Dynamic Memory Management Design Methodology for Reduced Memory Footprint in Multimedia and Wireless Network Applications

David Atienza*, Stylianos Mamagkakis†, Francky Catthoor‡, Jose M. Mendias*, Dimitris Soudris†
*DACYA/UCM, Juan del Rosal 8, 28040 Madrid, Spain. {datienza, mendias}@dacya.ucm.es
† VLSI Center-Demokritus Univ., Thrace, 67100 Xanthi, Greece.{smamagka, dsoudris}@ee.duth.gr
‡IMEC vzw, Kapeldreef 75, 3001 Heverlee, Belgium. Email: {name.surname}@imec.be *

Abstract

New portable consumer embedded devices must execute multimedia and wireless network applications that demand extensive memory footprint. Moreover, they must heavily rely on Dynamic Memory (DM) due to the unpredictability of the input data (e.g. 3D streams features) and system behaviour (e.g. number of applications running concurrently defined by the user). Within this context, consistent design methodologies that can tackle efficiently the complex DM behaviour of these multimedia and network applications are in great need. In this paper, we present a new methodology that allows to design custom DM management mechanisms with a reduced memory footprint for such kind of dynamic applications. The experimental results in real case studies show that our methodology improves memory footprint 60% on average over current state-of-the-art DM managers.

1 Introduction

The multimedia and wireless network applications to be ported to new embedded systems have lately experienced a fast growth in their variety and functionality. These new applications (e.g. MPEG21 or new network protocols) depend, with few exceptions, on Dynamic Memory (DM from now on) for their operations due to the unpredictable input data (e.g. 3D streams features). Designing embedded systems for the (static) worst case memory footprint of these new applications would lead to a too high overhead in memory footprint. 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 [8]. Moreover, these intermediate static solutions will not work in extreme cases of input data, whereas DM solutions can do it. Thus, DM management must be used in embedded realisations of these designs.

Many general DM management policies and implementations of them are nowadays available to provide relatively good performance and low fragmentation for general-purpose systems [19]. However, for embedded systems, such DM managers must be implemented inside their constrained Operating System (OS) and thus have to really consider the limited resources available to minimize memory footprint among other factors. Thus, recent embedded OSs (e.g. [12]) use specifically designed (or custom) DM managers according to the underlying memory hierarchy and the kind of applications that will run on them.

Custom DM managers are frequenly designed to improve performance [3, 19], but they can also be used to heavily optimize memory footprint compared to general-purpose managers. For example, in new 3D vision algorithms [15], a well designed custom DM manager can improve memory footprint up to 44.2% over conventional general-purpose DM managers [8].

When custom DM managers are used, their designs are manually optimized by the developer, typically considering only a limited number of design and implementation alternatives due to the lack of formal methodologies to explore the DM management search space. Thus, designers must define, construct and evaluate new custom implementations of DM managers and strategies manually, which has been proven to be programming intensive (and very time-consuming). Even if the embedded OS offers considerable support for standardized languages (e.g C or C++), the developer is still faced with defining the structure of the DM manager on a case per case basis.

In this paper, we present a new methodology that allows to design custom DM management mechanisms with the reduced memory footprint required for new dynamic embedded applications, i.e. multimedia and wireless network applications. First, this methodology defines the relevant design space of DM management decisions for a minimal memory footprint in multimedia and wireless network applications. Then, we propose a suitable order to traverse it according to the relative influence of each decision in memory footprint and to design custom DM managers according to the specific DM behaviour of the application under study.

The remainder of the paper is organized as follows. In

*This work is partially supported by the Spanish Government Research Grant TIC2002/0750 and the European founded program AMDREL IST-2001-34379.

1530-1591/04 $20.00 (c) 2004 IEEE

Section 2, we describe some related work. In Section 3, we present our DM management search space of decisions for a reduced memory footprint in dynamic applications. In Section 4 we define the order to traverse it to reduce the memory footprint of the application under analysis. In Section 5, we introduce our case studies and present the experimental results. Finally, in Section 6 we draw our conclusions.

2 Related Work

Currently the basis of an efficient DM management in a general context is already well established and much literature is available about software general-purpose DM management implementations and policies [19]. Also, research on custom DM managers that use application-specific behaviour to improve performance and locality of references to minimize fragmentation is also available [18, 19].

In new embedded systems where the range of applications to be executed is very wide (e.g. new consumer devices), variations of well-known state-of-the-art general-purpose DM managers are used. For example, Linux-based systems use as their basis the Lea DM manager [19] and Windows-based systems use the Kingsley DM manager [19]. Finally, recent real-time OSs for embedded systems (e.g. [12]) support dynamic allocation with custom DM managers based on simple region allocators [6] with a reasonable performance for the specific platform features.

Recent methods to refine the DM subsystem try to empirically evaluate it with customizable DM managers. In [18], a DM manager that allows to define multiple memory regions with different disciplines is presented. However, this approach cannot be extended with new functionality and is limited to a small set of user-defined functions for memory de/allocation. In [3], the abstraction level of customizable DM managers has been extended to C++. It proposes an infrastructure of C++ layers that can be used to improve performance of general-purpose managers. The main problem in this case is its extensibility for other metrics (e.g. energy dissipation), as embedded systems require.

Regarding memory optimizations and techniques to optimize memory footprint and other relevant factors in static data for embedded systems (e.g. power consumption or performance), much research has been done (see e.g. [13, 2] for good tutorial overviews). All these techniques are complementary to our work and perfectly applicable in the part of the code that accesses static data within the considered dynamic applications. Moreover, they are useful as back-end for our approach, once the dynamic data are allocated into memory pools that can be statically declared.

3 Relevant DM Management Search Space

Much literature is available about possible implementation choices for DM management mechanisms [19], but none of the earlier work provides a complete search space useful for a systematic exploration in multimedia and wireless network applications for embedded systems. Hence, in this section we first define our design search space of relevant DM management decisions for a reduced memory footprint and then we summarize the interdependencies observed within it, which partially allows us to order it.

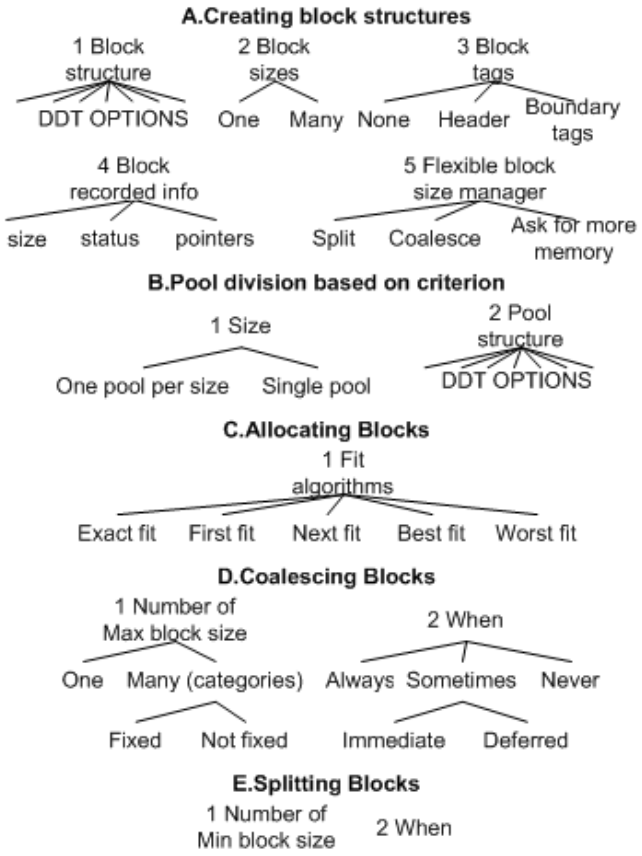

Figure 1. DM management search space of orthogonal decisions

3.1 Our DM Management Search Space for Reduced Memory Footprint

DM management basically consists of two separate tasks, i.e. allocation and deallocation. Allocation is the mechanism that searches for a block big enough to satisfy the request of a given application and deallocation is the mechanism that returns this block to the available memory of the system in order to be reused later. In real applications, the blocks are requested and returned in any order, thus creating ”holes” among used blocks. These holes are known as memory fragmentation. On the one hand, internal fragmentation occurs when a bigger block than the one needed is chosen to satisfy a request. On the other hand, if the memory to satisfy a memory request is available, but not contiguous (thus it cannot be used for that request), it is called external fragmentation. Hence, on top of memory de/allocation, the DM manager has to take care of fragmen-

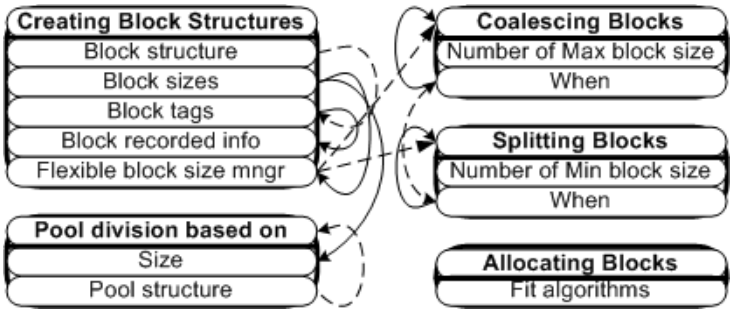

Figure 2. Interdependencies between orthogonal trees in the search space

tation issues. This is done by splitting and merging free blocks to keep memory fragmentation as small as possible. Finally, to support these mechanisms, additional data structures are built to keep track of the free and used blocks. Thus, to create an efficient DM manager, we have to systematically classify the design decisions that can be taken to handle the possible combinations of the previous factors (e.g. fragmentation, overhead of additional data structures).

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

Then, these trees have been grouped in categories according to the different main parts that can be distinguished in DM managements [19]. They are shown in Figure 1. This new approach allows us to reduce the complexity of the global design of DM managers in smaller subproblems that can be decided locally. In our search space, any combination of a leaf from each of the decision trees represents a valid DM manager. This fact leads to a huge amount of potential implementations that can be used not only to recreate any available general-purpose DM manager [19], but also create our own new highly-specialized DM managers.

In the following we describe the five main categories of Figure 1 and the important decision trees inside them for the creation of DM managers with a reduced memory footprint:

A. Creating block structures, which handles the way block data structures are created and later used by the DM managers to satisfy the memory requests. More specifically, the *Block structure* tree specifies the different blocks of the system and their internal control structures. In this tree, we have included all possible combinations of Dynamic Data Types (from now on called DDTs) required to represent and construct any dynamic data representation [4] used in the current DM managers. Secondly, the *Block sizes* tree refers to the different sizes of basic blocks available for DM management, which may be fixed or not. Thirdly, the *Block tags* and the *Block recorded info* trees specify the extra fields needed inside the block to store information used by the DM manager. Finally, the *Flexible block size manager* tree decides if the splitting and coalescing mechanisms are activated according to the the availability of the size of the memory block requested.

B. Pool division based on, which deals with the number of pools (or memory regions) present in the DM manager and the reasons why they are created. The *Size* tree means that pools can exist either containing internally blocks of several sizes or they can be divided so that one pool exists per different block size. In addition, the *Pool structure* tree specifies the global control structure for the different pools of the system. In this tree we include all possible combinations of DDTs required to represent and construct any dynamic data representation [4].

C. Allocating blocks, which deals with the actual actions required in DM management to satisfy the memory requests and couple them with a free memory block. Here we have included all the important choices available in order to choose a block from a list of free blocks [19].

D. Coalescing blocks, which concerns the actions executed by the DM managers to ensure a low percentage of external memory fragmentation, i.e. merging two smaller blocks into a larger one. Firstly, the *Number of max block size* tree defines the new block sizes that are allowed after coalescing two different adjacent blocks. Then, the *When* tree defines how often coalescing should be performed.

E. Splitting blocks, which refers to the actions executed by the DM managers to ensure a low percentage of internal memory fragmentation, i.e. splitting one larger block into two smaller ones. Firstly, the *Number of min block size* tree defines the new block sizes that are allowed after splitting a block into smaller ones. And the *When* tree defines how often splitting should be performed (these trees are not presented in full detail in Figure 1, because the options are the same as in the two trees in the Coalescing category).

3.2 Interdependencies between Orthogonal Trees

Although the decision categories and trees presented in Subsection 3.1 are orthogonal, certain leaves in some trees affect heavily the coherent decisions in other trees. Thus, they possess interdependencies to take into account when a DM manager is designed. The whole set of interdependencies for our search space is shown in Figure 2. They can be classified in two main groups. First, the interdependencies caused by leaves that disable the use of other trees or cate-

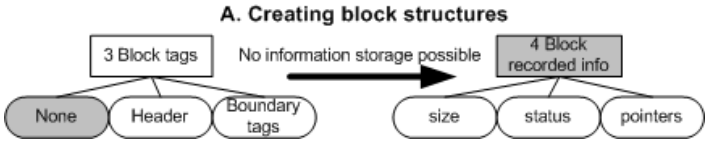

Figure 3. Example of interdependency between two orthogonal trees

gories (full arrows in Figure 2). An example of this kind of interdepencies is shown in Figure 3. It shows how the *Block tags* tree restricts the *Block recorded info* tree. The choice of the none leaf in the *Block tags* tree indeed prohibits the use of the *block recorded info* tree, because no space is reserved to store any information. Second, it also shows the interdependencies affecting other trees or categories due to their linked purposes (dotted arrows in Figure 2).

3.3 Construction of Global DM Managers

Real new multimedia and wireless network applications include different DM behaviour patterns, which are linked to their logical phases. Consequently, our methodology must be applied to each of these different phases separately in order to create an atomic custom DM manager for each of them. Then, the global DM manager of the application is the inclusion of all these atomic DM managers in one.

4 Order for Reduced DM Footprint

4.1 Factors of influence for DM footprint

The main factors that affect DM footprint are two:

1.- The *Organization overhead*, which is produced by the assisting fields and data structures that accompany each block and pool respectively. It depends on the following:

a) The fields (e.g. headers, footers) inside the memory blocks are used to store data of each specific block and are usually a few bytes long. The use of these fields is controlled by category A (Creating block structures).

b)The assisting data structures provide the infrastructure to organize the pool and to characterize its behaviour. They can be used to prevent fragmentation by forcing the blocks to reserve memory according to their size without having to split and coalesce unnecessarily. The use of these data structures is controlled by category B (Pool division based on criterion). The same effect on fragmentation prevention is also present in category C, because depending on the fit algorithm chosen, you can avoid internal fragmentation.

2.- The *Fragmentation memory waste* is caused by the internal and external fragmentation (discussed in Subsection 3.1), which depend on the following:

a)The internal fragmentation is mostly remedied by category E (Splitting blocks). This fragmentation affects mostly small data structures. E.g. if only 100-byte blocks are available inside the pools and you want to allocate 20-byte blocks, it would be wise to split each 100-byte block to 5 blocks of 20 bytes to avoid internal fragmentation.

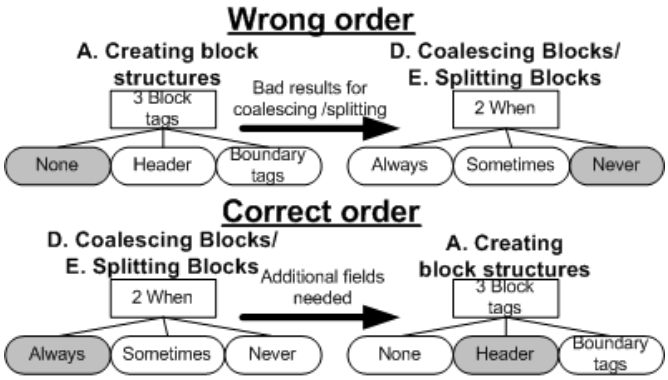

Figure 4. Example of the correct order between two orthogonal trees

b)The external fragmentation is mostly remedied by category D (Coalescing blocks). As expected, it affects mostly big data requests. E.g. if you want to allocate a 50-Kbyte block, but only 500-byte blocks exist inside the pools, it would be necessary to coalesce 100 blocks to provide the necessary amount of memory requested.

Note the distinction between categories D and E, which try to deal with fragmentation, as opposed to category B and C that try to prevent it.

4.2 Order of the trees for reduced memory size

Trees A2 and A5 are placed first to decide the global structure of the blocks. Then, experience suggests that most of the times it is more important to define how to deal with fragmentation (thus categories D and E go next) than try to prevented it with a convenient pool organization and allocation scheme [19], which are decided by categories B and C. Finally, the rest of the trees in category A (i.e. A1, A3 and A4) are decided. As a result, after also considering the inderdependencies, the final order is as follows: A2->A5->E2->D2->E1->D1->B4->B1->C1->A1->A3->A4.

If the order we have just proposed is not followed, unnecessary constraints are propagated to next decision trees, and thus the most suitable decisions cannot be taken in the remaining orthogonal trees. Figure 4 shows an explample of this. Suppose that the order was A3 and then D2/E2. When deciding the correct leaf for A3, the obvious choice to save memory space would be to choose the *None* leaf. This seems reasonable at first sight because the header fields would require a fixed amount of additional memory for each block that has to be allocated. However, now we are obliged to choose the *Never* leaf in D2/E2 because after propagating the constraints of A3, one block cannot be properly split or coealesced without storing information about the size of the block. Hence, the final DM manager uses less memory per block, but cannot deal with internal or external fragmentation by splitting or coalescing blocks.

However, if the application includes a variable amount of sizes, the fragmentation problem consumes more mem-

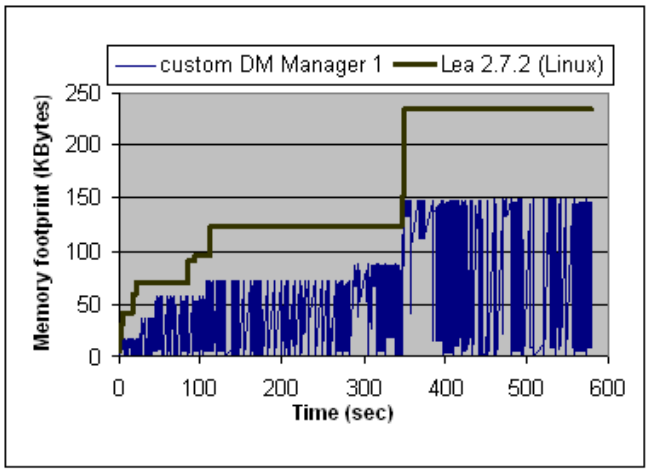

Figure 5. Memory footprint behaviour of Lea and our DM manager for the DRR application

ory than the extra header fields needed for coalescing and splitting. Therefore it is necessary to decide the D2/E2 tree and then propagate its constraints to tree A3. Hence, we select first the leaf *Always* in both cases, which are the leaves that deal better with internal (tree E2) and external fragmentation (D2). Then, after propagating these constraints to A3, we can select the leaf which better supports these decisions in D2 and E2, i.e. the *Header* leaf for A3.

5 Case Studies and Experimental Results

We have applied the proposed methodology to three case studies that represent different modern multimedia and network application domains: the first case study is a scheduling algorithm from the network domain, the second one is part of a new 3D image reconstruction system and the third one is a 3D rendering system based on scalable meshes.

All the results shown are average values after a set of 10 simulations for each DM manager, where all the final values were very similar (variations of less than 2%).

The first case study presented is the Deficit Round Robin (DRR) application [16] taken from the NetBench benchmarking suite [10]. It is a scheduling algorithm implemented in many routers today that tries to accomplish a fair scheduling by allowing the same amount of data to be passed and sent from each internal queue. It requires the use of DM because the real input can vary enormously depending on the network traffic. In our experiments, 10 real traces of internet network traffic up to 10 Mbit/sec [7] have been used to run realistic simulations of DRR.

In order to define the logical phases of the application and its atomic DM managers, we first profile its DM behaviour. Then, we create the global DM manager. First, we make a decision in tree A2 (Block sizes) and our decision is to have `many` block sizes to prevent internal fragmentation because the DRR application requests memory blocks that vary greatly in size (to store incoming packets of different sizes). Then, in tree A5 (Flexible block size manager) we choose to `split` and `coalesce`, so that every time a memory block with a bigger or smaller size than the current block is requested, the splitting and coalescing mechanims are invoked. In trees E2 and D2 (When) we choose `always`, thus we try to defragment as soon as it occurs. Then, in trees E1 and D1 (number of max and min block size) we choose `many` and `not fixed` because we want to get the maximum effect out of coalescing and splitting by not limiting the size of the produced blocks. After this, in trees B1 (Pool division based on size) and B2 (Pool structure), the simplest pool implementation possible is selected, i.e. `single pool`. Next, in tree C1 (Fit algorithms), we choose the `exact fit` to avoid as much as possible memory lost in internal fragmentation. Next, in tree A1 (Block structure), we choose the most simple DDT that allows coalescing and splitting, i.e. `double linked list`. Then, in the trees A3 (Block tags) and A4 (Block recorded info), we choose a `header` field to accomodate information about the `size` and `status` of each block to support the splitting and coalescing mechanisms. Finally, after taking these decisions following the order described in Section 4, we determine those decisions of the final custom DM manager that depend on its particular run-time behaviour in the application (e.g. final number of max block sizes) via simulation. To this end, we have developed a `C++` library that covers the decisions in our DM search space [5].

Then, we have compared our custom solution to state-of-the-art general-purpose managers, i.e. Lea [19] and Kingsley [19] (see Section 2 for more details). As Table 1 shows, our custom DM manager uses less memory than Lea or Kingsley because it have not got fixed sized blocks and tries to coalesce and split as much as possible, which is a better option in dynamic applications with very variable sizes. Moreover, when large coalesced chunks of memory are not used, they are returned back to the system for other applications. However, Lea and Kingsley create huge free-lists of unused blocks (in case they can be reused later), they coalesce and split seldomly (Lea) or never (Kingsley) and finally, they have fixed-sized blocks. This can be observed in Figure 5, which shows the DM footprint graphs of our DM manager (`custom DM manager 1`) and Lea during one run of the application. Hence, our custom DM manager improves the memory footprint by 36% compared to Lea and 93% compared to Kingsley.

Our second case study is one of the sub-algorithms of a 3D reconstruction algorithm [15] (see [17] for the full code of the algorithm, 1.75 million lines of `C++` code), where the relative displacement between frames is used to reconstruct the 3^{rd} dimension. It requires DM due to the unpredictable features of the input images at compile-time (e.g. number of possible corners to match varies on each image). The operations done on the images are memory intensive, e.g.

Dyn. Mem. managers	DRR scheduler	3D image reconst.	3D scalable rendering
Kingsley-Windows	2.09×10^6	2.26×10^6	3.96×10^6
Lea-Linux	2.34×10^5	-	1.86×10^6
Regions	-	2.08×10^6	-
Obstacks	-	-	1.55×10^6
our DM manager	1.48×10^5	1.49×10^6	1.07×10^6

Table 1. Maximum memory footprint results (Bytes) in real case studies

each image of 640×480 uses over 1Mb. Moreover, since the accesses to the images are randomized, classic image access optimizations as row-dominated versus column-wise accesses cannot be used to reduce memory footprint.

In this case study we have compared our custom DM manager with a manually designed implementation of the new kind of region managers [6] found in new embedded OSs (e.g. [12]). Also, we have compared our DM manager with Kingsley. The memory footprint results obtained are depicted in Table 1. They show that our custom DM manager obtains significant improvements compared to region managers (28.47%) and Kingsley (33.01%). These improvements occur because our DM manager reduces the fragmentation of the system in two ways. First, its behaviour varies according to the specific block sizes requested in the application. Second, it uses immediate coalescing and splitting to reduce fragmentation. In region managers, the block sizes of each region are fixed to one block size and the requests of several block sizes creates internal fragmentation. In Kingsley, an initial memory region is reserved and distributed among the different lists of block sizes. However, only a limited amount of block sizes is used and thus memory is misused.

Our third case study is a 3D video rendering application [20]. It belongs to a new category of video algorithms that adapt the quality of each object on the screen with scalable meshes [9] according to the position of the user watching at them at each moment of time (e.g. QoS systems [14]).

In this case, we have compared our DM manager with Lea, Kingsley and, due to its stack-like allocation behaviour in some phases of its execution, we have also used Obstacks [19], a well-known custom DM manager optimized for such behaviour. As Table 1 shows, Lea obtains better results in memory footprint (53%) than Kingsley. Then, Obstacks improves the memory footprint of Lea (17.70%). Finally, our custom manager improves the memory footprint of Obstacks (30%) because Obstacks cannot exploit its stack-like optimizations in the final phases of the rendering process. Thus, it suffers from a high memory footprint penalty in these phases whereas our DM manager does not.

Finally, to evaluate the design process with our methodology, we want to remark that the design and implementation of the final custom DM managers for each case study took us two weeks. These DM managers achieve the least memory footprint values with only a 10% overhead (on average) over the execution time of the fastest general-purpose DM manager observed in these case studies, i.e. Kingsley. Moreover, this decrease in performance is not relevant since our custom DM managers preserve the real-time behaviour required by the applications. Thus, the user will not notice any difference. Nevertheless, trade-offs between the relevant design factors (e.g. improving performance consuming a little more memory footprint) are possible using our methodology, if the requirements of the final design need it.

6 Conclusions

New consumer devices have improved their capabilities and, nowadays, very complex and dynamic applications can be mapped on them. However, to port these applications, new design methodologies must be available to efficiently use the memory available in the final embedded systems. In this paper we have presented a new methodology that defines and explores the DM management search space of relevant decisions, in order to design custom DM managers with a reduced memory footprint for such dynamic applications. Our results in real applications show significant improvements in memory footprint over state-of-the-art general-purpose and manually optimized custom DM managers, incurring only in a small overhead in execution time over the fastest of them.

References

[1] G. Attardi, T. Flagella, et al. A customizable mem. management framework for c++. *SW Practice and Experience*, 1998.
[2] L. Benini and G. De Micheli. System level power optimization techniques and tools. In *ACM TODAES*, 2000.
[3] E. D. Berger, B. G. Zorn, et al. Composing high-performance memory allocators. In *Proc. of PLDI*, 2001.
[4] E. G. Daylight, S. Wuytack, et al. Analyzing energy friendly phases of dyn. apps. in terms of sparse data. In *Proc. of ISLPED*, 2002.
[5] D. Atienza, S. Mamagkakis, et el. Fast system-level prototyping of power-aware DM managers for emb. syst. In *Proc. of COLP*, 2003.
[6] D. Gay, et al. Mem. manag. with regions. In *Proc. of PLDI*, 2001.
[7] LBN Lab. Internet traffic archive. `http://ita.ee.lbl.gov/`.
[8] M. Leeman, et al. Methodology for refinement of dyn. mem. manag. for embedded syst. in multimedia apps. In *Proc. SiPS*, Korea, 2003.
[9] D. Luebke, et al. *Level of Detail for 3D Graphics*, 2002.
[10] G. Memik, et al. Netbench: A benchmarking suite for network processors. CARES Technical Report, 2001.
[11] N. Murphy. Safe mem. usage with DM alloc. *Embedded Syst.*, 2000.
[12] Rtems, open-source real-time OS for embedded systems, 2002. `http://www.rtems.org`.
[13] P. R. Panda, F. Catthoor, et al. Data and memory optimizations for embedded systems. *ACM TODAES*, April 2001.
[14] N. Pham Ngoc, et al. Qos framework for interactive 3D apps. In *Proc. of Conf. on Computer Graphics and Vision*, 2002.
[15] M. Pollefeys, et al. Metric 3D surface reconst. from uncalibrated image seqs. In *Lect. Notes in Computer Science*, 1998.
[16] M. Shreedhar and G. Varghese. Efficient fair queuing using deficit round robin. In *Proc. of SIGCOMM*, 1995.
[17] Target jr, 2002. `http://www.targetjr.org`.
[18] K.-P. Vo. Vmalloc: A general and efficient memory allocator. In *SW Practice and Experience*, 1996.
[19] P. R. Wilson, et al. Dynamic storage allocation, a survey and critical review. In *Int. Workshop on Memory Management*, UK, 1995.
[20] M. Woo, et al. *OpenGL Programming Guide, 2nd Ed.* USA, 1997.