Simulation of High-Performance Memory Allocators

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

Abstract—**Current general-purpose memory allocators do not provide sufficient speed or flexibility for modern high-performance applications. To optimize metrics like performance, memory usage and energy consumption, software engineers often write custom allocators from scratch, which is a difficult and error-prone process. In this paper, we present a flexible and efficient simulator to study *Dynamic Memory Managers (DMMs)*, a composition of one or more memory allocators. This novel approach allows programmers to simulate custom and general DMMs, which can be composed without incurring any additional runtime overhead or additional programming cost. We show that this infrastructure simplifies DMM construction, mainly because the target application does not need to be compiled every time a new DMM must be evaluated. Within a search procedure, the system designer can choose the "best" allocator by simulation for a particular target application. In our evaluation, we show that our scheme will deliver better performance, less memory usage and less energy consumption than single memory allocators.**

Keywords-dynamic memory manager; memory allocation; embedded systems design; evolutionary computation; grammatical evolution

I. Introduction And Related Work

Many general-purpose memory allocators implemented for C and C++ provide good runtime and low memory usage for a wide range of applications [1, 2]. However, using specialized DMMs that take advantage of application-specific behavior can dramatically improve application performance [3-5]. Three out of the twelve integer benchmarks included in SPEC (`parser`, `gcc`, and `vpr` [6]) and several server applications, use one or more custom DMMs [7].

Current applications are developed using C++, where dynamic memory is allocated via the operator `new()` and deallocated by the operator `delete()`. Most compilers simply map these operators directly to the `malloc()` and `free()` functions of the standard C library.

On the one hand, studies have shown that dynamic memory management can consume up to 38% of the execution time in C++ applications [8]. Thus, the performance of dynamic memory management can have a substantial effect on the overall performance of C++ applications. On the other hand, new embedded devices must rely on dynamic memory for a very significant part of their functionality due to the inherent unpredictability of the input data. These devices also integrate multiple services such as multimedia and wireless network communications. It heavily influences the global memory usage of the system [9]. Finally, energy consumption has become a real issue in overall system design (both embedded and general-purpose) due to circuit reliability and packaging costs [10]. Thus, the optimization of the dynamic memory subsystem has three goals that cannot be seen independently: performance, memory usage and energy consumption. There cannot exist a memory allocator that delivers the best performance and least memory usage for all programs; however a custom memory allocator that works best for a particular program can be developed [11].

To reach high performance, for instance, programmers write their own ad hoc custom memory allocators as macros or monolithic functions in order to avoid function call overhead. This approach, implemented to improve application performance, is enshrined in the best practices of skilled computer programmers [12]. Nonetheless, this kind of code is brittle and hard to maintain or reuse, and as the application evolves, it can be difficult to adapt the memory allocator as the application requirements vary. Moreover, writing these memory allocators is both error-prone and difficult. Indeed custom and efficient memory allocators are complicated pieces of software that require a substantial engineering effort.

Therefore, to design "optimal" memory allocators, flexible and efficient infrastructures for building custom and general-purpose memory allocators have been presented in the last decade [7, 9, 13]. All the proposed methodologies are based on high-level programming where C++ templates and object-oriented programming techniques are used. It allows the software engineer to compose several general-purpose or custom memory allocator mechanisms. Thus, we define a *Dynamic Memory Manager (DMM)* as a composition of one or more memory allocators. The aforementioned methodologies enable the implementation of custom DMMs from their basic parts (e.g., de/allocation strategies, order within pools, splitting, coalescing, etc.). In addition, [9] and [13] provide a way to evaluate the memory usage and energy consumption, but at system-level. However, all the mentioned approaches require the execution of the target application to evaluate every candidate custom DMM, which is a very time-consuming task, especially if the target application requires human input (like video games). Thus, new approaches to measure performance, memory usage and energy consumption are needed when designing a custom or general-purpose DMM.

In this paper we present a flexible, stable and highly-configurable DMM simulator. By profiling of the target application, the proposed DMM simulator can receive offline the dynamic behavior of the application and evaluate all the aforementioned metrics. As a result, the simulator can be integrated into a search mechanism in order to obtain optimum DMMs.

The rest of the paper is organized as follows. First, Section II describes the design space of memory allocators. Then, Section III details the design and implementation of the DMM simulator, as well as some configuration examples . Section IV shows our experimental methodology, presenting the three benchmarks selected, whereas Section V shows the results for these benchmarks. Finally, Section VI draws conclusions and future work.

II. Dynamic Memory Management

In this Section, we summarize the main characteristics of dynamic memory management, as well as a classification of memory allocators, which is later used in the implementation of the simulator.

A. *Dynamic Memory Management*

Dynamic memory management basically consists of two separate tasks, i.e., allocation and deallocation. 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. Deallocation is the mechanism that returns a freed memory block to the available memory of the system in order to be reused subsequently. In current applications, the blocks are requested and returned in any order. The amount of memory used by the memory allocator grows up when the memory storage space is used inefficiently, reducing the storage capacity. This phenomenon is called fragmentation. Internal fragmentation happens when requested objects are allocated in blocks whose size is bigger than the size of the object. External fragmentation occurs when no blocks are found for a given object request despite enough free memory is available. Hence, on top of memory de/allocation, the memory allocator has to take care of memory usage issues. To avoid these problems, some allocators include splitting (breaking large blocks into smaller ones to allocate a larger number of small objects) and coalescing (combining small blocks into bigger ones to allocate objects for which there are no available blocks of their size). However, these two algorithms usually reduce performance, as well as consume more energy. To support these mechanisms, additional data structures are built to keep track of the free and used blocks.

B. *Classification of memory allocators*

Memory allocators are typically categorized by the mechanisms that manages the lists of free blocks (free lists). These mechanisms include segregated free lists, simple segregated storage, segregated fit, exact segregated fit, strict segregated fit, and buddy systems [5].

A Segregated free-list allocator divides the free list into several subsets, according to the size of the free blocks. A freed or coalesced block is placed on the appropriate list. An allocation request is serviced from the appropriate list. This class of mechanism usually implements a good fit or best fit policy.

Simple segregated storage is segregated free-list allocation mechanism, which divides the storage into pages or other areas, and only allocates objects of a single size, or small range of sizes, within each area. This approach makes allocation 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. There is an array of free lists, each holding free blocks of a particular range of sizes. The manager identifies the appropriate free list and allocates from it (often using a first-fit policy). If this mechanism fails, a larger block is taken from another list and split accordingly.

Exact segregated fit is a segregated fit allocator, which has a separate free list for each possible block size. The array of free lists may be represented sparsely. Large blocks may be treated separately. The details of the mechanism depend on the division of sizes between free lists.

Strict segregated fit is a segregated fit allocation mechanism which has only one block size on each free 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 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 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 of splitting. They include binary buddies (the most common type), Fibonacci buddies, weighted buddies, and double buddies [5].

III. DMM Simulator Design And Implementation

In this section we motivate and describe the proposed approach as well as outline its design goals..

As introduced in Section I, there are currently several libraries to implement general-purpose and custom DMMs. However, exploration techniques cannot be easily applied. Indeed, each custom design must be implemented, compiled and validated against a target application; hence, even if the

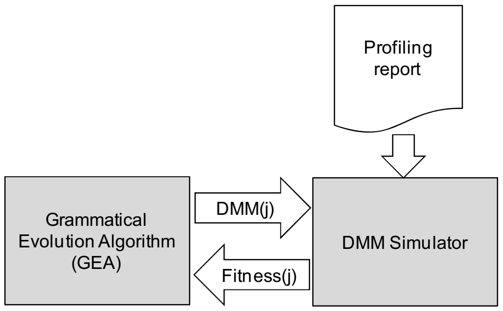


Figure 1. DMM generation and evaluation process.

DMM library is highly modular, this is a very time-consuming process. Thus, a simulator can greatly help in such optimization by being part of a higher optimization module that allows system designers to evaluate (in terms of performance, memory usage and energy consumption) the DMM for the target application.

Thus, the desired design goals for the development of this DMM exploration framework are:

- Efficiency: since the simulator needs to be included into search algorithms, the DMM simulator must improve the execution time of a real DMM.
- Flexibility: software engineers must be able to simulate any DMM as a composition of single memory allocators. Thus, the parameters of each allocator should be highly configurable.

Fig. 1 shows an illustrative example on how our proposed methodology operates. The input is a profiling report, which logs all the blocks that have been de/allocated. Our search algorithm is based on grammatical evolution (details about its implementation can be found in [11]). This algorithm is constantly generating different DMM implementation candidates. Hence, when a DMM is generated (DMM(j) in Fig. 1), it is received by the DMM simulator. Next, the DMM simulator 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. After the profiling report has been simulated, the simulator returns back the fitness of the current DMM to the search algorithm. After a given number of iterations, the search algorithm is able to find a custom DMM optimized for the target application in terms of performance, memory usage and energy consumption.

The global design of the simulator (available at [14]) architecture is shown in Fig. 2. First, the `sim` package constitutes the DMM simulation library. As Fig. 3 depicts, it includes: (1) a *Dynamic Memory Manager* class, which allows us to simulate standard allocation operations like `malloc()` or `free()`, (2) the *Simulator* class, which reads each line in the profiling report and calls the corresponding function in the DMM, and (3) the *Fitness* class, which evaluates the DMM in terms of performance, memory usage and energy consumption. The `gui` package in Fig. 2 includes *Graphical User Interfaces (GUIs)* to facilitate some standard simulations (based on general-purpose DMMs). The `util` package contains some extra required functionalities: reading of the profiling report, analysis of the current block sizes and frequency of use, and other relevant parameters. The `lib` package implements the simulation kernel, and it includes all the allocation mechanisms described in Section II-B. Fig. 4 depicts all the allocators implemented in the `allocator` package. We also have implemented the Kingsley memory allocator [5]. Fig. 5 shows the classes implemented in the `freelist` package. These classes include all the functionality reserved to the list of free blocks (allocation algorithms, policies and data structured selected) and the block, which maintains information about its size (including headers), time of creation and address in memory.

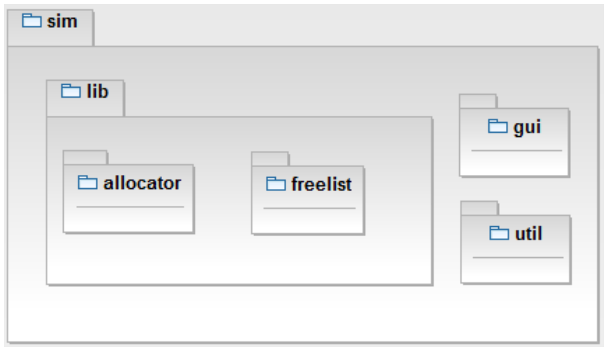


Figure 2. Package structure for DMM simulator.

In order to use the simulator, the software engineer must start with a profiling of the application. To this end, it is necessary to redefine/overload several functions (e.g. `malloc` and `free` in the case of C, and `new` and `delete` in case of C++). In the following, we illustrate how these functions can be overloaded in C:

```
extern FILE* log;
void* myMalloc(size_t size) {
        void* p = malloc(size);
        fprintf(log,"%f new %d %d\n", time(),p,size);
        return p;
}
void myFree(void* ptr) {
        fprintf(log,"%f delete %d\n", time(), ptr);
        free(ptr);
}
```

Next, after running the application, a profiling report is available and the system designer can test different DMMs using the same profiling report. Thus, the application must be executed just once during the whole study.

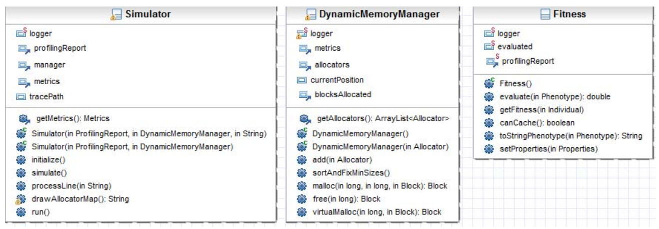

Simulator
logger
profilingReport
manager
metrics
tracePath
getMetrics(): Metrics
Simulator(in ProfilingReport, in DynamicMemoryManager, in String)
Simulator(in ProfilingReport, in DynamicMemoryManager)
initialize()
simulate()
processLine(in String)
drawAllocatorMap(): String
run()
DynamicMemoryManager
logger
metrics
allocators
currentPosition
blocksAllocated
getAllocators(): ArrayList<Allocator>
DynamicMemoryManager()
DynamicMemoryManager(in Allocator)
add(in Allocator)
sortAndFixMinSizes()
malloc(in long, in long, in Block): Block
free(in long): Block
virtualMalloc(in long, in Block): Block
Fitness
logger
evaluated
profilingReport
Fitness()
evaluate(in Phenotype): double
getFitness(in Individual)
canCache(): boolean
toStringPhenotype(in Phenotype): String
setProperties(in Properties)

Figure 3. Lib package class diagram

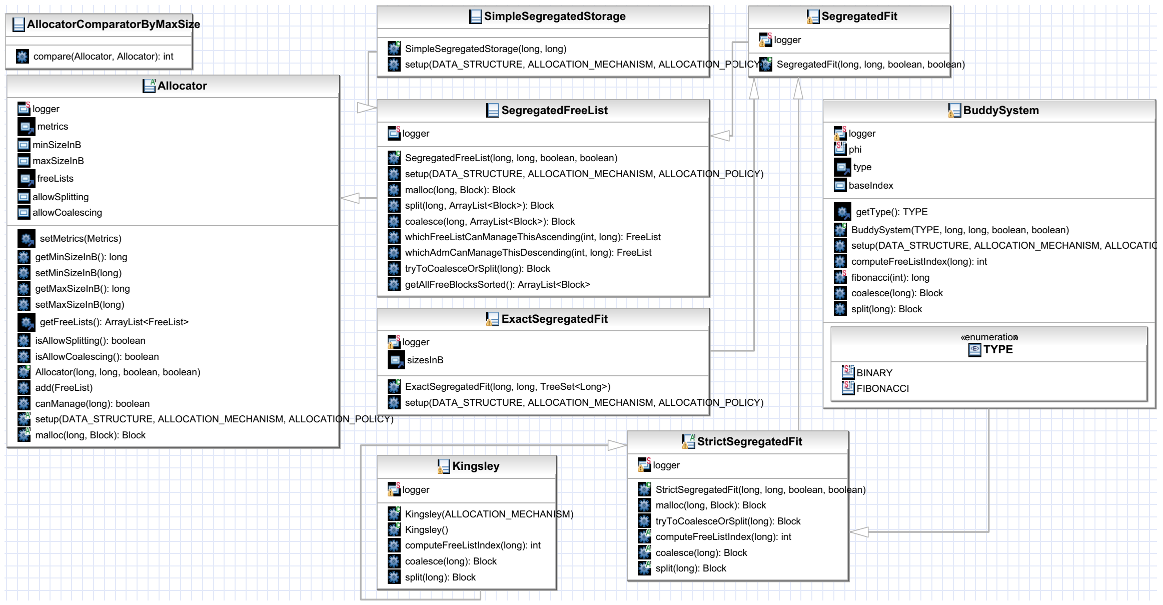

AllocatorComparatorByMaxSize
compare(Allocator, Allocator): int
Allocator
logger
metrics
minSizeInB
maxSizeInB
freeLists
allowSplitting
allowCoalescing
setMetrics(Metrics)
getMinSizeInB(): long
setMinSizeInB(long)
getMaxSizeInB(): long
setMaxSizeInB(long)
getFreeLists(): ArrayList<FreeList>
isAllowSplitting(): boolean
isAllowCoalescing(): boolean
Allocator(long, long, boolean, boolean)
add(FreeList)
canManage(long): boolean
setup(DATA_STRUCTURE, ALLOCATION_MECHANISM, ALLOCATION_POLICY)
malloc(long, Block): Block
SimpleSegregatedStorage
SimpleSegregatedStorage(long, long)
setup(DATA_STRUCTURE, ALLOCATION_MECHANISM, ALLOCATION_POLICY)
SegregatedFit
logger
SegregatedFit(long, long, boolean, boolean)
SegregatedFreeList
logger
SegregatedFreeList(long, long, boolean, boolean)
setup(DATA_STRUCTURE, ALLOCATION_MECHANISM, ALLOCATION_POLICY)
malloc(long, Block): Block
split(long, ArrayList<Block>): Block
coalesce(long, ArrayList<Block>): Block
whichFreeListCanManageThisAscending(int, long): FreeList
whichAdmCanManageThisDescending(int, long): FreeList
tryToCoalesceOrSplit(long): Block
getAllFreeBlocksSorted(): ArrayList<Block>
BuddySystem
logger
phi
type
baseIndex
getType(): TYPE
BuddySystem(TYPE, long, long, boolean, boolean)
setup(DATA_STRUCTURE, ALLOCATION_MECHANISM, ALLOCATIC
computeFreeListIndex(long): int
fibonacci(int): long
coalesce(long): Block
split(long): Block
«enumeration»
TYPE
BINARY
FIBONACCI
ExactSegregatedFit
logger
sizesInB
ExactSegregatedFit(long, long, TreeSet<Long>)
setup(DATA_STRUCTURE, ALLOCATION_MECHANISM, ALLOCATION_POLICY)
StrictSegregatedFit
logger
StrictSegregatedFit(long, long, boolean, boolean)
malloc(long, Block): Block
tryToCoalesceOrSplit(long): Block
computeFreeListIndex(long): int
coalesce(long): Block
split(long): Block
Kingsley
logger
Kingsley(ALLOCATION_MECHANISM)
Kingsley()
computeFreeListIndex(long): int
coalesce(long): Block
split(long): Block

Figure 4. Allocator package class diagram

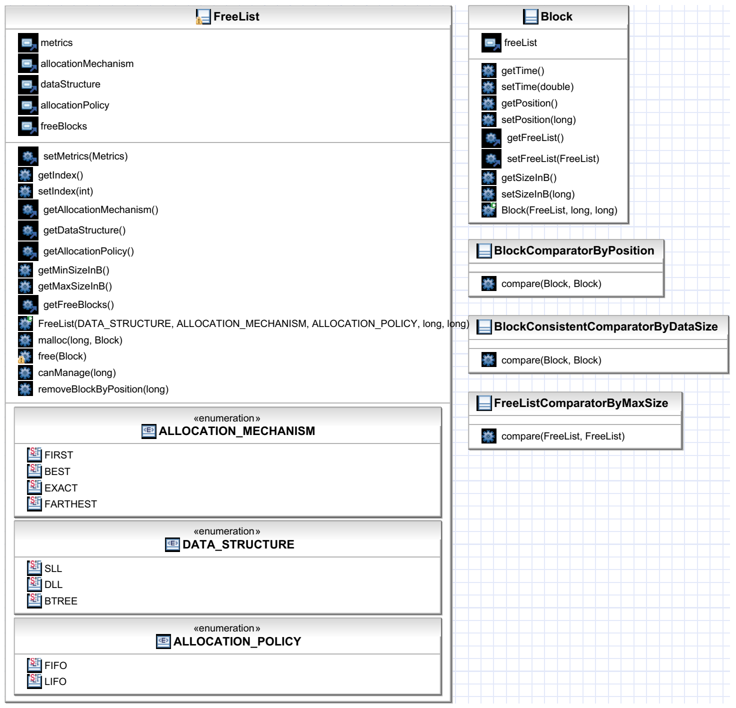


Figure 5. Freelist package class diagram.

The next step consists of: (a) performing an automatic exploration as described in Fig. 1, or (b) selecting the best design among a predefined set of DMM candidates.

As a consequence, the composition of a DMM candidate is not complex process for the designer. For instance, one common way of improving memory allocation performance is allocating all objects from a highly-used class from a per-class pool of memory. Because all these objects are of the same size, memory can be managed by a simple singly-linked free-list. Thus, programmers often implement these per-class allocators in C++ by overloading the `new` and `delete` operators for the class.

To evaluate the performance of the previous approach we can analyze, on the one hand, the profiling report to check how many different sizes appear in the target application. On the other hand, we can study the different classes used in the application as well as their sizes. Below we show how we can use our library to compose such complex DMM.

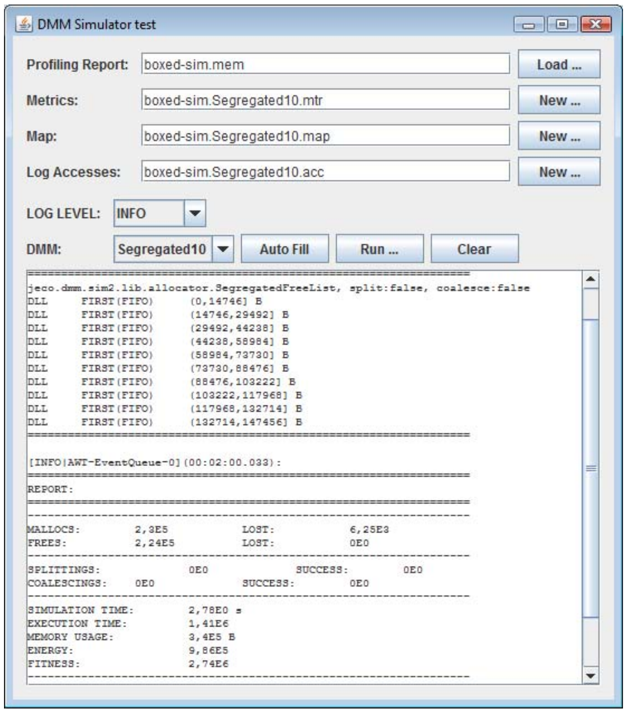
```
ProfilingReport profReport = new ProfilingReport();
profReport.load("profile.mem");
ExactSegregatedFit exact = new ExactSegregatedFit(
        0,
        profReport.getMaxSizeInB(),
        profReport.getSizesInB());
exact.setup(
        FreeList.DATA_STRUCTURE.SLL,
        FreeList.ALLOCATION_MECHANISM.FIRST,
        FreeList.ALLOCATION_POLICY.FIFO);
DynamicMemoryManager manager =
        new DynamicMemoryManager(exact);
```

In this example, when we build the *ExactSegregatedFit* allocator, we provide the constructor with the minimum block size in bytes, the maximum block size in bytes and the different sizes supported by this allocator. In the example above, the last two parameters are given by the profiling report, but they can be set manually. After that, we configure the allocator defining the data structure to be used (singly-linked list) the allocation mechanism (first-fit) and the allocation policy (first in, first out). Finally, we build the corresponding DMM. As defined before, a DMM may contain of one or more allocators in our case.

Finally, the simulator is invoked, and after a few seconds we obtain all the metrics needed to evaluate the current DMM:

```
Simulator simulator =
        new Simulator(profReport, manager);
simulator.start();
simulator.join();
```

At the same time that the simulation runs, several relevant metrics are computed, such us. number of de/allocations, splittings, coalescings, performance, memory usage, memory accesses, etc. All the previous parameters except the execution time can be calculated accurately. However, since the system is using simulation time instead of real time, the total execution time is calculated as the computational complexity or time complexity [15]. Finally, the energy is computed using the execution time, memory usage and memory accesses, following the model described in [11]. The following code snippet shows an illustrative example of how this task is performed in the proposed DMM optimization framework:



Profiling Report: boxed-sim.mem
Metrics: boxed-sim.Segregated10.mtr
Map: boxed-sim.Segregated10.map
Log Accesses: boxed-sim.Segregated10.acc
LOG LEVEL: INFO
DMM: Segregated10

```
==========================================================================
jeco.dmm.sim2.lib.allocator.SegregatedFreeList, split:false, coalesce:false
DLL      FIRST(FIFO)      (0,14746] B
DLL      FIRST(FIFO)      (14746,29492] B
DLL      FIRST(FIFO)      (29492,44238] B
DLL      FIRST(FIFO)      (44238,58984] B
DLL      FIRST(FIFO)      (58984,73730] B
DLL      FIRST(FIFO)      (73730,88476] B
DLL      FIRST(FIFO)      (88476,103222] B
DLL      FIRST(FIFO)      (103222,117968] B
DLL      FIRST(FIFO)      (117968,132714] B
DLL      FIRST(FIFO)      (132714,147456] B
==========================================================================

[INFO|AWT-EventQueue-0](00:02:00.033):
==========================================================================
REPORT:
==========================================================================
--------------------------------------------------------------------------
MALLOCS:        2,3E5          LOST:           6,25E3
FREES:          2,24E5         LOST:           0E0
--------------------------------------------------------------------------
SPLITTINGS:             0E0             SUCCESS:        0E0
COALESCINGS:    0E0            SUCCESS:        0E0
--------------------------------------------------------------------------
SIMULATION TIME:        2,78E0 s
EXECUTION TIME:         1,41E6
MEMORY USAGE:           3,4E5 B
ENERGY:                 9,86E5
FITNESS:                2,74E6
--------------------------------------------------------------------------
```

Figure 6. Simulation interface.

```
void firstFit(long sizeInB) {
// ...
  while(iterator.hasNext()) {
    counterForMetrics++;
    currentBlock = iterator.next();
    if(currentBlock.sizeInB>=sizeInB) {
      block = currentBlock;
      iterator.remove();
      break;
    }
  }

  metrics.addExecutionTime(counterForMetrics);
  metrics.addMemoryAccesses(2*counterForMetrics);
  // ...
}
```

The previous code excerpt shows a portion of the first-fit algorithm inside the simulator (*FreeList* class in Fig. 5). The main loop looks for the first block big enough to allocate the requested size. We count the number of iterations in the loop, and after that, both the execution time and memory accesses are updated accordingly (+1 for each cycle in the loop to compute the computational time and +2 for each cycle to count two accesses in the actual allocator: one to the current node in the free-list and another one to compute the size, i.e., subtraction of two pointers).

As Fig. 6 shows, to facilitate the use of the simulator, we have developed a GUI to test some general-purpose memory allocators, as well as to perform an automatic exploration of DMMs for the target application. Given a profiling report, the interface simulates the selected allocator, giving its “map” and some of the metrics computed. In any case, all the metrics are saved in external files.

IV. Experimental Methodology

In this section, we study the performance, memory usage and energy consumption implications of building general-purpose and custom allocators using the simulator. The custom DMM is obtained using the simulator and a grammatical evolution algorithm. Details about how the evolutionary algorithm is implemented can be found in [11].

TABLE I. STATISTICS FOR THE MEMORY-INTENSIVE BENCHMARKS USED IN THIS PAPER

Memory-Intensive Benchmark Statistics			
Statistics	***Boxed-sim***	***GCBench***	***Espresso***
Objects	229983	843969	4395491
Total memory (bytes)	2576780	2003382000	2078856900
Max in use (bytes)	339072	32800952	430752
Average size (bytes)	11.20	2373.76	472.95
Memory ops	453822	1687938	8790549

To evaluate each allocator runtime performance, memory usage and energy consumption, we use a number of memory-intensive programs, listed in Table I.

First, *Boxed-sim* is a graphics application that simulates spheres bouncing in a box [16]. The application represents a physical experiment where gravity is turned off, thus there is zero friction and no energy loss at each collision. Consequently, each sphere is given a random initial position, orientation and velocity, and zero initial angular velocity. Each run simulates a given amount of virtual time. Second, *GCBench* is an artificial garbage collector benchmark that allocates and drops complete binary trees of various sizes [17]. It maintains some permanent live data structures and reports time information for those operations. This benchmark appears to have been used by a number of vendors to aid in Java VM development. That probably makes it less desirable as a means to compare VMs. (It also has some know deficiencies, e.g. the allocation pattern is too regular, and it leaves too few "holes" between live objects.) It has been recently proposed as the most useful sanity test to be used by garbage collector developers. Finally, *Espresso* is an optimization algorithm for PLAs that minimizes boolean functions [6]. It takes as input a boolean function and produces a logically equivalent function, possibly with fewer terms. Both the input and output functions are represented as truth tables. The benchmark performs set operations such as union, intersect and difference. In particular, the sets are implemented as arrays of unsigned integers and set membership is indicated by a particular bit being on or off. These data structures are instantiated using dynamic memory.

To perform the profiling of the applications, we redefined the `malloc()` and `free()` functions (`new()` and `delete()` in the case of GCBench), as described in Section III. Then, all the applications were compiled with Visual Studio 2008 and run on an Intel Core 2 Quad processor Q8300 system with 4 GB of RAM, under Windows 7. This task must be performed just once within the whole DMM exploration process.

Table I includes the number of objects allocated and their average size. The applications' memory usages range from just 331.125 KB (for boxed-sim) to over 31.28 MB (for GCBench).

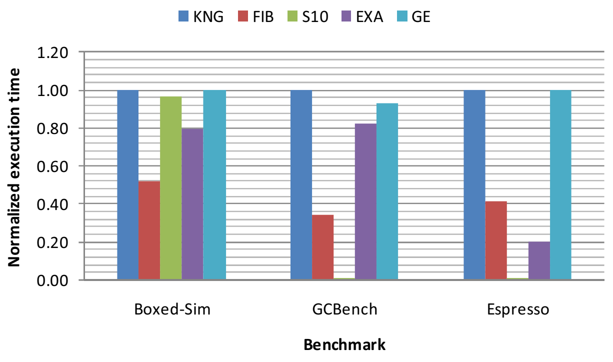


Figure 7. Execution time normalized to the Kingsley allocator (greater than 1 is better).

For all the programs, the ratio between total allocated the maximum amount of memory in use is large, and memory operations account for a significant portion of the runtime of all the considered applications. The next step consists of simulating the proposed general-purpose allocators, as well as performing an automatic exploration of feasible DMMs using a search algorithm (i.e., grammatical evolution in our case).

V. Results

We simulated the benchmarks in Table I with the Kingsley allocator (labeled as KNG in the following figures), a buddy system based on the Fibonacci allocation algorithm (labeled as FIB), a list of 10 segregated free-lists (S10), and an exact segregated free list allocator (EXA). Finally, we compare all the three benchmarks with the general-purpose allocators mentioned vs. the custom DMM obtained with our proposed automatic exploration process (GE).

In Fig. 7 we present a comparison of the execution time (computational complexity) of our benchmark applications normalized to the Kingsley allocator (i.e., greater than 1 is better). The best results in this case are obtained by Kingsley and the custom DMM. Only in the case of GCBench, the custom DMM is 0.98% worst than Kingsley, which is not relevant as Kingsley is an allocator highly optimized for performance, while tends to perform much worse than other managers regarding memory footprint and energy consumption [9]. Then, the Fibonacci-based buddy allocator, Segregated list and Exact fit, on the contrary, have different behaviors in function of the application, and these three allocators are worst than the Kingsley allocator and custom DMM in all conditions.

Next Fig. 8 shows the memory usage for the same benchmarks, normalized to the Kingsley allocator. We define memory usage as the high-water mark of memory requested from the virtual memory. In this case, we can observe that Kingsley is not the best allocator, as it suffers from a high level of internal fragmentation which in turn results in a larger memory utilization. As a notorious result, the exact segregated-fit allocator performs well in the case of Boxed-sim, but it presents the worst behavior in the case of espresso. It is because the last benchmark allocates a big amount of memory

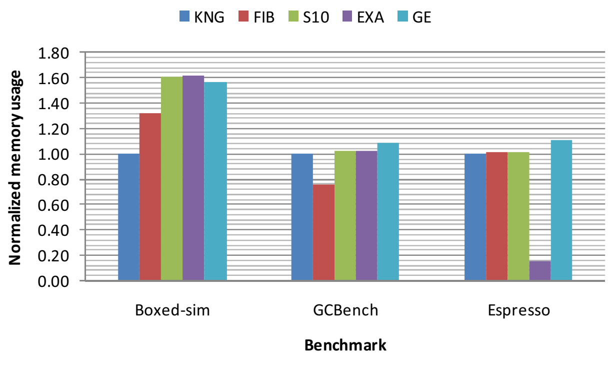


Figure 8. Space (memory usage) normalized to the Kingsley allocator (greater than 1 is better)

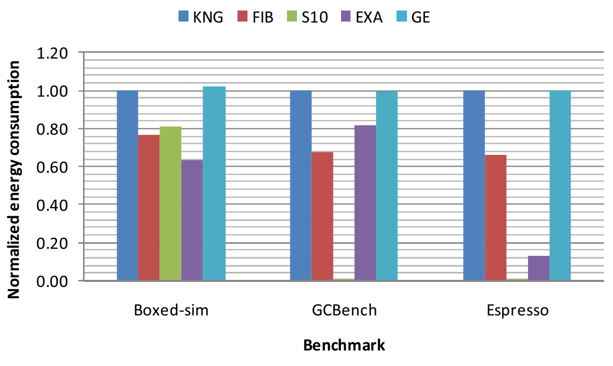


Figure 9. Energy consumption normalized to the Kingsley allocator (greater than 1 is better)

at the beginning of the application, and when the free blocks can be reused, they are too big for the new requests. It induces both internal and external fragmentation. Once more, the custom DMM is the best choice, 58% better than Kingsley in the case of Boxed-sim and 75% and 11% better than Kingsley in the case of GCBench and Espresso, respectively.

Finally, Fig. 9 shows the energy consumed by the four allocators. Since energy depends on execution time, the map of the energy consumption is quite equivalent to the execution time depicted in Fig. 7. However, energy consumed may vary greatly in function of the data structure used (singly-linked list, doubly-linked list, binary search tree, etc.), because it heavily influences in the number of memory accesses and thus, in energy. In this case, all the four allocators (included the custom DMM) use simple data structures (based on lists), so the energy is equivalent to the execution time. However, it is not a necessary condition, so the energy consumption is an independent metric and must be studied separately. In this case, both Kingsley and the custom DMM are good candidates for all the three benchmark problems. However, the custom DMM requires a lot less memory than the Kingsley allocator. As a result, the custom DMM is the best choice.

As a conclusion, we can state the exact fit allocation, a common practice among object-oriented applications, is not the best choice, especially for the Espresso benchmark. We can also observe the benefits of using our proposed DMM simulator in the memory exploration process, because it enables to compile and run the three applications just once to study the four different allocation mechanisms, which enables large savings in the dynamic memory optimization exploration.

Finally, we present the custom DMM obtained by our search algorithm for the Boxed-sim benchmark in Table II. The Boxed-sim benchmark includes 35 different block sizes, varying from 4 to 147456 bytes (144 KB). The automatically-obtained custom DMM presents 3 different internal allocators, all of them without splitting or coalescing. The first one is a binary buddy, where all the five free-lists are implemented as doubly-linked lists with exact allocation and LIFO policy. The allowed range by each free list is depicted in the last column of Table II. The second and third allocators follow a simple segregated storage mechanism and contain just one free-list. The first of them is implemented as a doubly-linked list with a first-fit allocation algorithm with FIFO policy. The second one, varying from 256 B to 144 KB is implemented as a singly-linked list, first-fit allocation algorithm and LIFO policy. According to the output of the simulator, the most requested block sizes are 4 and 16 bytes (111971 and 112069 times, respectively). Thus, that the best approach is managing the first region of block sizes using a binary buddy allocation, which prevents high internal fragmentation while being very fast in the allocation function.

TABLE II. CUSTOM DMM MAP FOR THE BOXED-SIM BENCHMARK.

Binary buddy (No splitting, No coalescing)			
DLL	EXACT	LIFO	(0..1]
DLL	EXACT	LIFO	(1..2]
DLL	EXACT	LIFO	(2..4]
DLL	EXACT	LIFO	(4..8]
DLL	EXACT	LIFO	(8..16]
Simple Segregated Storage (No splitting, No coalescing)			
DLL	FIRST	FIFO	(16..256]
Simple Segregated Storage (No splitting, No coalescing)			
SLL	FIRST	LIFO	(256..147456]

VI. CONCLUSION AND FUTURE WORK

Dynamic memory management continues to be a critical part of many recent applications in embedded systems for which performance, memory usage and energy consumption is crucial. Programmers, in an effort to avoid the overhead of general-purpose allocation algorithms, write their own custom allocation implementations trying to achieve (mainly) a better performance level. Because both general-purpose and custom allocators are monolithic designs, very little code reuse occurs between allocator implementations. In fact, memory allocator are hard to maintain and, as a certain application evolves, it becomes very complex to adapt the memory allocator to the changing needs of each application. In addition, writing custom memory allocators is both error-prone and difficult. Overall, efficient multi-objective (performance, memory footprint and energy efficient) memory allocators are complicated pieces of software that require a substantial engineering and maintaining effort.

In this paper, we have described a simulation framework in which custom and general-purpose allocators can be effectively constructed and evaluated. Our framework allows system designers to rapidly build and simulate high-performance allocators, both general and custom ones, while overcoming the tedious task of multiple profiling steps for each allocator instance within a target application. In particular, the proposed methodology avoids the compilation and execution of the target application in a case-by-case basis. Thus, the design of a custom DMM is quickly and free of (initial) implementation errors.

In addition to the proposed one-time DMM simulation approach, we have developed a complete search procedure to automatically find optimized custom DMMs for the application in study. Using this methodology, we have designed custom DMMs for three different benchmark applications, which important energy and memory footprint savings with respect to state-of-the-art memory allocation solutions.

Acknowledgment

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.

References

[1] M. S. Johnstone, and P. R. Wilson, “The memory fragmentation problem: solved?,” SIGPLAN Not., vol. 34, no. 3, pp. 26-36, 1999.

[2] "Doug Lea. A memory allocator.," http://g.oswego.edu/dl/html/malloc.html.

[3] D. A. Barrett, and B. G. Zorn, “Using lifetime predictors to improve memory allocation performance,” SIGPLAN Not., vol. 28, no. 6, pp. 187-196, 1993.

[4] D. Grunwald, and B. Zorn, “CustoMalloc: efficient synthesized memory allocators,” Softw. Pract. Exper., vol. 23, no. 8, pp. 851-869, 1993.

[5] P. R. Wilson, M. S. Johnstone, M. Neely et al., “Dynamic Storage Allocation: A Survey and Critical Review,” in Proceedings of the International Workshop on Memory Management, 1995.

[6] SPEC. "Standard Performance Evaluation Corporation," http://www.spec.org.

[7] E. D. Berger, B. G. Zorn, and K. S. McKinley, “Composing high-performance memory allocators,” in Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation, Snowbird, Utah, United States, 2001.

[8] B. Calder, D. Grunwald, and B. Zorn, “Quantifying behavioral differences between C and C++ programs,” Journal of Programming Languages, vol. 2, pp. 313–351, 1995.

[9] D. Atienza, J. M. Mendias, S. Mamagkakis et al., “Systematic dynamic memory management design methodology for reduced memory footprint,” ACM Trans. Des. Autom. Electron. Syst., vol. 11, no. 2, pp. 465-489, 2006.

[10] N. Vijaykrishnan, M. Kandemir, M. J. Irwin et al., “Evaluating Integrated Hardware-Software Optimizations Using a Unified Energy Estimation Framework,” IEEE Trans. Comput., vol. 52, no. 1, pp. 59-76, 2003.

[11] J. L. Risco-Martín, D. Atienza, R. Gonzalo et al., “Optimization of dynamic memory managers for embedded systems using grammatical evolution,” in Proceedings of the 11th Annual conference on Genetic and evolutionary computation, Montreal, Québec, Canada, 2009.

[12] S. Meyers, More Effective C++: 35 New Ways to Improve Your Programs and Designs: Addison-Wesley Longman Publishing Co., Inc., 1995.

[13] D. Atienza, S. Mamagkakis, F. Poletti et al., “Efficient system-level prototyping of power-aware dynamic memory managers for embedded systems,” Integr. VLSI J., vol. 39, no. 2, pp. 113-130, 2006.

[14] "JECO: Java Evolutionary Computation library," https://jeco.svn.sourceforge.net/svnroot/jeco.

[15] M. Sipser, Introduction to the Theory of Computation: International Thomson Publishing, 1996.

[16] T. M. Chilimbi, “Efficient representations and abstractions for quantifying and exploiting data reference locality,” SIGPLAN Not., vol. 36, no. 5, pp. 191-202, 2001.

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