Reducing Memory Accesses with a System-Level Design Methodology in Customized Dynamic Memory Management

David Atienza*, Stylianos Mamagkakis†, Francky Catthoor‡, Jose M. Mendias*, Dimitrios Soudris†
* DACYA/UCM Avda. Complutense s/n, 28040, Madrid, Spain.
† VLSI Center-Demokritus Univ., Thrace, 67100 Xanthi, Greece.
‡ IMEC, Kapeldreef 75, 3001 Heverlee, Belgium.

***Abstract*— Currently, portable consumer embedded devices are increasing more and more their capabilities and can now implement new algorithms (e.g. multimedia and wireless protocols) that a few years ago were reserved only for powerful workstations. Unfortunately, the original design characteristics of such applications do not often allow to port them directly in current embedded devices. These applications share complex and intensive memory use. Furthermore, they must heavily rely on dynamic memory 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). Thus they require that the dynamic memory subsystem involved is able to provide the necessary level of performance for these new dynamic applications. However, actual embedded systems have very limited resources (e.g. speed and power consumed in the memory subsystem) to provide efficient general-purpose dynamic memory management. In this paper we propose a new methodology to design custom dynamic memory managers that provide the performance required in new embedded devices by reducing the amount of memory accesses to handle these new dynamic multimedia and wireless network applications. Our results in real-life dynamic applications show significant improvements in memory accesses of dynamic memory managers , i.e. up to 58%, compared to state-of-the-art dynamic memory management solutions for complex applications.**

I. Introduction

Over the last decade, the differences between the applications to be executed on top line Digital Signal Processors (DSPs) and portable devices have been fading. The new complex applications (e.g. network protocols, 3D games, MPEG4 video rendering) introduced in DSPs designed exclusively for performance are now implemented on hand-held devices where memory resources and performance are much more limited. Therefore extensive (and very time-consuming) refinements must be applied to the initial design and implementation of these new applications in order to be suitably ported to current embedded consumer devices.

One of the fundamental characteristics of these new and complex multimedia applications (e.g. MPEG21) or wireless network protocols is that they extensively employ Dynamic Memory (or DM from now on) for their operations [9]. A high variation in the use of resources at each moment in time exists because of the strong internal dynamism and the unpredictable input data (e.g. input frames) of such applications. Thus a (static) worst-case memory footprint of these new applications would lead to an unacceptable overhead in memory footprint and memory bandwidth requirements for current embedded systems [2]. On the other hand, dynamic solutions can de-allocate memory before the end of the scope and re-allocate it at run time as needed. Therefore, DM solutions can continue to work in almost any case, because they only allocate memory for the dynamic data types [2], which are actually present. Hence, DM management should be used in cost- and failure-sensitive realisations of such embedded applications.

When DM management solutions are used in new consumer embedded devices, general-purpose DM management mechanisms [16] cannot be directly implemented because they imply too much overhead (e.g. memory accesses) for the limited resources of embedded devices. Therefore, in recently proposed real-time OSs for embedded systems [13], DM management is supported with specifically designed (or custom) DM managers considering the specific set of applications that will run on them and the underlying memory hierarchy. Note that they are still realised in the middleware and usually not in the hardware. However, flexible system-level approaches that can guide designers to explore the DM management design space for embedded systems do not presently exist. Thus, custom DM managers for embedded systems are designed by developers relying mainly on their own experience and intuition. As a result, only a very limited number of design alternatives are considered due to the very time-consuming effort of manually implementing and refining them.

In this paper, we propose a new methodology that allows developers to effectively define and create custom DM management solutions with the required performance for current embedded devices. This is done by reducing the amount of memory accesses of the data used inside the DM management subsystem to handle new dynamic multimedia and wireless network applications. It is based on a systematic definition and guided exploration of the DM management design space for embedded systems according to their particular DM behaviour. In contrast with our recent work [2] that focused on reduced memory footprint, we propose new factors of influence and a new order for the exploration of the relevant design space for reduced amount of memory accesses and an enhanced performance. The factors that influence our decisions for the formalization of the methodology are different and thus we design different custom DM managers. In the end, using our own DM libraries [1] we can automate the creation of the proposed customized DM managers. The rest of the paper is organized in the following way. In Section II, we describe some related work. In Section III we present our DM management design space of decisions for reduced amount of memory accesses in dynamic applications. In Section IV we define the order to traverse it, in order to achieve the required performance for current embedded systems. Later, in Section V, we introduce our real-life applications and present the experimental results obtained. Finally, in Section VI we conclude the paper with a summary of our main contributions.

II. Related Work

Currently the basis for an efficient DM management in a general context is already well established and work is available about general-purpose DM management implementations and policies [16]. Also, research exists on custom DM managers that improve performance and fragmentation (but not memory accesses) using locality of references [16].

In memory management for embedded systems [12], the DM is usually partitioned into fixed blocks to store the dynamic data. Then, the free blocks are placed in a single/doubly linked list [12] due to performance constraints with a simple (but fast) fit strategy, e.g. first fit [16]. Also, in new consumer embedded systems where the range of applications to be executed is very wide, variations of state-of-the-art general-purpose DM managers are frequently used. For instance, Linux-based systems use as their basis the Lea DM manager [16] and Windows-based systems include in their foundations the Kingsley DM manager [16]. Finally, recent real-time OSs for embedded systems (e.g. [13]) include support for dynamic allocation via platform-specific (custom) DM managers based on simple region allocators [16] with a reasonable performance.

Regarding methods to build DM managers, [5] proposes an infrastructure of C++ layers that can be used to improve performance of general-purpose managers. However, this approach lacks the required formalization to consistently design and profile custom DM managers.

Finally, optimizations for static data in embedded systems is available (see e.g. [4] for a good tutorial overview). These techniques are fully complementary to our work.

III. Methodology Description

Our methodology enables the design of custom DM management mechanisms for new dynamic embedded applications (e.g. multimedia) with reduced amount of data accesses inside the DM managers. To this end, we first define the relevant design space of DM management decisions in multimedia and wireless network applications for reduced amount of memory accesses. Then, we indicate a suitable order to traverse it according to the relative influence of each decision with respect to memory accesses of the de/allocated data of the DM manager. This order is decided with the results of the corresponding simulations, which provide valuable run-time behaviour information of the applications. Finally, a custom

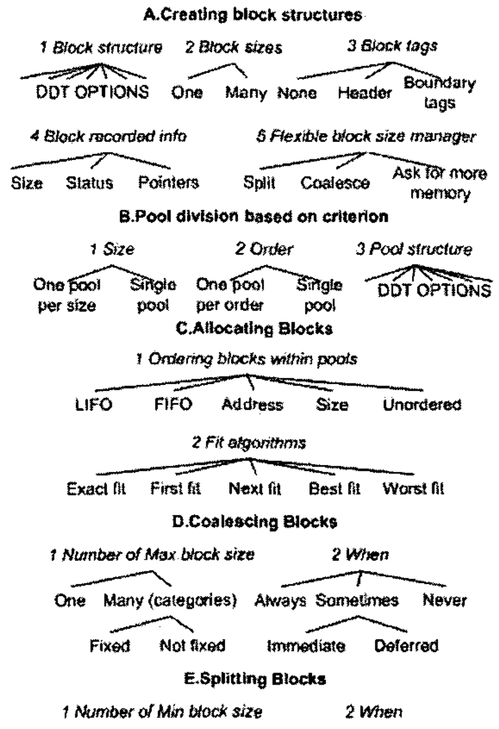

Fig. 1. DM management design space of relevant decisions for reduced amount of memory accesses exploration

DM manager is designed according to the specific DM behaviour of the application under study (e.g. allocation sizes).

In order to perform the final run-time evaluation, implementation and simulation of the custom DM managers, we have developed a C++ library that implements the decisions in our DM design space (see [1] for information about the corresponding profiling tool). Therefore, this library enables a semi-automatic code generation of the DM managers with common profiling for all of them. Apart from the profiling tool, in our recent work [2] we have proposed a DM methodology for reducing memory footprint, but no memory accesses have been addressed there. The most relevant design space decisions and the final order to traverse it are quite different for these two objectives. So this paper contains a brief summary in Subsection III-A of the entire design space (which is largely the same for both) but it also contains novel factors of influence to steer the memory accesses related decisions and traverse the design space (see Section IV).

A. Our DM management design space for reduced amount of memory accesses

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 fragmentation issues. This is done by splitting and merging free blocks to keep memory fragmentation as small as possible.

We have classified all the important design options that compose the design space of DM management in different decision trees. The design space was created after careful analysis of the most common DM managers [16] and many custom solutions (e.g. [13]) so that it can define any custom or general-purpose DM manager. 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 reduced amount of memory accesses (they are explained in detail in [2]):

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.

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.

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.

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.

E. Splitting blocks, which refers to the actions executed by the DM managers to limit internal fragmentation, i.e. splitting one larger block into two smaller ones. The leaves of E1 and E2 are the same as D1 and D2.

IV. Order for Reduced Amount of DM Data Accesses

A. Factors of influence for DM data accesses

The main factors that affect memory accesses of the data inside the DM manager are the accesses to traverse the pools, the accesses to traverse the blocks within the pools and the internal and external memory defragmentation function accesses:

I)The traversing pools accesses consist of the accesses used by the DM manager to find the correct pool to allocate, deallocate or use a specific memory block. The pool organization is controlled by category B *Pool division based on*. When memory pools contain many blocks, their internal organization frequently becomes more complex. Thus, more memory accesses are needed to find a single memory block (e.g. fewer accesses are required to find a block searching in 10 pools that contain 20 blocks each, than to find a block in 2 pools that contain 100 blocks each). Additionally, more complex memory pools will prevent the system from reaching high levels of fragmentation. For example, a system with several different allocation size requests is likely to have less fragmentation, when you have two pools with 400 bytes in total (one pool with 20 10-byte-blocks and one pool with 10 20-byte-blocks) compared to when you have just one pool with 20 20-byte-blocks (again 400 bytes), because the requested blocks can be distributed more evenly according to their sizes. Therefore, less defragmentation function accesses will be needed. As a result, we can conclude that category B is the most important category to minimize the memory accesses.

II)The traversing block accesses are those made by the DM manager to find a specific memory block to be allocated, deallocated or used within a certain pool. The category of the DM management design space, related to traversing blocks accesses, is category C *Allocating Blocks*. For example, fewer memory accesses are used, if the `First fit` algorithm is chosen, because the DM manager looks for the first available block inside the pool, not for the block that matches best to the requested size (`Best fit`). In this way, we achieve fewer accesses, but have more memory fragmentation, thus more defragmentation function accesses will be needed, if memory footprint is a design constraint.

III)The internal defragmentation function accesses are used to split big memory blocks to fit smaller block requests. In our design space, the internal defragmentation accesses are controlled by category E (*Splitting blocks*) and its main impact is on applications, that deal with small blocks [16]. E.g. if only 500-byte blocks are available inside the pools and you want to allocate 20-byte blocks, it would require at least 25 memory accesses to split one 500-byte block to 25 blocks of 20 bytes to avoid internal fragmentation.

IV)The external defragmentation function accesses are used to coalesce small memory blocks to fit bigger block requests. E.g. if you want to allocate a 50-Kbyte block, but only 500-byte blocks exist inside the pools, it would require at least 100 memory accesses to coalesce 100 blocks to provide the necessary amount of memory requested. In this case, the external defragmentation memory accesses are controlled by category D (*Coalescing blocks*) in our DM management design space. As expected, it its main impact is on applications, which deal with big blocks [16].

Note the distinction between categories D and E, which try to deal with internal and external fragmentation, as opposed to category B and C that try to prevent it. Prevention requires less memory accesses than the defragmentation functions.

B. Order of the trees for reduced amount of DM data accesses

Due to the big size of the design space and the many different options available to the designer, we have concluded to a sequential order of the decision trees and their corresponding leaves to achieve reduced amount of memory accesses. This order is derived from extensive simulations and the factors

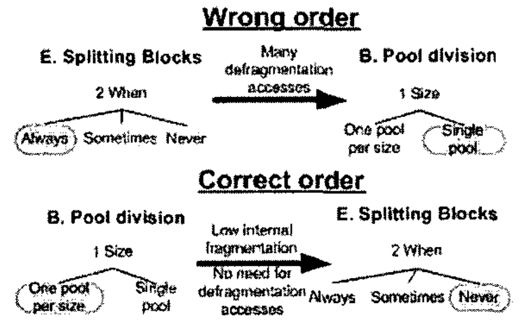

Fig. 2. Example of correct order between decision trees

of influence for DM accesses. Therefore, the correct leaf for tree A2 should be decided first to define the global structure of the blocks. Secondly, it is important to define the various pool structures of the DM manager. This is the most important step for the construction of the DM manager, because as we can see in Subsection IV-A the pool organization (controlled by category B) is the most crucial factor and can achieve a dramatic reduction in memory accesses and prevent fragmentation. Thirdly, it must be decided how to deal with internal and external fragmentation (thus categories D and E go next), which is an important issue among dynamic applications. Next, the trees in category C, which control the memory accesses for traversing the blocks within pools and prevent further memory fragmentation, are decided. Finally, the rest of the trees in category A (i.e. A1, A3 and A4), which control the memory size per block, are decided. As a result, after running the needed simulations to establish the importance of each factor of influence for memory accesses and also considering the interdependencies among the different options (they are explained in detail in [2]), the final order within the DM design space for the reduced amount of memory accesses exploration is as follows: A2->B3->B1->B2->A5->E2->D2->E1->D1->C1->C2->A1->A3->A4.

If the order we have just proposed is not followed, unnecessary constraints are propagated to subsequent decision trees, and thus the most suitable decisions cannot be taken in the remaining decision trees. Figure 2 shows an example of this. Suppose that the order was E2 and then B1. When deciding the correct leaf for E2, the usual choice would be to choose the `Always` leaf (e.g. [13]). This seems reasonable at first sight, because it makes sense to split a memory block if it is bigger than the block requested and reuse later the remaining block for another allocation [16]. Now, we can choose the `Single pool` leaf for the tree B1, because there is no need to have many pools with different sized blocks. If we need a block size that does not exist, we split a bigger sized block. Hence, the final DM manager always chooses the blocks available in a single pool and splits them if they are bigger than the requested size, thus increasing dramatically the accesses due to defragmentation purposes (i.e. splitting mechanisms). However, the internal fragmentation problem can be solved without wasting too many accesses on defragmentation mechanisms. Therefore, it is necessary to decide first the leaf for the B1 tree and then decide the leaf for the tree E2. Hence, we select first the leaf `One pool per size` in tree B1, so that we have many different block sizes available, but in separated pools. In this way, we create too little internal fragmentation to be considered important. Then, we have more freedom to select the correct leaf for E2, i.e. the `Never` leaf, which never invokes a splitting mechanism, thereby reducing the total memory accesses due to defragmentation.

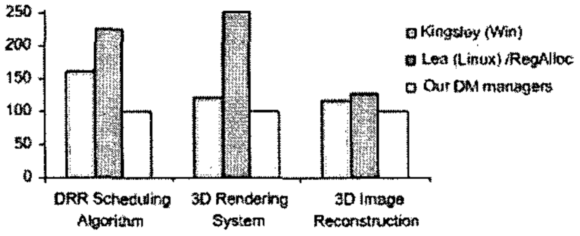

Fig. 3. Memory accesses comparison between DM Managers (results normalized to our custom DM managers)

V. Experimental Results

We have applied the proposed methodology to three case studies that represent different new multimedia and wireless network application domains: the first case study is a 3D rendering system where the objects are represented as scalable meshes, the second one is part of a new 3D image reconstruction system, and finally the third one is a scheduling algorithm from the network protocol domain. The results shown in this section 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 3%). The results were compared to the general/purpose DM managers Lea (Linux) and Kingsley (Windows XP), and to custom DM managers (e.g. Obstacks) whenever they were applicable. All the DM managers were created with the same DM library and the same profiling tools have been used for all of them [1].

The first case study presented is a 3D video rendering application [17]. It belongs to the new category of 3D algorithms with scalable meshes [10] that adapt the quality of each object displayed on the screen according to the position of the user watching them (i.e. Quality of Service). The objects are internally 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. The results were obtained with Visual C++ 6.0 on a Pentium III at 800 Mhz with 256 MBytes SDRAM and running Windows XP.

Initially, we profile the application to get a global view of its DM behaviour. Then, we take the decisions in the trees according to the order we have developed. The order of the decisions in the corresponding trees is different than [2] and therefore we will choose the right leaves to create the custom DM manager with the least memory accesses. 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 3D video rendering application requests memory blocks that vary greatly in size depending on the data structure to allocate (e.g. vertices, faces). In this way, we can avoid having defragmentation memory accesses later. After this, in trees B1 (*Pool division based on size*), B2 (*Pool division based on order*) and B3 (*Pool structure*), a complex pool implementation with tree different-sized freelists is selected, i.e. `One pool per size and order`. In this way, we prevent most of the possible fragmentation and provide fast finding of the correct block size, with few memory accesses (in [2] we had chosen to have a `Single pool` to save as much space as possible). Then, in tree A5 (*Flexible block size manager*) we choose to `Ask for more memory`, so that no memory accesses are spent on defragmentation functions (in contrast to [2], where the splitting and coalescing mechanisms were invoked in every memory allocation). In trees E2 and D2 (*When*) we choose `Never` and, in trees E1 and D1 (*Number of max and min block size*) we choose `One` (per pool), due to the interdependencies involved with the previous tree A5 (in [2] we had chosen the complete opposite leaves `Always` and `Not fixed`). Next, in tree C1 (*Fit algorithms*), we select `First fit` to minimize the amount of memory accesses to traverse the pools (in [2] we had chosen `Exact fit` to avoid fragmentation, but which increases significantly the amount of memory accesses). In tree C2 (*Ordering blocks within pools*), we select the `LIFO` ordering because the rendering algorithm de/allocates blocks in a phase-like fashion that is very suitable for a reuse structure with such kind of order. Next, in tree A1 (*Block structure*), we choose the fastest dynamic data type for this particular application, i.e. `Doubly linked list with memory for the last accessed element in any pool`. This effectively minimizes the amount of traversing accesses within the pools according to the de/allocation behavior of this particular application. Then, in the trees A3 (*Block tags*) and A4 (*Block recorded info*), we choose a `Header` field to accommodate information about the `Status` (in [2] we had chosen also `Size`). Finally, after taking these decisions following the order described in Section IV, we determine those decisions of the final custom DM manager that depend on its particular run-time behaviour in the application via simulation (e.g. for this applicaiton we obtained the best results with 4 different freelists for the *Pool structure* tree).

In this case, we have compared our custom DM manager with Lea, Kingsley and Obstacks [16]. The results depicted in Figure 3 shows that our custom DM manager produces less memory accesses than Lea and Obstacks, i.e. 58.5% and 30.2% less respectively. In addition, Kingsley requires less memory accesses than Obstacks because its optimizations for the stack-like behaviour of the first three phases of the rendering process cannot be used in the last three phases, which include many more additional maintenance accesses compared to Kingsley. Finally, our custom manager further reduces the amount of memory accesses produced by Kingsley (18.1% less). This improvement is due to the fact that our design exploration includes the knowledge of the specific DM behaviour of the application, which cannot be done in Kingsley since its design is general-purpose. Therefore, our custom manager includes similar stack-like optimizations as Obstack in the first three phases of the rendering process. Then, in the last three phases, the faces are used independently in a disordered pattern and they are freed separately. Thus, our DM manager includes separate pools for each of the specific sizes of the objects used in these phases, i.e. 40, 44 and 84 bytes. This very specific structure simplifies the complex infrastructure of Kingsley, where more than 20 different allocation sizes (and pools) are selected and frequently checked for a new allocation. Thus, Kingsley produces more memory accesses than our custom DM manager.

The second real-life application we have used to test our methodology is a 3D vision reconstruction system [14] (see [15]. The results were obtained with `gcc` v3.2 on a Pentium III at 800 Mhz with 256 MBytes SDRAM and running GNU/Linux 2.4. It heavily uses DM due to the variable features of input images. This implementation reconstructs 3D images by matching corners [14] detected in 2 subsequent frames. The operations on the images are particularly memory intensive, e.g. each matching process between two frames with a resolution of 640 × 480 uses over 1Mb, and the accesses of the algorithm (in the order of millions of accesses) to the images are randomized. Thus, classic image access optimizations such as row-dominated accesses versus column-wise accesses cannot be applied to reduce the amount of memory accesses.

In this real-life application we report the results obtained with our custom DM manager and those obtained with two frequently used DM managers for such kind of applications where performance is important, i.e. Kingsley and a manually optimized version of a typical region manager found in new embedded OSs [13]. Figure 3 shows the results obtained. These results indicate that our custom DM manager reduces the total amount of memory accesses compared to Kingsley (14.8%) and new region managers (22.7%).

These reductions in memory accesses occur, as in the previous case study, because our DM manager isolates the representative sets of allocation sizes (only few as often occurs in new multimedia applications) in several isolated pools with optimizations based on their particular de/allocation pattern (e.g. sequential allocation of memory blocks, stack-like deallocation, LIFO). Thus, a limited overhead in accesses is needed to identify the right pool for each allocation or deallocation of the data in each pool. In Kingsley, a much higher amount of accesses is required due to the wide range of block sizes (and lists of free blocks) it allows. Also, due to its general-purpose orientation, all the pools include the same organization and no particular access pattern de/allocation optimizations are applied for each allocation size. Therefore, the overall amount of memory accesses increases significantly. Similarly, the region manager does not allow to define different strategies for each region it includes. Furthermore, the overhead of using more than one size (and then more than one region) inside the region manager creates an additional overhead in accesses that cannot be removed due to the maintenance block of each region. This distributed organization implies more memory accesses than our custom DM manager, which includes only one unified table for all the pools.

The third case study is the Deficit Round Robin (DRR) application from the NetBench benchmarking suite [11]. It is a scheduling algorithm implemented in many routers today that performs a fair scheduling by sending the same amount of data from each internal queue. It uses DM because the real input can vary enormously depending on the network traffic. In our experiments, real traces of internet network traffic up to 50 Mbit/sec [8] have been used. The results were obtained with gcc v3.3 on a AMD Athlon at 1200 Mhz with 256 MBytes SDRAM and running GNU/Linux 2.4.

In this case, we have compared our custom solution to state-of-the-art general-purpose managers, i.e. Lea [16] and Kingsley [16] (see Section II for more details). As Figure 3 shows, our custom DM manager uses fewer memory accesses than Lea or Kingsley, because it sacrifices reasonably-increased memory footprint for reduced memory accesses. On the one hand, our custom DM manager chooses to prevent as much memory fragmentation as possible, but does not invoke defragmentation mechanisms like Lea, thus considerably reducing the amount of memory accesses. On the other hand, our custom DM manager employs more application-specific fit strategies and block ordering than Kingsley, thereby gaining advantage on memory accesses. To conclude, our custom DM manager reduces the amount of memory accesses by 37.7% compared to Kingsley and 56% compared to Lea.

Lastly, to evaluate the complexity of the design process with our methodology, note that the design and implementation of the source code of the custom DM managers for each case study took us only two weeks using the methodology presented in this paper and the semi-automatic source code generation with our DM library [1]. As Figure 4 shows, these DM managers improve the execution time up to 47% compared to the execution time of the fastest general-purpose DM manager observed in these case studies, i.e. Kingsley, thus increasing the performance of the DM subsystem. These reductions in memory accesses and the increased performance are achieved at the cost of increasing the total DM footprint in comparison with the results obtained by other DM managers, i.e. Lea or Obstacks, and our own previous results [2] (in the worst case, the memory footprint of the DRR scheduling algorithm was increased from 1.20 MBytes to 2.02 MBytes). This indicates that trade-offs exist within our DM design space between the relevant design factors for embedded system (e.g. reducing memory footprint by increasing the amount of more memory accesses). These trade-offs can be exploited by our methodology (by slightly modifying our exploration order and cost functions) if the requirements of the final design need it.

VI. Conclusions

Embedded devices have lately increased their capabilities and now complex applications (e.g. multimedia) can be ported

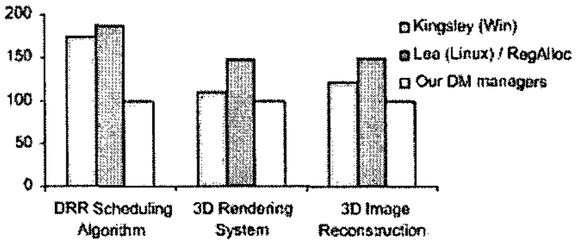

Fig. 4. Execution time comparisons between DM Managers (results normalized to our custom DM managers)

to them. Such applications include intensive DM requirements (e.g. memory accesses and performance) that must be heavily optimized for an efficient mapping on current embedded devices. Within this context, new design methodologies must be available to suitably use the resources available in these final embedded devices. In this paper we have presented a new methodology that defines and explores the DM management design space of relevant decisions in order to design custom DM managers with reduced amount of memory accesses for such dynamic applications. Due to the focus of our methodology to the data accesses inside the DM managers, the results achieved in real applications show significant improvements over state-of-the-art general-purpose and manually optimized custom DM managers.

Acknowledgments

This work is 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.

References

[1] D. Atienza et al. Modular Construction and Power Modelling of DM Managers for Embedded Systems. In *Proc. of PATMOS*, Greece, 2004.
[2] D. Atienza et al. DM Manag. Design Method. for Reduced Mem. Footprint in Multimedia and Network Apps. In *Proc. of DATE*, 2004.
[3] G. Attardi, et al. A customizable memory management framework for C++. *SW Practice and Experience*, 1998.
[4] L. Benini et al. System level power optimzation techniques and tools. In *ACM TODAES*, 2000.
[5] E. D. Berger, et al. Composing high-performance memory allocators. In *Proc. of ACM SIGPLAN PLDI*, USA, 2001.
[6] E. G. Daylight, et al. Analyzing energy friendly phases of dyn. apps. in terms of sparse data. In *Proc. of ISLPED*, USA, 2002.
[7] N. Jouppi. Western research laboratory, cacti, 2002. `http://research.compaq.com/wrl/people/jouppi/CACTI.html`.
[8] The Internet traffic archive, 2000. `http://ita.ee.lbl.gov/`.
[9] M. Leeman, et al. Methodology for refinement and optim. of DMM for emb. syst. in multimedia. In *Proc. of SiPS*, 2003.
[10] D. Luebke, et al. *Level of Detail for 3D Graps.* Morgan-Kauf., 2002.
[11] G. Memik, et al. Netbench: A benchmarking suite for network processors. CARES Tech. Report 2001-2-01.
[12] N. Murphy. Safe mem. usage with DM allocation. *Emb. Systems*, 2000.
[13] Rtems, Multiproc. Real-time OS, 2002. `http://www.rtems.org`.
[14] M. Pollefeys, et al. Metric 3D surface reconstruction from uncalibrated image sequences. In *LNCS*, vol. 1506, 1998.
[15] Target jr, 2002. `http://www.targetjr.org`.
[16] P. R. Wilson, et al. Dynamic storage allocation, a survey and critical review. In *Int. Workshop on Mem. Manag.*, UK, 1995.
[17] M. Woo, et al. *OpenGL Programming Guide.* Silicon Graphics, 1997.