Systematic Dynamic Memory Management Design Methodology for Reduced Memory Footprint

DAVID ATIENZA and JOSE M. MENDIAS
DACYA, Complutense University of Madrid
STYLIANOS MAMAGKAKIS, and DIMITRIOS SOUDRIS
VLSI Design and Test Center, Democritus University of Thrace
and
FRANCKY CATTHOOR
DESICS Division, IMEC

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 behavior (e.g., number of applications running concurrently defined by the user). Within this context, consistent design methodologies that can tackle efficiently the complex DM behavior of these multimedia and network applications are in great need. In this article, we present a new methodology that allows to design custom DM management mechanisms with a reduced memory footprint for such kind of dynamic applications. First, our methodology describes the large design space of DM management decisions for multimedia and wireless network applications. Then, we propose a suitable way to traverse the aforementioned design space and construct custom DM managers that minimize the DM used by these highly dynamic applications. As a result, our methodology achieves improvements of memory footprint by 60% on average in real case studies over the current state-of-the-art DM managers used for these types of dynamic applications.

Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-Based Systems**]: *Real-time and embedded systems*

General Terms: Design, Management, Measurement, Performance

This work was partially supported by the Spanish Government Research Grant TIC2002/0750, the European founded program AMDREL IST-2001-34379 and E.C. Marie Curie Fellowship contract HPMT-CT-2000-00031.

Authors' addresses: D. Atienza and J. M. Mendias, DACYA/UCM, Avda. Complutense s/n, 28040 Madrid, Spain. email: {datienza, mendias}@dacya.ucm.es; S. Mamagkakis and D. Soudris, VLSI Design and Testing Center – Democritus University, Thrace, 67100 Xanthi, Greece. email: {smamagka,dsoudris}@ee.duth.gr; F. Catthoor, IMEC vzw, Kapeldreef 75, 3001 Heverlee, Belgium. email: {francky.catthoor}@imec.be.

Additional Key Words and Phrases: Multimedia embedded systems, custom dynamic memory management, reduced memory footprint, memory management, operating systems

1. INTRODUCTION

In order to cope with the increasing complexity, the drastic rise in memory requirements, and the shortened time-to-market of modern consumer embedded designs, new system design methodologies are required. In the past, most applications that were ported to embedded platforms stayed mainly in the classic domain of signal processing and actively avoided algorithms that employ data de/allocated dynamically at run-time, also called Dynamic Memory (DM from now on). Recently, the multimedia and wireless network applications to be ported to embedded systems have experienced a very fast growth in their variety, complexity and functionality. These new applications (e.g., MPEG4 or new network protocols) depend, with few exceptions, on DM for their operations due to the inherent unpredictability of the input data. Designing the final embedded systems for the (static) worst-case memory footprint[1] of these new applications would lead to a too high overhead 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., approximately 25% more) than DM solutions [Leeman et al. 2003]. Furthermore, these intermediate static solutions do not work in extreme cases of input data, while DM solutions can do it since they can scale the required memory footprint. Thus, DM management mechanisms must be used in 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 [Wilson et al. 1995]. However, for embedded systems, such managers must be implemented inside their constrained Operating System (OS) and thus have to take into account the limited resources available to minimize memory footprint among other factors. Thus, recent embedded OSes (e.g., RTEMS [2002]) use custom DM managers according to the underlying memory hierarchy and the kind of applications that will run on them.

Usually custom DM managers are designed to improve performance [Berger et al. 2001; Wilson et al. 1995], but they can also be used to heavily optimize memory footprint compared to general-purpose DM managers, which is very relevant for final energy and performance in new embedded systems as well since many concurrent dynamic applications have to share the limited on-chip memory available. For instance, in new 3D vision algorithms [Pollefeys et al. 1998], a suitably designed custom DM manager can improve memory footprint by 45% approximately over conventional general-purpose DM managers [Leeman et al. 2003]. However, when custom DM managers are used, their designs have to be manually optimized by the developer, typically considering only a limited number of design and implementation alternatives, which are defined based on his experience and inspiration. This limited exploration is mainly

[1]Accumulated size of all the data allocated in memory and counted in bits.

restricted due to the lack of systematic methodologies to consistently explore the DM management design space. As a result, designers must define, construct and evaluate new custom implementations of DM managers and strategies manually, which has been proved to be programming intensive (and very time-consuming). Even if the embedded OS offers considerable support for standardized languages, such as C or C++, the developer is still faced with defining the structure of the DM manager and how to profile it on a case per case basis.

In this article, we present a new methodology that allows developers to design custom DM management mechanisms with the reduced memory footprint required for these new dynamic multimedia and wireless network applications. First of all, this methodology delimits the relevant design space of DM management decisions for a minimal memory footprint in new dynamic embedded applications. After that, we have studied the relative influence of each decision of the design space for memory footprint and defined a suitable order to traverse this design space according to the DM behavior of these new dynamic applications. As a result, the main contributions of our methodology are two-fold: (1) the definition of a consistent orthogonalization of the design space of DM management for embedded systems and (2) the definition of a suitable order for new dynamic multimedia and wireless network applications (and any other type of embedded applications that possesses the same dynamic de/allocation characteristics) to help designers to create very customized DM managers according to the specific dynamic behavior of each application.

The remainder of the article is organized in the following way: In Section 2, we describe relevant related work. In Section 3, we present the relevant DM management design space of decisions for a reduced memory footprint in dynamic applications. In Section 4, we define the order to traverse this design space in order to minimize the memory footprint of the application under analysis. In Section 5, we outline the global design flow proposed in our methodology to minimize the memory footprint in dynamic embedded applications. In Section 6, we introduce our case studies and present in detail the experimental results obtained. Finally, in Section 7, we draw our conclusions.

2. RELATED WORK

Currently the basis of an efficient DM management in a general-context are already well established. Much literature is available about general-purpose DM management software implementations and policies [Wilson et al. 1995]. Trying to reuse this extensive available literature, new embedded systems where the range of applications to be executed is very wide (e.g., new consumer devices) tend to use variations of well-known state-of-the-art general-purpose DM managers. For example, Linux-based systems use as their basis the Lea DM manager [Berger et al. 2001; Wilson et al. 1995] and Windows-based systems (both mobile and desktop) include the ideas of the Kingsley DM manager [Microsoft MSDN (a);(b); Wilson et al. 1995]. Finally, recent real-time OSes for embedded systems (e.g., RTEMS [2002]) support DM de/allocation via custom DM managers based on simple region allocators [Gay and Aiken 2001] with a reasonable level of performance for the specific platform features. All these

approaches propose optimizations considering general-purpose systems where the range of applications are very broad and unpredictable at design time, while our approach takes advantage of the special DM behavior of multimedia and wireless network applications to create highly customized and efficient DM managers for new embedded systems.

Lately, research on custom DM managers that take application-specific behavior into account to improve performance has appeared [Vo 1996; Wilson et al. 1995; Berger et al. 2001]. Also, for improving speed in highly constrained embedded systems, Murphy [2000] proposes to partition the DM into fixed blocks and place them in a single linked list with a simple (but fast) fit strategy, for example, first fit or next fit [Wilson et al. 1995]. In addition, some partially configurable DM manager libraries are available to provide low memory footprint overhead and high level of performance for typical behaviors of certain application (e.g., Obstacks [Wilson et al. 1995] is a custom DM manager optimized for a stack-like allocation/deallocation behavior). Similarly, Vo [1996] proposes a DM manager that allows to define multiple regions in memory with several user-defined functions for memory de/allocation. Additionally, since the incursion in embedded systems design of object-oriented languages with support for automatic recycling of dead-objects in DM (usually called garbage collection), such as Java, work has been done to propose several automatic garbage collection algorithms with relatively limited overhead in performance, which can be used in real-time systems [Bacon et al. 2003; Blackburn and McKinley 2003]. In this context, also hardware extensions have been proposed to perform garbage collection more efficiently [Srisa-an et al. 2003]. The main difference of these approaches compared to ours is that they mainly aim at performance optimizations and propose ad-hoc solutions without defining a complete design space and exploration order for dynamic embedded systems as we propose in this paper.

In addition, research has been performed to provide efficient hardware support for DM management. Chang et al. [1999] presents an Object Management Extension (i.e., OMX) unit to handle the de/allocation of memory blocks completely in hardware using an algorithm which is a variation of the classic binary buddy system. Shalan and Mooney [2000] proposes a hardware module called SoCDMMU (i.e., System-On-a-Chip Dynamic Memory Management Unit) that tackles the global on-chip memory de/allocation to achieve a deterministic way to divide the memory among the processing elements of SoC designs. However, the OS still performs the management of memory allocated to a particular on-chip processor. All these proposals are very relevant for embedded systems where the hardware can still be changed, while our work is thought for fixed embedded designs architectures where customization can only be done at the OS or software level.

Finally, a lot of research has been performed in memory optimizations and techniques to reduce memory footprint and other relevant factors (e.g., power consumption or performance) in static data for embedded systems (see surveys in Panda et al. [2001] and Benini and De Micheli [2000]). All these techniques are complementary to our work and are usable for static data that usually are also present in the dynamic applications we consider.

3. RELEVANT DYNAMIC MEMORY MANAGEMENT DESIGN SPACE FOR DYNAMIC MULTIMEDIA AND WIRELESS NETWORK APPLICATIONS

In the software community, much literature is available about possible design choices for DM management mechanisms [Wilson et al. 1995], but none of the earlier work provides a complete design space useful for a systematic exploration in multimedia and wireless network applications for embedded systems. Hence, in Section 3.1, we first detail the set of relevant decisions in the design space of DM management for a reduced memory footprint in dynamic multimedia and wireless network applications. Then, in Section 3.2, we briefly summarize the interdependencies observed within this design space, which partially allow us to order this vast design space of decisions. Finally, in Section 3.3, we explain how to create global DM managers for new dynamic multimedia and wireless network applications.

3.1 Dynamic Memory Management Design Space for Reduced Memory Footprint

Conventional DM management basically consists of two separate tasks, namely allocation and de-allocation [Wilson et al. 1995]. Allocation is the mechanism that searches for a block big enough to satisfy the request of a given application and de-allocation is the mechanism that returns this block to the available memory of the system in order to be reused later by another request. In real applications, the blocks can be 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. Therefore, on top of memory allocation and de-allocation, the DM manager has to take care of fragmentation issues as well. This is done by splitting and coalescing free blocks to keep memory fragmentation as small as possible. Finally, to support all these mechanisms, additional data structures should be built to keep track of all the free and used blocks, and the defragmentation mechanisms. As a result, to create an efficient DM manager, we have to systematically classify the design decisions that can be taken to handle all the possible combinations of these previous factors that affect the DM subsystem (e.g., fragmentation, overhead of the additional data structures, etc.).

We have classified all the important design options that constitute 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, thus covering the whole possible design space. Then, the relevance of a certain solution in each concrete system depends on its design constraints, which implies that some solutions in each design may not meet all timing and cost constraints for that concrete system. Furthermore, the decisions in the different orthogonal trees can be sequentially 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

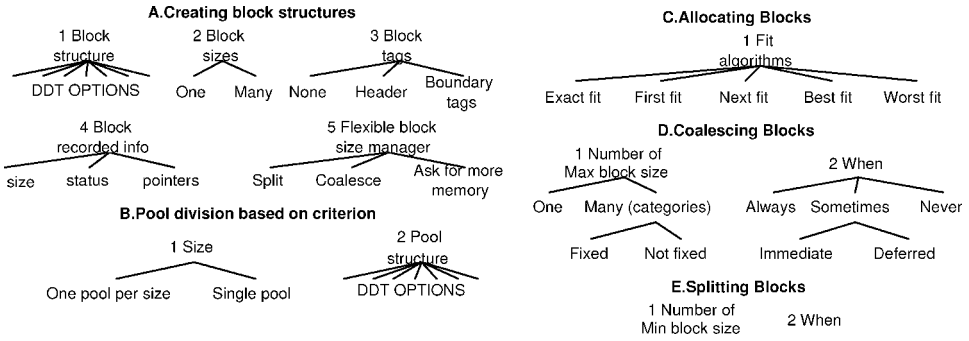

Fig. 1. Overview of our DM management design space of orthogonal decisions for reduced memory footprint.

has been taken in every tree, one custom DM manager is defined (in our notation, atomic DM manager) for a specific DM behavior pattern (usually, one of the DM behavior phases of the application). In this way, we can recreate any available general purpose DM manager [Wilson et al. 1995] or create our own highly specialized DM managers.

Then, these trees have been grouped in categories according to the different main parts that can be distinguished in DM management [Wilson et al. 1995]. An overview of the relevant classes of this design space for a reduced memory footprint is 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, making feasible the definition of a convenient order to traverse it.

In the following, we describe the five main categories and decision trees shown in Figure 1. For each of them, we focus on the decision trees inside them that are important 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 [Daylight et al. 2004; Leeman et al. 2003] used in the current DM managers. Second, the *Block sizes* tree refers to the different sizes of basic blocks available for DM management, which may be fixed or not. Third, 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 or extra memory is requested from the system. This depends on 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 of memory pools [Daylight et al. 2004; Leeman et al. 2003].

—*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 include all the important choices available in order to choose a block from a list of free blocks [Wilson et al. 1995]. Note that a Deallocating blocks category with the same trees as this category could be created, but we do not include it in Figure 1 to avoid adding complexity unnecessarily to our DM management design space. The fact is that the Allocating blocks category possesses more influence for memory footprint than the additional Deallocating blocks category. Moreover, these two categories are so tightly linked together regarding memory footprint of the final solution that the decisions taken in one must be followed in the other one. Thus, the Deallocating blocks category is completely determined after selecting the options of this Allocating block category.

—*D. Coalescing blocks*, which is related to the actions executed by the DM managers to ensure a low percentage of external memory fragmentation, namely merging two smaller blocks into a larger one. The *Number of max block size* tree defines the new block sizes that are allowed after coalescing two different adjacent blocks. 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, namely splitting one larger block into two smaller ones. The *Number of min block size* tree defines the new block sizes that are allowed after splitting a block into smaller ones. 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 of the Coalescing category).

3.2 Interdependencies between the Orthogonal Trees

After this definition of the decision categories and trees, in this section, we identify their possible interdependencies. They impose a partial order in the characterization of the DM managers. The decision trees are orthogonal, but not independent. Therefore, the selection of certain leaves in some trees heavily affects the coherent decisions in the others (i.e., interdependencies) when a certain DM manager is designed. The whole set of interdependencies for our design space is shown in Figure 2. These interdependencies can be classified in two main groups. First, the interdependencies caused by certain leaves, trees or categories, which disable the use of other trees or categories (drawn as full arrows in Figure 2). Second, the interdependencies affecting other trees or categories due to their linked purposes (shown as dashed arrows in Figure 2). The

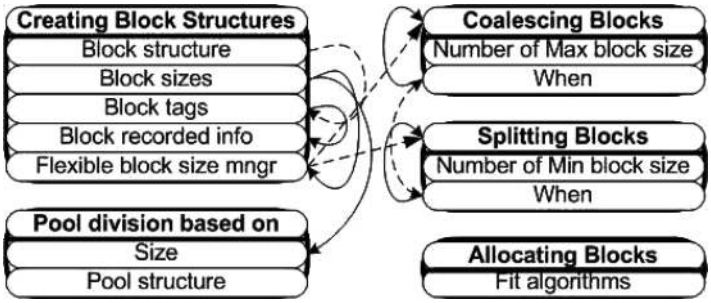

Fig. 2. Interdependencies between the orthogonal trees in the design space.

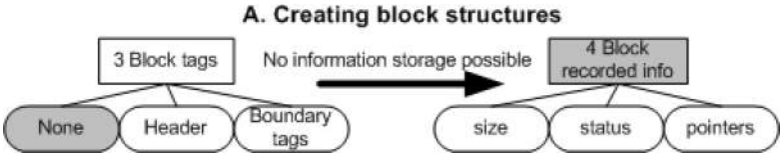

Fig. 3. Example of interdependency between two orthogonal trees of the DM management design space.

arrows indicate that the side without arrow ending affects the other one and must be decided first.

3.2.1 *Leaves or Trees that Obstruct the Use of Others in the New Design Space.* These interdependencies appear due to the existence of opposite leaves and trees in the design space. They are depicted in Figure 2 and are represented by *full* arcs:

—First, inside the Creating block structures category, if the *none* leaf from the *Block tags* tree is selected, then the *Block recorded info* tree cannot be used. Clearly, there would be no memory space to store the recorded info inside the block. This interdependency is graphically shown as example in Figure 3.

—Second, the *one* leaf from the Block sizes tree excludes the use of the *Size* tree in the Pool division based on criterion category. In a similar way, the *one* leaf from the Block sizes tree, excludes the use of the *Flexible block size manager* tree in the Creating block structures category. This occurs because the *one block size* leaf does not allow us to define any new block size.

3.2.2 *Leaves or Trees that Limit the Use of Others in the New Design Space.* These interdependencies exist since the leaves have to be combined to create consistent whole DM schemes. For example, the coalescing and splitting mechanisms are quite related and the decisions in one category have to find equivalent ones in the other one. These interdependencies are represented with *dashed* arcs in Figure 2.

—First, the *Flexible block size manager* tree heavily influences all the trees inside the Coalescing Blocks and the Splitting Blocks categories. Thus, according to the selected leaf for a certain atomic DM manager (i.e., the *split*

or *coalesce* leaf), the DM manager has to select some leaves of the trees involved in those decisions or not. For example, if the *split* leaf is chosen, the DM manager will not use the coalescing category and the functions of their corresponding trees inside that atomic the DM manager. However, it can be used in another atomic DM manager in a different DM allocation phase, thus the final global manager will contain both (see Section 3.3 for more details).

However, the main cost of the selection done in the *Flexible block size manager* tree is characterized by the cost of the Coalescing Blocks and Splitting Blocks categories. This means that a coarse grain estimator of their cost influence must be available to take the decision in the aforementioned *Flexible block size manager* tree. In fact, this estimator is necessary whether the derived decisions in other categories have a large influence in the final cost, which is not smaller than the influence of the tree that decides to use them or not [Catthoor and Brockmeyer 2000]. This does not happen in the tree ordering and then no estimator is required.

—Second, the decision taken in the *Pool structure* tree significantly affects the whole Pool division based on criterion category. This happens because some data structures limit or do not allow the pool to be divided in the complex ways that the criteria of this category suggest.

—Third, the *Block structures* tree inside the Creating block structures category strongly influences the decision in the *Block tags* tree of the same category because certain data structures require extra fields for their maintenance. For example, single-linked lists require a *next* field and a list where several blocks sizes are allowed has to include a *header* field with the size of each free block inside. Otherwise, the cost to traverse the list and find a suitable block for the requested allocation size is excessive [Daylight et al. 2004].

—Finally, the respective *When* trees from the Splitting and Coalescing Blocks categories are linked together with a double arrow in Figure 3 because they are very tightly related to each other and a different decision in each of these two trees does not seems to provide any kind of benefit to the final solution. On the contrary, according to our study and experimental results it usually increases the cost in memory footprint of the final DM manager solution. However, this double arrow is needed because it is not possible to decide which category has more influence in the final solution without studying the specific factors of influence for a certain metric to optimize (e.g., memory footprint, power consumption, performance, etc.). Thus, the decision about which category should be decided first has to be analyzed for each particular cost function or required metric to optimize in the system.

3.3 Construction of Global Dynamic Memory Managers

Modern multimedia and wireless network applications include different DM behavior patterns, which are linked to their logical phases (see Section 6 for real-life examples). 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. To this end, we have

developed a C++ library based on abstract classes or templates that covers all the possible decisions in our DM design space and enables the construction of the final global custom DM manager implementation in a simple way via composition of C++ layers [Atienza et al. 2004a].

4. ORDER FOR REDUCED DYNAMIC MEMORY FOOTPRINT IN DYNAMIC MULTIMEDIA AND WIRELESS NETWORK APPLICATIONS

Once the whole design space for DM management in dynamic embedded systems has been defined and categorized in the previous section, the order for different types of applications can be defined according to their DM behavior and the cost function/s to be optimized. In this case, the DM subsystem is optimized to achieve solutions with a reduced DM footprint. Therefore, first, in Section 4.1, we summarize the factors of influence for DM footprint. Then, in Section 4.2, we briefly describe the features that allow us to group different dynamic applications and focus on the common (and particular) features of new multimedia and wireless network applications, which enable to cluster these applications and define of a common exploration order of the design space. Finally, in Section 4.3, we present the suitable exploration order for these multimedia and wireless network application to attain reduced memory footprint DM management solutions.

4.1 Factors of Influence for Dynamic Memory Footprint Exploration

The main factors that affect memory size are two: the *Organization overhead* and the *Fragmentation memory waste*.

(1) The *Organization overhead* is the overhead produced by the assisting fields and data structures, which accompany each block and pool respectively. This organization is essential to allocate, deallocate and use the memory blocks inside the pools, and depends on the following parts:
   —The fields (e.g., headers, footers, etc.) inside the memory blocks are used to store data regarding the specific block and are usually a few bytes long. The use of these fields is controlled by category A (Creating block structures) in the design space.
   —The assisting data structures provide the infrastructure to organize the pool and to characterize its behavior. 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). Note that the assisting data structures themselves help to prevent fragmentation, but implicitly produce some overhead. The overhead they produce can be comparable to the fragmentation produced by small data structures. Nonetheless this negative factor is overcome by their main feature of preventing fragmentation problems, which is a more relevant negative factor for memory footprint [Wilson et al. 1995]. The same effect on fragmentation prevention is also present in tree C1. This happens because depending on the fit algorithm chosen, you can reduce the internal fragmentation of the system. For example, if you allocate a 12-byte block

using a next fit algorithm and the next block inside the pool is 100 bytes, you loose 88 bytes in internal fragmentation while a best fit algorithm will probably never use that block for the request. Therefore, in terms of preventing fragmentation, category C is equally important to B.

(2) The *Fragmentation memory waste* is caused by the internal and external fragmentation, discussed earlier in this article, which depends on the following:
   —The internal fragmentation is mostly remedied by category E (Splitting blocks). This mostly affects to small data structures. For example, if you have only 100-byte blocks inside your pools and you want to allocate 20-byte blocks, it would be wise to split each 100-byte block inside your pool to 5 blocks of 20 bytes to avoid too much internal fragmentation.
   —The external fragmentation is mostly remedied by category D (Coalescing blocks). It mostly affects to big data requests. For example, if you want to allocate a 50-Kbyte block, but you only have 500-byte blocks inside your 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 Analysis of De/Allocation Characteristics of Dynamic Embedded Multimedia and Wireless Network Applications

Current dynamic multimedia and wireless network embedded applications involve several de/allocation phases (or patterns) for their data structures, which usually represent different phases in the logical. We have classified these different DM allocation patterns [Wilson et al. 1995] in three orthogonal components, namely, *Predominant allocation block size*, *Main pattern* and *Proportion of intensive memory de/allocation phases*.

As we have empirically observed, using this previous classification based on component, new dynamic multimedia and wireless network applications share the main features regarding DM footprint. First, the predominant allocation block sizes are few (as our case studies indicate, 6 or 7 can account for 70–80% of the total allocation requests), but sizes with a large variation since some of them can be just few bytes and the others several Kbytes per allocation. Therefore, DM managers suitable for only large or very small sizes are not suitable, and combinations of solutions for both types of allocations are required in these applications. Second, the main pattern component that defines the dominant pattern of de/allocations, for example, ramp, peaks, plateaus, (see Wilson et al. [1995] for more details) indicates that very frequently new dynamic multimedia and wireless network applications possess very active phases where few data structures are dynamically allocated in a very fast and variable way (i.e., peaks) while others data structures grow continuously (i.e., ramps) and remain stable at a certain point for a long-time before they are freed from the system. According to our observations, this DM phases of creation/destruction of sets of data structures with certain allocation sizes are mostly influenced by the logical structure of phases defined by the designers in their application codes (e.g.,

rendering phases) or special event (e.g., arrival of packets in wireless network applications). Third, the proportion of intensive memory de/allocation phases defines how often the structure of the DM manager has to be changed to fit the new de/allocation sizes and pattern of the dynamic application. In this case, we have also observed very similar features in both new dynamic multimedia and wireless network applications because the run-time behavior of the system tends to follow a predefined order (i.e., defined by the designer) about how to handle the different phases and events. For example, in 3D games the sequence to service new events, as the movement of the camera by the players (e.g., update of the visible objects, rendering of the new background, etc.) is always fixed depending on the type of object. Similarly, it is fixed the way to handle the arrival of new packets in wireless network applications. Thus, all these previous features allow us to cluster these two initially different fields of applications considering their common DM behaviors and define a common order to traverse the design space.

4.3 Order of the Trees for Reduced Memory Footprint in Dynamic Multimedia and Wireless Network Applications

In this Section, we discuss the global order inside the orthogonal categories of our DM management design space according to the aforementioned factors of influence for a reduced memory footprint for dynamic multimedia and wireless network applications. We have defined this global order after extensive testing (i.e., more than 6 months of experiments with different orders and implementations of DM managers) using a representative set of 10 applications of new real-life dynamic embedded multimedia and wireless network applications with different code sizes (i.e., from 1000 lines to 700K lines of C++ code), including: scalable 3D rendering such as [Luebke et al. 2002] or MPEG 4 Visual Texture Coder (VTC) [MPEG-4], 3D image reconstruction algorithms [Pollefeys et al. 1998], 3D games [Quake; MobyGames] and buffering, scheduling and routing network applications [Memik et al. 2001].

Experience suggests that most of the times fragmentation cannot be avoided only with a convenient pool organization [Wilson et al. 1995]. Our experiments show that this holds especially true for embedded dynamic multimedia and wireless network applications. Therefore, categories D and E are placed in the order before categories C and B. The final order is as follows: the tree A2 is placed first to determine if one or more block sizes are used and A5 are placed second to decide the global structure of the blocks. Next, categories that deal with fragmentation, that is, categories D and E, are decided because, as we have mentioned, they are more important than categories that try to prevent fragmentation (i.e., C and B). Then, the rest of the organization overhead must be decided for these block requests. Thus, the rest of the trees in category A (i.e., A1, A3 and A4) are decided. As a result, taking into account the interdependencies, the final global order is the following: 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 the next decision trees. Hence, the most suitable decisions

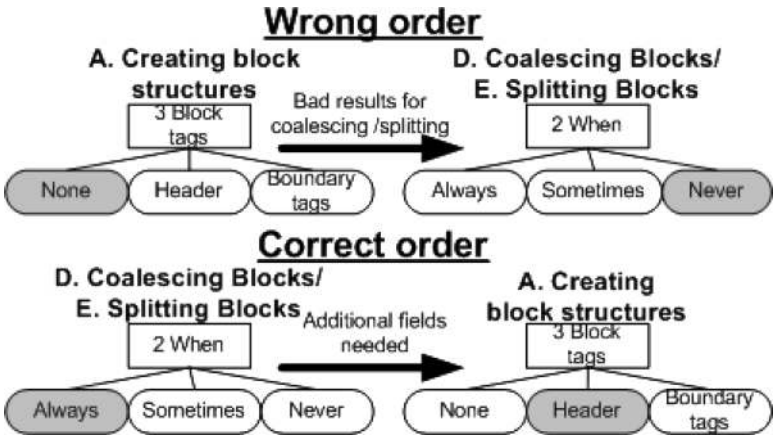

Fig. 4. Example of correct order between two orthogonal trees.

cannot be taken in the remaining orthogonal trees. An example of this is shown in Figure 4. Suppose that the order was A3 and then E2 and D2. When deciding the correct leaf for A3, the obvious choice to save memory footprint would be to choose the *None* leaf, which indicates that no header fields of any type should be used. This seems reasonable at first sight because the header fields would require a fixed amount of additional memory for each block that is going to be allocated. After the decision about the *None* leaf in tree A3, the leaves to use for the trees E2 and D2 are decided. Now, we are obliged to choose the *Never* leaf because after propagating the constraints of A3, one block cannot be properly split or coalesced 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. This solution could be seen as an appropriate decision for an application where the predominant block size is fixed and thus no serious effects exist due to internal or external fragmentation. However, if the application includes a variable amount of allocation sizes (typical behavior in many current dynamic multimedia and wireless network applications), the fragmentation problem (internal and external) is going to consume more memory than the extra header fields needed for coalescing and splitting since the freed blocks will not be able to be reused. Therefore, it is necessary to decide the E2 and D2 trees first, and then propagate the resulting constraints to tree A3.

To demonstrate the correctness of this explanation, in the following paragraphs the experimental results of memory footprint for a multimedia application with a varying predominant block size that uses each of the possible orders are shown, that is, A 3D algorithm with scalable meshes [Luebke et al. 2002] that is later described more in detail in our case studies and experimental results (Section 6). The results shown are average values after 10 runs for each experiments and groups of 10 bursts. First of all, the application is divided in several bursts of 1000 (de)allocations with a limited amount of blocks (10 different sizes that range from 20 bytes to 500 bytes). This is a typical way in which a multimedia application with several phases can run. In network applications,

Table I. Example of Two DM Managers with Different Order in the DM Design Space

DM Managers	Memory Footprint (KBytes)	Execution Time (secs)
(1) A3-D2/E2	2.392×10^2	7.005
(2) D2/E2-A3	4.682×10^1	11.687
Comparison 2–1	19.56%	166.83%

the range of sizes can vary even more due to the uncertainty in the workload and input packets. We have implemented two different versions of the same DM manager, which include a single pool where both the free and used blocks are doubly linked inside one list. The only difference between these two DM managers is the aforementioned explained order between A3 and D2/E2.

In the first solution, A3 is decided and then D2 and E2. Hence, no coalescing and splitting services are provided in the DM manager, but also less overhead in the blocks is required. In the second solution D2 and E2 are decided first and then A3 is determined. Thus, this DM manager includes both services (i.e., coalescing and splitting), but includes an overhead of 7 bytes per block in the list as additional header for maintenance: 2 bytes for the size, each link 2 bytes (2 required) and 1 byte (i.e., a bool field) to decide if it is used. The results obtained in both cases are shown in Table I. It shows clearly that the overhead of the additional headers is less significant than the overhead due to fragmentation, where 5 times less memory is required (see line `Comparison 2–1` in Table I). Therefore, as we propose in our global exploration order for reduced memory footprint (see Section 4.3 for more details), the second DM manager produces better results. In this second manager, first the D2 and E2 trees are decided and then their constraints are used to select the A3 tree leaf. Also note the difference in execution time: the first DM manager has a very low total execution time because it does not provide coalescing or splitting services, while the second one has an additional overhead in execution time of 66.83%. This shows that the performance requirement of the system under development must be taken into account when the choices in the design space are decided. Hence, if the system designer requires a certain level of performance, which is not achieved by the extreme reduced memory footprint solution shown in number 2, another solution closer to solution number 1 (but with a more balanced trade-off between memory footprint and performance) can be designed. As it has been mentioned before, we have performed a large amount of similar experiments for the rest decision orders and derived the proposed one. The main conclusion from our studies is that with the proposed global order the exploration methodology is not unnecessarily limited and only the necessary constraints are forwarded to the next decision trees, which is not possible with other orders for these types of new dynamic applications.

5. OVERVIEW OF THE GLOBAL FLOW IN THE DYNAMIC MEMORY MANAGEMENT METHODOLOGY

The main objective of our proposed DM management design methodology is to provide developers with a complete design flow of custom DM managers for

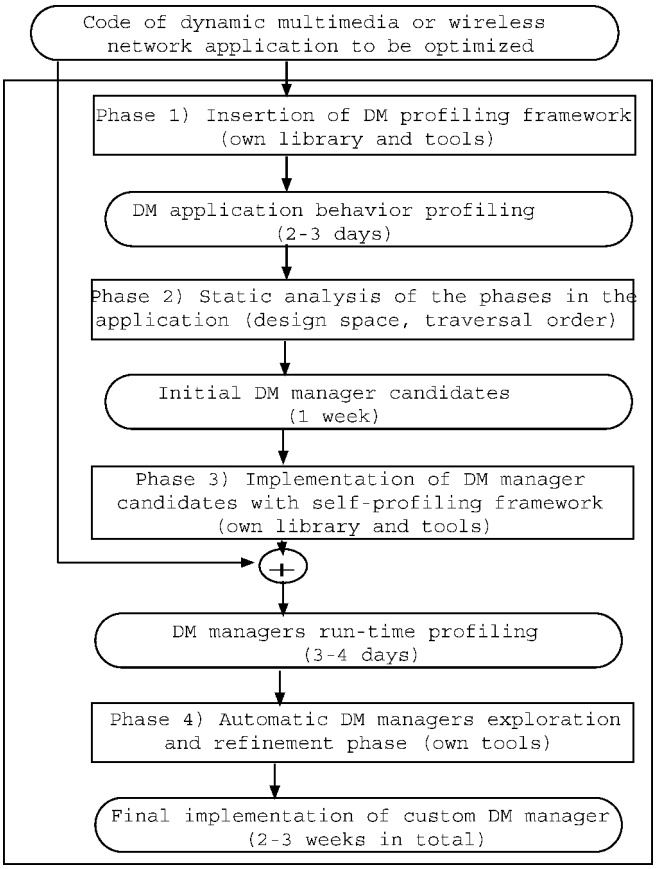

Fig. 5. Global overview of our proposed DM management design methodology flow.

new dynamic multimedia and wireless network systems. Our proposed global design flow is divided into four main phases, as indicated in Figure 5.

The first phase of our methodology is used to obtain detailed information about the DM sizes requested and the time when each dynamic de/allocation is performed, as depicted in the first oval shape of Figure 5. This phase is based on profiling the use of DM in the considered application because, according to our experience, it is the only realistic option for today's highly-dynamic multimedia and wireless network applications [Leeman et al. 2003; Atienza et al. 2004a]. Profiling is carried out as follows. First, we insert in the code of the application our own high-level profiling framework, which is based on a C++ library of profiling objects and additional graphical parsing tools [Leeman et al. 2003]. Then, we automatically profile a representative set of input cases of the application, in general between 10 and 15 different inputs, including the extreme cases of least and maximum memory footprint. This small set is provided by the designers of the application since they use it to create their dynamic data structures to store and retrieve the data in the application (e.g., dynamic arrays, doubly linked lists, binary trees, etc.) [Daylight et al. 2004]. As a result, this first phase

takes between 2-3 days in our methodology for real-life applications thanks to our tools with very limited user interaction.

Then, in the second phase of our DM management methodology, indicated as the second rectangular shape of Figure 5, the DM behavior of the application is characterized. The de/allocation phases are identified and linked to the logical phases of the application (e.g., rasterization, clipping, illumination, etc. in 3D rendering [Woo et al. 1997]) using our tools to analyze the profiling information obtained in the previous phase. These tools identify in different graphs and tables the main variations in memory footprint and the dynamically de/allocated sizes. Then, using our design space for DM management and the exploration order for dynamic multimedia and wireless network applications, proposed in Section 3 and Section 4, respectively, suitable sets of custom DM manager candidates are selected for each application. This second phase can take up to one week for very complex applications, namely with more than 20 de/allocation phases in the same application.

Next, in the third phase of our DM management methodology we implement the DM manager candidates and exhaustively explore their run-time behavior in the application under study, such as memory footprint, fragmentation, performance and energy consumption. For this purpose, as we have explained in Section 3.3, we have developed an optimized `C++` library that covers all the decisions in our DM design space and enables the construction and profiling of our custom DM managers implementations in a simple way via composition of `C++` layers [Atienza et al. 2004a]. As a result, the whole implementation space exploration is done automatically using the aforementioned library and additional tools, which execute multiple times the application with the different DM managers to acquire complete run-time profiling information. This third phase takes between 3 to 4 days in case of complex applications.

In the fourth and final phase of our DM management design flow, we use our own tools to automatically evaluate the profiling generated in the previous phase and to determine the final implementation features of the custom DM manager for the application, such as final number of max block sizes, number of allowed allocation memory sizes. This fourth phase takes between 2 to 3 days. Overall, the whole flow for custom DM managers requires between 2 to 3 weeks for real-life applications.

6. CASE STUDIES AND EXPERIMENTAL RESULTS

We have applied the proposed methodology to three realistic case studies that represent different modern multimedia and network application domains: the first case study is a scheduling algorithm from the network protocol 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.

In the following sections, we briefly describe the behavior of the three case studies and the proposed methodology is applied to design custom managers that minimize their memory footprint. All the results shown are average values after a set of 10 simulations for each application and manager implementation using 10 additional real inputs from the ones used to profile and design the

custom DM managers. All the final values in the simulations were very similar (variations of less than 2%).

6.1 Method Applied to a Network Scheduling Application

The first case study presented is the Deficit Round Robin (DRR) application taken from the NetBench benchmarking suite [Memik et al. 2001]. It is a scheduling algorithm implemented in many routers today. In fact, variations of the DRR scheduling algorithm are used by Cisco for commercial acess points products and by Infineon in its new broadband access devices. Using the DRR algorithm the router tries to accomplish a fair scheduling by allowing the same amount of data to be passed and sent from each internal queue. In the DRR algorithm, the scheduler visits each internal nonempty queue, increments the variable deficit by the value quantum and determines the number of bytes in the packet at the head of the queue. If the variable deficit is less than the size of the packet at the head of the queue (i.e., it does not have enough credits at this moment), then the scheduler moves on to service the next queue. If the size of the packet at the head of the queue is less than or equal to the variable deficit, then the variable deficit is reduced by the number of bytes in the packet and the packet is transmitted on the output port. The scheduler continues this process, starting from the first queue each time a packet is transmitted. If a queue has no more packets, it is destroyed. The arriving packets are queued to the appropriate node and if no such exists then it is created. 10 real traces of internet network traffic up to 10 Mbit/sec have been used [Lawrence Berkeley Lab 2000] to run realistic simulations of DRR.

To create our custom DM Manager, we have followed our methodology step by step. As a result, in order to define the logical phases of the application and its atomic DM manager, we first profile its DM behavior with our dynamic data structures profiling approach [Leeman et al. 2003]. Then, we can apply our DM management design exploration. First, we make the decision in tree A2 (Block sizes) to have `many` block sizes to prevent internal fragmentation. This is done because the memory blocks requested by the DRR application vary greatly in size (to store packets of different sizes) and if only one block size is used for all the different block sizes requested, the internal fragmentation would be large. Then, in tree A5 (Flexible block size manager) we choose to `split` or `coalesce`, so that every time a memory block with a bigger or smaller size than the current block is requested, the splitting and coalescing mechanisms are invoked. In 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/min block size) we choose `many` (categories) and `not fixed` because we want to get the maximum effect out of coalescing and splitting mechanisms by not limiting the size of these new blocks. After this, in trees B1 (Pool division based on size) and B2 (Pool structure), the simplest pool implementation possible is selected, which is a single pool, because if we do not have fixed block sizes, then no real use exists for complex pool structures to achieve a reduced memory footprint. Then, in tree C1 (Fit algorithms), we choose the `exact fit` to avoid as much as possible memory loses in internal fragmentation. Next, in tree A1 (Block structure),

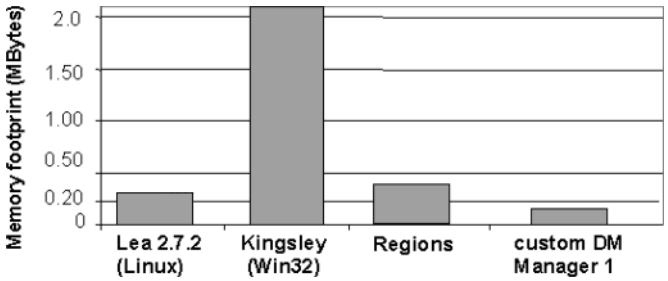

Fig. 6. Maximum memory footprint results in the DRR application.

we choose the most simple DDT that allows coalescing and splitting, which is a `doubly linked list`. Then, in trees A3 (Block tags) and A4 (Block recorded info), we choose a `header` field to accommodate information about the `size` and `status` of each block to support splitting and coalescing mechanisms. Finally, after taking these decisions following the order described in Section 4, according to the proposed design flow we can determine those decisions of the final custom DM manager that depend on its particular run-time behavior in the application (e.g., final number of max block sizes) via simulation with our own customizable `C++` library and tools [Atienza et al. 2004a] (see Section 5 for more details).

Then, we implement it and compare our custom solution to very well-known state-of-the-art optimized general-purpose managers, namely Lea v2.7.2 [Lea 2002] and Kingsley [Wilson et al. 1995]. The Lea allocator is one of the best generic managers (in terms of the combination of speed and memory footprint) [Berger et al. 2001] and several variations of it are integrated in the different distributions of the GNU Linux OS. It is a hybrid DM manager that includes different behaviors for different object sizes. For small objects it uses some kind of quick lists [Wilson et al. 1995], for medium-sized objects it performs approximate best-fit allocation [Wilson et al. 1995] and for large objects it uses dedicated memory (allocated directly with the `mmap()` function). Also, we compare our approach with an optimized version of the Kingsley [Wilson et al. 1995] DM manager that still uses power-of-two segregated-fit lists [Wilson et al. 1995] to achieve fast allocations, but it is optimized for objects larger than 1024 bytes, which take their memory from a sorted linked list, sacrificing speed for good fit. A similar implementation technique is used in the latest Windows-based OSes [Microsoft MSDN (a), (b)].

In addition, we have compared our custom DM solution with a manually designed implementation of the new region-semantic managers [Gay and Aiken 2001] that can be found in new embedded OSes (e.g., RTEMS [2002]).

As Figure 6 shows, our custom DM manager uses less memory than the Lea 2.7.2 (Linux OS), the Kingsley (Windows OS) and Region DM managers. This is due to the fact that our custom DM manager does not have fixed sized blocks and tries to coalesce and split as much as possible, which is a better option in dynamic applications with sizes with large variation. Moreover, when large coalesced chunks of memory are not used, they are returned back to the system for other applications. On the other hand, Lea and Kingsley create huge freelists of unused blocks (in case they are reused later), they coalesce and split seldom

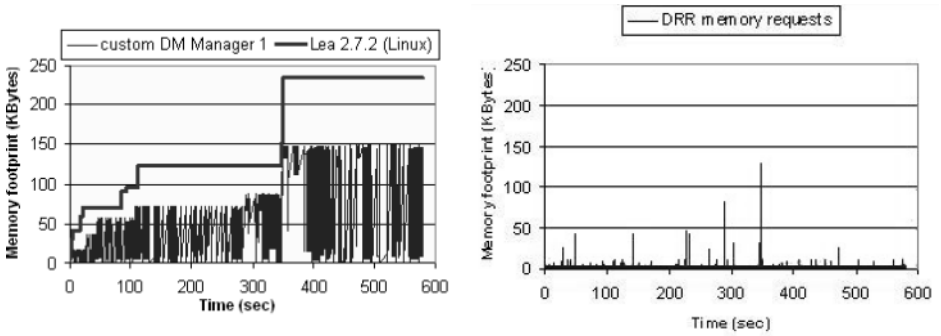

Fig. 7. Memory footprint behavior of the Lea DM manager vs our custom DM manager (left graph), and DM allocation requests in the DRR application (right side).

(Lea) or never (Kingsley) and finally, they have fixed sized block sizes. This can be observed in Figure 7, where we show the DM use graphs of our custom manager, the Lea DM manager and the DM requests of the application during one run. In the case of the Region DM manager, since no coalescing/splitting mechanisms are applied to reuse memory blocks, more memory footprint than Lea is consumed due to fragmentation.

Concerning the performance of the various DM managers (see Table II further on for a summary with the results of all our case studies), we can observe that in all cases 600 seconds is the time used to schedule real traces of 600 seconds of internet traffic. This means that all the DM managers studied fulfill the real-time requirements of the DRR application. Moreover, our DM manager improves extensively the memory footprint used by the other DM managers, but no extra time overhead is noticed due to its internal maintenance operations.

Finally, we have evaluated the total energy consumption of the final embedded system with each of the studied DM managers, using a cycle-accurate ARM-based simulator that includes a complete energy-delay estimation model for 0.13 um [Loghi et al. 2004]. The results indicate that our custom DM achieves very good results for energy, when compared to the state-of-the-art DM managers. This is due to the fact that most of the dynamic accesses performed internally by the managers to their complex management structures are not required in our custom manager, which uses a simpler and optimized internal data structures for the target application. Thus, our custom DM manager reduces by 12%, 15% and 16% the energy consumption values of Kingsley, Lea and Regions, respectively. It is important to mention that even though Kingsley has a smaller amount of DM management accesses, since it does not perform splitting or coalescing operations, it suffers from a large memory footprint penalty. This translates into very expensive memory accesses because bigger memories need to be used.

Consequently, for new dynamic network applications like the DRR application, our methodology allows to design very customized DM managers that exhibit less fragmentation than Lea, Regions or Kingsley and thus require less memory. Moreover, since this decrease in memory footprint is combined with

a simpler internal management of DM, the final system consumes less energy as well.

6.2 Methodology Applied to a New 3D Image Reconstruction System

The second case study forms one of the sub-algorithms of a 3D reconstruction application [Pollefeys et al. 1998] that works like 3D perception in living beings, where the relative displacement between several 2D projections is used to reconstruct the 3rd dimension. The software module used as our driver application is one of the basic building blocks in many current 3D vision algorithms: *feature selection and matching*. It has been extracted from the original code of the 3D image reconstruction system (see Target Jr [2002] for the full code of the algorithm with 1.75 million lines of high level C++), and creates the mathematical abstraction from the related frames that is used in the global algorithm. It still involves 600,000 lines of C++ code, which demonstrates the complexity of the applications that we can deal with our approach and the necessity of tool support for the analysis and code generation/exploration phases in our overall approach (see Section 5). This implementation matches corners [Pollefeys et al. 1998] detected in 2 subsequent frames and heavily relies on DM due to the unpredictability of the features of input images at compile-time (e.g., number of possible corners to match varies on each image). Furthermore, the operations done on the images are particularly memory intensive, that is, each image with a resolution of 640×480 uses over 1Mb. Therefore, the DM overhead (e.g., internal and external fragmentation [Wilson et al. 1995]) of this application must be minimized to be usable for embedded devices where more applications are running concurrently. Finally, note that the accesses of the algorithm to the images are randomized, thus classic image access optimizations as row-dominated accesses versus column-wise accesses are not relevant to reduce the DM footprint further.

For this case study, its dynamic behavior shows that only a very limited range of data type sizes are used in it [Leeman et al. 2003], namely 8 different allocation sizes are requested. In addition, most of these allocated sizes are relatively small (i.e., between 32 or 16384 Bytes) and only very few blocks are much bigger (e.g., 163 KBytes). Furthermore, we see that most of the data types interact with each other and are alive almost all the execution time of the application. Within this context, we apply our methodology and using the order provided in Section 4 we try to minimize the DM footprint wastage (e.g., fragmentation, overhead in the headers, etc.) of this application. As a result, we obtain a final solution that consists of a custom DM manager with 4 separated pools or regions for the relevant sizes in the application. The first pool is used for the smallest allocation size requested in the application, that is, 32 bytes. The second pool allows allocations of sizes between 756 bytes and 1024 bytes. Then, the third pool is used for allocation requests of 16384 bytes. Finally, the fourth pool is used for big allocation requests blocks (e.g., 163 or 265 KBytes). The pool for the smallest size has its blocks in a single linked list because it does not need to coalesce or split since only one block size can be requested in it. The rest of the pools include doubly linked lists of free blocks with headers that contain the size

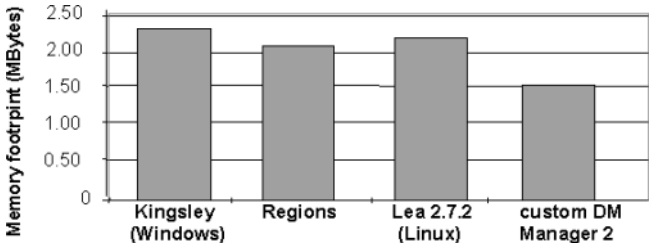

Fig. 8. Maximum memory footprint results in the 3D reconstruction application.

of each respective block and information about their current state (i.e., in use or free). These mechanisms efficiently support immediate coalescing and splitting inside these pools, which minimizes both internal and external fragmentation in the custom DM manager designed with our methodology.

In this case study we have compared our solution with new region-semantic managers [Gay and Aiken 2001; Wilson et al. 1995]. Also, we have tested our manager with the same optimized versions of Kingsley and Lea used in the previous example (i.e., DRR) since they are the types of DM managers found in embedded systems using Windows or Linux-based OSes, respectively. The memory footprint results obtained are depicted in Figure 8.

These results show that the values obtained with the DM manager designed using the proposed methodology obtains significant improvements in memory footprint compared to the manually designed implementation of a Region manager (28.47%), Lea (29.46%) and the optimized version of Kingsley (33.01%). These results are because our custom DM manager is able to minimize the fragmentation of the system in two ways. First, because its design and behavior varies according to the different block sizes requested. Second, in pools where a range of block sizes requests are allowed, it uses immediate coalescing and splitting services to reduce both internal and external fragmentation. In the new region managers, the blocks sizes of each different region are fixed to one block size and when blocks of several sizes are used, this creates internal fragmentation. In Lea, the range of block sizes allowed do not fit exactly the ones used in the applications and the mixed de/allocation sizes of few block sizes produce a waste of the lists of sizes not used in the system. Furthermore, the frequent use of splitting/coalescing mechanisms in Lea creates an additional overhead in execution time compared to the Region manager, which has better adjusted allocation sizes to those used in the application. Finally, in Kingsley, the coalescing/splitting mechanisms are applied, but an initial boundary memory is reserved and distributed among the different lists for sizes. In this case, since only a limited amount of sizes is used, some of the "bins" (or pools of DM blocks in Kingsley) [Wilson et al. 1995] are underused.

In addition, the final embedded system implementation using our custom DM manager achieves better energy results than the implementations using general-purpose DM managers. In this case study, our DM manager employs less management accesses to DM blocks and memory footprint than any other manager. Thus, our DM manager enables overall energy consumption savings of 10% with respect to Regions, 11% over Kingsley and 14% over Lea.

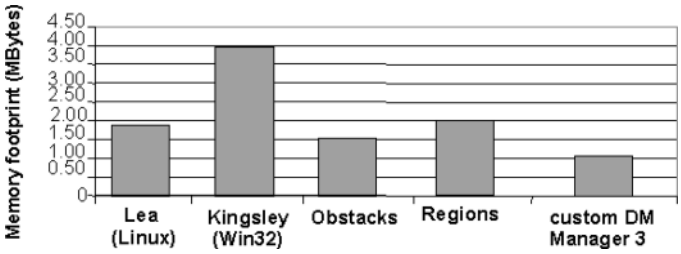

Fig. 9. Maximum memory footprint results in the 3D rendering application.

6.3 Methodology Applied to a 3D Rendering System

The third case study is the 3D rendering module [Woo et al. 1997] of a whole 3D video system application. This module belongs to the new category of 3D algorithms with scalable meshes [Luebke et al. 2002] that adapt the quality of each object displayed on the screen according to the position of the user watching at them at each moment in time (e.g., Quality of Service systems [Pham Ngoc et al. 2002]). Therefore, the objects are represented by vertices and faces (or triangles) that need to be dynamically stored due to the uncertainty at compile time of the features of the objects to render (i.e., number and resolution). First, those vertices are traversed in the first three phases of the whole visualization process, that is, modelview transformation, lighting process and canonical view transformation [Woo et al. 1997]. Finally, the system processes the faces of the objects in the next three phases (i.e., clipping, viewport mapping and rasterization [Woo et al. 1997]) of the visualization process to show the final object with the appropriate resolution on the screen. According to our experiments, this application closely resembles the DM behavior of the MPEG4 Visual Texture deCoder (VTC) implementation in the standard [MPEG-4].

In this case, we have compared our custom manager with Lea v2.7.2, the optimized version of Kingsley, the Region manager and due to its particular behavior with phases where intermediate values are built in a sequential way and are finally destroyed at the end of each phase (for the phases that handle vertices), we have also used Obstacks [Wilson et al. 1995]. Obstacks is a well-known custom DM manager optimized for applications with such stack-like behavior. For example, Obstacks is used internally by *gcc*, the GNU Compiler.

As Figure 9 shows, Lea and the Region manager obtain better results in memory footprint than the Kingsley DM manager. Also, due to the stack-like behavior of the application in the phases that handle triangles, Obstacks achieves even better results than Lea and region managers in memory footprint. However, the custom manager designed with our methodology improves further the memory footprint values obtained by Obstacks. The fact is that the optimized behavior of Obstacks cannot be exploited in the final phases of the rendering process because the faces are used all independently in a disordered pattern and they are required to be freed separately. Thus, Obstacks suffers from a high penalty in memory footprint due to fragmentation because it has not got a suitable maintenance structure of the DM blocks for such DM deallocation behavior.

Table II. Maximum Memory Footprint, Execution Time and Energy Consumption Results of Each DM Manager in Our Case Studies

Dynamic Memory Managers	DRR Scheduler	3D Image Reconstruction	3D Scalable Rendering
Kingsley- Windows (Bytes)	2.09×10^6	2.26×10^6	3.96×10^6
execution time (sec.)	600	1.52	6.05
energy consumption (nJ)	7.98×10^9	4.04×10^6	11.22×10^6
Lea-Linux (Bytes)	2.34×10^5	$2.11 \times$ lO^6	1.86×10^6
execution time (sec.)	600	2.01	8.12
energy consumption (nJ)	8.18×10^9	4.11×10^6	11.43×10^6
Regions (Bytes)	2.47×10^5	2.08×10^6	2.03×10^6
execution time (sec.)	600	1.87	7.03
energy consumption (nJ)	8.25×10^9	$4.01 \times$ lO^6	$11.61 \times$ lO^6
Obstacks (Bytes)	—	—	1.55×10^6
execution time (sec.)	—	—	6.55
energy consumption (nJ)	—	—	10.93×10^6
our DM manager (Bytes)	1.48×10^5	1.49×10^6	1.07×10^6
execution time (secs.)	600	1.24	6.91
energy consumption (nJ)	7.11×10^9	3.52×10^6	9.67×10^6

From an energy point of view, our custom DM manager also improves the results obtained with the studied general-purpose managers in the ARM-based simulator. In a similar way as in the previous case studies, our DM manager requires less memory management accesses than Lea and Regions, thus improving the global energy figures of the final embedded system by 18% 20%, respectively. In the case of Obstacks and Kingsley, they produce less memory accesses than our custom DM manager due their optimized management of DM blocks for performance, but their larger consumption of memory footprint finally enables 13% and 14% overall savings in energy consumption for our custom DM manager, respectively.

Finally, to evaluate the design process with our methodology, our proposed design and implementation flow of the final custom DM managers for each case study took us two weeks. Also, as Table II shows, 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, that is, Kingsley. Moreover, the decrease in performance is not relevant since our custom DM managers preserve the real-time behavior required by the applications. Thus, the user will not notice any difference. In addition, the proposed custom DM managers include optimized internal DM management organizations of memory blocks for each studied application, which usually produce a decrease in memory accesses compared to state-of-the-art general-purpose managers, such as Lea or Regions, that are designed for a wide range of memory requests and DM behavior patterns. Only Kingsley and Obstacks produce less memory accesses than our custom DM manager due to their performance-oriented designs, thus outperforming our custom DM managers but wasting a large part of memory footprint due to fragmentation. As a consequence, these managers demand larger on-chip memories to store the dynamic data and much more energy is required per access [Loghi et al. 2004], which counteracts the possible improvements in energy consumption because

of less memory accesses. In summary, our custom DM managers also reduce by 15% on average the total energy consumption of the final embedded system compared to general-purpose DM managers.

Although our methodology for designing custom DM managers has been driven by the minimization of the memory footprint, it can be perfectly retargeted towards achieving different trade-offs between any relevant design factors, such as improving performance or consuming a little more memory footprint to achieve more energy savings [Atienza et al. 2004b].

7. CONCLUSIONS

Embedded devices have improved their capabilities in the last years making feasible to map very complex and dynamic applications (e.g., multimedia applications) in portable devices. Such applications have grown lately in complexity and to port them to the final embedded systems, new design methodologies must be available to efficiently use the memory present in these very limited embedded systems. In this paper we have presented a new systematic methodology that defines and explores the dynamic memory management design space of relevant decisions, in order to design custom dynamic memory managers with a reduced memory footprint for new dynamic multimedia and wireless network 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

ATIENZA, D., MAMAGKAKIS, S., CATTHOOR, F., MENDIAS, J. M., AND SOUDRIS, D. 2004a. Modular construction and power modelling of dynamic memory managers for embedded systems. In *Proceedings of Workshop PATMOS*. Lecture Notes in Computer Science, vol. 3254, Springer-Verlag, New York.

ATIENZA, D., MAMAGKAKIS, S., CATTHOOR, F., MENDIAS, J. M., AND SOUDRIS, D. 2004b. Reducing memory accesses with a system-level design methodology in customized dynamic memory management. In *Proceedings of IEEE Workshop ESTIMEDIA*. IEEE Computer Society Press, Los Alamitos, CA.

BACON, D. F., CHENG P., AND RAJAN, V. T. 2003. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*. ACM, New York.

BENINI, L. AND DE MICHELI, G. 2000. System level power optimization techniques and tools. In *ACM Trans. Des. Automat. Embed. Syst.*

BERGER, E. D., ZORN, B. G., AND MCKINLEY, K. S. 2001. Composing high-performance memory allocators. In *Proceedings of Conference PLDI*. ACM, New York.

BLACKBURN, S. M. AND MCKINLEY, K. S. 2003. Ulterior reference counting: Fast garbage collection without a long wait. In *Proceedings of Conference OOPSLA*. ACM, New York.

CATTHOOR, F. AND BROCKMEYER, E. 2000. *Unified Low-Power Design Flow for Data-Dominated Multi-Media and Telecom Applications*. Kluwer Academic Publishers.

CHANG, J. M., LO, C.-T. D., AND SRISA-AN, W. 1999. OMX: Object management extension. In *Proceedings of Workshop CASES*, USA.

DAYLIGHT, E., ATIENZA, D., VANDECAPPELLE, A., CATTHOOR, F., AND MENDIAS, J. M. 2004. Memory-access-aware data structure transformations for embedded software with dynamic data accesses. *IEEE Trans. VLSI Syst.* 269–280.

GAY, D. AND AIKEN, A. 2001. Memory management with explicit regions. In *Proceedings of PLDI*. ACM, New York.

LAWRENCE BERKELEY NATIONAL LAB 2000. The Internet Traffic Archive. http://ita.ee.lbl.gov/.

LEA, D. 2002. The Lea 2.7.2 DM Allocator. http://gee.cs.oswego.edu/dl/.

LEEMAN, M., ATIENZA, D., CATTHOOR, F., DECONINCK, G., MENDIAS, J. M., DE FLORIO, V., AND LAUWEREINS, R. 2003. Power estimation approach of dynamic data storage on a hardware software boundary level. In *Proceedings of the Workshop PATMOS*. Lecture Notes in Computer Science, vol. 2799, Springer-Verlag, New York.

LOGHI, M., ANGIOLINI, F., BERTOZZI, D., BENINI, L., AND ZAFALON, R. 2004. Analyzing on-chip communication in a mpsoc environment. In *Proceedings of DATE*. IEEE Computer Society Press, Los Alamitos, CA.

LUEBKE, D., REDDY, M., COHEN, J., VARSHNEY, A., WATSON, B., AND HUEBNER, R. 2002. *Level of Detail for 3D Graphics*. Morgan-Kaufmann Publishers, USA.

MEMIK, G., MANGIONE-SMITH, B., AND HU, W. 2001. Netbench: A benchmarking suite for network processors. CARES Technical Report 2001-2-01.

MICROSOFT MSDN (a) Heap:pleasures and pains (for Windows NT Technologies). http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dngenlib/html/heap3.asp.

MICROSOFT MSDN (b) Heaps in Windows CE. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wcecor% eos5/html/wce50conheaps.asp.

MOBYGAMES. Moby Games, a game docs and review project. http://www.mobygames.com/.

MPEG-4 Implementation Reference. Iso/iec jtc1/sc29/wg11 Mpeg-4 standard features overview. http://www.chiariglione.org/mpeg/standards/mpeg-4/mpeg-4.htm.

MURPHY, N. 2000. Safe memory usage with dynamic memory allocation. *Embedded Systems*.

PANDA, P. R., CATTHOOR, F., DUTT, N. D., DANCKAERT, K., BROCKMEYER, E., AND KULKARNI, C. 2001. Data and memory optimizations for embedded systems. *ACM Trans. Des. Automat. Elect. Syst. 6,* 2, 142–206.

PHAM NGOC, N., VAN RAEMDONCK, W., LAFRUIT, G., DECONINCK, G., AND LAUWEREINS, R. 2002. Qos framework for interactive 3d applications. In *Proceedings of Conference CECGVC*.

POLLEFEYS, M., KOCH, R., VERGAUWEN, M., AND VAN GOOL, L. 1998. Metric 3D surface reconstruction from uncalibrated image sequences. In Lecture Notes in Computer Science, vol. 1506, Springer-Verlag, New York.

QUAKE, H. Handheld quake. http://handheldquake.sourceforge.net/.

RTEMS RESEARCH, O.-L. A. 2002. RTEMS, open-source real-time operating system for multiprocessor systems. http://www.rtems.org.

SHALAN, M. AND MOONEY V. J. II, 2000. A dynamic memory management unit for embedded real-time system-on-a-chip. In *Proceedings of Workshop CASES*.

SRISA-AN, W., DAN LO, C.-T., AND CHANG, J. M. 2003. Active memory processor: A hw garbage collector for real-time java embedded devices. *IEEE Trans. Mobile Comput. 2*, 89–101.

Target Jr 2002. Target jr. http://computing.ee.ethz.ch/sepp/targetjr-5.0b-mo.html.

VO, K.-P. 1996. Vmalloc: A general and efficient memory allocator. *Softw. Pract. Exp. 26*, 1–18.

WILSON, P. R., JOHNSTONE, M. S., NEELY, M., AND BOWLES, D. 1995. Dynamic storage allocation, a survey and critical review. In *Proceedings of the Workshop on Memory Management*, Lecture Notes in Computer Science, Springer-Verlag, New York.

WOO, M., NEIDER, J., DAVIS, T., AND SHREINER, D. 1997. *OpenGL Programming Guide, Second Edition*. Silicon Graphics, Inc.

Received March 2004; revised November 2004 and July 2005; accepted August 2005