Optimization of Dynamic Data Structures in Multimedia Embedded Systems Using Evolutionary Computation

David Atienza[1,2], Christos Baloukas[3], Lazaros Papadopoulos[3], Christophe Poucet[4], Stylianos Mamagkakis[4], Jose I. Hidalgo[1], Francky Catthoor[4], Dimitrios Soudris[3] and Juan Lanchares[1]
[1]DACYA/UCM, Madrid, Spain. datienza@dacya.ucm.es ; {hidalgo, julandan}@fis.ucm.es
[2] LSI/EPFL, Lausanne, Switzerland. david.atienza@epfl.ch
[3] VLSI Lab/DUTH, Thrace, Greece. {cmpalouk, lpapadop, dsoudris}@ee.duth.gr
[4]DDT/IMEC, Leuven, Belgium. {poucetc, mamagka, catthoor}@imec.be *

Abstract

Embedded consumer devices are increasing their capabilities and can now implement new multimedia applications reserved only for powerful desktops a few years ago. These applications share complex and intensive dynamic memory use. Thus, dynamic memory optimizations are a requirement when porting these applications. Within these optimizations, the refinement of the Dynamically (de)allocated Data Type (or DDT) implementations is one of the most important and difficult parts for an efficient mapping onto low-power embedded devices.

In this paper, we describe a new automatic optimization approach for the DDTs of object-oriented multimedia applications. It is based on an analytical pre-characterization of the possible elementary DDT blocks, and a multi-objective genetic algorithm to explore the design space and to select the best implementation according to different optimization criteria (i.e., memory accesses, memory footprint and energy consumption). Our results in real-life multimedia applications show that the best implementations of DDTs can be obtained in an automated way in few hours, while typically designers would require days to find a suitable implementation, achieving important savings in exploration time with respect to other state-of-the-art heuristics-based optimization methods for this task.

1 Introduction

In forthcoming technologies of embedded systems a great amount of applications (e.g., 3D games, video-players) coming from the general-purpose domain need to be integrated in an extremely compact device. However, embedded systems struggle to execute these complex applications because they come from desktop systems, with very different restrictions regarding memory use features, and more concretely not concerned with an efficient use of the dynamic memory. In fact, a desktop computer typically includes between 512 and 1024 MB of RAM memory at least, as opposed to the 32 or 64 MB present in latest embedded systems. Therefore, one of the main tasks to be developed during the porting process of multimedia applications for embedded multimedia systems is the optimization of the dynamic memory subsystem. To this end, it is required to suitably choose the *Dynamic Data Types (DDTs)* according to the specific application and final system requirements.

To optimize the use of dynamic memory, the designer must choose among a number of possible DDT implementations [1, 23] (dynamic arrays, linked lists, etc.) the best one in each case, 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's front [8], which would try to cover all the optimal implementation points for the aforementioned required design metrics. The construction of this Pareto's front is a very time-consuming process, sometimes even unaffordable. For instance, in the case of an embedded application including 30 different DDTs and 20 design parameters that need to be explored for 16 basic relevant implementations of DDTs for multimedia applications (as proposed in [12]), the number of experiments (i.e., multiple runs of the application) that need to be performed is 9600 implementations, which is not feasible to be manually tested.

Moreover, due to the inter-dependencies of DDTs, namely, one DDT implementation behavior may affect the performance or memory footprint of another one [8], the re-

*Francky Catthoor is also Professor at ESAT/K.U.Leuven

finement process must explore overall combinations of the different DDTs. Thus, the number of experiments to be carried out typically becomes unaffordable even for a small number of DDTs. Also, optimization techniques relying on partial cost estimators to enable covering and pruning of the design space of possible DDT implementations have been proposed for embedded systems [2]. However, in these approaches (even with a predefined set of possible DDT implementation alternatives), the exploration phase for complex applications still takes days due to the lack of methods to capture the aforementioned inter-dependencies and collateral effects of multiple DDTs interacting together, as it occurs in the latest dynamic multimedia applications ported to embedded systems (e.g., games or scalable video rendering applications [21, 22, 8]).

In addition, extending the set of available DDTs with new implementations of multi-layered (complex) DDTs often proves to be programming intensive. Even when standardized languages (if used at all) offer considerable support, the developer still has to define the access pattern on a case-by-case basis. Thus, the optimization of DDT implementations overall constitutes one of the most difficult design challenges when mapping state-of-the-art dynamic multimedia applications on low-power and high-speed processors, since the target platforms are often not equipped with extensive hardware and system support for dynamic memory.

This paper presents a novel and automatic optimization approach for the DDTs of multimedia applications, which 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)* [18]) to model the existing inter-dependencies of using different DDTs implementations. Then, this modeling of inter-dependencies is utilized to prune the design space, and to select the best choice according to the user's metrics. Hence, given an application to be optimized for a certain embedded system, the proposed optimization framework makes use of application profiling and, in a completely automated way, returns for each of the included dynamic variables in the target application the best multi-layer DDT implementation for a concrete user-defined optimization metric (i.e., memory footprint, memory accesses, energy consumption or linear combinations of them), or a number of overall solutions that respect the defined user constraints (Pareto's front).

This paper is organized as follows. In Section 2, we overview 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) and important research work has been already started through the optimization of dynamic data storage (or DDTs) for embedded systems [10].

Regarding DDT refinement, in general-purpose software and algorithms design [1, 23], primitive data structures are commonly implemented as mapping tables. They are used to achieve software implementations with high performance or with low memory footprint. Additionally, the Standard Template C++ Library (STL) [19] or other proposed templates [4] provide many basic data structures to help designers to develop new algorithms without being worried about complex DDT implementation issues. 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 and while they can be considered as acceptable general-purpose solutions, they 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 started to be proposed for multimedia systems [8, 2]. However, these approaches require the development of efficient pruning function costs 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 method using evolutionary computation achieves. Also, several transformations of data structures for compilers have simplified local loops in embedded programs [16]. 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 [20, 3, 5] 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. Thus, it is possible to perform optimizations in non-convex regular functions,

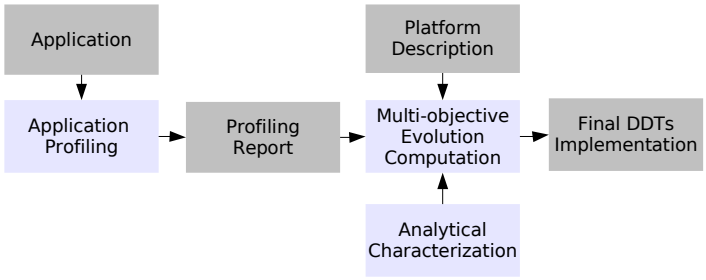

Figure 1: Overview of the DDTs optimization flow

and also to select the order of algorithmic transformations in concrete types of source codes [13, 14, 17]. However, such techniques are not applicable in DDT implementations, due to the initially unpredictable nature of the data to be stored at compile-time.

3 DDTs Global Optimization Flow

The proposed optimization framework uses three different phases to perform the automatic exploration of DDT implementations using evolutionary computation. Figure 1 shows an overview of the different phases (in light gray) and the inputs (in dark gray) required to perform the overall DDTs optimization. In the first phase, there is an initial profiling of the iterator-based access methods to the different DDTs used in the application (Section 3.1). Next, using this detailed report of the accesses to the DDTs done by the application, by using our own analytical characterization of the basic and multi-level DDTs (Section 3.2) and the characteristics of the final platform, an exploration of the design space of DDTs implementation is performed using multi-objective evolution computation (Section 3.3).

3.1 Iterator-Based DDTs Profiling Library

To enable the exploration of different data-type refinements, 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 therefore necessary to profile them to get an accurate view of the different demands of the data-types at runtime. As we want to explore the different potential implementations for the DDTs, it is necessary that this profiling happens not at the memory level, but at the interface level. Thus, we have expanded our profiling library [7] with several higher level profiling packets.

According to the standard interface defined by STL [19], we have re-implemented a sequence type, `vector`, that logs all the different semantical operations. One concrete and more practical advantage to sticking to a commonly used interface, is that limited changes are required in the sources to profile an application, and they can be performed automatically by searching for the declaration of data types in the code sources of the application with a standard parser of a C++ compiler, and then changing the type of the found data types from the STL vector-type to the profiling vector-type, without requiring a modification of the remainder of the application where the data-type is actually being accessed.

A careful analysis of the sequence interface indicates that not only operations of the container, but also the iterator operations used to access the stored elements [5, 1] must be logged. To enable us to couple the logging of memory accesses to specific containers, 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. 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, no distinguishing exists between reads and writes.

Additionally, the iterator methods to access an element or the updating of an iterator, either incrementally or using random offsets, are distinguished as well. The reasoning to differentiate these two types of operations is that this gives different trade-offs for the implementation of a DDT, which need to be explored in our optimization process using evolutionary computation (see Section 3.3). For instance, for an array-like data type, randomly moving the iterator through the contents of the array has O(1) access and large O(n) memory footprint requirements, while for a compact list-like implementation of a sequence, this has O(n) access requirements.

3.2 Analytical Modeling of DDT Implementations

In a first pre-characterization phase we have defined the equations to evaluate the different basic blocks of DDT implementations. A DDT is a software abstraction by means of which we can manipulate and access data. 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 we have classified the DDT implementations in basic DDT and multi-layer implementations relevant for embedded multimedia applications,

DDT implem.	Sequential (NA_s) Access Count	Random (NA_r) Access Count	Average Size (S_{av})
SLL(AR)	$3 \times N_e + N_a$	$\frac{N_e}{2 \times N_a} + 3$	$5s_w + \frac{N_e}{N_a}$
SLL(ARO)	$3 \times N_e + N_a$	$\frac{N_e}{2 \times N_a} + \frac{3 \times N_a}{N_e} + 1$	$7s_w + \frac{N_e}{N_a} \times (3s_w + s_T)$
DLL(AR)	$3 \times N_e + N_a$	$\frac{N_e}{4 \times N_a} + 3$	$6s_w + \frac{N_e}{N_a} \times (4s_w + s_T)$
DLL(ARO)	$3 \times N_e + N_a$	$\frac{N_e}{4 \times N_a} + \frac{N_a}{5 \times N_e} + \frac{5}{4}$	$8s_w + \frac{N_e}{N_a} \times (4s_w + s_T)$
SLL	$3 \times N_e$	$\frac{N_e}{2} + 1$	$4s_w + N_e(2s_w + s_T)$
SLL(O)	$3 \times N_e$	$\frac{N_e}{2} + \frac{1}{N_e}$	$6s_w + N_e(2s_w + s_T)$
DLL	$3 \times N_e$	$4N_e + 1$	$5s_w + N_e(3s_w + s_T)$
DLL(O)	$3 \times N_e$	$\frac{N_e}{4} + \frac{2}{N_e} + \frac{1}{4}$	$7s_w + N_e(3s_w + s_T)$
AR	N_a	1	$N_a \times s_T$
AR(P)	$2 \times N_a$	2	$N_a(s_T + s_w)$

Table 1: Analytical characterization of DDT implementations considered in the exploration

as proposed in [11, 8]. 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 vectors of type s_T. Each element of the list is connected with the next element through a pointer.
- `Double Linked List (DLL)`: is a double linked list of vectors of type Te. Each element of the list is connected with the next and the previous element with two pointers (of size s_w), one pointing to the previous element and one to the next.

In addition, we have included in our exploration the fundamental key variations of basic DDTs for embedded multimedia applications [8, 2] in order to cover effectively the design space of DDT implementations for latest multimedia consumer systems, 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 T_{ref}) useful to access a particular element of a list with less accesses in case of iterator-based access patterns. For instance, for an array if you access element $n + 1$ immediately after element n, your average access count is $1 + 1$ instead of $n/2 + 1$.

Then, these simple DDTs can be combined in multi-layered structures that offer different trade-offs between memory use, performance and energy consumption. These trade-offs are shown in the analytical characterization of the multi-layer DDTs used in our exploration, presented in Table 1. In this table NA_r refers to the number of accesses needed to the retrieve one value with a random access pattern to the DDT, NA_s relates to the total number of accesses required to access all the values stored in the DDT with an iterator-based access pattern. Then, S_{av} indicates the average memory footprint used by the DDT during the execution of the application.

The analyses for both types of access counts and the average memory footprint are based on a number of parameters, which can be extracted from the initial profiling phase of the presented method. These parameters are the following ones: 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, and grouping-level of the basic elements. Then, these equations are used in the evaluation phase of our exploration process of DDT implementations using evolutionary computation, which is explained next (Section 3.3), since dynamic multimedia embedded applications are made of multiple dynamic variables and the use of a concrete DDT implementation for a particular variable can affect the overall cost of the rest of the variables in the multi-objective optimization process.

3.3 Multi-Objective Optimization of DDTs

GAs [9] 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. 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 interdependencies 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 [15].

0 to 1	2 to 5	6 to 8	9 to 12	13 to 14	15 to 18	19 to 21	22 to 25	Bit positions
Levels	Basic Fields	Elements	DS	Levels	Basic Fields	Elements	DS	Meaning
Variable 1				Variable 2				

Figure 2: Example of a 26-bit chromosome

3.3.1 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 variables 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. To this end, we define the implementation of the variables of a program by storing the following information on each chromosome:

- Data Structure (DS): this field represents one of the 16 different possibilities using the previous DDTs analytically characterized (Table 1) and 6 additional basic key merging and splitting combinations, as [8] has proposed for multimedia applications. Therefore, using a binary encoding we need 4 bits.
- Number of elements (Elements): this field represents the grouping factor of elements, up to eight elements (3 bits), which can create optimal access patterns in dynamic multimedia applications, as outlined by [8].
- Number of Levels of the Data Structure (Levels): this field can specify up to 4 levels of basic DDT implementations grouped together (2 bits).
- Basic Fields: Our methodology enables in its encoding the representation of up to 16 fields (4 bits) to cover a large exploration space in this field. However, according to our experience with real-life applications, we typically would not find optimal data structures with more than 7 or 8 basic fields, which can enable reducing the size of this field to improve further the exploration time. Nevertheless, even with this large size for this field, the observed exploration time for real-life applications is very limited, i.e., few hours (see Section 4).

Consequently, using this chromosome structure we need (4+3+2+4)=13 bits to represent the solution proposed for each variable, and if an application has N variables, each chromosome has to be constituted by N*13 bits (genes). For example, the first application we have tested in our experimental results (i.e., Simblob, see Section 4) uses two dynamic variables. A potential solution would be represented by a 26-bit chromosome (see Figure 2). Our current implementation of the exploration framework is able to explore applications with up to 40 DDTs at the same time, which can cover all the real-life embedded multimedia applications we are aware of.

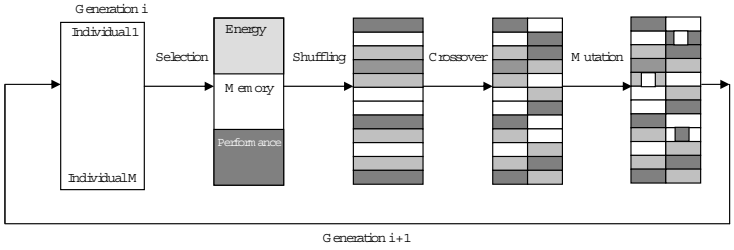

Figure 3: VEGA-based design space exploration method

3.3.2 Fitness function

After performing once the profiling of the real application for a realistic input set, the information required for the analytical characterization of the DDTs implementations considered is available (see Section 3.2 for more details), as well as the number of read (N_r) and write (N_r) accesses to each DDT during execution. Thus, for each DDT implementation available in a certain generation we can compute the performance (`Perf` related to the number of ac-

cesses to layers of the memory hierarchy), memory footprint (`AvMem` in Bytes) and energy values (`Energy` in nJ) using the equations of Table 1, according to the different types of access methods (sequential and random) used in each target embedded application. These values are evaluated for each individual DDT, which represents a solution, by means of a fitness function.

The objective of our algorithm is to obtain DDT implementations that optimize the aforementioned metrics for each variable in the application (energy, memory use and performance). Therefore, the fitness process starts with the decoding of the individuals. Next, for each possible variable (and its valid DDTs in the current generation of the exploration) we compute the following equations:

$$Perf = (NA_r + (NA_s * N_{ran}) * (N_r + N_w) + + (N_{pa}/4) * T_{amem} + (NA_{cd} * 2) \quad (1)$$

$$AvMem = S_{av} \quad (2)$$

$$Energy = (N_{pa} * E_{pa}) + (N_{rw} * E_{rw}) + + (S_{av} * E_{est}) \quad (3)$$

In this work we consider a basic memory hierarchy that consists of a main shared memory and a L1 data-cache. Other memory hierarchies could be modeled as well by modifying the previous equations. In this case, N_{ran} is the number of random accesses to elements stored in the DDT made by the running application, N_{rw} is the number of reads/writes to the L1 data-cache memory and N_{pa} is the number of misses in the data-cache. We consider that each cache line contains four blocks; Thus, the amount of misses is divided by this constant. Then, T_{amem} is the average cycle time that requires an access to the main shared memory and NA_cd is the cycle time cost of creating/destructing the DDT implementation. It has to be included twice since in our modeling all the data structures are created at the beginning and deleted at the end, no DDTs remain allocated when the considered application finishes its execution. Finally, regarding the energy calculations, we consider in this work in-place sharing, as the DDTs lifetimes are short. Then, E_{pa} is the energy consumed per access to main memory, E_{rw} is the energy consumption per access to the cache, and E_{est} is the static energy consumed by the main memory.

Note that according to our empirical validation with several multimedia applications [7], 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.

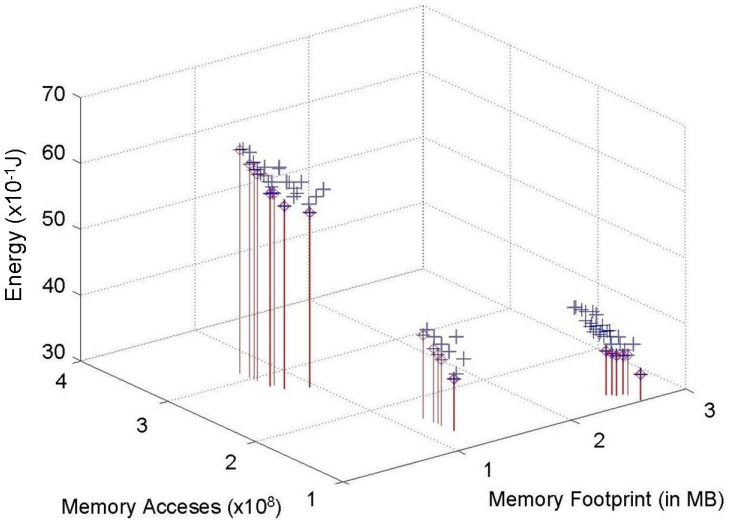

Figure 4: 3D Pareto curve (memory footprint, memory accesses and energy consumption) of combined DDTs implementation solutions for Simblob obtained using the proposed evolutionary-based optimization framework)

3.3.3 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, and 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 [17]. 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's curves for problems of great difficulty, several multi-objective evolutionary algorithms have been recently proposed [6]. Among them, one that has been demonstrated to be very efficient is the approximation proposed by Schaffer [18]. The main idea is an extension of the Simple Genetic Algorithm, which was called VEGA, and that 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 subpopulations 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 [18]. The basic principle states that a

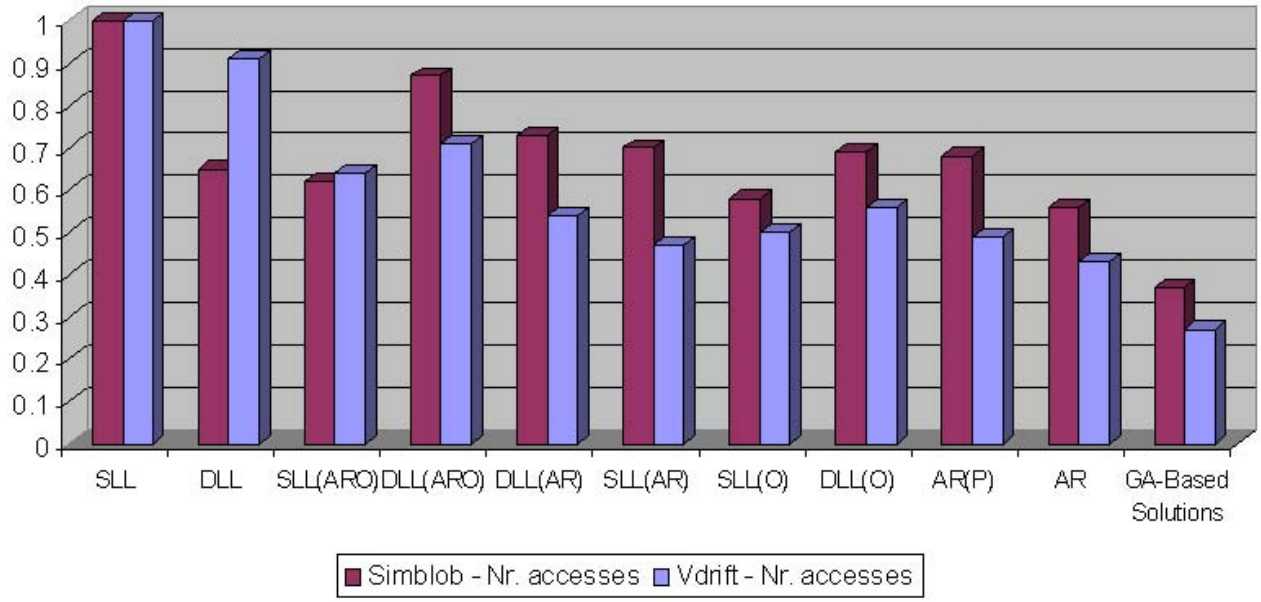

Figure 5: Overall results obtained in the exploration process of minimizing memory accesses for different sets of DDT implementations solutions of Simblob and Vdrift (values normalized to the SLL DDT implementation)

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 (i.e., we could have the evolution of solutions within the population which excel on different objectives). Thus, for our problem with 3 objectives and a population size of M individuals, tree sub-populations of size M/3 each are generated. These sub-populations are mixed together to obtain a new population of size M, where we then apply the GA operators (crossover and mutation) to refine further the solution, as illustrated in Figure 3. 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's front of optimal solutions for the different optimization metrics can be generated (see Section 4 for some examples).

4 Experimental Results

In the first set of experiments, we have used our methodology to explore the optimized configuration of DDTs variables for Simblob [21] (a 3D environment builder) and Vdrift [22] (a racing simulator). The Pareto's front with the major combinations of DDT implementations for Simblob is presented in Figure 4. This figure shows that the presented approach using evolutionary computation can provide a range of possible DDT implementation solutions to the designer according to the weight defined for each optimization metric (i.e., memory accesses, memory footprint and energy consumption in our case). Also, our results trying to minimize the metrics of memory accesses or memory footprint for both applications are shown in Figure 5 and Figure 6, where the `GA-Based Solution` represents the points in the Pareto's fronts for the target applications that the exploration returns to the user (see Figure 4). These solutions achieve the minimal value for the explored metrics (memory footprint or memory accesses, respectively). A similar graph has been obtained for energy, but with other different DDTs implementations as optimal results. For verification purposes, we have implemented the different DDTs shown in the figures and run the applications with each of them, which has shown that the results of relative comparisons between the different DDTs and the solution obtained by our multi-objective GA-based exploration were correct for both applications.

The obtained results illustrate that in real-life applications the solutions found by our multi-objective GA-based exploration for memory footprint are the best possible DDT implementations, in comparison with other possible manual solutions where the 6 main DDTs of Vdrift and all the DDTs of Simblob are implemented using variations of one DDT implementation, as it is usually done in STL-based solu-

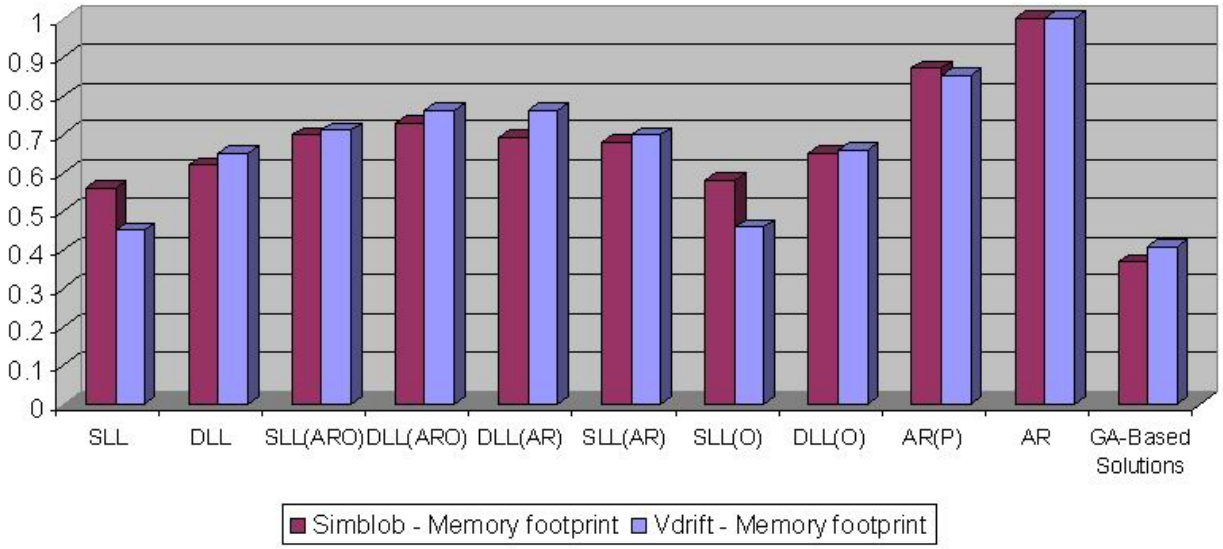

Figure 6: Overall results minimizing the memory footprint metric for different sets of DDT implementations solutions of Simblob and Vdrift (values normalized to the AR DDT implementation)

tions for simplification purposes on handling DDTs. In fact, our GA-based method uses multiple generations of possible solutions (5 generations in the case of Simblob and 23 in the case of Vdrift) to find the correct combination of different DDT implementations for each variable in the two applications, which is a very time-consuming process for a designer to manually tune since there is a large set of different combinations of DDTs implementations. For instance, in the case of Vdrift (for 25 DDTs), the best overall solution found with our methodology uses a combination of DLL, SLL, AR(P),AR, SLL(O) and DLL(O) implementation. Hence, the presented results show the utility of the proposed methodology for designers since they do not need to waste any time on implementing and debugging DDTs implementations that are not useful for their final applications and target metrics, and can just focus on the ones found by the proposed methodology.

In a second set of experiments we have utilized our approach to test its exploration speed in comparison to different alternative methods. The results obtained for both applications for the different tested exploration methods are shown in Table 2. First, we have compared with an almost exhaustive exploration, where we have supposed that a designer starts with all DDTs implementations presented in Section 3.2 already available. Note that we are removing one of the main issues for designers (and clear benefit on our side in the first set of experiments), because the coding and debugging of all variations of DDT implementations is already a very time-consuming process in the overall exploration time. Second, as Table 2 depicts, we have also compared our algorithm with state-of-the-art pruning and optimization methods for DDT implementations presented in [11, 12, 2].

In this case several function costs and deep-first and branch&bound exploration heuristics are used to minimize overall memory accesses, memory footprint and energy consumption figures in embedded multimedia applications.

The results in Table 2 outline that the exploration process with our method is much faster than the optimization process performed using directly the implementations of DDTs, namely 5 minutes versus 13 hours (156x times faster) in the case of Simblob and 20 minutes instead of 9 days in the case of Vdrift (648x times faster). 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 between 16% and 19% for Simblob, and 25% and 29% for Vdrift, respectively. The main reasons for these improvements are the use in our methodology of only an initial profiling phase to characterize the dynamic behavior of the application for all possible DDTs, and 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 few 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).

DDTs Optimization Methods	Simblob	Vdrift
Mem. accesses minimization		
Exhaustive exploration	13 hours	9 days
Deep-First exploration	7 minutes	31 minutes
Branch & Bound exploration	6 minutes	27 minutes
GA-Based proposed method	5 minutes (16% gains)	20 minutes (25% gains)
Mem. footprint minimization		
Exhaustive exploration	14 hours	9 days
Deep-First exploration	9 minutes	37 minutes
Branch & Bound exploration	6.2 minutes	28 minutes
GA-Based proposed method	5 minutes (19% gains)	21 minutes (29% gains)

Table 2: Exploration time to minimize memory accesses or memory footprint of DDT implementations for Simblob and Vdrift using different exhaustive exploration and heuristic-based optimization methods versus the proposed multi-objective GA-based approach

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 to refine the DDT implementations in an automated way; As a result, the proposed approach leads to important savings in overall system integration time for dynamic applications, while achieving optimal implementations of DDT structures with respect to key designer's metrics (memory footprint, energy consumption and performance). Moreover, our experimental results with two real-life multimedia embedded applications show that the presented optimization approach significantly reduces the exploration time with respect to state-of-the-art methods to optimize DDTs implementations while still achieving complete Pareto's 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. Among these lines it is very challenging the study of the possible benefits of more complex and parallel GAs in the efficient exploration of the design space of DDT implementations. 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.

6 Acknowledgements

This work is partially supported by the Spanish Government Research Grant TIN 2005-5619, the Swiss FNS Research Grant 20021-109450/1 and PENED 03 funded by GSRT of Greek Ministry of Development. The authors would like to thank Sergio Belmar Argudo, Cesar M. Gonzalez Iñiguez and Pablo Virseda Benito for their implementation of the MOGA/VEGA algorithm and initial tests done with the two real-life applications used in the paper to validate the exploration of DDTs.

References

[1] James L. Antonakos and Kenneth C. Mansfield Jr. *Practical Data Structures using C/C++*. Prentice Hall, London, UK, 1999.

[2] Sven Wuytack, Francky Catthoor, and Hugo De Man. Transforming set data types to power optimal data structures. *IEEE Transactions on Computer-aided Design*, 15(6):619–629, June 1996.

[3] Luca Benini and Giovanni De Micheli. System level power optimization techniques and tools. In *ACM Transactions on Design Automation for Embedded Systems (TODAES)*, April 2000.

[4] C++ Standardisation Comittee. Programming languages – C++ – ISO/IEC 14882. Technical report, American National Standards Institutes, 11 West 42nd Street, New York, New York 10036, USA, September 1998.

[5] Francky Catthoor, K. Danckaert, C. Kulkarni, Eric Brockmeyer, P.G. Kjeldsberg, Tanja Van Achteren, and T. Omnes. *Data access and storage management for embedded programmable processors*. Kluwer Academic Publishers, Boston, USA, 2002.

[6] Carlos A. Coello, David A. Van Veldhuizen, and Gary B. Lamont. *Evolutionary Algorithms for Solving Multi-Objective Problems*. Kluwer Academic Publishers, New York, NY, USA, 2002.

[7] Christophe Poucet, David Atienza, and Francky Catthoor, Template-Based Semi-Automatic Profiling of Multimedia Applications, In *Proceedings of the IEEE International Conference on Multimedia and Expo*, Toronto, Canada, July 2006. IEEE Computer and Systems Societies.

[8] E.G. Daylight, David Atienza, Anrout Vandecappelle, Francky Catthoor, and Jose M. Mendias. Memory-access-aware data structure transformations for embedded software with dynamic data accesses. *IEEE Transactions on VLSI Systems*, pages 269–280, 2004.

[9] John Holland. *Adaptation in Natural and Artificial Systems*. Univ. of Mitchigan Press, 1975.

[10] Ahmed Jerraya and Wayne Wolf. *Multiprocessor Systems-on-Chips*. Morgan Kaufmann, Elsevier, 2005.

[11] Marc Leeman, Chantal Ykman, David Atienza, Vincenzo De Florio, and Geert Deconinck. Automated dynamic memory data type implementation exploration and optimization. In *Proceedings on Annual Symposium on Very Large System Integration*, Tampa, FL, February 2003. IEEE Computer Society.

[12] Mark Leeman. *Interactive Strategies and Analysis Method for Dynamic Data Type Transformation and Refinement in Multimedia Applications*. PhD thesis, Katholieke Universiteit Leuven, October 2003.

[13] Zbigniew Michalewicz. *Genetic Algorithms + data structures = Evolution Programs*. Springer-Verlag, 1996.

[14] Chris Houck, Jeff Joines, and Mike Kay. *A Genetic Algorithm for Function Optimization: A Matlab Implementation*. NCSU-IE Technical Report 95-09, 1995.

[15] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, USA, 1996.

[16] S Muchnick. *Advanced compiler design & implementation*. Morgan Kaufmann Publisher, San Francisco, CA, 1997.

[17] Andrzej Osyczka. Multicriteria optimization for engineering design. In John S. Gero, editor, *Design Optimization*, pages 193–227. Academic Press, 1985.

[18] J. David Schaffer. Multiple objective optimization with vector evaluated genetic algorithms. In *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 93–100, Mahwah, NJ, USA, 1985. Lawrence Erlbaum Associates, Inc.

[19] SGI. Standard template library, 2006. `http://www.sgi.com/tech/stl/`.

[20] Asim Smailagic, Daniel P. Siewiorec, Drew Anderson, Chris Kasaback, Tom Martin, and John Stivoric. Benchmarking an interdisciplinary concurrent design methodology for electronic/mechanical systems. In *Proceedings of the 32nd ACM/IEEE conference on Design Automation Conference (DAC)*, pages 514 – 519, New York, NY, 1995. ACM Press.

[21] Sourceforge. Simblob - the 3d environment builder framework. `http://sourceforge.net/projects/simblob`.

[22] Sourceforge. Vdrift racing simulator. `http://sourceforge.net/projects/vdrift`.

[23] Derick Wood. *Data structures, algorithms, and performance.* Addison-Wesley Longman Publishing Co., USA, 1993.