Improving Reliability of Embedded Systems through Dynamic Memory Manager Optimization using Grammatical Evolution*

J. Manuel Colmenar‡, José L. Risco-Martín⋆, David Atienza†⋆, Oscar Garnica⋆,
J. Ignacio Hidalgo⋆, Juan Lanchares⋆
‡ C.E.S. Felipe II, Complutense University of Madrid, 28300 Aranjuez, Spain
jmcolmenar@cesfelipesegundo.com
⋆ Dept. of Computer Architecture and Automation, Complutense University of Madrid, 28040 Madrid, Spain
{jlrisco,ogarnica,hidalgo,julandan}@dacya.ucm.es
† Embedded Systems Laboratory (ESL), EPFL, 1015 Lausanne, Switzerland
david.atienza@epfl.ch

ABSTRACT

Technology scaling has offered advantages to embedded systems, such as increased performance, more available memory and reduced energy consumption. However, scaling also brings a number of problems like reliability degradation mechanisms. The intensive activity of devices and high operating temperatures are key factors for reliability degradation in latest technology nodes. Focusing on embedded systems, the memory is prone to suffer reliability problems due to the intensive use of dynamic memory on wireless and multimedia applications. In this work we present a new approach to automatically design dynamic memory managers considering reliability, and improving performance, memory footprint and energy consumption. Our approach, based on *Grammatical Evolution*, obtains a maximum improvement of 39% in execution time, 38% in memory usage and 50% in energy consumption over state-of-the-art dynamic memory managers for several real-life applications. In addition, the resulting distributions of memory accesses improve reliability. To the best of our knowledge, this is the first proposal for automatic dynamic memory manager design that considers reliability.

Categories and Subject Descriptors

C.3 [Special-Purpose and Application-Based Systems]: Real-time and embedded systems; I.2.8 [Artificial Intelligence]: Problem Solving, Control Methods and Search - *Heuristic Methods*

*This work has been supported by Spanish Government grants TIN2008-00508 and MEC Consolider Ingenio CSD00C-07-20811 of the Spanish Council of Science and Technology, and partially supported by the Swiss National Science Foundation (SNF) grant 200021-127282.

GECCO'10, July 7–11, 2010, Portland, Oregon, USA.

General Terms

Design, Reliability, Performance

Keywords

Genetic Programming, Grammatical Evolution, Evolutionary Computation, Embedded Systems Design

1. INTRODUCTION AND MOTIVATION

Modern portable multimedia devices are able to run applications coming from desktop systems, but they usually have less powerful resources and they are limited by the battery power constraints. As a consequence, real-time processing, low energy consumption, cost effectiveness and reliable operation are key requirements that current embedded systems must fulfill.

A common way to increase performance is to improve the hardware resources through very deep sub-micron process technologies. However, technology scaling has driven to couple both design and fabrication process [10]. New technologies become far less mature than earlier ones, leading to potentially less reliable products, specially under 40 nm [15]. Progressive degradation instead of abrupt failure of electrical characteristics of transistors and wires becomes a reality as an intrinsic consequence of the smaller feature sizes and interfaces, as well as increasing electric fields and operating temperatures [7]. Device activity, and the way this is translated into operating conditions of the devices and wires is forecasted to have a major impact on the actual dynamics of the degradation phenomena [15].

As embedded systems are able to run wireless and multimedia applications, the intensive use of dynamic memory is a common issue [4], [5]. Therefore, the memory system is prone to suffer progressive degradation, and has to be designed considering reliability. As a consequence, the design of the *Dynamic Memory Manager (DMM)* may greatly influence on requirements like reliability, performance and energy consumption. Moreover, in terms of performance, a general-purpose DMM may consume up to 38% of the execution time in C++ applications [6]. In terms of memory footprint, despite current DMMs consider block splitting and coalescing in order to improve the memory usage, these managers use standard block sizes, which are not suitable for specialized

applications like multimedia ones [3], [20]. Finally, energy consumption, a key point for embedded systems, is not a priority for that kind of DMM. In summary, general-purpose DMM implementations consider neither reliability nor power consumption or other limitations of target embedded platforms where DMMs must run on [17], [11], [20]. Therefore, these implementations are never optimal for the final target platform, and produce large power and performance penalties.

In [3] and [4], the authors present a new methodology based on high-level programming where C++ mixins [19] are applied. This technique enables the implementation of custom DMMs from their basic parts (e.g., de/allocation strategies, order within pools, splitting, coalescing, etc.), and provides a way to evaluate their power consumption at system-level. The proposal describes an almost-exhaustive exploration of the DMM design space defined for embedded systems. The aim of the authors is to reduce power consumption [3], memory footprint [4] and energy [11]. However, this methodology requires at least two human interventions: (1) the reduction of the initial design space, and (2) the "live" execution of the target application in order to evaluate every candidate custom DMM, which is a very time-consuming process.

A recent work presents a genetic programming method that automatically and efficiently explores the DMM design space exploiting *Grammatical Evolution* (GE) [17] . This approach allows developers to design custom DMMs with the reduced accesses, memory usage and power consumption required for dynamic multimedia applications, with no manual intervention in the exploration effort. This proposal starts from the methodology described in [3] and [4], automatically defining the relevant design space of dynamic memory management decisions for minimal memory accesses, memory usage and energy consumption. Then, applying GE, the design space is traversed according to the dynamic memory behavior of the target application. The evaluation of each generated DMM is automatically obtained by simulation-mode extension of the evaluation technique proposed in [3]. As a result, the work defines the initial design space for particular multimedia embedded applications and presents an evolutionary-based automatic exploration of DMMs for dynamic multimedia applications.

The aim of the work presented in this paper is to complement the work proposed in [17] by considering reliability. Dynamic memory device activity may be measured through the number of accesses to each memory block. As the number of consecutive accesses to a given block increases, the device temperature gets higher, leading to gradual reliability loss and delay reduction [15]. Therefore, one plausible technique to naturally improve reliability is to distribute the memory accesses both in space and time. After reviewing literature, and up to our knowledge, this is the first attempt to provide developers a method to automatically obtain custom DMMs with the optimized performance, memory usage and energy consumption required for the new multimedia applications, being aware of memory reliability. This approach develops a non-supervised exploration of the space of solutions obtaining a result where the execution time, memory usage and energy consumption are reduced in relation to general-purpose DMMs, obtaining a distribution of memory accesses that improves reliability.

The rest of the paper is organized as follows. First, Section 2 describes the design space of DMMs for embedded systems. Then, Section 3 details how GE is applied in this context of DMM search, while Section 4 presents the new reliability-aware method to optimize the search for DMMs. Section 5 shows the results of this method for two real-life applications and, finally, Section 6 draws conclusions and discusses future work.

2. DESIGN SPACE OF DMM FOR EMBEDDED SYSTEMS

The design of DMMs is an extensive field, where multiple solutions have been implemented [20]. However, few of them consider a complete search space useful for a systematic exploration in multimedia applications for embedded systems. Among these approaches, we focus on the works presented in [3], [4], [11] and [17], since we have extended the DMM exploration and optimization methodology the authors developed.

Dynamic memory management basically consists of two separate tasks, i.e., allocation and deallocation. Allocation is the mechanism that searches for a block big enough to satisfy the request of a given application. Deallocation is the mechanism that returns this block to the available memory of the system in order to be reused later. In real applications, the blocks are requested and returned in any order, thus creating "holes" among used blocks. Moreover, the size of the requested block may be bigger than the required size, with the consequent memory underuse. Hence, on top of memory de/allocation, the DMM has to take care of those memory issues. Finally, to support these mechanisms, additional data structures are built to keep track of the free and used blocks. As a consequence, a large memory footprint usually represents an inefficient use of memory due to problems like those described above.

In order to create an efficient DMM, the design decisions that can be taken to handle the possible combinations of the previous factors must been classified. Therefore, we define the taxonomy to classify such design decisions.

A DMM is formed by one or more *Atomic Dynamic Memory Managers (ADMs)*. Then, an ADM may be described as the combination of features coming from the four different categories depicted in Figure 1 and described below:

A. *Creating block structures.* This category handles the way block data structures are created and later used by the ADM to satisfy the memory requests. More specifically, the *Block structure* tree in Figure 1 specifies the data structure that manages the free blocks (singly-linked list, doubly-linked list, AVL trees, etc). The *Block sizes* tree refers to the different sizes of basic blocks available in the ADM, which may be fixed or not. The *Block tags* and the *Block recorded info* trees specify the extra fields needed inside the block to store information used by the ADM. Finally, the *Flexible block size manager* tree decides if the splitting and coalescing mechanisms are activated according to the availability of the size of the memory block requested.

B. *Allocating blocks.* It deals with the allocation policy applied to the ADM. Here we may include all the important choices available in order to choose a block from a list of free blocks [20].

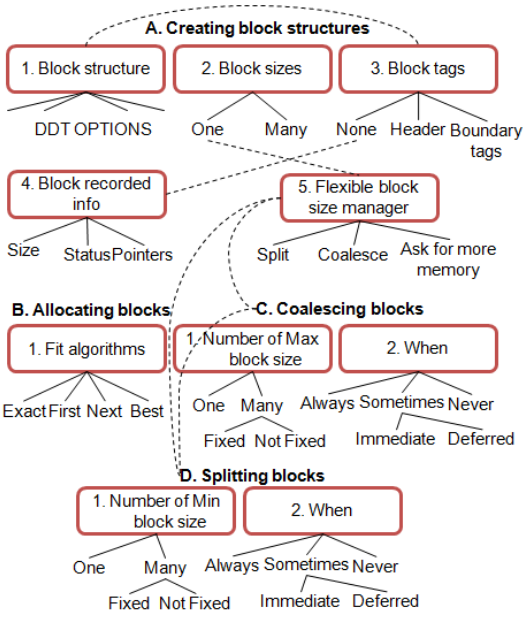

Figure 1: DMM search space of decisions.

C. *Coalescing blocks.* It concerns the actions executed by the ADM to merge two smaller blocks into a larger one. Firstly, the *Number of max block size* tree defines the new block sizes that are allowed after coalescing two different adjacent blocks. Then, the *When* tree defines how often coalescing should be performed.

D. *Splitting blocks.* This category refers to the actions executed by the ADM to split one larger block into two smaller ones. Firstly, the *Number of min block size* tree defines the new block sizes that are allowed after splitting a block into smaller ones. And the *When* tree defines how often splitting should be performed.

Basically, when one decision has been taken in every tree, one custom ADM is defined. However, the decision categories and trees presented above include some interdependencies (constraints, in our design space). Therefore, the selection of certain leaves in some trees heavily affects the coherent decisions in the others. These dependencies are represented using dashed lines in Figure 1.

This approach allows us the reduction of the complexity of the DMM global design in smaller sub-problems, which correspond to the decisions to be taken in order to design the different ADMs that could form a DMM.

We have developed a C++ library based on abstract classes and templates [19] that covers all the possible decisions in the DMM design space depicted in Figure 1. To this end, we have followed the same structure described in [5] and [3], developing several data structures and de/allocation policies, utility layers, object representations, etc. This template-based approach largely simplifies the complex engineering process of designing custom DMMs, allowing the developers to cover a vast part of the implementation space (e.g., different strategies of the DMM, internal blocks of the allocators, etc.) with a minimal programming and modeling effort.

Thus, our library enables the construction of the final global custom DMM via composition of C++ layers. In general terms, the basic interface defined in such DMM library, called *AtomicDMM*, is based on a C++ template. As stated before, every DMM is formed by a set of ADMs, and each ADM is defined by the following class prototype:

```
template<class DataStructure, class Selector,
         class Migration, class NextADM>
class AtomicDMM {
  ...
  inline void* malloc (size_t sz) { ... }
  inline void  free   (void* ptr) { ... }
  ...
};
```

where

- *DataStructure* is the data structure of the ADM designed for a certain region of memory. It should include the type of data structure and policies for blocks sorting and selection that are used in that manager (trees A.1, A.3, A.4 and B.1 in Figure 1).
- *Selector* includes the set of conditions determining the range of block sizes that will be attended by this ADM. If there are several ADMs with the same range, every memory request is attended in descending order as the ADMs are created in the code, in such a way that the last ADM attends requests when there are no free blocks on the previous atomic DMMs (tree A.2).
- *Migration* defines the set of rules determining the range of block sizes that are returned (freed) by this atomic DMM. Using this parameter, block migration policies between different ADMs can be defined, that is, coalescing and splitting policies (trees A.5, C.1, C.2, D.1 and D.2 in Figure 1).
- *NextADM* is the next ADM in the global manager' structure. If there are no more ADMs, it represents the interface used by the *Operating System (OS)* to de/allocate memory (*sbrk()*, *mmap()*, *malloc()*, etc.).

For illustration purposes, in the following example we design a DMM formed by four ADMs. Thus, such DMM manages four different regions of memory. Every region is selected according to the block size that the application needs to de/allocate. The first ADM uses a singly-linked list of blocks with *First Fit (FF)* allocation policy (*FirstFitSLL* data structure). This atomic manager attends de/allocation for 8-bytes-size objects, and does not use coalescing or splitting (*SizeSelector* may be used as both de/allocation size and migration policy). The second atomic manager implements the same behavior, although it is used for 16-bytes-size objects on a doubly-linked list (*FirstFitDLL*), and so forth. Finally, the last region is used for all the requests that cannot be managed by the previous three atomic managers, and corresponds to the operating system layer.

```
typedef CustomDMM <
AtomicDMM< FirstFitSLL<SizeHeader>,
           SizeSelector<8>,SizeSelector<8>,
  AtomicDMM< FirstFitDLL<SizeHeader>,
             SizeSelector<16>,SizeSelector<16>,
    AtomicDMM< FirstFitSLL<SizeHeader>,
               SizeSelector<2^32>,SizeSelector<2^32>,
      OperatingSystem<
        SbrkHeap<EmptyHeader>,2048KB,SizeHeader >
    >
  >
>> GlobalHeap;
```

Moreover, we have extended the DMM library with a *simulation mode*. It allows us not only to execute a real DMM for a certain application, but also to emulate the behavior of a DMM to obtain the performance, the memory used and energy consumed by the embedded application in independent cases or memory allocation situations (see Section 4 for more details). In this way, we are able to evaluate a DMM with an initial profiling of the application, faster than previous approaches, where every DMM is evaluated running the application in real time with a predefined DMM. As a result, the evaluation of a DMM can be performed relatively fast, and exploration algorithms can be included in the searching process since the system designer does not have to implement the DMMs, and try to evaluate them by running the application in real time, in a case-by-case basis.

3. DMM OPTIMIZATION USING GE

Grammatical Evolution (GE) [14] is a grammar-based form of *Genetic Programming (GP)* [16]. It combines principles from molecular biology to the representational power of formal grammars. GE's rich modularity gives a unique flexibility, making it possible to use alternative search strategies (evolutionary, deterministic or some other approach) and to radically change its behavior by merely changing the grammar supplied. Since a grammar is used to describe the structures that are generated by GE, the modification of the output structures may be done by simply editing the plain text grammar. This is one of the main advantages that makes the GE approach so attractive. The genotype-phenotype mapping also means that instead of operating exclusively on solution trees, as in standard GP, GE allows search operators to be applied on the genotype (e.g., integer or binary chromosomes), in addition to partially derived phenotypes, and the fully formed phenotypic derivation trees themselves. When tackling a problem with GE, a suitable *Backus Naur Form (BNF)* grammar definition must initially be defined. Then, in a simulation run, GE can evolve programs in any language described by a BNF.

A grammar can be represented by the tuple $\{N, T, P, S\}$, where N is the set of non-terminals, T the set of terminals, P a set of production rules that maps the elements of N to T, and S is a start symbol that is a member of N. When there are a number of productions that can be applied to one element of N, the choice is delimited with the "|" symbol.

A GE's individual uses a variable-length encoding scheme where each gene holds an integer value that will be mapped to previously labeled production rules of a given BNF by the decoding process. The genotype is used to map the start symbol, as defined in the grammar, onto terminals by reading codons to generate a corresponding integer value. For illustration purposes, we show in Figure 2 an excerpt of a simple grammar that defines a language for DMM implementations. Also, consider the genome example of Figure 3, which has 10 genes with values ranging from 0 to 255 (8-bit number). Since an 8-bit integer is far more than the number of production rules, the modulus operation is needed to decode the genes properly.

The decoding process reads the first gene, namely, 204, that corresponds to the first group of rules of the grammar (**I**). Since there is only one production rule headed by `<CustomDMM>` , the selected one is the production labeled 0 (204 mod 1 = 0) (`<CustomDMM> ::= AtomicDMM ( <DataStructure>, <Selector>, <Migration>, <NextADM>`

```
N = {<CustomDMM>, <DataStructure>, <Header>,
     <Selector>, <Migration>, <NextADM>}
T = {AtomicDMM, FirstFitSLL, BestFitSLL,
     EmptyHeader, SizeHeader, TrueSelector,
     RangeSelector, SizeSelector, Coallesceable,
     OperatingSystem, (, )}
S = <CustomDMM>
P = I   <CustomDMM>     ::= AtomicDMM(<DataStructure>,
                                  <Selector>,
                                  <Migration>,
                                  <NextADM>)        (0)
    II  <DataStructure> ::= FirstFitSLL(<Header>)   (0)
                          |BestFitSLL(<Header>)     (1)
    III <Header>        ::= EmptyHeader             (0)
                          |SizeHeader               (1)
    IV  <Selector>      ::= TrueSelector            (0)
                          |RangeSelector            (1)
                          |SizeSelector             (2)
    V   <Migration>     ::= SizeSelector            (0)
                          |Coallesceable            (1)
    VI  <NextADM>       ::= CustomDMM               (0)
                          |OperatingSystem          (1)
```

Figure 2: Grammar describing a set of DMM implementations.

204	142	55	224	12	7	75	191	233	1

Figure 3: A GE individual's genome.

). Next, there are two productions pointed to by the symbol `<DataStructure>`, so the second gene is read and its value (142), after the modulus operation (142 mod 2), results in 0; therefore, the production `<DataStructure> ::= FirstFitSLL(<Header>)` is picked up. Next, the gene 55 is read and, after the modulus operation (55 mod 2) results in 1 and the production `<Header> ::= SizeHeader` is selected. So far the full decoded expression after the 6th gene will be:

```
AtomicDMM(FirstFitSLL(SizeHeader),
                      SizeSelector,
                      SizeMigration,
                      OperatingSystem)
```

An important advantage of the shown GE representation is that it uses a linear genome. Therefore, GE can directly use all standard genetic algorithm operators.

In order to evolve different DMMs in an optimization process, we have defined a grammar that covers any possible DMM that our template library can instantiate. This grammar is complete enough to implement any well-known DMMs and to explore custom DMM implementations for real-life multimedia embedded applications (see Section 5). Moreover, although this methodology was originally developed to optimize embedded system designs (which introduce strong performance, memory and energy constraints), it can be applied for multi-objective DMM optimization to any computer platform where the use of C++ is allowed.

Just before the GE starts, all the parameters needed in the grammar are initialized according to hardware and software specifications. Then, every individual in the population defines the implementation of a DMM, which is instantiated and simulated over a one-time profiling of the application. This simulation returns the fitness of each individual and the GE continues, selecting after each cycle the best DMM

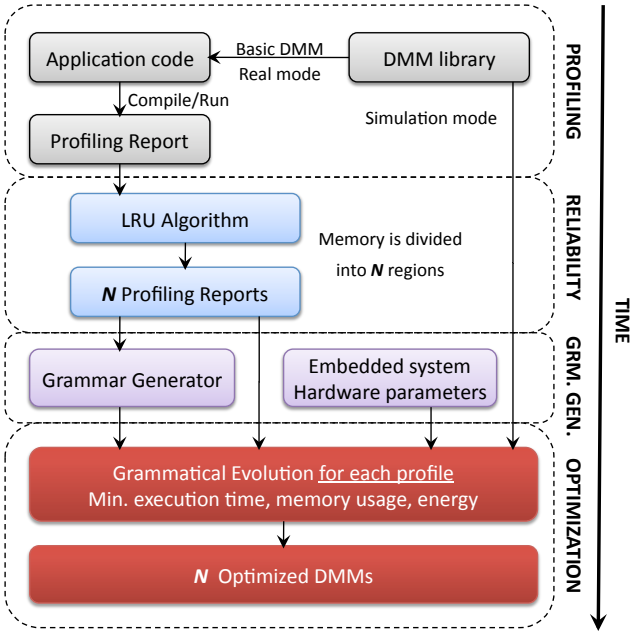

Figure 4: Optimization flow considering reliability.

found in the overall population. In the next section, the phases of this optimization process are detailed.

4. RELIABILITY-AWARE DMM OPTIMIZATION FLOW

The workflow proposed in this work extends the one presented in [17] by adding a new layer and optimization approach that deals with reliability. The resulting four steps required to perform the overall DMMs optimization are shown in Figure 4. Next, we detail these four phases of our proposed optimization flow.

In the first phase we run the application under study using a basic DMM implemented with the DMM library. Such process logs all the required information in an external file: identification of the object created/deleted, operation (allocation, deallocation, read and write) object size in bytes and memory address. To this end, we must only include the DMM library in the source code of the application (only one line of code at the beginning of the application to optimize). As a result, this first phase takes between 30-45 minutes in our methodology for real-life applications, thanks to our tools with very limited user interaction. Then, it comes the reliability-aware phase.

As stated before, degradation of memory devices is strongly related to their activity. Therefore, the higher the number of consecutive accesses to a given memory block (or to its neighbors), the higher the temperature and the lower the reliability. So, in the second phase we automatically distribute the blocks given by the application profile over a number of memory regions by following the *Least Recently Used (LRU)* algorithm. This method guarantees that the most recently used region will be accessed as late as possible, enabling (1) the reduction of the consecutive accesses to the same region and, consequently, (2) the localized degradation of the involved devices. Figure 5 shows a basic example of this phase considering four memory regions. The result of this phase

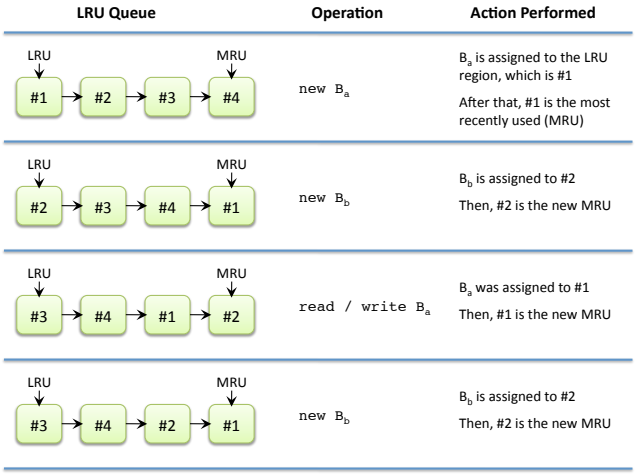

Figure 5: Example of reliability phase considering LRU on four memory regions, denoted as *#i*, and operations for memory blocks denoted as Bi .

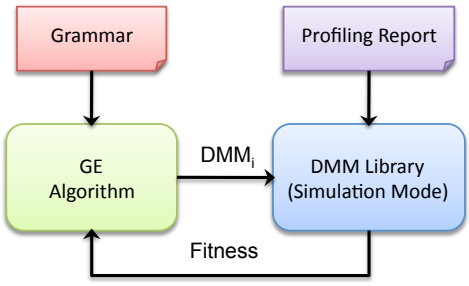

Figure 6: DMM generation and evaluation process.

is a different profile report for each one of the defined memory regions. Each one of the profiles includes the operations related to the blocks assigned to the corresponding region. As a consequence, the objects are spread along the different memory regions, distributing the accesses on a LRU basis.

In the following phase, as Figure 4 shows, we automatically examine the information contained in all the profiling reports, and using the memory size of the embedded system, we obtain a sub-grammar of the original one. Consequently, this phase also reduces the search space. Moreover, some incomplete rules in the original grammar, such as the size of the selectors or the memory size of the embedded system, are automatically defined according to the obtained profiling. To this end, we have developed a tool called *Grammar Generator*. Overall, this phase takes no more than 1-2 minutes with no user interaction.

The last phase is the optimization process. As Figure 4 depicts, this phase consists of a GE algorithm that takes as inputs: (1) the sub-grammar generated in the previous phase, (2) the hardware parameters (e.g., memory size and power consumption model for the embedded memory [12]) of the target embedded system, and (3) the N profiling reports of the application. It also uses the DMM library, extended to simulate the behavior of every DMM generated by the grammar when it is used in the application.

Figure 6 shows a diagram on how our methodology per-

forms. Our GE algorithm is constantly generating different DMM implementations from the grammar file for each one of the memory regions. When a DMM is generated (DMM_i in Figure 6), it is received by the DMM library. Next, the DMM library, working in simulation mode, emulates the behavior of the application debugging every line in the profiling report. This emulation does not de/allocate memory from the computer like the real application, but maintains useful information about how the structure of the selected DMM evolves in time. This proposed exploration methodology is much faster than previous approaches proposed in the literature ([5], [4]), and allows the system designer to use automatic exploration algorithms instead of compiling and running the application for every new DMM. After the profiling has been simulated, the DMM library returns back the fitness of the current DMM to the GE algorithm.

The fitness is computed as a weighted sum of the execution time (performance), memory usage and energy consumed by the proposed DMM for the target embedded system and application under study. Such parameters are indirectly calculated by the DMM simulator in its source code. To this end, the DMM simulator calculates the computational complexity [18] to compute performance, the size of memory de/allocated by the DMM to compute memory usage, and the number of memory accesses to compute energy. In this regard, every portion of the code in the simulator that emulates the behavior of a DMM is accompanied by its corresponding added execution time, memory accesses and memory usage.

When the optimization process ends, the GE algorithm returns the best DMM found, with minimal fitness. In the two real-life benchmarks tested, this phase varies from 10 to 16 hours with no user interaction. It mainly depends on the size of the profiling report. In the performed tests, we have applied GE to profiling reports varying from 3 to 5 GB. Note that in previous approaches, this phase typically takes days or weeks, and for every DMM generated the application must be compiled and executed to evaluate the fitness function [4], [11]. Moreover, the proposed methodology requires much less time than state-of-the-art solutions to this problem [4] because we work with a profiling report, instead of simulating multiple times the complete original application. Furthermore, we do not compile the original application every time a new DMM must be evaluated, which makes our framework even more stable and the overall results more easily comparable, as there is no possible change by the compiler in subsequent system binaries.

5. EXPERIMENTAL RESULTS

We have applied the optimization methodology to two case studies that represent different modern multimedia application domains: the first case study is VDrift [2], which is a driving simulation game. The game includes as main features: 19 different tracks, 28 types of cars, artificial intelligent players and a networked multi-player mode. The second benchmark is a 3D Physics Engine (Physics3D) for elastic and deformable bodies [8], which is a 3D engine that displays the interaction of non-rigid bodies

For both applications we show five different experiments labeled as `Kingsley`, `Custom`, `Custom 2`, `Custom 4` and `Custom 8`. The first one corresponds to the performance of the Kingsley memory allocator [20], and will be the normalized reference. Although Kingsley is quite fast and extensively

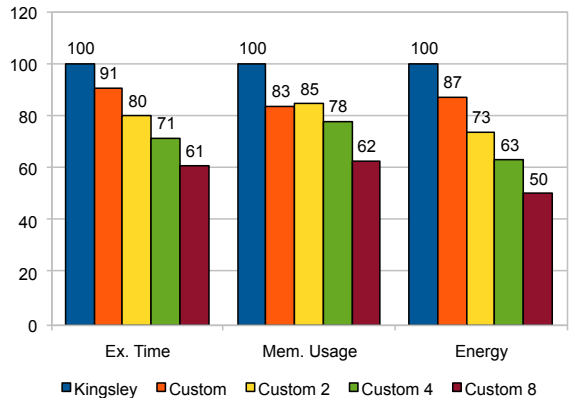

Figure 7: Results of execution time, memory usage and energy consumption of the custom DMMs for 1, 2, 4 and 8 memory regions (normalized to Kingsley) in the VDrift application.

used in embedded operating systems (e.g., RTEMS [1] or Free BSD [9]), it can potentially present a considerable fragmentation due to its use of power-of-two segregated-fit lists. The second experiment, named `Custom`, corresponds to the DMM obtained after the optimization considering one memory region, using our GE proposal. Then, we have run the algorithm considering 2, 4 and 8 different memory regions, and we have collected their performance results. Figures 7 and 9 depict these results labelled as `Custom 2`, `Custom 4` and `Custom 8` respectively. All the experiments were run 10 different times, obtaining the same results on each run.

Table 1: Parameters for the GE algorithm.

Parameter	Value
Population size	60
Number of generations	100
Probability of crossover	0.80
Probability of mutation	0.02

The parameters employed in the GE algorithm for both applications are shown in Table 1. To implement our GE algorithm, we have used GEVA [13], a well-known GE tool implemented in Java.

5.1 Optimization for VDrift

The dynamic behavior of the VDrift case study shows that only a very limited range of data type sizes are used in it, namely 11 different allocation sizes are requested. In addition, most of these allocated sizes are relatively small (i.e., between 32 or 8192 Bytes) and only very few blocks are much bigger (e.g., 151 KBytes). Furthermore, we see that most of the data types interact with each other and are alive almost all the execution time of the application. Within this context, we apply our methodology using the flow presented in Figure 4, minimizing at the same time the execution time, memory usage and energy consumed by the DMM, dividing the memory on different number of regions.

As Figure 7 shows, dividing the memory into regions with a particularly customized DMM for each one leads to reduc-

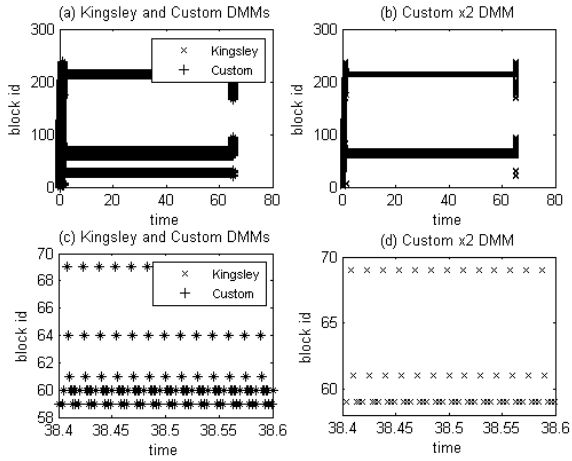

Figure 8: Spatial and temporal accesses distribution for the VDrift application.

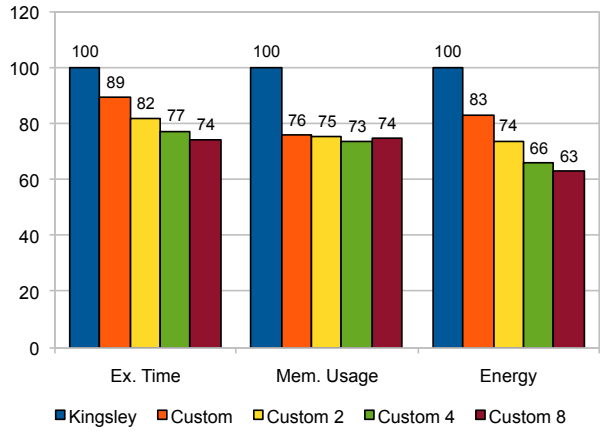

Figure 9: Results of execution time, memory usage and energy consumption of the custom DMMs for 1, 2, 4 and 8 memory regions (normalized to Kingsley) in the Physics3D application.

tions up to 39% on execution time and up to 50% on energy. These results are obtained because the custom DMMs are able to improve the memory usage in two ways. First, because their design and behavior vary according to the different block sizes requested. Second, in pools where a range of block sizes requests are allowed, they use immediate coalescing and splitting services to reduce memory loss. In Kingsley, since only a limited amount of sizes is used, some of the "bins" (or pools of dynamic memory blocks in Kingsley) [20] are underused. Therefore, the custom DMMs employ less memory usage than Kingsley, and this trend is more pronounced when the number of regions increases, obtaining up to 38% on memory saving. However, the reductions on execution time do not include the overhead due to the LRU algorithm, which we will study in future work.

Regarding reliability, we have obtained the distribution of memory accesses for all the five DMMs by accounting the number of times each block was accessed (spatial distribution), and the moment when each block was accessed (temporal distribution). As Figure 8(a) shows, Kingsley and Custom DMMs present a similar distribution where the accesses for both DMMs are overlapped. Figure 8(c) enlarges the upper diagram showing that the spatial and temporal distribution is similar for both. Therefore, the Custom DMM does not improve reliability versus Kingsley because their distributions of memory accesses are similar.

On the other hand, the Custom 2 DMM produces a different pattern. Figure 8(b) depicts a distribution where the accesses to both two regions are overlapped. This figure shows less horizontal lines, which means that less blocks are frequently reused. This detail can be also seen in Figure 8(d), where the upper graph is enlarged. The accesses to each block are separated by longer delays than in Kingsley, as indicated by the spaces between the marks. Therefore, the Custom 2 DMM memory accesses are less concentrated in terms of space and time, which definitely improves reliability. More precisely, Custom 2 DMM distributes accesses over time better than Kingsley by 1.47%. In the same way, Custom 4 and Custom 8 DMMs distribute accesses better than Kingsley by 3.62% and 7.19% respectively. Nevertheless, we have omitted the corresponding figures for the sake of space.

5.2 Optimization for Physics3D

To create our custom DMMs, we have followed again the proposed methodology flow shown in Figure 4. Then, we have compared their behavior with the Kingsley DMM.

As Figure 9 shows, the custom DMM for one memory region reduces the execution time by 11%, because it includes an array to provide direct access to every ADM, as well as direct access to different regions. In addition, our custom DMM manages exact sizes for small and medium blocks, so it does not round up to the next power of two, as Kingsley does, thus improving performance. Custom DMMs with more than one memory region obtain better performances because they manage shorter structures. This occurs because with the same set of blocks, the more the number of regions, the smaller the size of the structures, so they obtain execution time reductions up to 26%. The memory usage is reduced in all custom DMMs, obtaining up to 27% compared to Kingsley. This is due to the fact that the custom DMM managers do not have fixed size blocks to try with multiple accesses, and they attempt to coalesce and split to efficiently use the existing memory, which is a better option in dynamic applications with large variations in requested sizes. Moreover, when large coalesced chunks of memory are not used, they are returned back to the system for other applications. This behavior is similar despite the number of regions. Furthermore, the custom DMMs achieve significantly better results for energy when compared to Kingsley, because most of the dynamic accesses performed internally by Kingsley to its complex management structures are not required in the custom DMMs. As shown in Figure 9, increasing the number of regions reduces the energy consumption by 37%. Again, this occurs because of the smaller size of the internal structures, which require less energy to be managed. Even though Kingsley does not perform splitting or coalescing operations, it suffers from a large memory footprint penalty. This translates into more traffic of memory (and expensive ones, because larger memories would need to be used for the final system due to memory fragmentation) with respect to our custom DMMs. Consequently, for Physics3D, our method-

ology allows to design a set of very customized DMMs that exhibit less fragmentation than Kingsley and, thus, require less memory. Moreover, since this decrease in memory usage is combined with a simpler internal management of dynamic memory, the final DMMs perform less memory accesses and obtain significant reductions in energy consumption as well.

The distribution of memory accesses for Physics3D follows a similar trend than for VDrift. As the number of memory regions increase, the accesses are more distributed both in space and time. Therefore, Custom DMMs with more than one region obtain better reliability.

6. CONCLUSIONS AND FUTURE WORK

Current high-end embedded systems like smart phones, PDAs or portable video game stations execute complex applications where dynamic memory is an important issue. In addition, new sub-micron technologies, that allow increasing performance of those platforms, present reliability problems due to the device activity. These problems are likely to be present on memories, due to their intensive use in embedded systems. Therefore, the optimization of the DMM should take into account reliability in order to get the best results.

In this paper we present an optimization workflow that allows the automatic design of DMMs for embedded systems considering reliability. This approach considers the division of the memory into different regions under the LRU algorithm. Then, applying GE, our method obtains optimized DMMs for each one of the defined regions. As shown in the experimental results, the execution time, memory usage and power consumption obtained by our optimized DMMs using GE are significantly better than those obtained with one of the fastest and frequently used general-purpose managers, optimized for latest embedded systems, the Kingsley dynamic memory manager. Our custom DMMs outperforms the Kingsley DMM in all the configurations for different memory regions and present distributions of memory accesses that improve reliability. Our future work includes the automatic exploration of the optimal number of regions for the target application, the evaluation of the LRU algorithm overhead and the transformation of our proposal into a multi-objective algorithm.

7. REFERENCES

[1] Real-Time Operating System for Multiprocessor Systems (RTEMS). http://www.rtems.com, 2008.

[2] VDrift racing simulator. http://vdrift.net, 2008.

[3] D. Atienza, S. Mamagkakis, F. Poletti, J. M. Mendias, F. Catthoor, L. Benini, and D. Soudris. Efficient system-level prototyping of power-aware dynamic memory managers for embedded systems. *Integr. VLSI J.*, 39(2):113–130, 2006.

[4] D. Atienza, J. M. Mendias, S. Mamagkakis, D. Soudris, and F. Catthoor. Systematic dynamic memory management design methodology for reduced memory footprint. *ACM Trans. Des. Autom. Electron. Syst.*, 11(2):465–489, 2006.

[5] E. D. Berger, B. G. Zorn, and K. S. McKinley. Composing high-performance memory allocators. *SIGPLAN Not.*, 36(5):114–124, 2001.

[6] B. Calder, D. Grunwald, and B. Zorn. Quantifying behavioral differences between C and C++ programs. *Journal of Programming Languages*, 2:313–351, 1995.

[7] G. Groeseneken, R. Degraeve, B. Kaczer, and P. Roussel. Recent trends in reliability assessment of advanced CMOS technologies. In *Microelectronic Test Structures, 2005. ICMTS 2005. Proc. of the 2005 Intl. Conference on*, pages 81–88, April 2005.

[8] L. Kharevych and R. Khan. 3D physics engine for elastic and deformable bodies. University of Maryland, College Park, 2002.

[9] J. Kozubik. FreeBSD and solid state devices. Available at: http://www.freebsd.org/doc/en/articles/solid-state/index.html, 2001.

[10] K. Maex, M. Stucchi, M. Bamal, E. Grossar, W. Dehaene, A. Papanikolaou, M. Miranda, and F. Catthoor. Technology aware design and design aware technology. In *Integrated Circuit Design and Technology, 2005. ICICDT 2005. 2005 Intl. Conference on*, pages 77–81, May 2005.

[11] S. Mamagkakis, D. Atienza, C. Poucet, F. Catthoor, and D. Soudris. Energy-efficient dynamic memory allocators at the middleware level of embedded systems. In *EMSOFT '06: Proc. of the 6th ACM & IEEE Intl. Conference on Embedded software*, pages 215–222, New York, NY, USA, 2006. ACM.

[12] M. Mamidipaka and N. Dutt. eCACTI: An enhanced power estimation model for on-chip caches. Technical Report TR-04-28, CECS, UC Irvine, 2004.

[13] M. O'Neill, E. Hemberg, C. Gilligan, E. Bartley, J. McDermott, and A. Brabazon. GEVA - grammatical evolution in Java. *SIGEVOlution*, 3(2):17–22, 2008.

[14] M. O'Neill and C. Ryan. Grammatical evolution. *IEEE Trans. Evolutionary Computation*, 5(4):349–358, 2001.

[15] A. Papanikolaou, H. Wang, M. Miranda, F. Catthoor, and W. Dehaene. *VLSI-SoC: Research Trends in VLSI and Systems on Chip*, chapter Reliability Issues in Deep Deep Submicron Technologies: Time-Dependent Variability and its Impact on Embedded System Design, pages 119–141. Boston: Springer, 2007.

[16] R. Poli, W. B. Langdon, and N. F. McPhee. *A field guide to genetic programming*. Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk., 2008.

[17] J. L. Risco-Martín, D. Atienza, R. Gonzalo, and J. I. Hidalgo. Optimization of dynamic memory managers for embedded systems using grammatical evolution. In *GECCO '09: Proc. of the 11th Annual conference on Genetic and evolutionary computation*, pages 1609–1616, New York, NY, USA, 2009. ACM.

[18] M. Sipser. *Introduction to the Theory of Computation*. Course Technology, 2005.

[19] Y. Smaragdakis and D. Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2):215–255, 2002.

[20] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *IWMM '95: Proc. of the Intl. Workshop on Memory Management*, pages 1–116, London, UK, 1995. Springer-Verlag.