Garbage Collector Refinement for New Dynamic Multimedia Applications on Embedded Systems

Jose M. Velasco⋆, David Atienza⋆, Francky Catthoor‡,
Francisco Tirado⋆, Katzalin Olcoz⋆, Jose M. Mendias⋆*
⋆DACYA/UCM, Avda. Complutense s/n, 28040 Madrid, Spain. Email: mvelascc@fis.ucm.es,
{datienza, ptirado, katzalin, mendias}@dacya.ucm.es
‡IMEC vzw, Kapeldreef 75, 3001 Heverlee, Belgium.
Email: {Francky.Catthoor}@imec.be

Abstract

New consumer embedded devices must execute concurrently multiple services (e.g. multimedia applications) that are dynamically triggered by the user. For these new embedded multimedia applications, the dynamic memory subsystem is currently one of the main sources of power consumption and its inattentive management can severely affect the performance and power consumption of the whole system. Therefore, the use of suitable automatic mechanisms to reuse the dynamic computer storage (i.e. garbage collector mechanisms) taking into account the underlying embedded devices would allow the designers to more efficiently design these systems. However, methodologies to explore and implement convenient garbage collector mechanisms for embedded devices have not been developed yet. In this paper we propose a new system-level method to define and explore the vast design space of possible garbage collector mechanisms, which enables to define custom garbage collector implementations for the final embedded devices.

1 Introduction

Currently, with the increasing importance of new embedded and portable applications, new design methods are required. The increasing need for efficient systems has motivated a large body of research on low power memory optimizations and performance improvement techniques for static data in embedded systems [14, 3]. However, the new application domains of these embedded systems include mobile terminals (e.g. for multimedia applications), automotive, game processors, etc. [13]. These domains currently include two sources of dynamism that have to be taken into account when they are designed.

First, new dynamic applications (e.g. MPEG21) are internally dynamic, which means that they heavily depend on Dynamically allocated Memory (DM from now on) due to the inherent unpredictability of the input data and their respective behaviour (e.g. the data and calculations of a motion estimation vector vary greatly depending on how the user moves the camera). Designing the final embedded systems for the (static) worst case memory footprint of these new applications would lead to a too high overhead in memory footprint and power consumption for them. Even if average values of possible memory footprint estimations are used, these static solutions will result in higher memory footprint figures (i.e. 22% more) than DM solutions [10]. Moreover, these intermediate static solutions will not work in extreme cases of input data, whereas solutions using dynamic allocation can do it. Thus, DM management mechanisms must be used in embedded realisations of these designs.

Second, new portable consumer devices include another degree of dynamism, what we can call external dynamism, because the code (and eventually the number of different applications) to be executed on these platforms is user dependent. The user behaviour determines the number of applications and which ones to run. Clearly, these two key factors make it very hard to appropriately handle the use of DM in new consumer embedded devices. Therefore, efficient automatic DM reclamation mechanisms to decide the DM still in use at run-time (i.e. garbage collectors) are in great need. However, current embedded devices are not prepared to support general-purpose Garbage Collector (GC from now on) mechanisms due to their limited resources (e.g. power, memories, etc.). Thus, new methods to design custom GCs for embedded systems must be developed.

In this paper we propose a new method that allows the design of custom GCs for new dynamic applications (e.g.

*This work is partially supported by the Spanish Government Research Grant TIC2002/0750.

multimedia) taking into account the constraints (e.g. performance, power consumption, etc.) of the final embedded devices where these applications must be executed. The remainder of this paper is organized in the following way. In Section 2 we describe some related work. In Section 3 we present the proposed method to define the design space of GC mechanisms and we outline a main order to traverse it. In Section 4, we show how state-of-the-art and classical GCs can be characterized within our GC design space. In Section 5, we briefly introduce our benchmark applications and present the experimental results obtained. Finally, in Section 6 we draw our conclusions.

2 Related Work

Nowadays a very wide variety of well-known techniques for uniprocessor GCs (e.g. reference counting, mark-sweep collection, copying garbage collector) are available in a general-purpose context within the software community [18]. Also, recent research on GC policies show important gains for performance in general-purpose systems using generational GCs [9]. Moreover, further improvements in performance are achieved by real implementations of GCs including application-specific behaviour in their designs [17, 18].

Regarding methods to refine the structure of GCs for specific scenarios, usually designers use simulation and lifetime predictors to analyze an extensive amount of traces of objects allocations and their run-time behaviours [9]. However, this is prohibitively expensive in time both for data acquisition and profiling analysis. Thus, new methods include algorithms to generate typical and representative traces of the application under analysis, which allows to prune the exploration design space [18]. In both cases, the exploration is limited by the number of GC candidates available in the library used and the time to implement them and profile them at run-time. Moreover, these GCs have to be defined by the designers based on their own experience since the design space of possible decisions for GC mechanisms is not defined.

Finally, in memory management for highly-constrained embedded systems, the DM is usually partitioned into fixed blocks to store the dynamic data and these free blocks are placed in a single linked list [11]. Also, in recent real-time operating system synthesis approach for embedded systems, dynamic allocation is supported with custom DM managers based on region allocators [19] for the specific platform features. However, all these systems do not include GCs and only rely on manual DM management.

Finally, a large body of research on memory optimizations and techniques exists for static data in embedded systems (see e.g. [14, 3] for good tutorial overviews). All these techniques are complementary to our work and are applicable in the part of the code that accesses static data in the dynamic applications under study. Furthermore, they are useful as back-end for our approach, once the amount of DM used by the system is limited and the data are allocated into memory pools that can be statically declared and optimized.

3 Garbage Collectors Design Space

3.1 Design Space of Orthogonal Decisions for Garbage Collectors

In manual memory management, the programmer has full control of the DM subsystem of the system. Thus, he must explicitly deallocate the objects that are no longer used. However, in complex dynamic applications (e.g. multimedia), this full control is very difficult to be properly applied. As a result, very frequently, memory problems must be debugged in the design process due to inappropriate free function calls because the object should still be available or the DM is not freed at the right moment but later. Within this context, the GC replaces the programmer in this task. To do so, the GC must perform two phases. First, during the tracing phase it has to be able to detect the dead objects in the application running presently at a certain moment of the execution time. Secondly, during the reclaiming phase the GC has to recycle these dead objects and free the DM for future program requests. Usually, in the literature we can observe that to accomplish the previous two phases, GCs can be categorized regarding the way they work:

- Stop-the-world GC. The running application is paused during the two phases of the GC to avoid inconsistencies in the references to DM in the system. This pause is the main obstacle for applications using GCs to be executed with real-time requirements.
- Incremental GC. We can diminish the pause by dividing the tracing phase, the reclaiming phase or both in smaller subphases (named ”increments”) that check subparts of the heap. In this case, the problem appears on the complexity of the GC design to define the appropriate moments to stop temporarily the application. The fact is that since the GC will not have time to finish its garbage collection fully in one increment and it is unknown which parts of the memory are going to be modified next by the application, severe inconsistencies may appear in the DM of the system.
- Concurrent versus Parallel. The meaning of these terms is rather ambiguously defined across the literature. Here we use the term concurrent when a GC thread is running concurrently with the rest of the program threads. In addition, we reserve the term parallel when two or more GC threads are running concurrently [5]. Hence, in a concurrent GC, the application

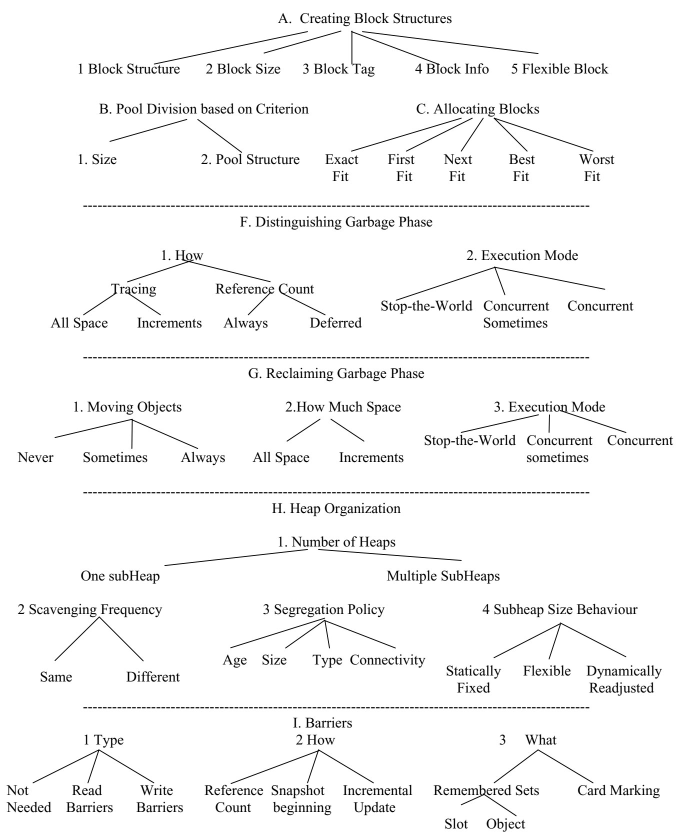

Figure 1. GC design space of orthogonal decisions

running presently is not explicitly paused, but the overhead of the GC thread can hamper meeting the necessary real-time requirements.

As we have just explained, an extensive amount of possible GC strategies (and implementations for them) exists. Therefore, all these options have to be enumerated to cover exhaustively the GC design space. In our method, we have classified all the relevant decisions that can compose the design space of GC mechanisms in different orthogonal decision trees (see Figure 1). Orthogonal means here 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). In addition, all possible solutions in the design space should be spanned by a combination of leaves in the orthogonal trees, just like any point in a geometrical space can be represented in a set of orthogonal axes. Moreover, the decisions in the different orthogonal trees can be ordered in such a way that traversing the trees can be done without decision 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 GC is defined (in our notation, atomic GC mechanism) for a specific DM behaviour pattern. Note that in Figure 1, categories A, B and C are not only related to GCs, but also relevant in the design space of DM managers [2]. In the following we focus on the four main categories for GCs mechanisms of Figure 1 (i.e. categories F, G, H and I) and the important decision trees inside them for the creation of custom GCs. In the original description [2] this has not been described at all. Therefore, it forms the main contribution of the paper:

F. Distinguishing Garbage Phase relies on determining which objects are not pointed to by any living program object. Two strategies exist within this context, as tree F1 in Figure 1 shows, which give name to different families of GCs, i.e. Tracing garbage collectors and Reference-Counting collectors:

- A *Tracing collector* traverses the graph of references of the program to find the objects that can be reached at each moment of the execution. Then, the unreachable objects at a certain moment are garbage. In order to avoid the overhead, as we have mentioned before, instead of tracing the graph across the whole heap, we can focus in smaller parts (i.e. increments).
- A *Reference-Counting collector* acts in a way much closer to explicit deallocation. Each object has a header field with a reference counter. Whenever a pointer is assigned/unassigned, the counter is incremented/decremented. If the count becomes zero, we can recycle the object. A Deferred reference count (tree F1)is a mixed technique with tracing collection. The cost of reference counting can be very high depending on the work done by the application running, so a hardware-assisted solution [20] can benefit this policy. This is perfectly possible in our design space, the only variation then would be to include two different cost functions (i.e. cost of the hardware solution and cost of the software solution) for that leave of the GC design space when the custom GC solution is determined. Finally, our last tree in this category (F2) is used to decide the way the GC runs in the system (e.g. concurrently, alone). It allows to avoid long time pauses in certain DM behaviours of the applications by creating concurrent GCs (or mostly concurrent with the application).

G. Reclaiming Garbage Phase. Once we have differentiated the dead objects from the alive ones in the Distinguishing phase, three options exist in the reclaiming phase (represented by tree G1 in Figure 1) . First, the use of the DM used by the "garbage objects" as available memory for subsequent memory requests of the application, thus the memory can be reused. Second, moving and compacting living objects, reducing this way the total DM fragmentation [19]. If the GC never moves data, we have the classical Mark& Sweep GC [18]. In case the GC always moves objects, we have the classical GC Copying policy [18]. Finally, a trade-off between both is the Mark&Sweep with a compacting phase when necessary [9], which is the leaf SOMETIMES in tree G1. In this case, this general option must be particularized with several thresholds to determine when compacting the DM. Then, tree G2 defines how these previous mechanisms for the reclaiming phase are applied to the heap, i.e. in the whole space at once or in increments. Finally, as in category F, we include in our GC design space the options to decide the way the GC runs in the system (e.g. serialized, concurrently, etc.).

H. Heap Organization. To improve global performance and to avoid long pauses, GC designers have devised different new strategies that basically rely on dividing the heap in regions or subHeaps [9] (tree H1, Figure 1). The main policies for segregating the data into these regions are as Figure 1, tree H2 depicts: the size, the age, the type or the connectivity of the objects (minimization of traversing between regions). Then, the GC can manage all the subheaps with the same policy or assign different strategies to each of them. This way and by combination, the powerful option to use separate hybrid GCs arises, as our GC design space allows (see Section 5 for real examples of these hybrid GCs). Moreover, the GC can apply different scavenging frequencies to each region (tree H3 in Figure 1), which is the main point in the Generational collection strategy [9]. Typically for this case, the borders of each subheap are statically fixed and no variations in the size of each subheap are allowed at run-time (tree H4 in Figure 1). Alternatively, in our GC design space (see Figure 1) it is possible to have a more flexible policy, which assigns space as it is available, like in the Appel Collector [1]. Furthermore, using our GC design space, it is possible to use an adaptive strategy that dynamically readjusts the regions size. In this case, the GC must take this kind of decisions based on dynamic feedback gathered at run time (see Section 5 for its application in GCs for real examples). The required maintenance data structures to gather this information need to be supported by the general DM manager of the system [2], thus we include in our GC design space (Figure 1) categories A and B.

I. Barriers. We need barriers in two different scenarios. As we mentioned earlier, some implementations of the GC can scan only part of the heap (i.e. increments) and not the whole heap. In this case, every time the GC searches for live objects within a "subheap" it needs to know the references that point into the present subheap and have their sources in the rest of the heap. Furthermore, if the GC runs concurrently with the application, it has to take into account that the relationship graph of data structures can change while it is traversed. Therefore, the GC must know when the application is referencing data that has not traced yet. To achieve safe concurrent cooperation between the application and the GC, the Tri-Color marking algorithm [18] was introduced in GCs. This algorithm deals with the distinguishing garbage phase in Tracing GC [18]. It requires both read and write barriers, thus these two different uses of barriers are included in our GC design space, tree I1, Figure 1. Furthermore, our GC design space allows to use these barriers to create an incremental GC copying algorithm [18], which has been proposed as an alternative GC useful for certain DM behaviours [6]. Thus, we include in trees I2 and I3 all types and ways to implement such write/read barriers (e.g. snapshot at the beginning, card marking, etc.). However, the cost of the barriers is very high and they should be used only in particular DM behaviours of the applications under study (see Section 5 for more details). Therefore, they must be combined with other GC mechanisms to improve the overall results (e.g. performance) of these GCs, as we show in Section 5 for our case studies.

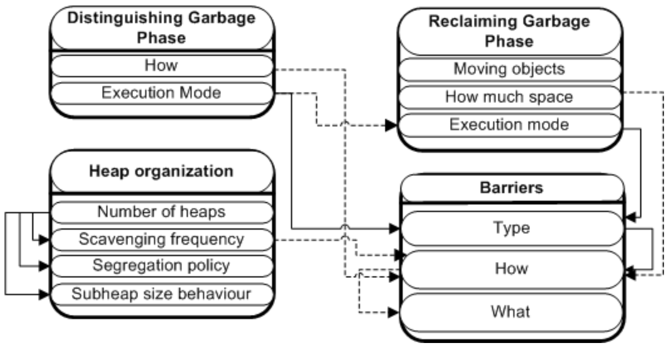

Figure 2. Interdependencies between orthogonal trees in the GC design space

3.2 Interdependencies within the Garbage Collectors Design Space

Although the decision categories and trees presented in Subsection 3.1 are orthogonal, certain leaves in some trees strongly affect the coherent decisions in other trees. Thus, they include interdependencies to take into account when a GC is designed. The whole set of interdependencies for our design space is shown in Figure 2. They can be classified in two main groups. The interdependencies caused by leaves that can disable the use of other trees or categories and those affecting other trees or categories due to their linked purposes (full and dotted arrows respectively, in Figure 2). An example of these interdependencies is shown in Figure 3, where we can see that the use of the Reference count strategy (tree F1) requires to use of Write barriers (tree I1).

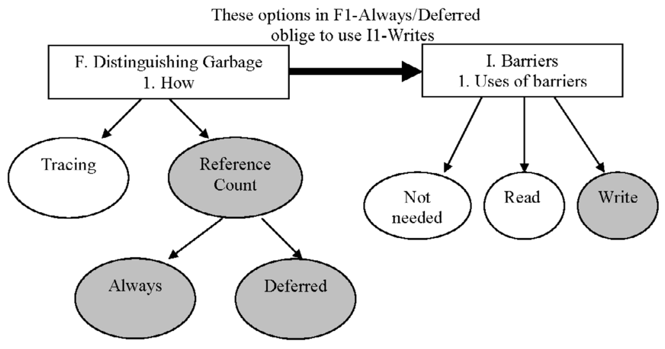

Figure 3. Example of interdependencies within the GC design space

4 Garbage Collectors Characterization and Evaluation

In this section, we show how our design space can be used to characterize the most representative state-of-the-art and classical GCs found in the literature. First, we show which leaves must be selected in our GC orthogonal design trees of Figure 1 also respecting the interdependencies of Figure 2 to define two well-known classical policies, namely Reference Counting and non-concurrent Mark&Sweep [18]:

- *Reference Counting*: F1-Reference Count-Always; F2-Concurrently always; G1-Never; G2-All space; G3-Concurrently always; H1-One subheap; I1-Read barriers; I2-Reference count.
- *Mark&Sweep*: F1-Tracing-All space; F2-Stop-the-world; G1-Never; G2-All space; G3-stop-the-world; H1-One subheap; I1-Not needed.

Also, more complex GC can be univocally defined using our GC design space, which allows to more easily cover and explore all the possible GC candidates. For example, the Copying policy [18] has a traditional improvement that can be easily characterized within our GC design space: since the cost of moving large objects is very high, we can reserve a special subheap for these objects with a non-moving strategy. This subheap is called the Large Object Space (LOS) in the literature [18]. In this case, using our GC design space both options of the Copying policy can be easily defined in the following way:

- *Copying with LOS (Mark&Sweep) Space*: F1-Tracing-All space; F2-Stop-the-world; H1-One subheap; H2-Same; H3-size; H4-Statically fixed; I1-Not needed
- *Copying Space*: G1-Always; G2-All space; G3-Stop-the-world. And then, the LOS space: G1-Never; G2-All space; G3-Stop-the-world.

In the Copying policy [18], the immortal or the long life data are copied repeatedly during the execution. This produces a useless overhead. To avoid this problem, the Generational strategy divides the Heap in areas according to data age [9]. Therefore, when an object is created, it is assigned to the youngest generation, the nursery space. As objects survive different executions of the GC, they get a more "mature" state and are copied into older generations (different areas). The frequency with which the GC runs is smaller in older generations. In the following we show how a GC with two generations spaces (i.e. nursery space for young objects and mature space for older ones) can be easily defined with our GC design space. It uses copying policy in the nursery and Mark&Sweep in the mature. The subheaps are statically bounded. We can find a real example with such GC structure in the IBM Jikes RVM [7]:

- *Generational Collection (genMS)*: F1-Tracing-increments; F2-stop-the-world; H1-Multiple subheaps; H2-Different; H3-Age; H4-Statically fixed; I1-Write barriers; I2-Incremental Update; I3-Remembered sets-Slot. Then, for the Nursery Space: G1-Always; G2-All space; G3-Stop-the-world. Finally, the Mature space: G1-Never; G2-All space; G3-Stop-the-world.

In addition, Appel [1] has proposed a variation of this scheme. His GC assigns all free space to the nursery space. When the nursery space is full, it copies surviving objects to the mature space, and then reduces the nursery size by that volume. The GC repeats this process until the nursery space falls down a certain threshold, then it performs a full heap collection. This GC can be expressed within our GC design space as follows:

- *Generational Collection (genMS,Appel)*: F1-Tracing-Increments; F2-stop-the-world; H1-Multiple subheaps; H2-Different; H3-Age; H4-Flexible; I1-Write barriers; I2-Incremental Update; I3-Remembered sets-slot. Then, the Nursery space is defined in the following way: G1-Always; G2-All space; G3-Stop-the-world. Finally, the Mature space: G1-Never; G2-All space; G3-Stop-the-world.

A new and interesting generational GC is the one proposed by S. Blackburn and McKinley [4]. It has two generations for the objects (i.e. nursery space and mature space), copying policy in the nursery space and Reference count in the mature space. This GC can be defined in our GC design space in the following way:

- *Ulterior Reference Counting*: H1-Multiple subheaps; H2-Different; H3-Age; H4- Statically fixed. Then, the Nursery space is defined like this: F1-Tracing-All space; F2-stop-the-world; G1-always; G2-All space;

G3-stop-the-world; I1-Write barriers; I2-Incremental Update; I3-Remembered sets-slot. Finally, the Mature space is defined as follows: F1-Refence Count-deferred; F2-stop-the-world; G1-Never; G2-All space; G3-Stop-the-world; I1-Write barriers; I2-Reference count.

Stefanovic has developed an incremental variation of the Copying GC strategy [17]. In this case the heap is divided in increments (called "windows" in [17]). Then, the GC collects the window with the oldest allocated objects. This strategy can be an alternative to Generational collection [9] or they can be used together as we can see in Beltway [4]. Here, we describe the simplest non-generational Older-first strategy [17]:

- *Older-first*: F1-Tracing-increments; F2-Stop-the-world; G1-Always; G2-Increments; G3-Stop-the-world; H1-Multiple subheaps; H2-Same; H3-age; H4-Statically fixed; I1-Write barriers; I2-Incremental update; I3-Remembered sets-Slot.

Finally, we show how very complex GCs with several strategies can be also defined with our GC design space (we show the practical use and benefits of these custom GCs in Section 5 with our case studies). In this case we define a GC that uses a fine-grained increment Mark&Sweep scheme, intended for Real-Time systems [5]. If memory fragmentation increases, this GC compacts the pools with high fragmentation into new pools. The incremental tracing and reclaiming phase is achieved thanks to a read barrier and a `Snapshot beginning` write barrier:

- *Real-Time Garbage Collector (Mark&Sweep)*: F1-Tracing-increments; F2-stop-the-world; G1-Sometimes; G2-Increments; G3-Stop-the-world; H1-One subheap; I1-Read and write barriers; I2-Snapshot beginning.

5 Case Studies and Experimental Results

We have applied the proposed GC design space analysis and exploration method to the most representive benchmarks regarding DM behaviour and dynamic data allocation included in the suite SPECjvm98 [16]. They are the following:

- _205_Raytrace: it is an engine to raytrace a scene into a memory buffer that is then displayed on the screen.

- _202_Jess: it is the Java version of NASA CLIPS [12], the tool to construct rule and object based expert systems. It is an application that shows a very significant use of DM and a very variable DM behaviour, like new dynamic multimedia applications.

- _228_Jack: it is a parser based on the Purdue Compiler Construction Tool Set [15]. A parser determines the syntactic structure of a chain of symbols received from the exit of the lexical analyzer. As the previous benchmark (i.e. Jess) , Jack shows a very variable DM behaviour that is interesting to study since it resemblances the typical DM behaviour of multimedia applications.

The results shown in this section are average values after 10 simulations for each GC, where all the final values were very similar (variations of less than 7%). These results have been obtained after modifying significantly the code of Jikes RVM (Research Virtual Machine) from the Watson Research Center of IBM [7]. Jikes RVM is a Java virtual machine designed for research. It is written in Java and the components of the virtual machine are Java objects [8], which are designed as a modular system that enables the possibility of modifying extensively the source code to implement different GC strategies and custom GCs. We have used version 2.2.0 along with the recently developed memory manager JMTk (Java Memory management Toolkit) [7]. The simulations were performed on a Pentium III processor at 866 Mhz with 1024 MBytes SDRAM and running GNU/Linux 2.4.

In all the following experiments we have composed custom GCs for the previously explained benchmarks using our GC design space and taking into account the DM behaviour of each benchmark. As we show, the most suitable GC strategies for each benchmark vary enormously according to the system requirements (e.g. performance) and the available resources for the heap (e.g. memory footprint). We have experimented with a range of possible heap sizes varying from the minimum needed (i.e. no fragmentation in the GCs) to 7 times this value. Then, we have built custom GCs using our GC design space to refine state-of-the-art general-purpose GCs [9, 18] according to our benchmarks specific dynamic behaviours. As a result, the lifetime data behaviour of the benchmarks used in our experiments have led us to define custom GCs with three different strategies and heap organizations (so the global custom GC is the inclusion of these three atomic GC strategies in one if the DM available in the system varies at run time in the range explored for the heap). Our final custom GCs for the benchmarks studied have major changes between these three zones and minor refinements inside each zone for each benchmark (according to the final suitable implementation for each benchmark, e.g. thresholds, etc.). These regions are depicted in Figure 4 and are the following:

1. *A-Zone*: This zone starts with the minimum heap size needed for executing each benchmark until 2.5 or 3.5 times this value, depending on the specific benchmark. In this case we have observed that the best results

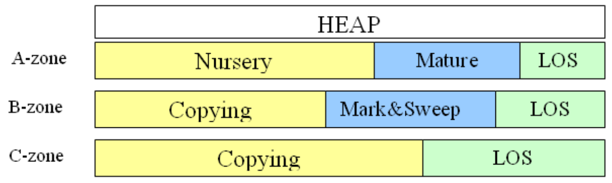

Figure 4. Aspect of the heap used with the different zones distinguished by our custom GCs

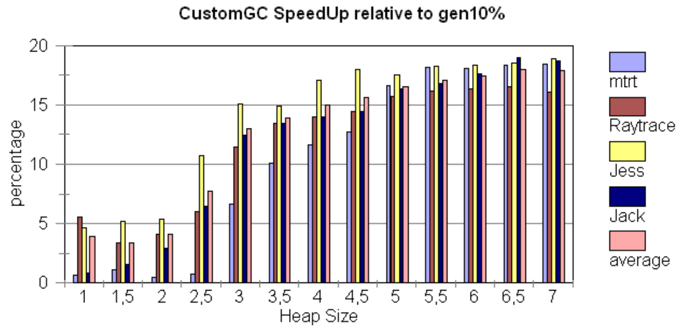

Figure 5. Speedups in total execution time of our custom GC compared to a state-of-the-art Generational GC

are obtained using a Generational Copying collector (with two generations: nursery and mature space) and a bounded Large Object Space (i.e. LOS) [18]. Then, the nursery space is managed with a semispace Copying policy [18] and both mature and LOS with a Mark&Sweep strategy [18]. The border between the nursery and the mature space is not flexible, but we adjust it for each benchmark and the total heap size as well. The LOS size remains constant and the same occurs with the object size threshold that indicates when an object has to be considered a Large Object and thus allocated in the LOS space.

2. *B-Zone*: This zone starts from the A-zone's maximum size (2.5 or 3.5) to 4.5 or 5 times the minimum heap size needed to run the appplication. In this zone we have observed that the best results are obtained using a Non-Generational GC [18] and defining 3 regions as in the A-zone. First, a LOS region with Mark&Sweep policy [18]. Then, another region with Copying policy [18]. Finally, a region that is filled with the survivors from the first region and that is managed also in a Mark&Sweep style [18]. Inside this B-zone, we design our custom GCs by readjusting the three regions sizes and the object size threshold to be included in LOS for each benchmark.

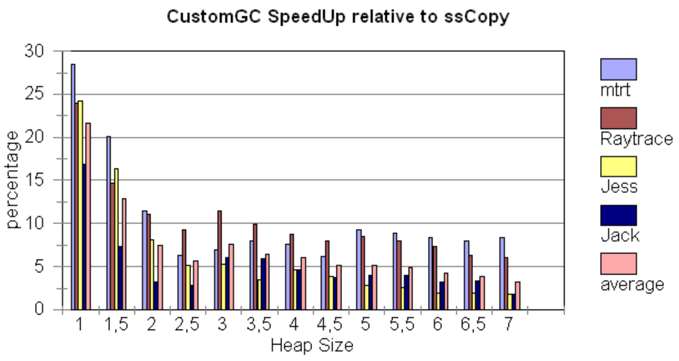

Figure 6. Speedups in total execution time of our custom GC compared to an optimized Semispace Copying Collector

3. *C-Zone*: This zone starts from the B-zone's limit size (4.5 or 5) until the end of our experimental heap size range, i.e. 7 times the minimum heap size. Going further in the maximum heap size allowed does not seem to vary the design of the custom GC, which means that this part of the GC design space would yield in this C-Zone GC implementation. In this zone we have observed the best results using a simple Non-Generational SemiSpace Copying collector with a Mark&Sweep Large Object Space [9]. Thus, we use two regions, and again we fix the size for them according to the DM behaviour of each benchmark and heap size. Furthermore, we need to adjust the object size threshold used for allocating data in LOS to get the best results.

We have compared our custom GCs with two completely different and representative general-purpose GCs (see Figure 5, Figure 6 and Figure 7). First, *ssCopy* is the classical Semispace Copying Collector with a Large Object Space (LOP) of a 4K threshold for the objects sizes [9, 7]. In addition, we have compared our custom GC with a state-of-the-art Generational Copying collector using a fixed nursery size of 10% of the heap size, i.e. *gen10%*. We have used this value because Blackburn [4] reports that the best range for fixed generational collectors is between 5% and 15%. Thus, we have chosen this intermediate value of 10% to measure our custom GC against.

The results obtained in the benchmarks comparing our custom GCs with these previous reference GCs show that our custom GCs reduces significantly their execution times. This speedup depends on the amount of memory available for the GC (see Figure 5 and Figure 6) and when more memory is available, more gains are achieved. As we explained in Subsection 3.1, it is possible in our GC design space to create a global GC as combination of several policies, which allows to reduce the weaknesses of each GC policy and ef-

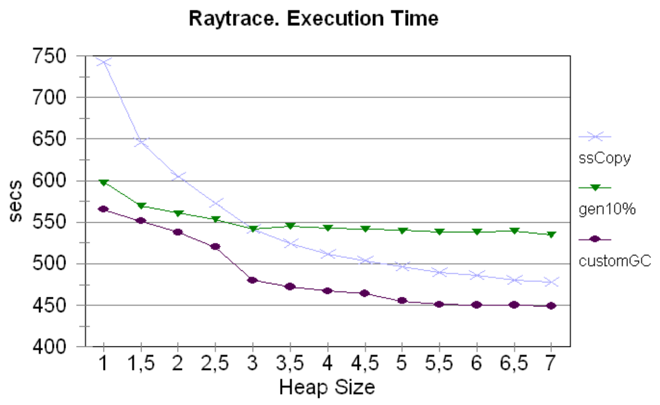

Figure 7. Execution time comparisons results for the Raytrace benchmark application

ficiently use the total DM available in the system. This is depicted in Figure 7, which shows the execution time results for the Raytrace benchmark application. Our custom GC accomplishes always the best results by combining application-specific versions of both state-of-the-art strategies in the same global GC and applying each strategy when they accomplish the best DM management of the system.

At last, to evaluate the complexity of the design process with our method, we want to remark that the design and implementation of the final custom GC managers for each case study presented took us only one week. These custom GCs are mainly optimized for execution time. However, trade-offs between the relevant design factors (e.g. reducing power consumption by increasing slightly the execution time) are possible using our method if the designer needs it.

6 Conclusions

New embedded devices can execute presently complex and dynamic applications (e.g. multimedia). These new applications include intensive dynamic memory requirements that must be heavily optimized (i.e. memory footprint, power and performance) for an efficient mapping on current consumer embedded devices. System-level exploration and refinement methodologies have started to be proposed to consistently perform these optimizations. Within them, the creation of convenient custom GC designs for these new multimedia applications is one of the most time-consuming and programming intensive parts. In this paper we have presented a new system-level method to design custom GC mechanisms. This method largely simplifies the complex engineering process of analyzing and designing suitable custom GCs for embedded devices, allowing the developers to cover a vast part of the GC design space (e.g. different garbage distinguishing mechanisms, incremental strategies, barriers to use, etc.) with a limited designing effort. In our future work we plan to investigate the ways to automate and formalize the trade-off exploration within our custom GCs (e.g. power consumption, performace, etc.) and the possible benefits of a mixed hardware-software implementation solution for them.

References

[1] A. Appel. Simple generational garbage collection and fast allocation. *SW Practice and Experience*, 1989.

[2] D. Atienza, S. Mamagkakis, et al. DM manag. design methodology for reduced mem. footprint in multim. and wireless network apps. *Accepted for Proc. of DATE '04*, 2004.

[3] L. Benini and G. De Micheli. System level power optimization techniques and tools. In *ACM TODAES*, April 2000.

[4] S. M. Blackburn et al. Ulterior reference counting: Fast GC without a long wait. In *Proc. ACM OOPSLA, USA, 2003*.

[5] P. C. David, F. Bacon et al. A real-time GC with low overhead and consistent utilization. In *Proc. ACM POPL, 2003*,

[6] J. Henry G. Baker. The treadmill: Real-time GC without motion sickness. *ACM SIGPLAN Notices*, 1992.

[7] IBM. Jikes' research virtual machine 2.2.0., 2003. `http://oss.software.ibm.com/developerworks/oss/jikesrvm/`.

[8] The source for java technology, 2003. `http://java.sun.com`.

[9] R. Jones. *Garbage Collection: Algorithms for Automatic DM Management*. John Wiley and Sons, July 2000.

[10] M. Leeman, D. Atienza, et al. Methodology for refinement and optim. of DM manag. for embedded syst. in multimedia apps. In *Proc. of SiPS*, 2003.

[11] N. Murphy. Safe memory usage with DM allocation. *Embedded Systems*, May 2000.

[12] NASA. Clips: A tool for building expert systems, 2003. `http://www.ghg.net/clips/CLIPS.html`.

[13] Oka and Suzuoki. Designing and programming the emotion engine. *IEEE Micro*, 1999.

[14] P. R. Panda, F. Catthoor, et al. Data and memory optimizations for embedded systems. *ACM TODAES*, April 2001.

[15] T. Parr. Purdue compiler construction tool set, 1992. `http://www.ece.purdue.edu/~hankd/PCCTS/`.

[16] SPEC. Specjvm98 documentation, March 1999. `http://www.specbench.org/osg/jvm98/`.

[17] D. Stefanovic, M. Hertz, et al. Older-first GC in practice: Evaluation in a java virtual machine. In *Proc. Workshop MSP, 2002*.

[18] P. R. Wilson. Uniprocessor garbage collection techniques. In *Proc. of Int. Workshop on Mem. Management*, 1992.

[19] P. R. Wilson, M. S. Johnstone, et al. Dynamic storage allocation, a survey and critical review. In *Int. Workshop on Mem. Management*, UK, 1995.

[20] D. S. Wise, et al. Research demonstration of a HW reference counting heap. Tech. report, Indiana Univ., 1997.