Multi-Objective Optimization of Dynamic Memory Managers using Grammatical Evolution*

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

ABSTRACT

The dynamic memory manager (DMM) is a key element whose customization for a target application reports great benefits in terms of execution time, memory usage and energy consumption. Previous works presented algorithms to automatically obtain custom DMMs for a given application. Nevertheless, those approaches are based on grammatical evolution where the fitness is built as an aggregate objective function, which does not completely exploit the search space, returning the designer the DMM solution with best fitness. However, this approach may not find solutions that could fit in a concrete hardware platform due to a very low value of one of the objectives while the others remain high, which may represent a high fitness. In this work we present the first multi-objective optimization methodology applied to DMM optimization where the Pareto dominance is considered, thus providing the designer with a set of non-dominated DMM implementations on each optimization run. Our results show that the multi-objective optimization provides Pareto-optimal alternatives due to a better exploitation of the search space obtaining better hypervolume values than the aggregate objective function approach.

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, Algorithms

Keywords

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

1. INTRODUCTION AND MOTIVATION

Dynamic memory is an important resource to be managed, specially in multimedia applications. In fact, studies have shown that dynamic memory management can consume up to 38% of the execution time in C++ applications [5]. Thus, the performance of dynamic memory management can have a substantial effect on the overall performance of highly dynamic applications.

In this way, general-purpose *Dynamic Memory Managers (DMMs)* implemented on current operating systems offer a reasonable performance for most of the applications. However, this performance can be improved by custom DMMs built up considering the issues of a target application. In this regard, three out of the twelve integer benchmarks included in SPEC (`parser`, `gcc`, and `vpr` [16]) and multiple server applications, use one or more custom DMMs [3]. Nevertheless, designing and coding a custom DMM for an application is a complex and error-prone task.

Past works have proposed to solve this problem using flexible infrastructures to build any DMM for C++ applications [3, 2, 1]. Based on these proposals, several recent works proposed optimization methods to automatically obtain optimal custom DMMs through grammatical evolution [14, 13, 7]. In these works, the authors describe different implementations of optimization algorithms and contribute with several approaches like parallel implementations or reliability-aware optimizations. All these proposals were based on the optimization of three different, but combined, objectives: execution time, memory usage and energy consumption. However, these works define the fitness as a single aggregate objective function to be minimized. Therefore, the multi-objective problem is solved through scalarisation [11], which is a widely reported method of solving a multi-objective problem as a mono-objective one. Although these works obtain significant improvements in performance and memory usage with respect to general-purpose DMMs, their approach cannot identify all non-dominated solutions because

*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'11, July 12–16, 2011, Dublin, Ireland.

the aggregation of objectives leads to explore only the convex part of the Pareto front [8].

In fact, the limitation of the fitness function applied to a multi-objective problem is important in the optimization of DMMs, because the algorithm may overlook solutions that could fit in a concrete hardware platform. For example, embedded multimedia systems like smartphones or tablets need low energy consumption, while server computers are more concerned about execution time. Thus, if the algorithm produces a DMM with the shortest execution time, but with a high memory usage, this DMM could be the best choice for a server computer, but it also could be discarded by the selection method based on fitness due to its high memory usage.

On the other hand, a multi-objective optimization, where the concept of non-dominated solutions and Pareto-optimal front (POF) is considered, could provide a set of different solutions with minimum values on any (or all) of the objectives. As a consequence, a designer could select the DMM implementation from the resulting POF that better fits on the target platform for a given application.

In addition, this new multi-objective optimization may obtain DMM implementations with similar objective values but different implementations. Therefore, the designer could select the simplest DMM implementation because the optimization process returns a set of non-dominated solutions.

Following these ideas, in this paper we present a novel Multi-Objective Grammatical Evolution (MOGE) optimization method that allows the designer to automatically obtain a set of non-dominated custom DMMs for a given target application. We considered three optimization objectives, i.e., execution time, memory usage and energy consumption. To the best of our knowledge, this is the first time where the concepts of non-dominated solutions and Pareto-optimal front are applied to obtain optimal custom DMMs. The multi-objective optimization we present has been implemented by merging grammatical evolution and the NSGA-II algorithm [9]. Our experimental results show that the solutions obtained by MOGE are not dominated by the solution obtained through the aggregate objective function approach in almost all cases. In addition, the MOGE optimization obtains a higher number of non-dominated solutions and better hypervolume values than the aggregate objective approach.

The rest of the paper is organized as follows. Section 2 describes the DMM design space and how we model it using grammatical evolution. Section 3 details the new multi-objective optimization algorithm where we merge our multi-objective approach and grammatical evolution. Section 4 shows the results of our method for several benchmarks and, finally, Section 5 draws conclusions and future work.

2. BACKGROUND: DESIGN SPACE OF DMM AND GRAMMATICAL EVOLUTION

Dynamic memory management basically consists of two separate tasks, i.e., allocation and deallocation. On the one hand, allocation is the mechanism that searches for a memory block big enough to satisfy the memory requirements of an object request in a given application. On the other hand, deallocation is the mechanism that returns a freed memory block to the available memory of the system to be reused.

In current applications, the blocks are requested and returned in any order. Thus, the DMM task consists of guiding the allocation and deallocation of memory blocks after each application request. To this end, there exist multiple free block administration policies, data structures to be managed, and also techniques like splitting and coalescing may be implemented in order to improve the behavior of the DMM. Therefore, we face a large design space of solutions that we summarize next.

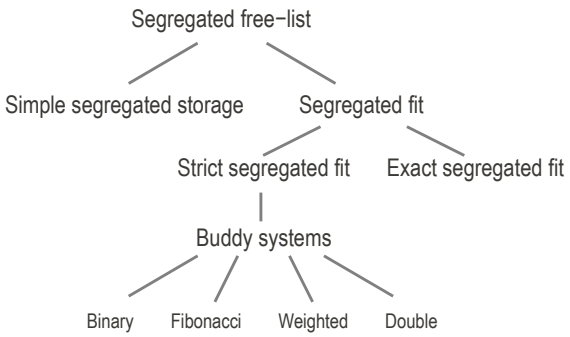

Figure 1: Classification of memory allocators.

2.1 Design space of DMMs

As shown by previous works in the literature [1, 14], a custom DMM that takes advantage of the application-specific behavior is usually the best approach for dealing with dynamic memory operations. Thus, we need an optimization flow able to represent and produce any DMM implementation in the previously described design space.

Among all the previous works about DMM design, we focus our approach in the proposal followed by [1], [2], [14] and [7]. Hence, we consider a DMM as a set of free-blocks lists where different policies can be implemented. As a result, we can define a hierarchy of memory allocators, as shown in Figure 1, that we briefly describe next according to [17].

A segregated free-list allocator divides the free-blocks list into several subsets, taking into account the size of the free blocks. Both allocations and blocks are solved choosing the appropriate list. This class of mechanism usually implements a good fit or best fit policy.

Simple segregated storage is a segregated free-list allocation mechanism which divides the storage into areas, allocating objects of a single size, or of a small range of sizes, within each area. Then, allocation is fast and avoids headers, but may lead to high external fragmentation, as unused parts of areas cannot be reused for other object sizes.

Segregated fit is another variation of the segregated free-list class of allocation mechanisms. It maintains an array of free-blocks lists, each list holding free blocks of a particular range of sizes. The manager identifies the appropriate free-blocks list and allocates from it (often using a first-fit policy [2]). If this mechanism fails, a larger block is taken from another list by splitting it accordingly.

Strict segregated fit is a segregated-fit allocation mechanism, which has only one block size on each free-blocks list. A requested block size is rounded up to the next provided size, and the first block on that list is returned. The sizes must be chosen so that any block of a larger size can be split into a number of smaller sized blocks.

Buddy systems are special cases of strict segregated-fit allocators, which make splitting and coalescing fast by pairing

each block with a unique adjacent buddy block. To this end, an array of free-blocks lists exists, namely, one for each allowable block size. Allocation rounds up the requested size to an allowable size and allocates from the corresponding free list. If the free-blocks list is empty, a larger block is selected and split. A block may only be split into a pair of buddies. A block may only be coalesced with its buddy, and this is only possible if the buddy has not been split into smaller blocks. Different sorts of buddy system are distinguished by the available block sizes and the method for splitting. They include binary buddies (the most common type), Fibonacci buddies, weighted buddies, and double buddies.

Exact segregated fit is a segregated fit allocator, which has a separate free-blocks list for each possible block size. The array of free-blocks lists may be represented sparsely, so large blocks may be treated separately. The implementation depends on the distribution of block sizes between lists.

2.2 Grammatical Evolution-Based Exploration

In order to automatically obtain a custom DMM for a target application, we selected *Grammatical Evolution (GE)* [11] as the optimization method. GE is a grammar-based form of *Genetic Programming (GP)* [12] that is able to evolve hierarchical representations like the memory allocators classification proposed for the DMMs.

Based on the method proposed by the authors of [13], our optimization process begins with a profile of the memory operations of the target application. This profile can be used to create a grammar, which is based on the previous DMM hierarchy, improved by including some application-specific data like the block sizes and the non-terminal symbols. The grammar is subsequently taken to decide if a DMM individual is valid, and also to produce new individuals through the genetic operators.

Figure 2 shows the grammar for the Boxed-sim benchmark optimization process. In this case we find a set of non-terminals symbols, N, a set of terminals, T, a set of production rules, P, that maps the elements of N to T, and the start symbol S, which is a member of N.

Following the production rules, a DMM is built up as a list of allocators (see production rules I and II in Figure 2). There are two kinds of allocators (rule II) whose configuration is made recursively by applying the rest of the production rules. Hence, a DMM is created after taking a set of decisions on the tree represented by the grammar.

Once the grammar is defined, the multi-objective optimization produces different DMM implementations, i.e., individuals that are evaluated and evolved to obtain a set of non-dominated solutions.

```
N = {<DynamicMemoryManager>, <Allocators>,
    <AllocatorSize>, <AllocatorMaxSize>,
    <AllocatorClass>, <Size>, <MaxSize>,
    <AllowSplitting>, <AllowCoalescing>,
    <DataStructure>, <AllocationMechanism>,
    <AllocationPolicy>}
T = {SegregatedFreeList, SimpleSegregatedStorage, ExactSegregatedFit,
      BuddySystemBinary, BuddySystemFibonacci, 4, 8, 12, 16, 20, 24, 32,
      36, 40, 48, 64, 72, 80, 96, 104, 112, 124, 144, 192, 256, 264, 272,
      288, 312, 508, 520, 1584, 2036, 2048, 2304, 4000, 6192, 8256, 24576,
      147456, true, false, SLL, DLL, BTREE, FIRST, BEST, EXACT, FIFO, LIFO}
S = <DynamicMemoryManager>
I    <DynamicMemoryManager> ::= <Allocators>
II   <Allocators>           ::= <AllocatorSize>
                            |<AllocatorMaxSize>
III  <AllocatorSize>         ::= <AllocatorClass>
                             <AllowSplitting>
                             <AllowCoalescing>
                             <Size>
                             <DataStructure>
                             <AllocationMechanism>
                             <AllocationPolicy>
                             <Allocators>
IV   <AllocatorMaxSize>      ::= <AllocatorClass>
                             <AllowSplitting>
                             <AllowCoalescing>
                             <MaxSize>
                             <DataStructure>
                             <AllocationMechanism>
                             <AllocationPolicy>
V    <AllocatorClass>        ::= SegregatedFreeList
                            |SimpleSegregatedStorage
                            |ExactSegregatedFit
                            |BuddySystemBinary
                            |BuddySystemFibonacci
VI   <Size>                  ::=1|4|5|6|8|16|24|28|32
                            |52|128|544|853|2329|4658
VII  <MaxSize>               ::= 147456
VIII <AllowSplitting>        ::= true|false
IX   <AllowCoalescing>       ::= true|false
X    <DataStructure>         ::= SLL|DLL|BTREE
XI   <AllocationMechanism>   ::= FIRST|BEST|EXACT
XII  <AllocationPolicy>      ::= FIFO|LIFO
```

Figure 2: Grammar file obtained for Boxed-sim.

$$\begin{aligned} &\textit{Minimize} \quad \vec{z} = (f_1(\vec{x}), f_2(\vec{x}), f_3(\vec{x})) \\ &\textit{subject to} \qquad\quad \vec{x} \in X \end{aligned} \tag{1}$$

where $\vec{z}$ is the objective vector with 3 objectives to be minimized: execution time (f_1), memory usage (f_2) and energy consumption (f_3); $\vec{x}$ is the decision vector, which corresponds to a DMM implementation, and X is the feasible region in the decision space, which corresponds to all the possible DMMs produced by the grammar defined in the previous section.

As stated before, previous approaches solved the DMM optimization by considering a fitness function defined as a single aggregate objective function:

$$\textit{fitness}(\tilde{x}) = c_1 f_1(\tilde{x}) + c_2 f_2(\tilde{x}) + c_3 f_3(\tilde{x})$$

Although this is a common approach [8], the solution obtained will depend on the relative values of the specified weights. Moreover, the weighted sum method is essentially subjective, as the decision manager needs to supply the weight of every objective.

In addition, the approach of aggregated objectives cannot identify all non-dominated solutions, but only solutions located on the convex part of the Pareto front can be found. The objective way of solving multi-objective problems requires a Pareto-compliant ranking method, favoring non-dominated solutions, as seen in current multi-objective evolutionary approaches such as NSGA-II and SPEA2. Here, no weight is required and thus no *a priori* information on the problem is needed [8].

Therefore, we tackle the DMM optimization problem by considering the concept of dominance. Hence, a solution

3. PROPOSED MULTI-OBJECTIVE DMM OPTIMIZATION

3.1 Problem definition

The goal of the multi-objective optimization is to simultaneously optimize several objectives that could be sometimes contradictory. For such kind of problems, a single optimal solution does not exist, and some trade-offs among the different involved variables need to be considered.

In the case of DMM optimization, we consider the following 3-objective minimization problem:

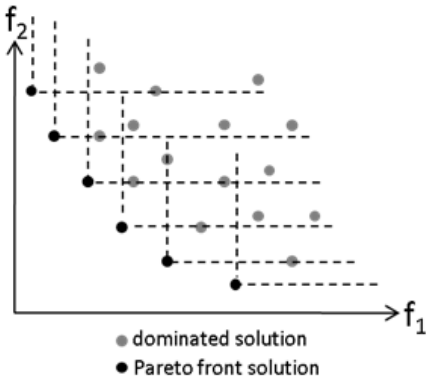

Figure 3: Non-dominated solutions of a set of solutions in a two objective space.

$\vec{x} \in X$ is said to dominate another solution $\vec{y} \in X$ (denoted as $\vec{x} \prec \vec{y}$) iff the following two conditions are satisfied:

$$\forall i \in \{1,2,3\}, f_i(\vec{x}) \leq f_i(\vec{y})$$
$$\exists i \in \{1,2,3\}, f_i(\vec{x}) < f_i(\vec{y}) \quad (2)$$

Then, a DMM implementation DMM_i dominates a different solution DMM_j iff all the execution time, memory usage and energy consumption values of DMM_i are lower or equal to those of DMM_j and, in addition, almost one of those values of DMM_i is strictly lower than in DMM_j.

A decision vector $\vec{x} \in X$ is non-dominated with respect to X if another $\vec{x}' \in X$, such that $\vec{x}' \prec \vec{x}$, does not exist.

A solution $\vec{x}^* \in X$ is called Pareto-optimal if it is non-dominated with respect to X.

An objective vector is called Pareto-optimal if the corresponding decision vector is Pareto-optimal. Therefore, the non-dominated set of the entire feasible search space X is the *Pareto-optimal Set (POS)*. Then, the image of the POS in the objective space is the *Pareto-optimal Front (POF)* of the multi-objective problem at hand. Figure 3 shows a particular case of the POF in the presence of two objective functions. Consequently, a multi-objective optimization problem is solved, when its complete POS is found.

In particular, the POF and POS correspond to the analytical solution of a multi-objective problem. However, the multi-objective optimization we propose in this work obtains a set of solutions corresponding to non-dominated DMM implementations that may belong to the POS. Hence, this is an experimental approach, thus we can state that the optimization method obtains an approximated set of non-dominated solutions [10]. This approach is appropriate for the DMM optimization problem because the obtention of the analytical solution is not affordable.

In the following subsection we detail the internals of the multi-objective optimization method we propose.

3.2 New MOGE optimization

To solve the aforementioned problem, we propose a new optimization method, namely, the Multi-Objective Grammatical Evolution (MOGE) approach, which consists of a multi-objective optimization algorithm that merges GE and the main mechanisms of the NSGA-II, i.e., fast non-dominated sorting algorithm based on crowding distance and elitism.

Figure 4 shows the optimization flow of MOGE. As this figure shows, the flow starts by considering an initial population of individuals representing DMMs. In this scenario, the generation of the initial population is done by using the grammar that comes from the profile of the target application. After that, each iteration of the algorithm follows the same pattern. Firstly, the non-dominated solutions (elites) are extracted from the population. Then, selection is applied on the rest of the population. In our implementation, selection consists on a tournament where dominance is considered and, in case of mutual dominance between two individuals, crowding distance is applied. After that, the crossover and mutation operators are applied in the same way as the authors of [14] and [7] propose. Next, the objective evaluation is performed by using the simulator proposed by the authors of [15]. Finally, the population is updated with the resulting individuals from the previous operations merged with the formerly extracted elites. At the end of each iteration the population maintains the same number of individuals as stated in the configuration of the algorithm, because the worst solutions in terms of dominance and crowding distance are discarded on the normalization step.

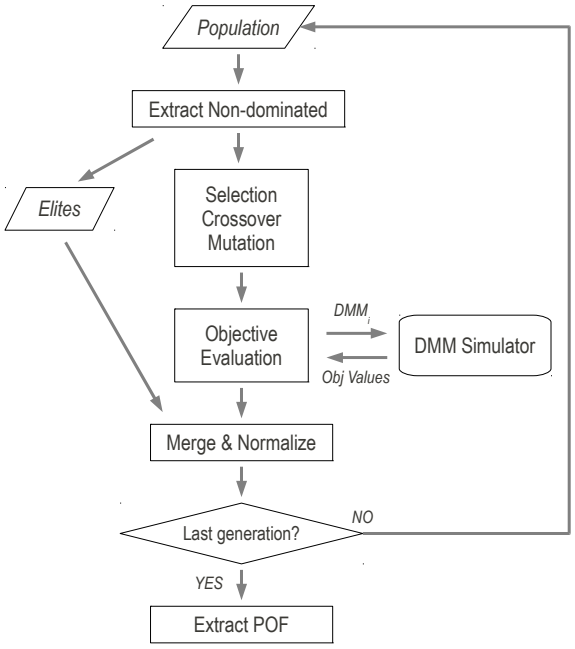

Figure 4: MOGE optimization flow.

4. EXPERIMENTAL SETUP AND RESULTS

In order to evaluate the multi-objective optimization, we have selected six benchmark programs with different dynamic memory allocation/deallocation patterns, namely: *Lindsay*, *Boxed-sim*, *Cfrac*, *GCBench*, *Espresso* and *Roboop*. *Lindsay* is a hypercube network communication simulator coded in C++ [3]. *Boxed-sim* is a graphics application that simulates spheres bouncing in a box [6]. *Cfrac* performs the factorization of an integer to two nearly equal factors [19]. *GCBench* is an artificial garbage collector benchmark that allocates and drops complete binary trees of various sizes [4]. *Espresso* is an optimization algorithm for PLAs that minimizes boolean functions [16]. *Roboop* is a C++ robotics ob-

ject oriented programming toolbox suitable for synthesis and simulation of robotic manipulator models in an environment that provides "MATLAB like" features for the treatment of matrices [3].

Table 1 shows several statistics illustrating their features as memory-intensive programs. These statistics are the number of managed objects, the total memory used (in bytes), the peak or maximum memory in use (in bytes), the average size of the required blocks (in bytes) and the total number of memory operations, which we have used to sort the benchmarks on the table.

Then, we have performed two different optimization experiments. First, we have run the optimization algorithm where the fitness is defined as a single aggregate objective function. We have named this algorithm Aggregated Sum GE (or ASGE). Second, we have run our proposed new MOGE approach on the benchmarks. Both optimization algorithms were run 10 times for each benchmark using the same configuration parameters, as depicted in Table 2.

Table 2: Parameters for both ASGE and MOGE optimization algorithms.

Parameter	Value
Population size	40
Number of generations	1000
Probability of crossover	0.80
Probability of mutation	0.02

In our experiments, apart from the designed custom DMMs using MOGE and ASGE, we have compared with *Kingsley*, which is a general-purpose memory allocator frequently implemented in Windows-based systems [17].

By definition, MOGE's best solution consists of a set of non-dominated individuals (cf. Section 3), whereas ASGE's best solution is defined by the individual with less fitness. Then, after running the optimizations, MOGE obtained a set of results that represent different non-dominated DMM implementations for each benchmark optimization, while ASGE obtained one DMM for each optimization. For the sake of clarity we chose to display the results for two of the three objectives: execution time and memory usage, normalizing the result to the worst value of each set of experiments. We selected these two objectives because in some cases the execution time and the energy consumption are proportional. On the other hand, the execution time and memory usage are always contradictory objectives.

Therefore, Figure 5 displays three different kinds of solutions for each benchmark. First of all, it shows the behavior of Kingsley. Second, labeled as ASGE, the figure shows the solution with the best fitness obtained after running the mono-objective optimizations 10 times. Finally, labeled as MOGE, the Figure shows the non-dominated solutions obtained after 10 runs of multi-objective optimization. Figure 5 does not display those solutions obtained by the MOGE optimization that, representing different DMMs, present equal objective values. These solutions will be later analyzed. In addition, the solutions obtained by MOGE in the 10 optimization runs were joined in order to present the non-dominated solutions of the entire set of results for each benchmark.

Our results indicate that Kingsley is the worst DMM in terms of memory usage (i.e., being the value to normalize to in all benchmarks). However, Kingsley obtains good execution times in all benchmarks but Cfrac. These results are consistent with the Kingsley implementation [17], which organizes the available memory in power-of-two block sizes and rounds up all allocation requests to the next power of two value. This rounding, in the worst case, allocates twice as much memory as requested, which is the reason of its high memory usage values. On the other hand, the power-of-two organization and the avoiding of splitting and coalescing leads to optimal execution times.

The ASGE optimization result is placed in very good position for almost all the benchmarks. In all cases the memory usage is lower than Kingsley, and the execution time is short, close to Kingsley. These values, as stated in the referenced literature, prove that a custom DMM obtains better results than a general-purpose allocator.

In addition, the MOGE optimization algorithm obtains solutions that improve al least one of the objectives of both Kingsley and ASGE results in all the benchmarks. As Figure 5 confirms, the MOGE optimization produces different non-dominated DMM alternatives for each binary. These alternatives range from the ones with high memory usage and low execution time, next to the Kingsley performance in all cases, to the ones that need less memory than the ASGE solution, but consume more execution time.

4.1 Analysis of found POF of solutions

Next, we analyze the results considering that the non-dominated solutions may belong to the Pareto-optimal Front (POF). We make this statement because, as explained in Section 3.1, this is a profiling-based exploration proposal whereas a pure analytical solution is not possible due to the complexity of the DMM design space of solutions [2, 3].

In Lindsay, 6 of the 10 MOGE solutions obtain shorter execution time than ASGE but need more memory, whereas 4 of the 10 MOGE obtain less memory usage than ASGE. Then, the ASGE solution will belong to the POF.

The ASGE solution for Boxed-sim also belongs to the POF because 5 of 6 MOGE solutions obtain shorter execution time but higher memory usage, while just one MOGE obtains less memory usage with longer execution time than ASGE.

In Cfrac we find the same pattern. Just 1 of 3 MOGE solutions obtains shorter execution time than ASGE, while 2 of 3 obtain less memory usage. Then, ASGE belongs to POF again.

In GCBench, 2 of 4 MOGE solutions obtain shorter execution time than ASGE and 1 of 4 MOGE obtains less memory usage than the ASGE solution. In this case one of the MOGE solutions obtains the same exact values than the ASGE. Therefore, the ASGE belongs to the POF.

In Espresso, 5 of 10 MOGE solutions obtain shorter execution time than ASGE, while 2 of 10 obtain less memory usage than ASGE. Therefore, the ASGE solution dominates 3 of 10 MOGE solutions, which leads to a POF formed by 7 MOGE solutions plus the ASGE one.

Finally, in Roboop, 5 of 10 MOGE solutions obtain shorter execution time while 4 of 10 obtain less memory usage than ASGE. In this case we found that ASGE dominates one of the MOGE solutions, which lead to a POF formed by 9 of the MOGE solutions plus the ASGE one.

Table 1: Statistics for the memory-intensive benchmarks considered in this paper

Memory-intensive benchmark statistics					
Benchmark	Objects	Total memory (bytes)	Max in use (bytes)	Average size (bytes)	Memory ops
Lindsay	102143	5795534	1497149	56.74	204153
Boxed-sim	229983	2576780	339072	11.20	453822
Cfrac	570014	2415228	6656	4.24	1140009
GCBench	843969	2003382000	32800952	2373.76	1687938
Espresso	4395491	2078856900	430752	472.95	8790549
Roboop	9268234	321334686	12802	34.67	18536420

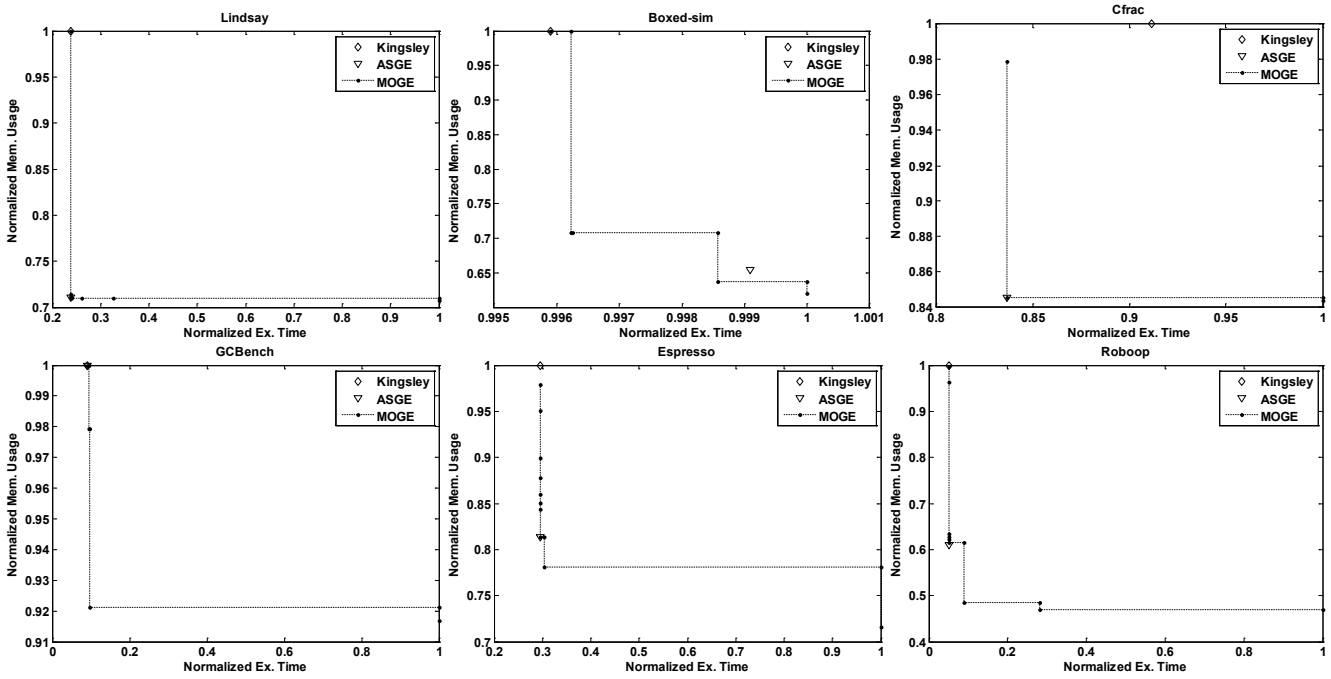

Figure 5: Results including Kingsley, ASGE optimized and MOGE optimized DMMs, normalized in relation to the worst value for both the execution time (x axis) and memory usage (y axis) objectives.

Therefore, the MOGE optimization obtains solutions than improve ASGE in one of the objectives for almost all the cases. In other words, MOGE provides the DMM designer a set of DMM implementations that are Pareto-optimal solutions. These DMMs could fit into different target platforms. For example, the MOGE solutions with less memory requirements (lower than ASGE in all the experiments) could be applied to a portable device, while the solutions with less execution time (shorter than ASGE in all the experiments) could be given to a server computer, for instance. As a consequence, the MOGE optimization provides a higher contribution to the DMM designer than the ASGE one.

4.2 Analysis of number of non-dominated solutions and hypervolume

Regarding the number of found solutions, we have measured how many of the non-dominated solutions are kept after the last iteration of both the ASGE and the MOGE optimization algorithms. We have to note that the MOGE algorithm will maintain two non-dominated individuals that have the same objective values if they correspond to two different DMM implementations. This choice was made because two DMM individuals with, for example, slightly different data structures, may obtain the same exact performance, but their different implementation could lead to different offspring individuals with different performance.

```
Solution #8 for MOGE run #0
=====================================================
BuddySystem, split:false, coalesce:false
SLL     EXACT(FIFO)     (0,1] B
SLL     EXACT(FIFO)     (1,2] B
SLL     EXACT(FIFO)     (2,4] B
SLL     EXACT(FIFO)     (4,8] B
SLL     EXACT(FIFO)     (8,16] B
=====================================================
BuddySystem, split:false, coalesce:false
DLL     EXACT(LIFO)     (16,21] B
DLL     EXACT(LIFO)     (21,34] B
DLL     EXACT(LIFO)     (34,55] B
=====================================================
SimpleSegregatedStorage, split:false, coalesce:false
SLL     BEST(FIFO)      (48,147456] B
=====================================================
```

Figure 6: DMM description of a non-dominated solution obtained by MOGE for Boxed-sim.

As an example of this kind of individuals, Figures 6 and 7

```
Solution #14 for MOGE run #0
=====================================================
BuddySystem, split:false, coalesce:false
SLL     EXACT(FIFO)     (0,1] B
SLL     EXACT(FIFO)     (1,2] B
SLL     EXACT(FIFO)     (2,4] B
SLL     EXACT(FIFO)     (4,8] B
SLL     EXACT(FIFO)     (8,16] B
=====================================================
BuddySystem, split:false, coalesce:false
DLL     FIRST(LIFO)     (16,21] B
DLL     FIRST(LIFO)     (21,34] B
DLL     FIRST(LIFO)     (34,55] B
=====================================================
SimpleSegregatedStorage, split:false, coalesce:false
BTREE   EXACT(FIFO)     (48,147456] B
=====================================================
```

Figure 7: DMM description of a non-dominated solution obtained by MOGE for Boxed-sim.

show the description of two non-dominated solutions, namely #8 and #14, obtained by the MOGE optimization for the Boxed-sim binary in the optimization run #0. Both DMMs obtained the same exact values of execution time, memory usage and energy consumption. As one may observe, these DMMs are formed by three allocators: two Buddy in charge of the smaller block sizes, and a Simple Segregated Storage allocator for the bigger block sizes. The differences between #8 and #14 DMMs are the policy used in the second allocator, which is FIFO exact fit in #8 and LIFO exact fit in #14; and the different data structure of the third allocator. DMM #8 uses a sigle-linked list with a FIFO best fit policy, while DMM #14 uses a B-tree with a FIFO exact fit policy.

Table 3 presents the average number non-dominated solutions obtained after the last iteration of the optimization runs for both ASGE and MOGE algorithms, as well as the interquartile range. As this table shows, the average number of non-dominated solutions of the MOGE optimization is higher than the value of ASGE in all benchmarks. Therefore, the ASGE algorithm explores the convex part of the Pareto front, while the MOGE extends the exploration obtaining more non-dominated solutions.

Table 3: Average and interquartile range ($\overline{X}_{IQR}$) of number of non-dominated solutions for ASGE and MOGE.

Benchmark	ASGE	MOGE
Lindsay	7.1_{2}	$18.3_{2.75}$
Boxed-sim	$11.4_{2.25}$	$21.7_{5.5}$
Cfrac	$7.9_{5.5}$	17.7_{1}
GCBench	$5_{1.5}$	10.9_{0}
Espresso	$4.3_{2.75}$	12.9_{1}
Roboop	$6.4_{1.75}$	$15.9_{2.75}$

On the other hand, it is important to compare both ASGE and MOGE optimizations using Pareto-compliant indicators for multi-objective problems [10]. Then, despite mono-objective optimizations produce just one solution (the one with the best fitness), we collected the non-dominated solutions after the last iteration of ASGE in all the benchmarks, as in the analysis of the number of solutions. Hence, we are able to compare both kind of optimizations using the hypervolume measure. This metric calculates the volume (in the objective space) covered by members of a non-dominated set of solutions [18]. However, in this case we have no reference points, which produces negative hypervolumes. Therefore, taking absolute values, the higher the hypervolume, the better the set of solutions.

Table 4: Average and interquartile range ($\overline{X}_{IQR}$) of hypervolume values for ASGE and MOGE.

Benchmark	ASGE	MOGE
Lindsay	$1.3187_{0.0003}$	$1.3190_{0.0014}$
Boxed-sim	$1.2830_{0.0175}$	$1.2853_{0.0335}$
Cfrac	$1.2604_{0.1050}$	1.3297_{0}
GCBench	1.2006_{0}	$1.2967_{0.0588}$
Espresso	$0.8989_{0.0332}$	$0.9382_{0.0004}$
Roboop	$0.9469_{0.0234}$	$1.0459_{0.1653}$

Thus, we have measured the hypervolume of each one of the sets of solutions obtained after each simulation run. Therefore, we have obtained 10 hypervolume values for the ASGE and 10 hypervolume values for the MOGE algorithms on each benchmark. Then, we have obtained the average and interquartile range values for each benchmark, as shown in Table 4. In all the benchmarks, the average hypervolume value of ASGE is improved by MOGE.

It should be noted that the difference between ASGE and MOGE hypervolumes in Lindsay and Boxed-sim is very short. The reason of this behavior comes from the fact that the less the number of memory operations, the less the number of different block sizes, and the less the efficient DMM configurations. In other words, if a benchmark performs a low number of operations, almost any DMM configuration will obtain good results. As a consequence, the ASGE optimization obtains solutions that are very similar to those obtained by MOGE. According to Table 1, Lindsay and Boxed-sim present the least number of memory operations.

On the contrary, if a benchmark performs a high number of operations over multiple block sizes, there will be multiple and diverse free blocks to be managed. Thus, specialized DMMs will probably perform well, and MOGE is more prone to generate different DMMs because the exploitation of the search space is better than in ASGE. This is the case of Cfrac and GCBench where the MOGE algorithm obtains, in both cases, a DMM implementation with the highest execution time but the least memory usage. This results in higher hypervolume values for the MOGE approach.

From the algorithmical point of view, the ASGE optimization minimizes the fitness, which is a single aggregate objective function. Then, this function explores the convex part of the Pareto front. As a result, the solutions use to converge, and the non-dominated DMMs obtained after the last generation are very similar, which does not result on those specialized DMMs that will improve the hypervolume.

On the other hand, the MOGE optimization keeps the non-dominated solutions. Therefore, a DMM where two different objectives present high values may persist over the generations if the third objective value is very low. This kind of non-dominated solutions may lead to specialized DMMs which allow the final set of solutions to obtain good hypervolume values. This trend increases when the number of

memory operations and block sizes grows. Table 4 show how Cfrac, GCBench, Espresso and Roboop, which present a higher number of operations, obtain a wider gap between the hypervolume values of ASGE and MOGE.

5. CONCLUSIONS AND FUTURE WORK

Dynamic memory is a key resource for any memory-intensive application because its management deeply impacts not only in the memory usage, but also in the execution time and the energy consumption. General purpose DMMs present acceptable behaviors, but it has been proven that custom DMMs obtain better performance by taking advantage of application-specific knowledge.

Previous works proposed automatic ways to obtain optimized custom DMMs considering three objectives: execution time, memory usage and energy consumption. However, this multi-objective problem was solved through a mono-objective approach where the fitness was built as a single aggregate objective function. This approach explores the convex part of the Pareto front, which leads to a reduced exploration of the search space and a lower exploitation of the individuals.

In this work we present, up to our knowledge, the first multi-objective optimization methodology able to obtain a set of non-dominated DMMs considering execution time, memory usage and energy consumption as objectives. This optimization technique was tested against the aggregate objective function optimization. The results show that our proposal provides Pareto-optimal alternatives to the solution coming from the aggregate objective function approach, better exploiting the search space. Therefore, our proposal provides the DMM designer more alternatives in order to select the implementation that could better fit on a concrete target platform.

In addition, we proved that our multi-objective implementation keeps more non-dominated solutions than the aggregate objective function approach after the last iteration of the algorithm. This is consistent with the idea of a wider exploration of the Pareto front.

Finally, we have measured the hypervolume value of the non-dominated solutions in both algorithms, and we found that our multi-objective approach obtains better results than the aggregate objective function one.

Our current work is focused in exploring other MOEA approaches in order to both improve the results and include new objectives like the memory temperature.

6. REFERENCES

[1] 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.

[2] 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.

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

[4] H. Boem. GCBench. http://www.hpl.hp.com/personal/Hans_Boehm/gc/gc_bench.html, 2010.

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

[6] T. M. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. *SIGPLAN Not.*, 36(5):191–202, 2001.

[7] J. M. Colmenar, J. L. Risco-Martin, D. Atienza, O. Garnica, J. I. Hidalgo, and J. Lanchares. Improving reliability of embedded systems through dynamic memory manager optimization using grammatical evolution. In *GECCO '10: Proceedings of the 12th Annual conference on Genetic and evolutionary computation*, pages 1227–1234, 2010.

[8] K. Deb. *Multiobjective Optimization using Evolutionary Algorithms*. John Wiley and Son Ltd., 2001.

[9] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[10] J. Knowles. A summary-attainment-surface plotting method for visualizing the performance of stochastic multiobjective optimizers. In *Proceedings of the 5th International Conference on Intelligent Systems Design and Applications*, pages 552–557, 2005.

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

[12] R. Poli, W. B. Langdon, and N. F. McPhee. *A field guide to genetic programming*. http://www.gp-field-guide.org.uk., 2008.

[13] J. L. Risco-Martín, D. Atienza, J. M. Colmenar, and O. Garnica. A parallel evolutionary algorithm to optimize dynamic memory managers in embedded systems. *Parallel Computing*, 36:572 – 590, 2010.

[14] 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: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1609–1616, 2009.

[15] J. L. Risco-Martin, J. M. Colmenar, D. Atienza, and J. I. Hidalgo. Simulation of high-performance memory allocators. In *13th Euromicro Conference on Digital System Design*, pages 275–282, 2010.

[16] SPEC. Standard Performance Evaluation Corporation. http://www.spec.org, 2010.

[17] 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.

[18] E. Zitzler. *Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications*. PhD thesis, Swiss Federal Institute of Technology (ETH), 1999.

[19] B. Zorn. The measured cost of conservative garbage collection. *Softw. Pract. Exper.*, 23(7):733–756, 1993.