destructor, copy constructor and swap operation are logged as separate packets of our profiling tool, thereby giving a mapping between the addresses that a `vector` owns and the addresses that an iterator accesses. Other similar operations are the accessing of an element, the addition of an element, the removal of an element and the clearing of the container. Since it is possible to obtain references to an element in a container, we do not need to distinguish between read and write operations.

3.3. Phase 3 – multi-objective optimization of DDTs

GAs (Mitchell, 1996; Coello et al., 2002) are stochastic optimization heuristics where the exploration of the solution space of a certain problem is carried out by imitating the population genetics stated in Darwin's theory of evolution. Selection, crossover and mutation operators, derived directly from natural evolution mechanisms, are applied to a population of solutions, thus favoring the birth and survival of the best solutions. GAs have been successfully applied to many NP-hard combinatorial optimization problems and work by encoding potential solutions (individuals) to a problem by bit strings (chromosomes), and by combining their codes and, hence, their properties. In order to apply GAs to a problem, a genetic representation of each individual has first to be found (Coello et al., 2002; Atienza and Baloukas, 2007). Furthermore, an initial population has to be created, as well as defining a cost function to measure the fitness of each solution.

As a second step we need to design the genetic operators that will allow us to produce a new population of DDT solutions from a previous one, by capturing the inter-dependencies of the different DDT implementations working concurrently. Then, by iteratively applying the genetic operators to the current population, the fitness of the best individuals in the population converges to targeted solutions, according to the metric/s to be optimized and the weight of each of these metrics. For an overview of GAs the reader is referred to (Mitchell, 1996).

3.3.1. Fitness function

The objective of our algorithm is to obtain a multi-layer DDT representation for each container in the original application that optimize energy, memory use and performance. Such complex DDT is formed by a combination of up to three levels of the basic DDTs proposed in Section 3.1. To this end, we must evaluate the candidate solution by means of a fitness function. Table 4 presents all the variables used in this section.

After profiling the real application, the information required for the analytical characterization of the DDT implementations considered is available (see Section 3.1 for more details). Thus, for each individual available in a certain generation we can compute the performance (Perf related to the number of accesses to layers of the memory hierarchy), memory footprint (AvMem in Bytes) and energy values (Energy in nJ). Note that all the parameters, such as N_r and N_w are obtained by profiling the application using our `vector` DDT, which can be modeled as a AR. This means that N_r and N_w must be scaled in terms of the DDTs proposed by the GA and using Tables 3 and 2.

Table 4
Definition of variables used in this section

Variable	Definition
N_r	Number of read accesses to a DDT
N_w	Number of write accesses to a DDT
N_{ran}	Number of random accesses to a DDT
N_{rw}	Number of read/write accesses to the L1 data cache memory
N_{pa}	Number of misses in the data cache
T_{amem}	Average cycle time that an access to the main memory requires
NA_{cd}	Cycle time cost of creating/destructing the DDT
E_{pa}	Energy consumed per access to main memory
E_{rw}	Energy consumed per access to cache memory
E_{est}	Static energy consumed by the main memory

Therefore, the fitness process starts with the decoding of the individuals. Next, for each possible container (and its valid multi-layer DDTs proposed by our algorithm) we compute the following equations:

$$\begin{aligned}\mathrm{Perf} &= (\mathrm{NA_s} + (\mathrm{NA_r} * N_{ran})) * (N_r + N_w) \\ &\quad + (N_{pa}/4) * T_{amem} + (\mathrm{NA_{cd}} * 2) \qquad (5)\\ \mathrm{AvMem} &= S_{av} \qquad (6)\\ \mathrm{Energy} &= (N_{pa} * E_{pa}) + (N_{rw} * E_{rw}) + (S_{av} * E_{est}) \qquad (7)\end{aligned}$$

In our work, we consider that each cache line contains four blocks, thus, the amount of misses is divided by this constant. NA_{cd} has to be included twice since in our modeling all the containers are created at the beginning and deleted at the end. Also, regarding the energy calculations, we consider in this work in-place sharing, as the containers lifetimes are short. Moreover, we consider a basic memory hierarchy that consists of a main shared memory and a L1 data cache. Note that according to our empirical validation with several multimedia applications (Poucet et al., 2006), we assume in our energy calculations an average miss rate of the cache memory below 5% of the overall memory accesses. However, this value is user-configurable in our VEGA-based exploration process and even additional multi-level cache miss rate effects can be configured. In addition, it is possible to introduce some constraints and weights for the metrics to be optimized. For example, we can fix maximum values of performance, memory use and energy if the final embedded system requires it.

Other memory hierarchies could be modeled as well by modifying the previous equations. First of all, it must be made clear that the equations to calculate performance and energy are based on the fact that only one processor is accessing the bus to get the information of the data structure (i.e. conflicts are not being modeled). Thus, as long as the modeling refers to accesses without conflicts in the bus or memory hierarchy, they are valid, indistinctly of having one or multiple processors. In this case, if we have multiple levels of caches for a certain processor, the memory hierarchy can be easily modeled by extending the formulas with multiple N_{pa1}, N_{pa2}, etc. modeling the number of misses in each layer of the memory hierarchy, the respective N_{rw1}, N_{rw2}, etc. for the number of reads in each layer, and T_{amem1}, T_{amem2}, etc. for the average cycle time to access each cache level (in the case of only L1, it was assumed that it would have an access time of 1 clock cycle and therefore it is not included in the equations).

Furthermore, in the case of several parallel L1 caches, they could be modeled with a number of extra coefficients/terms representing the extra access time penalty due to conflicts in the bus or to the external main memory (or any additional shared level). Moreover, if the data of the DDTs can be shared between different processors, the equations and modeling have to be completely rethought, as they would need to model the effects of caches from different cores competing for shared dynamic data stored in the DDTs (e.g. contention mechanisms, starvation, etc.) and other issues like false data sharing between multiple cache lines of different processors.

3.3.2. Multi-objective algorithm

Multi-objective optimization could be defined in our case as the problem of finding a vector of decision variables which meets a set of constraints. Then this vector of decision variables is used to optimize a vector function whose elements represent the objective functions. These functions form a mathematical description of performance criteria which are usually in conflict with each other. Hence, the term optimize means finding such a solution, which would give acceptable values to all the objective functions (energy, performance and memory in our problem) for the designer (Osyczka, 1985). The notion of acceptable values is defined by the weight that the designer gives to each optimization metric, enabling linear combinations of the aforementioned metrics in our case and creating Pareto curves of solutions. In order to find these Pareto curves for problems of great difficulty, many multi-objective evolutionary algorithms have been developed. They can be classified into two broad categories: non-elitist and elitist also called first and second generation multi-objective evolutionary algorithms (Coello et al., 2002). In the elitist approach, GAs store in an external set the best solutions of each generation. This set will then be a part of the next generation. Thus, the best individuals in each generation are always preserved, and this helps the algorithm to get close to its POF. Algorithms such as PESA-II, MOMGA-II, NSGA-II and SPEA2 are examples of this category (Coello et al., 2002). In contrast, the non-elitist approach does not guarantee preserving the set of best individuals for the next generation. Examples of this category include MOGA, HLGA, NPGA and VEGA (Coello et al., 2002).

Since our design framework is independent of the GA utilized, we selected VEGA as one of the quicker and simpler GAs implementations. It has been demonstrated to be very efficient (David Schaffer, 1985). The main idea of VEGA is an extension of the simple genetic algorithm, which was called *vector evaluated genetic algorithm (VEGA*). The algorithm differs from the first one only in the way the selection is performed. This operator was modified in such a way that after every generation a certain number of sub-populations are obtained. Hence, VEGA generates a set of possible solutions with different trade-offs among the objectives and this set of solutions is found using the Pareto dominance concept (David Schaffer, 1985). The basic principle states that a given solution x1 dominates another solution x2 if and only if:

– Solution x1 is not worse than solution x2 in any of the objectives; and
– Solution x1 is strictly better than solution x2 in at least one of the objectives.

As a consequence of its basic principle, VEGA-based algorithms generate solutions that are locally non-dominated, but not necessarily globally non-dominated. In fact, VEGA presents the so-called speciation problem (Coello et al., 2002; David Schaffer, 1985) (i.e. we could have the evolution of solutions within the population which excel on different objectives). Thus, as shown in Fig. 3, for our problem with three objectives and a population size of M individuals, three sub-populations of size $M/3$ each are selected. These sub-populations are shuffled to obtain a new population of size M, where we then apply the GA operators (crossover and mutation) to refine further the solution. Regarding crossover operator, a uniform crosspoint function is used to select randomly the crossover point for each pair of chromosomes of two genetic populations, called one-point crossover. A single crossover point on both parents' chromosome strings is selected. All data beyond that point in either chromosome string is swapped between the two parent chromosomes. The resulting chromosomes are the children. With respect to mutation operator, it is used to maintain genetic diversity from one generation of a population of chromosomes to the next. It is analogous to biological mutation. The classic example of a mutation operator involves a probability that an arbitrary bit in a genetic sequence will be changed from its original state. The method applied involves generating a random variable for each bit in a sequence. This random variable tells whether or not a particular bit will be modified.

This process is repeated until no improvement occurs in any of the possible combinations generated in the last generation and in any of the target metrics. At this point, a Pareto front of optimal solutions for the different optimization metrics can be generated (see Section 4 for some examples). After different tests performed in an initial validation phase to select the optimal values, we have fixed the parameters of the genetic algorithm to the values indicated in Table 5, where N represents the number of containers logged in the original application.

3.3.3. Genetic representation

In order to apply a GA correctly we need to define a genetic representation of the design space of all possible DDT implementations alternatives. Moreover, to be able to apply the VEGA optimization process and cover all possible inter-dependencies of DDT implementations for different dynamic containers of an application, we must guarantee that all the chromosomes represent real and feasible solutions to the problem and ensure that the search space is covered in a continuous and optimal way (Atienza and Baloukas, 2007; David Schaffer, 1985). To this end, we define the

Table 5
Parameters for VEGA

Parameter	Value
Population size	$10 \times N$
Max. number of generations	$120 \times N$
Probability of crossover	0.80
Probability of mutation	0.01

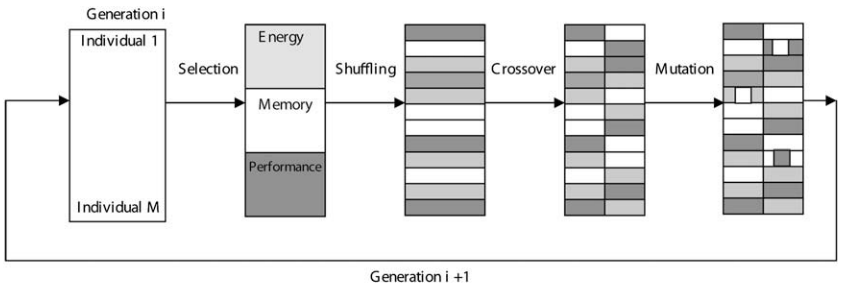

Fig. 3. VEGA-based design space exploration method.

Table 6
Example of a 48-bit chromosome

Container 1						
0–2	3–7	8–10	11–15	16–18	19–23	Bit
DDT^{c1}_{L1}	Na^{c1}_{L1}	DDT^{c1}_{L2}	Na^{c1}_{L2}	DDT^{c1}_{L3}	Na^{c1}_{L3}	Field
Container 2						
24–26	27–31	32–36	37–39	40–44	45–47	Bit
DDT^{c2}_{L1}	Na^{c2}_{L1}	DDT^{c2}_{L2}	Na^{c2}_{L2}	DDT^{c2}_{L3}	Na^{c2}_{L3}	Field

implementation of the containers of a program by storing the following information on each chromosome, as shown in Table 6:

- Level 1 DDT ($0 < DDT_{L1} < 7$): this field represents one of the six different possibilities using the previous DDTs analytically characterized (Table 3), as Daylight et al. (2004) has proposed for multimedia applications. Therefore, using a binary encoding we need 3 bits. Note that we must guarantee that at least one level of DDTs is selected, i.e. ($000 < DDT_{L1} < 111$).
- Initial size of level 1 ($0 < Na_{L1} \leqslant Nve$): this field represents the initial number of elements for the DDT of level 1. The number of bits for this chromosome depends on the average number of elements logged in the profiling report, i.e. $2^{n_b} \geqslant Nve$, where n_b is the number of bits for this field.
- Level 2 DDT ($0 \leqslant DDT_{L2} < 7$): this field represents the DDT selected for the second level, no DDT included ($DDT_{L2} = 0$). As in the previous level, we need 3 bits.
- Maximum size of level 2 ($0 \leqslant Na_{L2} \leqslant Nve$): It represents the maximum number of elements that a DDT of level 2 may contain. The number of bits for this chromosome are the same as in the Na_{L1} field.
- Level 3 DDT ($0 \leqslant DDT_{L3} < 7$): this field represents the DDT selected for the third level. As in the second and first levels, we need 3 bits.
- Maximum size of level 3 ($0 \leqslant Na_{L3} \leqslant Nve$): this field represents the maximum number of elements that a DDT of level 3 may contain.

Although n_b is calculated automatically by our application, it can be set manually. However, if n_b is set to a large value, the algorithm will select just one level of arrays: AR DDTs. This is because our array is the dynamic data type with the best performance, memory footprint and energy consumption when its size is a constant.

Consequently, using this chromosome structure we need $9 + 3 \cdot n_b$ bits to represent the solution proposed for each container. So, if an application has N containers, each chromosome has to be constituted of $(9 + 3n_b)N$ bits (genes). For instance, in the case of an application that uses two dynamic containers and 32 elements each on average ($n_b = 5$), a potential solution would be represented by a 48-bit chromosome (see Table 6).

We applied a repair algorithm to handle constraint violations. If no DDT is selected for level 2 ($DDT_{L2} = 0$), our algorithm removes the DDT from level 3 ($DDT_{L3} = 0$) and updates $Na_{Li}, i = 1, 2, 3$ accordingly. In addition, the total number of elements stored (Ne) must satisfy that $Ne = \prod_{i=1}^{3} Na_{Li} \forall Na_{Li} \neq 0$.

Our current implementation of the exploration framework is able to explore applications with up to 40 containers at the same time, which can cover all the real-life embedded multimedia applications we are aware of.

4. Experimental results

In this section we evaluate the proposed optimization framework for three 3D applications. A 3D environment builder (Simblob), a racing simulator (Vdrift) and a 3D physics engine for elastic and deformable bodies (University of Maryland). Each one of these applications is initially profiled automatically two times using our extended profiler tool (Poucet et al., 2006). This tool provides a profiling report, which includes all the accesses made to every container during the execution of the application. Finally, the actual GA-based exploration is driven by the profiling report, the analytical characterization of DDTs, as well as any platform description that may be targeted.

In the first set of experiments, we have used our methodology to explore the optimized configuration of DDTs for all applications. In fact, our GA-based method uses multiple generations of possible solutions to find the correct combination of different DDT implementations for each container in the three applications. This is a very time-consuming process for a designer to manually tune since there is a large set of different combinations of DDTs implementations (three levels for six DDTs reaches $6^3 = 216$ possibilities for each container, without taking into account the initial and maximum sizes of different levels). Furthermore, in order to have comparative view of how our solution behaves against standard DDT implementation, we implemented and tested each one of the 6 variations presented in Table 3. Then, the output of our methodology is compared with the case where every container in an application is implemented using a particular DDT, for example all containers implemented as SLL. Finally, we compare our optimized results with the evaluation of the original application.

The configuration used in the experimental process is presented in Table 7, while the L1 cache model used is described in Shivakumar and Jouppi (2001).

The three applications used in the exploration process are representative multimedia applications with heavy use of data structures. Table 8 presents the characteristics of these applications in terms of number of containers, the access pattern utilized by those containers and the type of the original container (from the C++ standard template library). As can be seen, there is enough variability in the applications to demonstrate the applicability and usefulness of our approach.

In the sections below, the results of the optimization process are presented for each of the three applications. In all case studies, VEGA will always keep 50 individuals for each iteration of the process. We used two profilings of each application.

4.1. Simblob

SimBlob is a project to develop simulations that focus on interaction with the environment. The user can create a variety of natural elements like mountains, lakes, forests, etc. that have their own impact in the simulation environment. SimBlob uses the OpenGL and GLUT graphics libraries to render the objects on the

Table 7
Hardware configuration for experimental results

	Speed (MHz)	Energy (mW)	Type	Latency (ns)	Bandwidth (MB/s)	Size (MB)
Memory	100	19.5	Embedded DRAM	19.5	50	16
Processor	100	168	N/A	N/A	N/A	N/A

Table 8
Characteristics of the benchmarks used in the exploration process

Application	Number of containers	Access pattern	Original container
Simblob	1	Sequential	Vector
Vdrift	37	Sequential/random	Vector, list
Physics engine	11	Sequential/random	Vector

screen. There is one container of sequence type inside the code that holds water source objects. For each water source created a new instance is placed inside the sequence container. During the execution the water sources list is traversed using an iterator, implying a sequential access pattern most of the time. This fact favors simple implementations like SLL and DLL. Indeed the Pareto optimal DDTs for this application are a singly linked list and a doubly linked list as came out from our exhaustive exploration.

Fig. 4 depicts the Pareto front for this application and both profiling reports. Results are quite similar. One of the Pareto points operates the embedded system in high performance mode (low number of accesses, thus higher speed) but gives a large memory footprint and has higher energy consumption. The other optimizes the system for low memory footprint and power consumption, but it loses in terms of performance (high number of accesses). As it was mentioned above, the Pareto optimal DDTs may be calculated for SimBlob (SLL and DLL). In this application, the corresponding DDTs to the Pareto front represented in Fig. 4 are precisely a singly linked list and a doubly linked list. Our genetic algorithm reached its best population after 17 generations.

Then, in Fig. 5 we present how the other DDT implementations and the original application performed in comparison with the GA output, where the minimal objectives found by VEGA are presented for both profiling reports. All the three objectives have been normalized to the AR DDT and represented in logarithmic scale. It is clear that VEGA selected the best implementations that manages to minimize all three design metrics, namely memory footprint, memory accesses and energy consumption. The relatively less energy consumption and memory accesses compensate for the slightly higher memory footprint than using ARP or the original application.

Finally, note that although SimBlob is the simplest among the three presented real-life applications, it already shows how the designer can benefit from our GA-based exploration methodology. Design time can be saved by using directly a singly or doubly linked list instead of any other DDT implementation.

4.2. Vdrift

Vdrift is an open source racing simulator that uses STL vector (SGI, 2006) to handle its dynamic behavior. The application uses very realistic physics to simulate the car's behavior and also includes a full 3D environment for interaction. Vdrift uses 37 dynamic DDTs to hold its dynamic data that are all sequences. The objects put inside the containers vary from wheel objects to float numbers required by the game's physics. During the game, some containers get accessed sequentially, while others exhibit a random access pattern. This means that the applications requests regard objects, which are not successive in the list structure, requiring complex data structures to cope with this access pattern.

After 500 generations we reached the best population. Fig. 6 represents the Pareto front, in other words the optimum operation states of the embedded system. Due to the large number of DDTs, the Pareto front is wider than the one for Simblob, offering more solutions to the designer.

In the case of Vdrift the output of our exploration method for both profiling reports is a combination of AR, SLL, DLL, AR(SLLO) and AR(DLLO). For comparison reasons we present Fig. 7 to illustrate the optimization process that our methodology performs. In this test, the set of containers was successively implemented using SLL, DLL, etc., and finally the original application. Thus, in the end, compared to the combination proposed by our framework. The figure shows clearly the achieved level of optimization and final gains after applying the proposed optimization flow. Note that the second profiling report consume less energy in the case of sequential data structures.Since the profiling report is much bigger, the capacity of the array is overloaded and as a consequence, arrays needs more energy (there was more cache misses, creation and destructions) than a sequential data structure.

Vdrift is a typical real life example, where the designer has to choose a DDT implementation based only on his experience, due to the large number of possible solutions that prohibit any exploration effort. Instead, using the proposed methodology, the designer can achieve an optimized combination of DDT implementations with a very limited effort.

4.3. Physics engine

The last tested application is a physics engine for elastic and deformable bodies (University of Maryland). The purpose of this project is to create a 3D engine that displays the interaction of non-rigid bodies. Every object is modeled as a set of points connected by springs of given elasticity factor. The model uses spring forces and damping to give elasticity to the objects. Object collisions are determined using our collision detection engine. After

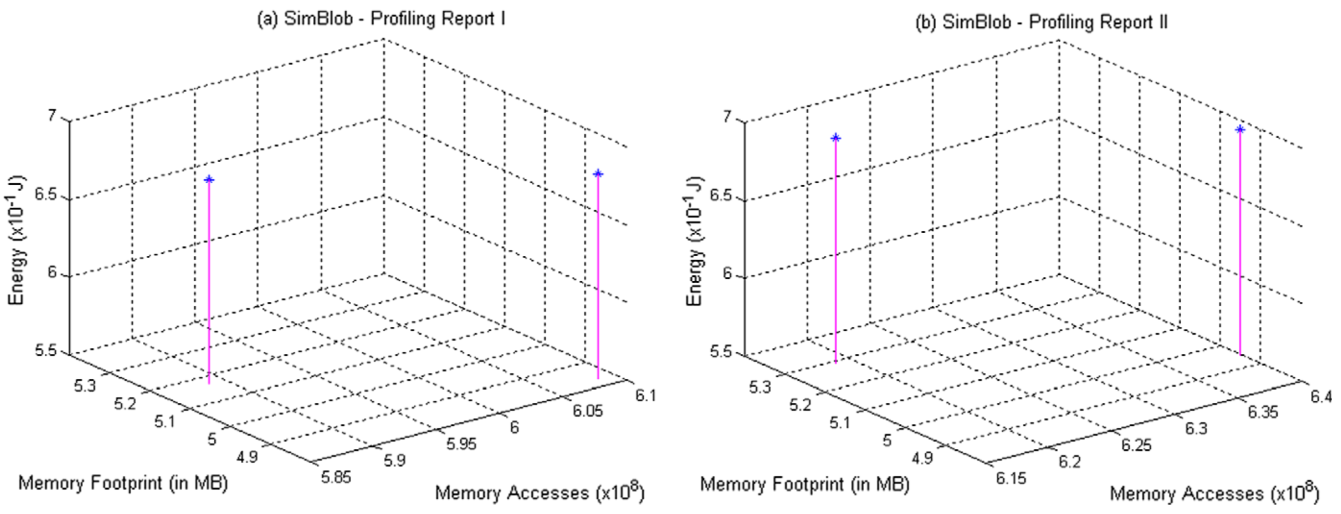

Fig. 4. 3D Pareto fronts (memory footprint, memory accesses and energy consumption) of combined DDTs implementation solutions for SimBlob obtained using the proposed evolutionary-based optimization framework and two profiling reports (a and b).

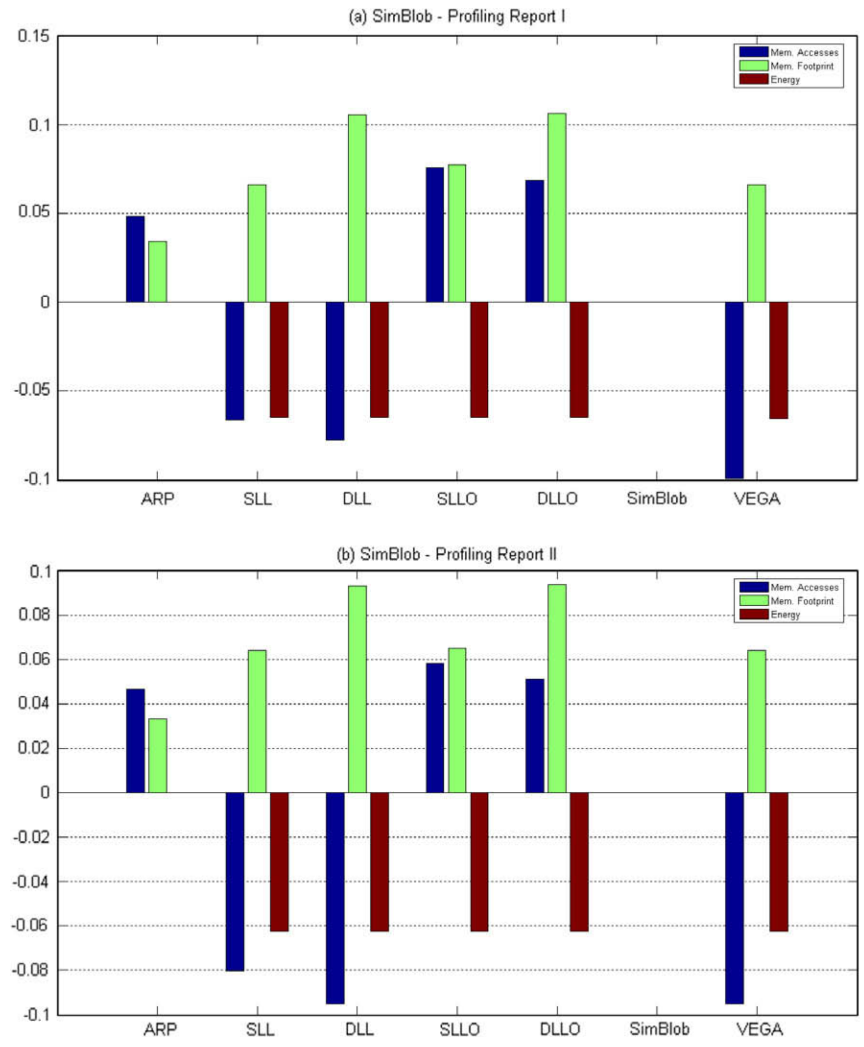

Fig. 5. Overall results for different design metrics coming from various sets of DDT implementations and the original application (values normalized to the AR DDT implementation, logarithmic scale).

collision occurs the response is calculated using physics laws. This physics engine has 11 containers, but creates more objects than the previous applications. This gives much more space for optimization, as dynamic structures are now called to host a large number of objects. A simple variation in their design can make a significant difference optimizing design metrics.

In the physics engine case our genetic algorithm reached its best population after 614 generations. Fig. 8 depicts the Pareto-front obtained. Thus, various trade-offs are provided to the designer in the multi-objective space (i.e. energy, memory footprint and memory accesses) using our methodology.

Finally, as an overview of the behavior of our solutions against standard solutions and the original application, we present Fig. 9 that shows the total number of accesses, memory footprint, and energy consumption for both profiling reports. It can be seen how in the second profiling report sequential DDTs give more benefits than arrays, because as the time of execution increases, the initial capacity of the arrays becomes obsolete and they must be destroyed and recreated. In this case the output of our exploration method for both profiling reports is a combination of SLL, DLL, SLL(AR) and DLL(AR).

The obtained results illustrate that in real-life applications like 3D games, physics engines and 3D environment creation programs, the solutions found by our multi-objective GA-based exploration for memory footprint,memory accesses and energy consumption are the best possible DDT implementations, in comparison with other possible manual solutions. In such solutions that are also based on STL, the 37 containers of Vdrift, 11 containers for physics engine and one container of Simblob are implemented using variations of one DDT implementation, mainly for simplification purposes on handling DDT's.

4.4. Comparison of exploration speed

In a second set of experiments we have compared the exploration speed of our GA-based optimization methodology for DDTs in comparison to different alternative exploratory methods. The results obtained for all applications for the different tested exploration methods are shown in Table 9. First, we have compared our approach with an almost exhaustive exploration. We assume that a designer starts with all DDTs implementations presented in Section 3.1 already available. Also, the optimization targets a subset of all the containers of the application. It is important to stress, however, that it is unmanageable for the designer to get a totally complete exploration of all the possible DDT implementation combinations using the traditional way for real-life complex applications. Considering the physics engine case, for example, even if we needed only 1 s for each combination's run, we would need

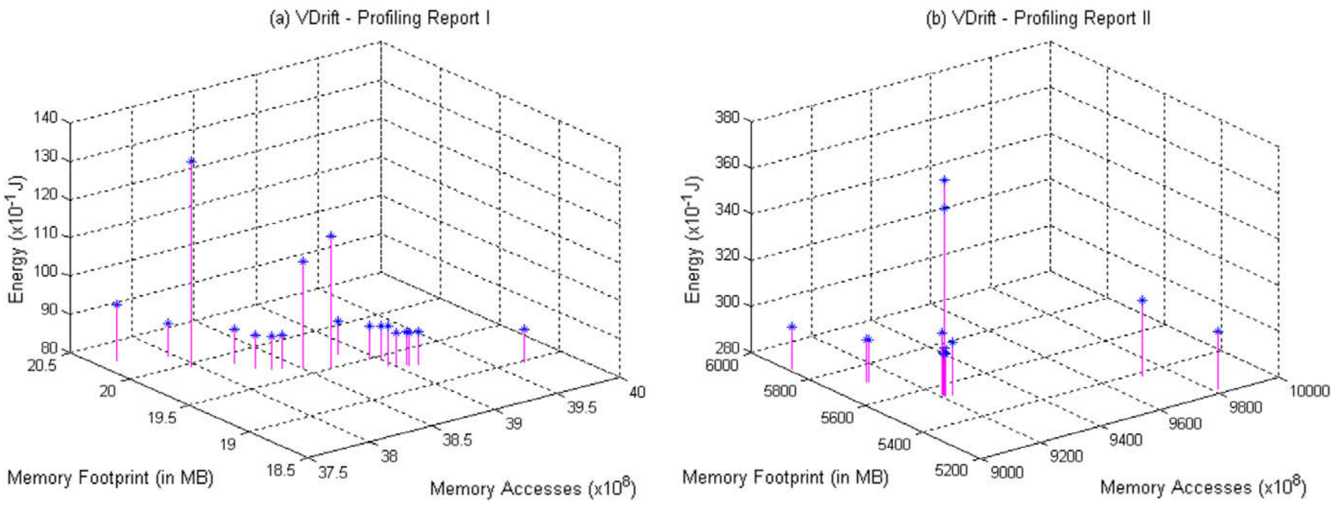

Fig. 6. 3D Pareto curve (memory footprint, memory accesses and energy consumption) of combined DDTs implementation solutions for VDrift obtained using the proposed evolutionary-based optimization framework and two profiling reports.

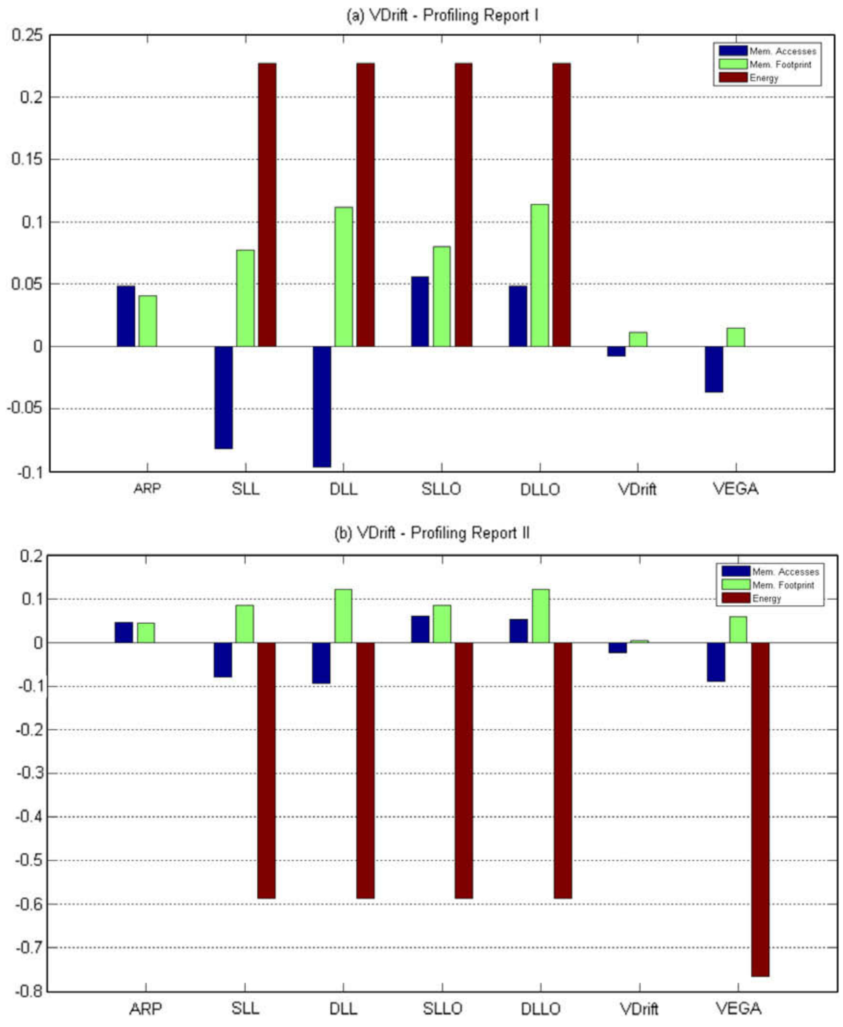

Fig. 7. Overall results for different design metrics coming from various sets of DDT implementations and the original application (values normalized to the AR DDT implementation and logarithmic scale).

$(6^3)^9$ s to run all the simulations, which is nearly 3.24×10^{13} years (without taking into account the number of elements for each level). Moreover, if we add the compilation time for each different combination the situation gets even worse. The figures in Table 9 represent an exhaustive exploration of a subset of all the DDTs in the tested applications, simplified to one level of DDTs. Second,

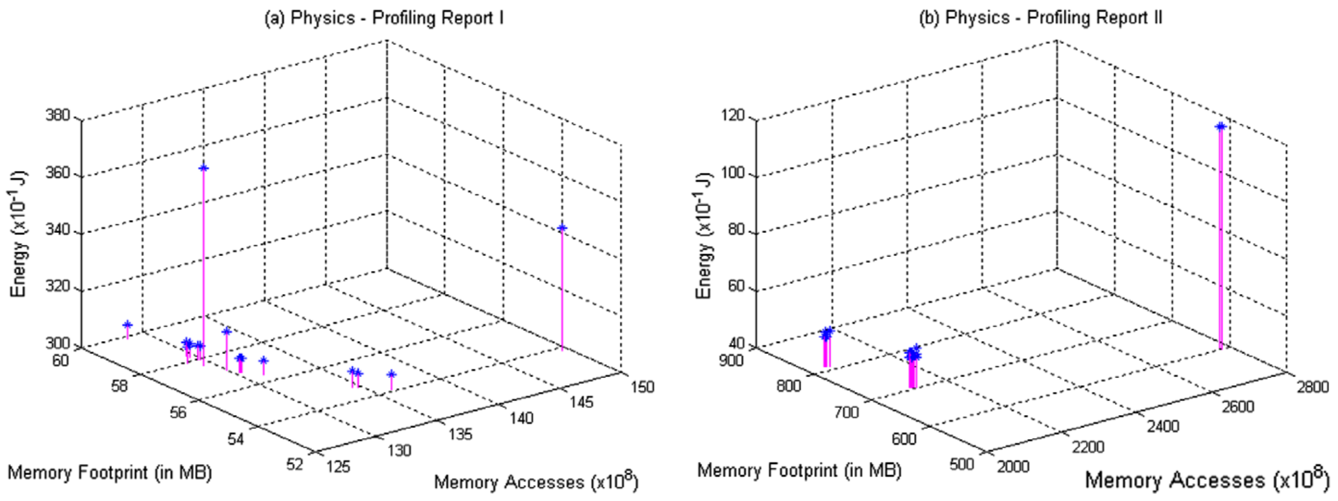

Fig. 8. 3D Pareto curve (memory footprint, memory accesses and energy consumption) of combined DDTs implementation solutions for Physics obtained using the proposed evolutionary-based optimization framework and two profiling reports.

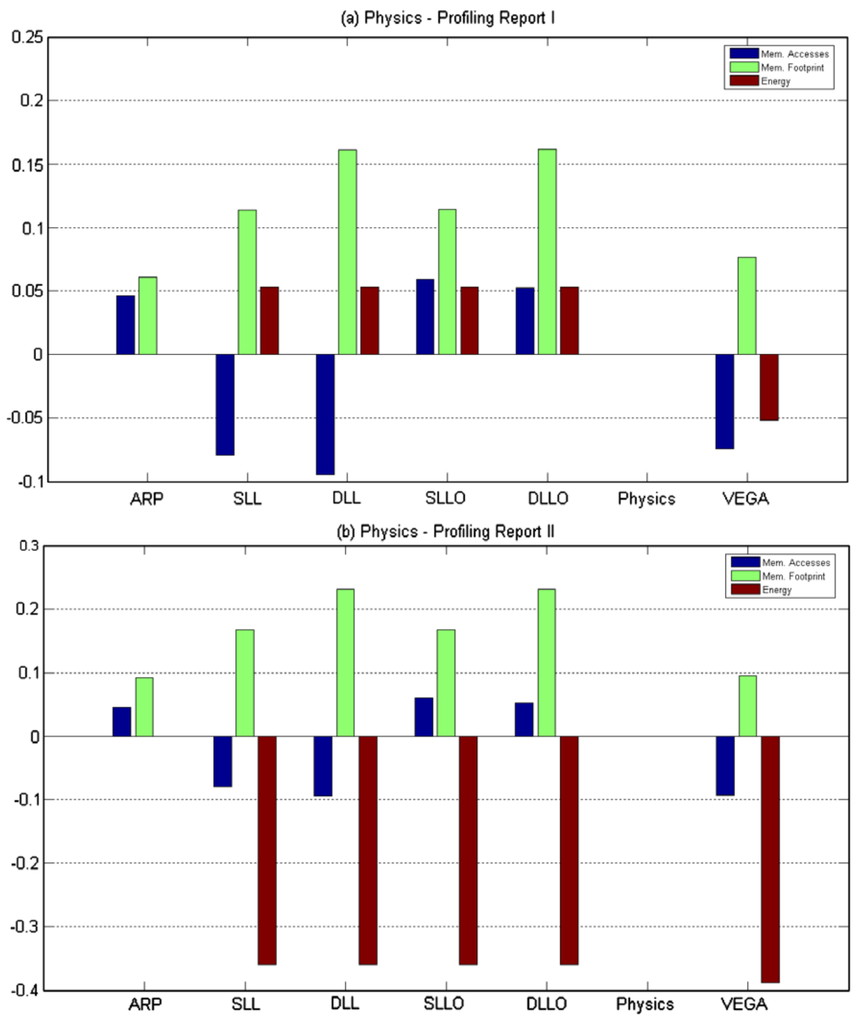

Fig. 9. Overall results for different design metrics coming from various sets of DDT implementations and the original application (values normalized to the AR DDT implementation and in logarithmic scale).

as Table 9 depicts, we have also compared our algorithm with state-of-the-art pruning and optimization methods for DDT implementations presented in Leeman (2003), Wuytack et al. (1996).

Breadth-first, depth-first and branch and bound algorithms were implemented in two phases. In a first phase, we obtain up to three levels of DDTs minimizing a simplified model of the

Table 9
Exploration time to minimize memory accesses or memory footprint of DDT implementations for Simblob, Vdrift and physics engine using different exhaustive exploration and heuristic-based optimization methods versus the proposed multi-objective GA-based approach

DDTs optimization methods	Simblob	Vdrift	Physics
Exhaustive exploration	1 h	9 days	41 days
Breadth-first exploration	14 s	3 days	5 days
Depth-first exploration	13 s	25 min	2 h
Branch and bound exploration	14 s	7 min	32 min
GA-based proposed method	9 s	59 s	210 s
	1.56 × gains	7.12 × gains	9.14 × gains

weighted sum of three objectives (memory accesses, memory footprint and energy consumption). In a second phase we explore the initial and maximum size for all the three levels. Thus, these algorithms, as well as the exhaustive approach, explore solutions only in single-objective space. Our GA-based framework is able to explore solutions in a multi-objective space, directly offering a complete pareto front of multi-layer DDTs. This corresponds to different optimal choices for the operation of an embedded system, so that the designer can select the best for his design constraints. Moreover, these Pareto fronts are crucial for use in dynamically varying situations where different working points have to be selected (and traversed) during the application life time (Yang et al., 2001). In such dynamically varying contexts, the conventional solution with only a single working point would become highly suboptimal.

Because of the differences in the mathematical model, we compared execution times instead of fitness values. The results in Table 9 outline that the exploration process with our method is orders of magnitude faster than the optimization process performed using directly the implementations of DDTs, namely 9 s versus 1 hour in the case of Simblob, 59 s instead of 9 days in the case of Vdrift and 210 s versus 41 days for the physics engine. In addition, and more importantly, the proposed GA-based method finds the optimal solutions of DDT implementations faster than the compared state-of-the-art DDTs optimization methods using different heuristics, achieving speed-ups of 1.56× for SimBlob, 7.12× for Vdrift, and 9.14× for physics engine respectively. The main reasons for these improvements are initially the use of only an initial profiling phase to characterize the dynamic behavior of the application for all possible DDTs. The other reason is the effective use of the VEGA exploration method in combination with our analytical models of DDT implementations to study the inter-dependencies of variables in the application. Hence, we can prune the design space in a more effective way than other heuristics. As a consequence, in a limited number of generations of possible sets of DDT implementations solutions, our GA-based optimization method can converge to an optimal solution according to the concrete user-defined constraints (i.e. memory footprint, memory accesses and/or energy consumption).

5. Conclusions

New embedded devices have increased their capabilities and now complex applications can be ported to them. Such applications include intensive dynamic memory requirements that must be heavily optimized for an efficient mapping on embedded devices. To efficiently use dynamic memory in this applications, designers need to select suitable complex DDT implementations (dynamic arrays, linked lists, etc.) for the variables used in the running applications with respect to their specific embedded systems requirements (e.g. performance, memory footprint or energy consumption).

In this paper, we have presented a new multi-objective optimization method based on evolutionary computation that can be used to optimize the complex DDTs implementations from multimedia applications. This method largely simplifies the exploration effort of multi-layered DDTs for developers and enables the refinement of DDT implementations in an automated way. As a result, the proposed approach leads to important savings in overall system integration time for dynamic applications. In the same time it achieves optimal implementations of DDT structures with respect to key designer's metrics. Moreover, our experimental results with three real-life multimedia embedded applications show that the presented optimization approach significantly reduces the exploration time up to 9.14× with respect to state-of-the-art methods to optimize DDTs implementations while still achieving complete Pareto fronts of solutions for the considered applications.

The results obtained so far have outlined other interesting future research lines in the area of DDT implementation optimizations using multi-objective evolutionary computation. Initially, analytical models for more and more complex DDTs can be extracted and added in our model to allow exploration of applications utilizing tree structures. Furthermore, the study of the possible benefits of more complex and parallel GAs in the efficient exploration of the design space of DDT implementations, is very challenging. Also, for practical reasons in large multimedia embedded applications with many dynamic variables, the evaluation of the influence of more complex memory hierarchies in the suitable pruning process of individuals is a key research problem to be considered.

Acknowledgement

This work is partially supported by Spanish Government Research Grants TIN2005-5619, and MEC Consolider Ingenio 2010 2007/2011 of the Spanish Council of Science and Technology.

References

Antonakos, James L., Mansfield Jr., Kenneth C., 1999. Practical Data Structures using C/C++. Prentice Hall, UK.
Atienza, David, Baloukas, Christos, et al., 2007. Optimization of dynamic data structures in multimedia embedded systems using evolutionary computation. In: Proceedings of the 10th International Workshop on Software and Compilers For Embedded Systems (SCOPES), April 2007.
Atienza, David, Leeman, Marc, Catthoor, Francky, Deconinck, Geert, Mendias, Jose M., De Florio, Vicenzo, Lauwereins, Rudy, 2004. Fast prototyping and refinement of complex dynamic data types in multimedia applications for consumer devices. In: Proceedings of the International Conference on Multimedia and Expo (ICME), June 2004.
Bartzas, Alexandros, Mamagkakis, Stylianos, Pouiklis George, Atienza, David, Catthoor, Francky, Soudris, Dimitrios, Thanailakis, Antonios, 2006. Dynamic data type refinement methodology for systematic performance-energy design exploration of network applications, Design, Automation and Test in Europe (DATE), March 2006.
Benini, Luca, De Micheli, Giovanni, 2000. System level power optimization techniques and tools, ACM Transactions on Design Automation for Embedded Systems (TODAES), April 2000.
C++ Standardisation Committee, 1998. Programming languages – C++ – ISO/IEC 14882. Technical report. American National Standards Institutes, September 1998.
Catthoor, Francky, Danckaert, K., Kulkarni, C., Brockmeyer, Eric, Kjeldsberg, P.G., Van Achteren, Tanja, Omnes, T., 2002. Data Access and Storage Management for Embedded Programmable Processors. Kluwer Academic Publishers.
Coello, Carlos A., Van Veldhuizen, David A., Lamont, Gary B., 2002. Evolutionary Algorithms for Solving Multi-Objective Problems. Kluwer Academic Publishers..
Cormen, Thomas H., Leiserson, Charles E., et al., 2001. Introduction to Algorithms, second ed. The MIT Press.
David Schaffer, J., 1985. Multiple objective optimization with vector evaluated genetic algorithms. In: Proceedings of the First International Conference on Genetic Algorithms.
Daylight, E.G., Atienza, David, Vandecappelle, Anrout, Catthoor, Francky, Mendias, Jose M., 2004. Memory-access-aware data structure transformations for embedded software with dynamic data accesses. IEEE Transactions on VLSI Systems.
Houck, Chris, Joines, Jeff, Kay, Mike, 1995. A genetic algorithm for function optimization: a matlab implementation. NCSU-IE Technical Report 95-09.
Jerraya, Ahmed, Wolf, Wayne, 2005. Multiprocessor Systems-on-Chips. Morgan Kaufman Elsevier.

Leeman, Mark, 2003. Interactive strategies and analysis method for dynamic data type transformation and refinement in multimedia applications, PhD thesis. Katholieke Universiteit Leuven, October 2003.
Michalewicz, Zbigniew, 1996. Genetic Algorithms + Data Structures = Evolution Programs. Springer-Verlag.
Mitchell, Melanie, 1996. An Introduction to Genetic Algorithms. MIT Press.
Muchnick, S., 1997. Advanced Compiler Design and Implementation. Morgan Kaufman Publisher, San Francisco.
Osyczka, Andrzej, 1985. Multicriteria optimization for engineering design. Design Optimization. Academic Press.
Poucet, Christophe, Atienza, David Catthoor, Francky, 2006. Template-based semi-automatic profiling of multimedia applications. In: Proceedings of the IEEE International Conference on Multimedia and Expo (ICME), July 2006.
SGI, 2006. Standard Template Library, http://www.sgi.com/tech/stl/.
Shivakumar, P., Jouppi, N.P., 2001. Cacti 3.0: an integrated cache timing, power, and area model. Technical Report 2001/2, Compaq Computer Corporation, August 2001.
Smailagic, Asim, Siewiorec, Daniel P., Anderson, Drew, Kasaback, Chris, Martin, Tom, Stivoric, John, 1995. Benchmarking an interdisciplinary concurrent design methodology for electronic/mechanical systems. In: Proceedings of the 32nd ACM/IEEE Conference on Design Automation Conference (DAC).
Sourceforge. Simblob – the 3d environment builder framework. http://sourceforge.net/projects/simblob.
Sourceforge. Vdrift racing simulator. http://sourceforge.net/projects/vdrift.
University of Maryland. 3D physics engine for elastic and deformable bodies. http://www.cs.umd.edu/Honors/reports/kharevych.html.
Wood, Derick, 1993. Data Structures, Algorithms and, Performance. Addison-Wesley Longman Publishing Co.
Wuytack, Sven, Catthoor, Francky, De Man, Hugo, 1996. Transforming set data types to power optimal data structures. IEEE Transactions on Computer-aided Design (June).
Yang, P., Wong, C., Marchal, P., Catthoor, F., et al., 2001. Energy-aware runtime scheduling for embedded multi-processor SOCs. IEEE Design and Test of Computers (special issue on Application-specific multi-processor mapping, September 2001).

Christos Baloukas received his Diploma and M.Sc. Degree in Electrical and Computer Engineering from the Democritus University of Thrace, Greece, in 2004 and 2006, respectively. He is currently a Ph.D. researcher in the VLSI Design and Testing Center in the Democritus University of Thrace. His research interests include dynamic access and storage optimization on communications applications for low power and high performance, embedded systems and high-level design optimizations.

Jose L. Risco-Martin is an Assistant Professor in Complutense University of Madrid, Spain. He received his Ph.D. from Complutense University of Madrid in 2004. His research interests are computational theory of modeling and simulation, with emphasis on DEVS, dynamic memory management of embedded systems, and net-centric computing. He can be reached at jlrisco@dacya.ucm.es.

Prof. David Atienza received his M.Sc. and Ph.D. degrees in Computer Science from Complutense University of Madrid (UCM), Spain, and Inter-University Micro-Electronics Center (IMEC), Leuven, Belgium, in 2001 and 2005, respectively. Currently he is Professor and Director of the Embedded Systems Laboratory (ESL) at EPFL, Switzerland and Adjunct Professor at the Computer Architecture and Automation Department (DACYA) of UCM. His research interests focus on design methodologies for integrated systems and high-performance embedded systems, including new modelling frameworks to explore thermal management techniques for Multi-Processor System-on-Chip, novel architectures for logic and memories in forthcoming nano-scale electronics, dynamic memory management and memory hierarchy optimizations for embedded systems, Networks-on-Chip interconnection design, and low-power design of embedded systems. In these fields, he is co-author of more than 90 publications in prestigious journals and international conferences, such as, IEEE TCAD, IEEE Micro, IEEE T-VLSI Systems, ACM TODAES, Elsevier-Integration: The VLSI Journal, DAC, ICCAD, DATE, ASP-DAC, etc. Also, he is part of the Technical Program Committee of the DATE, ICCAD, GLSVLSI, VLSI-SoC, RTAS, SBCCI and PATMOS conferences, and Associate Editor of IEEE Transactions on CAD (in the area of System-Level Design) and Elsevier Integration: The VLSI Journal. He is an elected member of the Executive Committee of the IEEE Council of Electronic Design Automation (CEDA) since 2008.

Christophe Poucet received his Diploma and M.Sc. Degree in Electrical Engineering from the ESAT department of the Katholieke Universiteit Leuven, in 2003. While writing this paper he was a Ph.D. researcher at IMEC and the ESAT department of the Katholieke Universiteit Leuven. His research interests include dynamic access and storage optimization for low power and high performance, embedded systems, high-level data-access optimizations and functional languages. Currently, he is working at Google as a Site Reliability Engineer.

Lazaros Papadopoulos received his Diploma in Electrical and Computer Engineering from the Democritus University of Thrace, Greece, in 2005. He is currently a post graduate student in the VLSI Design and Testing Center in the Democritus University of Thrace. His research interests include Networks-on-Chips and system-level design optimizations in embedded platforms.

Stylianos Mamagkakis received his Master and Ph.D. degree in Electrical and Computer Engineering from the Democritus Uni. Thrace (Greece) in 2004 and 2007, respectively. Since 2006, he coordinates a team of Ph.D. students within the NES division at IMEC, Leuven, Belgium. His research activities mainly belong to the field of system-level exploration, with emphasis on MPSoC run-time resource management and system integration. He has published more than 25 papers in International Journals and Conferences. He was investigator in 9 research projects in the embedded systems domain funded from the EC as well as national governments and industry.

Dimitrios Soudris received his Diploma in Electrical Engineering from the University of Patras, Greece, in 1987. He received the Ph.D. Degree in Electrical Engineering, from the University of Patras in 1992. Since 1995, he was working for more than thirteen years in Department of Electrical and Computer Engineering, Democritus University of Thrace, Greece. He is currently working as Associate Professor in School of Electrical and Computer Engineering of National Technical University of Athens, Greece. His research interests include embedded systems design, low power design, parallel architectures, and VLSI signal processing. He has published more than 180 papers in international journals and conferences and co-edited/authored four books. He was leader and principal investigator in numerous research projects funded from the Greek Government and Industry as well as the European Commission (ESPRIT II-III-IV and 5th, 6th and 7th IST). He has served as General Chair and Program Chair for PATMOS 1999 and 2000, respectively, and General Chair of IFIP/IEEE VLSI-SOC 2008. He received an award from INTEL and IBM for the project results of LPGD #25256 (ESPRIT IV). He is a member of the IEEE, the VLSI Systems and Applications Technical Committee of IEEE CAS and the ACM.

Ignacio Hidalgo is an Associate Professor at Complutense University of Madrid since 2002. He received the MS degree in Physics from the UCM in 1994. After that, he joined Cibertronic Ltd. where he worked as Electronic Design Engineer. Then he joined ArTeCS group, where he has held several research and teaching positions. He received his Ph.D. degree in December 2001. His advisor was Dr. Juan Lanchares. His dissertation, titled "Partitioning and Placement for Multi-FPGA systems using Genetic Algorithms" proposed several techniques for implemented digital circuits onto boards containing several FPGA devices. He is also Academic Organizer of the UCM Computer Science College at Aranjuez (Madrid). As a Ph.D. student he visited the Italian National Research Center (INSTI-CNR) (Pisa). His research interests include processor design, and hardware optimizations for Asynchronous and Embedded designs. For these tasks he has a lot of expertise on applying Evolutionary Computation techniques.

Francky Catthoor received a Ph.D. in El. Eng. from the K.U.Leuven, Belgium in 1987. Since then, he has headed several research domains in the area of architectural methodologies and system synthesis for embedded multimedia and telecom applications, all within the DESICS division at IMEC, Leuven, Belgium. His current research activities mainly belong to the field of system-level exploration, with emphasis on data storage/transfer and concurrency exploitation, both in customized and programmable (parallel) instruction-set processors.

Juan lanchares is associate professor in the Department of Computer Architecture and System Engineering of the Complutense University, Spain. His research interest are SMT processors , Asynchronous Techniques for System Design and the Genetic CAD tools for HW design. He received the MS degree in Physics from the UCM in 1990 and he received his Ph.D degree in the of 1995.