Optimization of Dynamic Memory Managers for Embedded Systems Using Grammatical Evolution

José L. Risco-Martín*, David Atienza†*, Rubén Gonzalo*, J. Ignacio Hidalgo*
* DACYA-Complutense University, Madrid, 28040 Madrid, Spain
E-mail: {jlrisco,rubenma,hidalgo}@dacya.ucm.es
† Embedded Systems Laboratory (ESL)-EPFL, 1015 Lausanne, Switzerland
E-mail: david.atienza@epfl.ch

ABSTRACT

New portable consumer embedded devices must execute multimedia applications (e.g., 3D games, video players and signal processing software, etc.) that demand extensive memory accesses and memory usage at a low energy consumption. Moreover, they must heavily rely on Dynamic Memory (DM) due to the unpredictability of the input data and system behavior. Within this context, consistent design methodologies that can tackle efficiently the complex DM behavior of these multimedia applications are in great need. In this article, we present a novel design framework, based on genetic programming, which allows us to design custom DM management mechanisms, optimizing memory accesses, memory use and energy consumption for the target embedded system. First, we describe the large design space of DM management decisions for multimedia embedded applications. Then, we propose a suitable way to traverse this design space using grammatical evolution and construct custom DM managers that minimize the DM used by these highly dynamic applications. As a result, our methodology achieves significant improvements in memory accesses (23% less on average), memory usage (38% less on average) and energy consumption (reductions of 21% on average) in real case studies over the current state-of-the-art DM managers used for these types of dynamic applications. To the best of our knowledge, this is the first approach to efficiently design DM managers for embedded systems using evolutionary computation and grammar evolution.

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*

General Terms

Design, Performance

GECCO'09, July 8–12, 2009, Montréal Québec, Canada.

Keywords

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

1. INTRODUCTION AND RELATED WORK

Modern multimedia embedded systems must be able to run applications coming from desktop systems. However, one of the most important problems that system designers face today is the fast integration of applications coming from a general-purpose domain into a highly-constrained memory device [17], where power consumption is a crucial design and optimization priority, both at the hardware and software level. In the past, most of the implementations that were ported to these embedded platforms stayed mainly in the classic domain of signal processing and actively avoided algorithms that employ *Dynamic Memory (DM)*. Recently, with the emerging market of new portable devices that integrate multiple services such as multimedia and wireless network communications, the need for an efficiently use of DM in embedded low-power systems has arisen.

New consumer applications (e.g., 3D video applications) are now mixed signal- and control-dominated. They must rely on DM for a very significant part of their functionality due to the inherent unpredictability of the input data, which heavily influences global performance and memory usage of the system. Designing them using static worst-case memory usage solutions would lead to an overhead in memory usage and power consumption for these systems [3]. In addition, power consumption has become a real issue in overall system design (both embedded and general-purpose) due to circuit reliability and packaging costs [20]. Thus, optimization in general (and especially for embedded systems) has three goals that cannot be seen independently: memory usage, power consumptions and memory accesses.

Since the DM subsystem heavily influences performance and is a very important source of power consumption and memory usage, flexible system-level implementation and evaluation mechanisms for these three factors must be available at an early stage of the design flow for embedded systems. Current implementations of *DM Managers (DMMs)* can provide a reasonable level of performance for general-purpose systems [21]. However, these implementations do not consider power consumption or other limitations of target embedded platforms where these DMMs must run on. Thus, these general-purpose DMMs implementations may produce large power and performance penalties. Consequently, system designers currently face the need to manually optimize the implementations of the initial DMMs on

a case-per-case basis. However, adding new implementations of (complex) custom DMMs manually often prove to be a very programming-intensive and error-prone task that consumes a very significant part of the time spent in system integration of DM management mechanisms (even if standardized languages such as C or C++ offer considerable support).

Recently, a new high-level programming and profiling approach has been presented [4], [5]. Such methodology, based on abstract derived classes or mixins in C++, is able to implement complex custom DM managers from its basic parts (e.g., de/allocation strategies, order within pools, splitting, coalescing, etc.) [21] in a modular way and to evaluate their power consumption at system-level. Based on their library, authors also define a methodology to implement a DMM by means of an almost-exhaustive exploration. Such exploration is realized in a complete design space for dynamic embedded systems. They aim to reduce power consumption [4] and memory footprint [5]. In addition, this study may be particularized for each embedded application. Finally, in [12], using the same methodology described in [4], they balance the two factors (memory footprint and memory accesses) in order to achieve the most energy efficient. However, they need two preliminary phases that require at least two human decisions: (1) the significant reduction of the initial design space, and (2) the execution of every DMM to evaluate the performance of the final design, whereas our goal in this paper is to automatically and efficiently explore the DMM design space while exploiting grammatical evolution.

In this article we propose a new method, using genetic programming, to automatically generate optimal DMM implementations, thus improving the state-of-the-art exploration approaches. We present a novel approach that allows developers to design custom DM management mechanisms with the reduced memory accesses, memory usage and power consumption required for these new dynamic multimedia applications, with no manual intervention in the exploration effort. First, starting from all the possible DMM implementations that the aforementioned methodology proposes ([4], [5]), we automatically define the relevant design space of DM management decisions for a minimal memory accesses, memory usage and energy consumption. After that, using grammatical evolution, we traverse this design space according to the DM behavior of these new dynamic applications. To evaluate each DMM implementation found by our evolutionary algorithm, we have extended the DMM library developed in [4] to work in simulation mode. It enables a relatively fast evaluation of each DMM implementation for fine-tuning our DM design space exploration results during the optimization process. As a result, the main contributions of this research work are two-fold: (1) the definition of a set of rules that delimits the initial design space for a particular multimedia embedded application and (2) a novel evolutionary-based automatic exploration of DMMs for new dynamic multimedia applications to help designers to create very customized DMMs according to the specific dynamic behavior of each application, since the user can fix maximum values of memory accesses, memory use and energy if the final embedded system requires it.

The remainder of the article is organized in the following way. First, Section 2 presents the construction method for DM managers. Then, in Section 3, we present our design flow to automatically explore DMMs, optimizing memory accesses, memory usage and energy consumption. Section 4 explains how grammatical evolution is applied to generate DMMs. In Section 5, we detail our experimental setup and we discuss the obtained results in two real-life embedded applications. Finally, Section 6 summarizes the main conclusions of this paper as well as our future work.

2. DYNAMIC MEMORY MANAGEMENT DESIGN SPACE FOR EMBEDDED SYSTEMS

Much literature is available about possible implementation choices for DM management mechanisms [21], but just few of them are related to a complete search space useful for a systematic exploration in multimedia applications for embedded systems. First, we summarize the design search space of relevant DM management decisions presented in [4], [5] and [12], since our evolutionary computation methodology is based on the proposed search space.

DM 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 and 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. These holes are known as memory fragmentation. On the one hand, internal fragmentation occurs when a bigger block than the one needed is chosen to satisfy a request. On the other hand, if the memory to satisfy a memory request is available, but not contiguous (thus it cannot be used for that request), it is called external fragmentation. Hence, on top of memory de/allocation, the DM manager has to take care of fragmentation issues. This is done by splitting and merging free blocks to keep memory fragmentation as small as possible. Finally, to support these mechanisms, additional data structures are built to keep track of the free and used blocks. Therefore, to create an efficient DMM, the design decisions that can be taken to handle the possible combinations of the previous factors (e.g., fragmentation, overhead of additional data structures) must been classified.

In [5], all the important design options that can compose the design space of DM management in different orthogonal decision trees have been classified. Orthogonal means that any decision in any tree can be combined with any decision in another tree, and the result should be a potentially valid combination (which does not necessarily mean that it meets all timing and cost constraints). Moreover, the decisions in the different orthogonal trees can be ordered in such a way that traversing the trees can be done without iterations, as long as the appropriate constraints are propagated from one decision level to all subsequent levels. Basically, when one decision has been taken in every tree, one custom DMM is defined for a specific DM behavior pattern.

Then, these trees have been grouped in categories according to the different main parts that can be distinguished in DM managements [21]. They are shown in Figure 1. This new approach allows the reduction of the complexity of the DMM global design in smaller sub-problems that can be decided locally. In such a search space, any combination of a leaf from each of the decision trees represents a valid DMM. This fact leads to a huge amount of potential implementa-

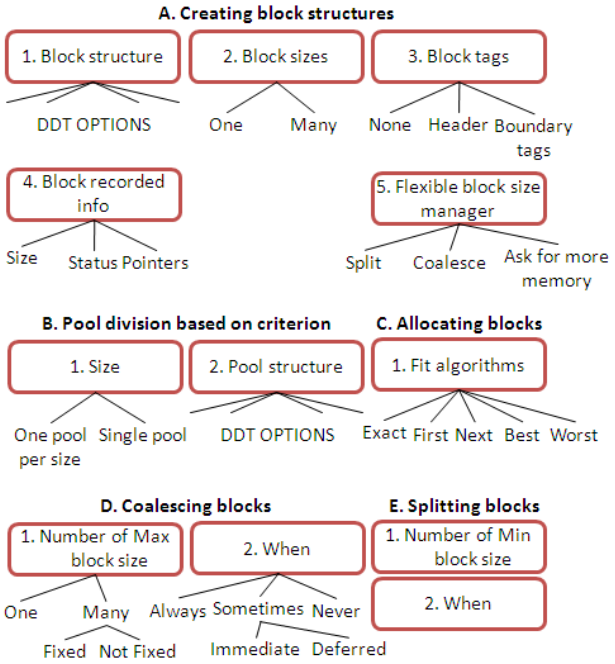

Figure 1: DM management search space of orthogonal decisions

tions that can be used not only to recreate any available general-purpose DMM [21], but also to create new highly-specialized DMMs [4]. The five main categories of Figure 1 and the important decision trees inside them for the creation of DMMs are fully described in [5].

Atienza *et. al* [4] 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. It enables the construction of the final global custom DM manager implementation in a simple way via composition of C++ layers. In general terms, the basic interface defined in such DMM library, called *HeapList*, is based on a C++ template. Then, every DMM is formed by a set of atomic DMMs, and each atomic DMM is defined by the following class prototype:

```
template<class Heap, class AllSel, class FreeSel,
         class Tail>
class HeapList {
  ...
  inline void* malloc (size_t sz) { ... }
  inline void  free   (void* ptr) { ... }
  ...
};
```

where

- *Heap* is the data structure of the atomic DMM 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.
- *AllSel* includes the set of conditions determining the range of block sizes that will be attended by this atomic DMM. If there are several atomic DMMs with the same range, every memory request is attended in descending order, as the atomic DMMs are created in the code. Thus, the last atomic DMM attends requests when there are no free blocks on the previous atomic DMMs.
- *FreeSel* 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 atomic DMMs can be defined.
- *Tail* is the next atomic DMM in the global manager's structure. If there are no more atomic DMMs, it represents the interface used by the *Operating System (OS)* to de/allocate memory (*sbrk()*, *nmap()*, *malloc()*, etc.).

For illustration purposes, in the following example we design a complex DMM formed by three atomic DMMs. Thus, such DMM manages three different regions of memory. Every region is selected accordingly to the block size that the application needs to de/allocate. The first atomic DMM attends de/allocation for 40-bytes-size objects. This atomic manager uses a single-linked list of blocks with *First In, First Out (FIFO)* allocation policy. The second atomic manager is used for 80-byte size objects. Similarly, the last region is used for all the requests that cannot be managed by the previous two atomic managers.

```
typedef SingletonHeap <
HeapList<
 FIFOSLFixedListHeap<SizeHeader>,
 SizeSelector<40>,
 SizeSelector<40>,
 HeapList<
  FIFOSLFixedListHeap<SizeHeader>,
  SizeSelector<80>,
  SizeSelector<80>,
  HeapList<
   FIFOSLBestFitHeap<SizeHeader>,
   TrueSelector,
   TrueSelector,
   FixedHeap<SbrkHeap<EmptyHeader>,
             2048, SizeHeader>
  >
 >
>> GlobalHeap;
```

In the DMM C++ library presented in [4], the authors have developed several heaps, utility layers, object representations, selectors, headers, 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 heap, internal blocks of the allocators, etc.) with a minimal programming and modeling effort. In addition, this library provides a logging layer, which reports at runtime the number, type and size of objects de/allocated by the application under study.

In this research work, we have extended this 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 number of memory accesses, the memory used and energy consumed in independent cases or memory allocation situations (see Section 3.3 for more details). Thus, we are able to evaluate a DMM with an initial profiling of the application much faster than previous approaches, where every DMM needs to be evaluated running the application in real-time with a predefined DMM. As a result, the evaluation of DMM can be performed relatively fast, and exploration algorithms can be included in the searching process. It is indeed much faster than the methodology presented in [4], since the system designer does not have to implement the DMMs, and try to evaluate them on a one-by-one basis by running the application under study in real-time.

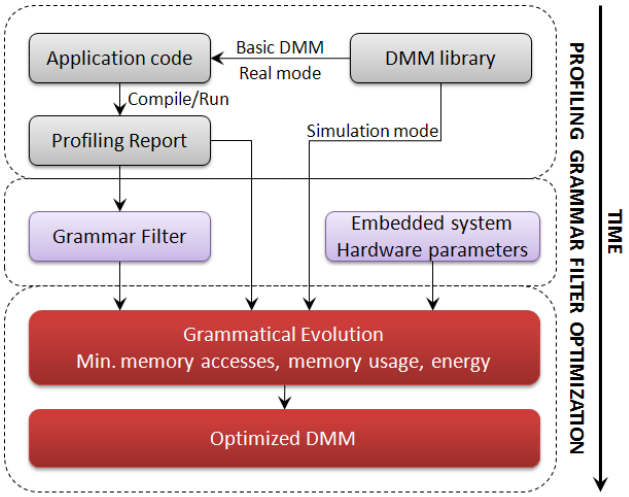

Figure 2: DMMs optimization flow

3. DMM OPTIMIZATION FLOW

The proposed optimization framework uses three different phases to perform the automatic exploration of DMMs using *Grammatical Evolution (GE)*. Figure 2 shows the different phases required to perform the overall DMMs optimization. In the first phase, we generate an initial profiling of the de/allocation pattern of the different objects instantiated by the application. In the second phase, we automatically analyze the profiling report and provide the hardware parameters of the target embedded system to generate a sub-grammar of the original one, more specialized for the application and final embedded system under study. Consequently, such phase also reduces the search space. Finally, in the third phase an exploration of the design space of DMMs implementation is performed using GE. Next, we describe the three phases of our proposed optimization flow.

3.1 Profiling of the application

First, we run the application under study using a basic DM manager implemented with the DMM library. Such process logs all the required information in an external file: identification of the object created/deleted, operation (allocation or deallocation) object size in bytes and memory address. To this end, we must only include the DMM library in the source code of the application (one line of code per variable to profile), as proposed in [4]. As a result, this first phase takes between 2-3 hours in our methodology for real-life applications, thanks to our tools with very limited user interaction.

3.2 DMM grammar filter

In the following phase, as Figure 2 shows, we automatically examine all the information contained in the profiling report, and using the memory size of the embedded system, we obtain a sub-grammar of the original one presented in Section 4. 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 Filter*. This phase takes no more than 1-2 minutes with no user interaction.

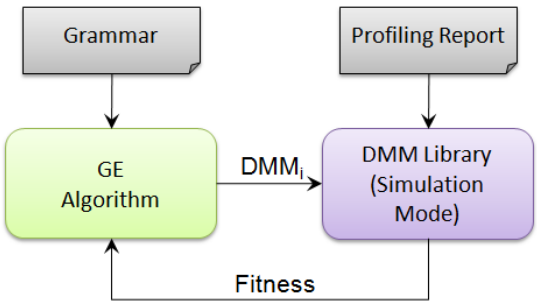

Figure 3: DMM generation and evaluation process.

3.3 Optimization

The last phase is the optimization process. As Figure 2 depicts, this phase consists of a GE algorithm that takes as input: (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 [13]) of the target embedded system, and (3) the profiling report 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 3 shows an illustrative example on how our methodology performs. Our GE algorithm is constantly generating different DMM implementations from the grammar file. When a DMM is generated (DMM_i in Figure 3), 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. Such 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. Such methodology is much faster than previous approaches proposed in the literature ([6], [5]), 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 memory accesses, memory usage and energy consumed by the proposed DMM for the target embedded system and application under study. The amount of memory accesses and memory used is directly calculated by the DMM simulator in its source code. With respect to dynamic and static power consumption, we assume that it is proportional to the execution time of the application using the DMM being evaluated (t_{ex}):

$$Energy\,(DMM_i) \propto t_{ex}\,(DMM_i) \qquad (1)$$

In several research works, *cache misses* have been computed in order to estimate the energy consumed by applications [4], [9], [12], [20]. However, cache misses are costly to compute, since it requires a complex analytical model or a cache-simulator. In this work, we are interested in evaluating a DMM as fast as possible, while being able to explore global trade-offs between different DM managers, so we mainly focus in (accurate enough) high-level exploration, rather than evaluating cycle by cycle the execution of the application in the internal buses of the embedded system.

Furthermore, as it is showed in [9], the most important factor in an energy model is the execution time and the number of memory accesses, which we model accurately.

To measure the execution time of the application under study (and using the DMM proposed by the GE algorithm), the DMM simulator calculates the computational complexity or time complexity [18]. 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 used. The following code snippet shows an illustrative example of how this task is performed:

```
inline void malloc(size_t sz) {
 exTime += 2; memAcc += 2;
 object* ptr = head.next;
 if(ptr!=&tail) {
  exTime += 2; memAcc += 5;
  head.next = ptr->next;
  if(head.next==&tail) {
   exTime++; memAcc += 2; memUsed -= ptr->size();
   tail.next = &head;
  }
  exTime++;
  return (void*)ptr;
 }
 exTime++;
 return 0;
}
```

The first sentence computes the execution time of the pointer assignment (`ptr`) and the evaluation of the `if` condition. The second one takes into account two memory accesses: one for the `head.next` sentence (i.e., access operator) and one because of the `&tail` sentence. This process is repeated until the end of the function, updating the execution time, memory accesses and memory used when needed. In the example presented, the memory used is reduced in `ptr->size()`: this is correct because the DMM does not need to manage this portion of memory, unless it is freed.

When the optimization process ends, the GE algorithm returns the best DMM found, with minimal weighted sum of memory accesses, memory used and energy consumed. This phase takes no more than few 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 2.4 to 3.1 GB. Note that in previous approaches, this phase typically takes days, and requires that the application does not demand user interaction [5]. In any case, our methodology requires a lot less time than state-of-the-art solutions to this problem [5] 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 results more easily comparable overall.

4. DMM OPTIMIZATION USING GRAMMATICAL EVOLUTION

Grammatical Evolution (GE) (e.g., [15], [8], [7]) 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, it is trivial to modify the output structures by simply editing the plain text grammar. When tackling a problem with GE, a suitable *Backus Naur Form (BNF)* grammar definition must initially be defined. The BNF can be either the specification of an entire language or, perhaps more usefully, a subset of a language geared towards the problem at hand. In a simulation run, GE can theoretically evolve programs in any language described by a BNF.

A simplified version of the grammar we have used to explore DMMs in multimedia embedded applications, but that illustrates the principles used in its definition within the proposed exploration framework, is as follows:

```
<GlobalHeap>::=<SingletonHeap>
              |<CoalesceableHeap>
<SingletonHeap>::=SingletonHeap(<HeapList>)
<CoalesceableHeap>::=CoalesceableHeap(<HeapList>)
<HeapList>::=HeapList(<Heap>,
                      <AllSel>,
                      <FreeSel>,
                      <Tail>)
<Heap>::=<FIFOBestFitHeap>
        |<FIFOFirstFitHeap>
        |<FIFOFixedListHeap>
        |<LIFOBestFitHeap>
        |<LIFOFirstFitHeap>
        |<LIFOFixedListHeap>
<FIFOBestFitHeap>::=FIFOBestFitHeap(<Header>)
<FIFOFirstFitHeap>::=FIFOFirstFitHeap(<Header>)
<FIFOFixedListHeap>::=FIFOFixedListHeap(<Header>)
<LIFOBestFitHeap>::=LIFOBestFitHeap(<Header>)
<LIFOFirstFitHeap>::=LIFOFirstFitHeap(<Header>)
<LIFOFixedListHeap>::=LIFOFixedListHeap(<Header>)
<Header>::=EmptyHeader
          |LeaHeader
          |SizeHeader
<AllSel>::=<SizeSelector>
          |TrueSelector
<FreeSel>::=<SizeSelector>
           |TrueSelector
<SizeSelector>::=SizeSelector(<SizeSelectorInBytes>)
<SizeSelectorInBytes>::=#Size1
                       |#Size2
                       |#...
                       |#SizeN
<Tail>::=<FixedHeap>
        |<HeapList>
<FixedHeap>::=FixedHeap(<SbrkHeap>,
                        <MemorySizeInKB>,
                        <Header>)
<SbrkHeap>::=SbrkHeap(EmptyHeader)
<MemorySizeInKB>::=#MemSize
```

In particular, the shown grammar does not include all the DMM library developed in [4] due to space limitations. Nonetheless, this grammar is complete enough to implement many well-known DMMs and to explore custom DMM implementations for the two real-life case studies used in this work (see Section 5 for more details). Moreover, it is straightforward to extend it, following the principles of the shown excerpt, with new classes to create the complete DMM library proposed [4].

The presented grammar includes two global heaps: (1) *CoalesceableHeap*, which allows the DMM to split and coalesce memory, and (2) *SingletonHeap*, which is the most simple DMM allowed since it does not permit neither splitting nor coalescing. As the grammar shows, every global heap contains a *HeapList*. Every *HeapList* is formed by its current heap, an allocator, free-selector and the next heap in the

list. On the one hand, both allocators and free-selectors can be implemented as a *TrueSelector* or a *SizeSelector*. A *TrueSelector* allows the DMM to place an object in an atomic DMM in any case. On the contrary, a *SizeSelector* only allows the DMM to place an object of size `<SizeSelectorInBytes>`. Note that this rule is not complete, since there exists some parameters, i.e., `#Size1`, `#Size2`, ..., `#SizeN` that are defined after the application has been examined, and the final memory subsystem configuration of the target device is inserted in our exploration framework. On the other hand, every heap in a HeapList can be implemented as six different *First-In First-Out (FIFO)* and *Last-In First-Out (LIFO)* reuse strategies (see `<Heap>` rule). Finally, the *HeapList* must end with a heap layer that provides memory directly from the system. It happens when the rule `<Tail>` produces a `<FixedHeap>` rule. This thin wrapper over system-based memory allocators include *SbrkHeap* (built using `sbrk()` for UNIX systems and an `sbrk()` emulation for Windows). The parameter `#MemSize` depends on the embedded system under study and it is defined when the exploration starts. Finally, we have also defined three headers or object representations: *EmptyHeader* which represents just the object, *SizeHeader* that maintains object size in a header just preceding the object, and LeaHeader that does the same but also records whether each object is free in the header of the next object in order to facilitate coalescing.

Just before the GE starts, all the parameters needed in the grammar (`<SizeSelectorInBytes>` and `<MemorySizeInKB>`) are initialized accordingly to hardware and software specifications. After that, every individual in the population defines the implementation of a DMM, which is instantiated and simulated over a profiling of the application. This simulation returns the fitness of each individual and the GE continues, selecting after each cycle the best DMM found in the overall population. In the next section, this process is explained in detail using two real-life embedded multimedia applications.

5. CASE STUDIES AND EXPERIMENTAL RESULTS

We have applied the proposed methodology to two case studies that represent different modern multimedia application domains: the first case study is a 3D Physics Engine (Physics3D) for elastic and deformable bodies [10], which is a 3D engine that displays the interaction of non-rigid bodies. The second benchmark 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. To compare our results with a well-known DMM, we have implemented one of the fastest general-purpose DM managers using our DMM library, namely, the Kingsley memory allocator [21]. Although Kingsley is quite fast and extensively used in embedded operating systems (e.g., RTEMS [1] or Free BSD [11]), it can present a considerable fragmentation due to its use of power-of-two segregated-fit lists.

The parameters employed in the GE algorithm for both applications are shown in Table 1. To implement our GE algorithm, we have used GEVA [14], a well-known GE tool written in Java.

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

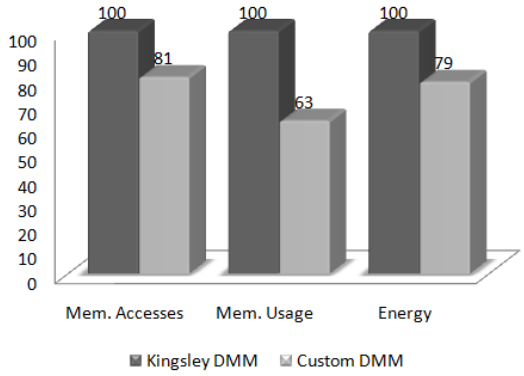

Figure 4: Results of memory accesses, memory usage and energy consumption of the custom DMM obtained by our design framework, normalized to Kingsley, in the Physics3D application.

5.1 Method applied to Physics3D

To create our custom DMM, we have followed the proposed methodology flow shown in Figure 2. We first profile the behavior of the application using a basic DMM implementation. Then, we run our *Grammar Filter* tool with the following results: first, it makes the decision to have many block sizes to prevent internal fragmentation. This is done because the memory blocks requested by the Physics3D application vary greatly in size (to store bodies of different sizes) and if only one block size is used for all the different block sizes requested, the internal fragmentation would be large. Next, the tool chooses splitting or coalescing, so that every time a memory block with a bigger or smaller size than the current block is requested, the splitting and coalescing mechanisms are invoked. In addition, an exact fit to avoid as much as possible memory losses in internal fragmentation is selected. Finally, the Grammar Filter tool selects a header field to accommodate information about the size and status of each block to support splitting and coalescing mechanisms. At the end of this phase, we obtain a reduced grammar file, which is used by the GE algorithm.

Then, we run the optimization phase and compare our custom solution with the Kingsley [21] DM manager, who is a well-known state-of-the-art general-purpose manager for embedded systems. As Figure 4 shows, our custom DMM uses less memory (reduction of 36%) and memory accesses (19% less) than Kingsley. This is due to the fact that our custom DMM manager does not have fixed sized blocks to try with multiple accesses, and tries to coalesce and split as much as needed to efficiently used 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. Furthermore, the results indicate that our custom DMM achieves significantly better results for energy (a 21% reduction), when compared to Kingsley, because most of the dynamic accesses performed internally by Kingsley to its complex management structures are not required in our custom DM manager, which uses a simpler and optimized internal data structures for the target application. Thus, our custom DMM reduces by 21% the energy consumption values of Kingsley. Even though Kingsley does not perform splitting or coalescing operations, it suffers from a large memory footprint penalty and performs unnecessary accesses to traverse all its storage bins in order to find the closest size for each new requested memory allocation. This translates into many unnecessary accesses (and expensive ones, because bigger memories need to be used) with respect to our custom DM manager. Consequently, for Physics3D, our methodology allows to design a very customized DMM that exhibits less fragmentation than Kingsley and, thus, requires less memory. Moreover, since this decrease in memory usage is combined with a simpler internal management of DM, the final DM manager performs less memory accesses and obtains significant reductions in energy consumption as well.

5.2 Method applied to 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 order provided in Figure 2, optimizing memory accesses, memory usage and energy consumed by the DMM. As a result, we obtain a final solution that consists of a custom DMM with 4 separated pools or regions for the relevant sizes in the application. The first pool is used for the smallest allocation size requested in the application, that is, 32 bytes. The second pool allows allocations of sizes between 756 bytes and 1024 bytes. Then, the third pool is used for allocation requests of 8192 bytes. Finally, the fourth pool is used for big allocation requests blocks (e.g., 151 or 265 KBytes). The pool for the smallest size has its blocks in a single-linked list because it does not need to coalesce or split since only one block size can be requested in it. The rest of the pools include doubly-linked lists of free blocks with headers that contain the size of each respective block and information about their current state (i.e., in use or free). These mechanisms efficiently support immediate coalescing and splitting inside these pools, which minimizes both internal and external fragmentation in the custom DMM designed with our methodology. We have tested our obtained DMM Kingsley used in the previous example (i.e., Physics3D). The memory accesses, memory used and energy consumed by both DMMs are depicted in Figure 5.

These results show that the values obtained with the DMM designed using the proposed methodology obtains significant improvements in memory usage compared to the manually designed implementation of Kingsley (38%). This result is obtained because our custom DMM is able to minimize the

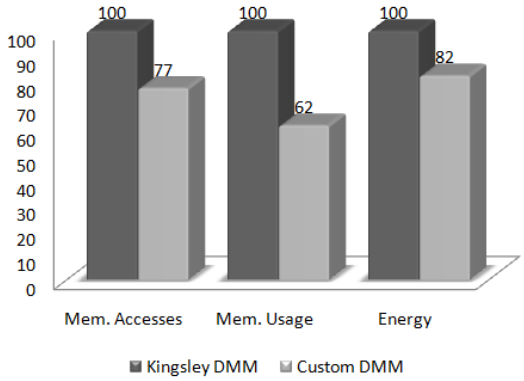

Figure 5: Results of memory accesses, memory usage and energy consumption of the custom DMM obtained by our design framework (normalized to Kingsley) in the VDrift application.

fragmentation of the system in two ways. First, because its design and behavior varies according to the different block sizes requested. Second, in pools where a range of block sizes requests are allowed, it uses immediate coalescing and splitting services to reduce both internal and external fragmentation. In Kingsley, the coalescing/splitting mechanisms are applied, but an initial boundary memory is reserved and distributed among the different lists for sizes. In this case, since only a limited amount of sizes is used, some of the "bins" (or pools of DM blocks in Kingsley) [21] are underused. Therefore, our DM manager employs less memory accesses to DM blocks than Kingsley (i.e., a reduction of 23%). In addition, the final embedded system implementation using our custom DMM achieves better energy results than the implementations using the Kingsley DMM (18% less consumed energy).

6. CONCLUSIONS AND FUTURE WORK

Nowadays, high-performance embedded devices (e.g., PDAs, advanced mobile phones, portable video games stations, etc.) need to execute very dynamic embedded applications. These applications, typically coming general-purpose computers, are very complex for embedded systems and currently demand intensive DM requirements that must be heavily optimized (i.e., memory accesses, memory usage and power consumption) for an efficient mapping on latest low-power embedded devices. System-level exploration and refinement methodologies have started to be proposed to consistently perform that refinement. Within this context, the manual exploration and optimization of the DMM implementation is one of the most time-consuming and programming-intensive parts. In this paper we have presented a new system-level approach based on genetic programming to automatically characterize custom DMMs with an integrated profiling method. This approach largely simplifies the complex engineering process of designing and profiling several implementation candidates, allowing the developers to automatically cover a vast part of the DM management design space (e.g., different strategies of the heap, internal blocks of the allocators, etc.) without any programming and modelling effort. Furthermore, we have shown in our

case studies that the profiling results obtained for memory accesses, memory usage and power consumption by our optimized DMM 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 DM manager. Our future work includes the exploration of possible ways to parallelize the proposed GE algorithm in order to improve further the execution time of the exploration process.

7. ACKNOWLEDGMENTS

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. We would like to express our acknowledgements to "La Caixa" for all their support during the realization of this research.

8. 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. Catthoor, J. M. Mendias, and D. Soudris. Dynamic memory management design methodology for reduced memory footprint in multimedia and wireless network applications. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 10532, Washington, DC, USA, 2004. IEEE Computer Society.

[4] 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.

[5] 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.

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

[7] A. Brabazon, M. O'Neill, and I. Dempsey. An introduction to evolutionary computation in finance. *Computational Intelligence Magazine, IEEE*, 3(4):42–55, Nov. 2008.

[8] I. Dempsey, M. O'Neill, and A. Brabazon. Constant creation in grammatical evolution. *Int. J. Innov. Comput. Appl.*, 1(1):23–38, 2007.

[9] J. I. Hidalgo, J. L. Risco-Martín, D. Atienza, and J. Lanchares. Analysis of multi-objective evolutionary algorithms to optimize dynamic data types in embedded systems. In *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1515–1522, New York, NY, USA, 2008. ACM.

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

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

[12] 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: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 215–222, New York, NY, USA, 2006. ACM.

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

[14] M. O'Neill, E. Hemberg, C. Gilligan, E. Bartley, J. McDermott, and A. Brabazon. Geva - grammatical evolution in java. Technical report, Natural Computing Research & Applications Group - UCD Complex & Adaptive Systems Laboratory, University College Dublin, Ireland, 2008.

[15] M. O'Neill and C. Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. Kluwer Academic Publishers, 2003.

[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, J. I. Hidalgo, and J. Lanchares. A parallel evolutionary algorithm to optimize dynamic data types in embedded systems. *Soft Comput.*, 12(12):1157–1167, 2008.

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

[19] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[20] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim, W. Ye, and D. Duarte. Evaluating integrated hardware-software optimizations using a unified energy estimation framework. *IEEE Trans. Comput.*, 52(1):59–76, 2003.

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