Modular Construction and Power Modelling of Dynamic Memory Managers for Embedded Systems*

David Atienza[1], Stylianos Mamagkakis[3], Francky Catthoor[2**],
J.M. Mendias[1], and D. Soudris[3]

[1] DACYA/U.C.M., Avda. Complutense s/n, 28040 - Madrid, Spain
[2] IMEC vzw, Kapeldreef 75, 3000 Leuven, Belgium
[3] VLSI Design Center-Democritus Univ., Thrace, 67100 Xanthi, Greece

Abstract. Portable embedded devices must presently run multimedia and wireless network applications with enormous computational performance requirements at a low energy consumption. In these applications, the dynamic memory subsystem is one of the main sources of power consumption and its inappropriate management can severely affect the performance of the system. In this paper, we present a new system-level approach to cover the large space of dynamic memory management implementations without a time-consuming programming effort and to obtain power consumption estimates that can be used to refine the dynamic memory management subsystem in an early stage of the design flow.

1 Introduction

Recently, with the emerging market of new portable devices that integrate multiple services such as multimedia and wireless network communications, the need to efficiently use Dynamic Memory (DM from now on) in embedded low-power systems has arisen. New consumer applications (e.g. 3D video applications) are now mixed signal and control dominated. They must rely on DM for a very significant part of their functionality due to the inherent unpredictability of the input data, which heavily influences global performance and memory footprint of the system. Designing them using static worst case memory footprint solutions would lead to a too high overhead in memory footprint and power consumption for these systems [5]. Also, power consumption has become a real issue in overall system design (both embedded and general-purpose) due to circuit reliability and packaging costs [14]. Thus, optimization in general (and especially for embedded systems) has three goals that cannot be seen independently: memory footprint, power consumptions and performance.

Since the DM subsystem heavily influences performance and is a very important source of power consumption and memory footprint, flexible system-level implementation and evaluation mechanisms for these three factors must be available at an early stage of the design flow for embedded systems. Unfortunately, general approaches that integrate all of them do not exist presently at this level of abstraction for the DM managers implementations involved.

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

** Also professor at ESAT/KUL.

E. Macii et al. (Eds.): PATMOS 2004, LNCS 3254, pp. 510–520, 2004.

Current implementations of DM managers can provide a reasonable level of performance for general-purpose systems [15]. However, these implementations do not consider power consumption or other limitations of target embedded platforms where these DM managers must run on. Thus, these general-purpose DM managers implementations are never optimal for the final target platform and produce large power and performance penalties. Consequently, system designers must face the need to manually optimize the implementations of the initial DM managers in a case per case basis and without detailed profiling of which parts within the DM managers implementations (e.g. internal data structures or links between the memory blocks) are the most critical parts (e.g. in power consumption) for the system. Moreover, adding new implementations of (complex) custom DM managers often proves to be a very programming intensive and error prone task that consumes a very significant part of the time spent in system integration of DM management mechanisms. In this paper, we present a new high-level programming and profiling approach (based on abstract derived classes or mixins [11] in `C++`) to create complex custom DM managers and to evaluate their power consumption at system-level. This approach can be used to effectively obtain early design flow estimates and implementation trade-offs for system developers.

The remainder of the paper is organized as follows. In Section 2, we describe some related work. In Section 3 we present the proposed construction method for DM managers and the necessary profile framework to obtain detailed power consumption estimations. In Section 4, we shortly introduce our drivers and present the experimental results obtained. Finally, in Section 5 we draw our conclusions.

2 Related Work

In the software community much literature is available about DM management implementations and policies to be used in general-purpose systems [15]. In memory management for embedded systems [8], the DM is usually partitioned into fixed blocks to store the dynamic data. Then, the free blocks are placed in a single linked list [8] due to performance constraints with a simple (but fast) fit strategy, e.g. first fit or next fit [15]. Also, in recent real-time operating system synthesis approach for embedded systems [9], dynamic allocation is supported with custom DM managers based on region allocators [15] for the specific platform features.

Another recent method to improve performance of the DM subsystem is to simulate the system with partially customizable DM management frameworks. In [1], a `C++` framework where you can partially redefine some functionality (e.g. `malloc()` function) of the DM subsystem has been proposed, but it does not consider changes in the implementation structure of DM managers. Also, [2] outlines an infrastructure to improve performance of general-purpose managers. However, its definition for performance exploration of general-purpose DM managers restricts its flexibility to isolate and explore the influence of basic implementation parts of custom DM managers (e.g. fit algorithms [15]) for other metrics (e.g. power consumption).

Regarding profiling of the DM subsystem, recent work has been done to obtain profiling from assembly code and even a higher abstraction level [14]. Nevertheless, current methods do not yet include detailed enough run-time profiling analysis to evaluate the

influence on power consumption of the basic implementation components of DM managers (e.g. fit algorithms or maintenance data structures). Hence, they are not sufficient for modern dynamic applications. In addition, several analytical and abstract power estimation models at the architecture-level have received more attention lately [3]. However, they do not focus on the DM hierarchy of the system and the power consumed by DM managers at the software level.

3 Construction and Profiling of Layered DM Managers

The implementation space of DM managers is very broad and we need to cover it in a flexible and extensible way. Therefore, we use a C++ approach which combines abstract derived classes or mixins [11] with template C++ classes [13]. In the remainder of the text, we use the definition of mixins as used in [11]: a method of specifying extensions of a class without defining up-front which class exactly it extends.

In Figure 1 we show the basic concepts used in this approach. In the first example of Figure 1, a subclass of `SuperClass` is declared with `SuperClass` itself being a template argument and consequently also the subclass is defined. Then, `MyMixin` class is reusable for one or more parent classes that will be specified in the different instantiations of `MyMixin` class. In the second example of Figure 1, another class is defined (i.e. `MyClass`), where the template argument is not used as a parent class, but instead as internal private data members. In our approach, as we show in the following sections, the first concept is used to refine the functionality of the custom DM managers and the second one is used to specify its main components, e.g. heaps, data structures, etc. As a result of this very modular approach, we can combine both concepts to build very customized DM managers starting from their basic structures (e.g. data structures, fit algorithms, etc.) and later on add detailed profiling for each of these basic structures. In conventional approaches [1,2,15] this kind of modeling and detailed profiling of basic structures of DM managers is not possible. The main reason is that in such approaches the DM managers are built as complex software engineering modules where all the different components (e.g. fit algorithms, data structures) are combined and deeply embedded in their implementations. Thus, only a limited number of variations in the final

```
// Example 1: Basic MyMixin Class
    template <class SuperClass>
        class MyMixin : public SuperClass{
                // MyMixin class definitions };

// Example 2: Abstract parent class inside MyClass
    template <class SuperClass>
        class MyClass{
                SuperClass* data;
                // template class definitions };
```

Fig. 1. Parametrized Inheritance used with mixins in C++

implementation can be explored by the designer due to the time-consuming effort of reprogramming their global structures.

3.1 Construction of Layered DM Managers with Profile Support

Using the previously explained concepts of abstract derived classes and template C++ classes, we have redefined the library proposed in [2] and integrated our own profile framework (see Subsection 3.2 for a detailed description of this framework) to be able to explore and profile power consumption, memory footprint and memory accesses in the basic construction categories we distinguish for DM managers. These categories are the following: creating block structures, pool division based on different criterion, fit algorithms, order of the blocks within the pools (address, size, etc.), coalescing (or merging blocks) and splitting blocks [16]. In Figure 2 we show an example of the construction of a DM manager with basic blocks and how our profile framework can be added to any part of it with a fine granularity. First, the basic heaps of the manager and the basic allocation blocks requested to the system are defined (class `BasicHeap` in Figure 2). Second, the two basic data structures to test within the manager, i.e. double linked lists (DLList) and binary trees (BTTree) are implemented. Third, they are instantiated for the basic sizes to use in the DM manager. Then, the profile objects (see Subsection 3.2 for

```
// Basic blocks for heap requests to the system
template<typename MyT>
        class BasicHeap: public TypeClass<MyT,mheap>;
// Data types of the dynamic memory manager
template<typename MyT, class SuperClass>
        class DLLList {// Implementation of a double link list
                       // list with generic data size MyT };
template<typename MyType, class SuperClass>
        class BTTree { //Implementation of binary tree
                       // with generic data size MyT };
// Two basic data types instantiated for the memory manager,
class I_DLList : public DLList<int, BasicHeap<int> >{};
class D_BTTree : public BTTree<double, BasicHeap<double> >{};
// Declaration of profile objects to profile the manager
_profile *prof1, *prof2, *profileGlobal;
// Memory manager with 2 segregated-fit lists of different data types,
// best or fit policy and profile objects
class DMMHeap: public
    SegLists<profileGlobal, // Global profile object
        list_Sizes, // List of sizes for the segList
        numElemFirstList, // Number of lists with type 1st segList
        BestFit<I_DLLList<prof1> >, // 1st segList
        FirstFit<D_BTTree<prof2> > // 2nd segList
    > {};
```

Fig. 2. Example of custom DM manager with profiling objects

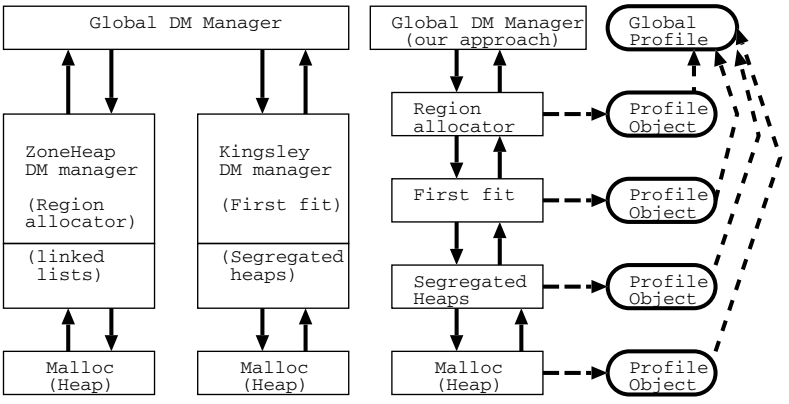

Fig. 3. Example of the structure of a custom DM manager. On the left, built with the approach proposed in [2] where the layers are really interdependent. On the right, our own approach

more details about their use) to obtain the necessary information about power consumption, memory footprint and memory accesses are created. Finally, the DM manager is created as a combined structure of two different segregated lists [15], which are formed by different dynamic data structures inside (DLList or BTTree) and different allocation policies (best fit or first fit) [15].

As Figure 3 shows, we can build custom DM managers from its basic blocks and obtain power estimations from them in a much more flexible way than the structure proposed in [2]. For example, if due to the characteristics of the final system it is necessary to combine two different allocation strategies from two different general-purpose managers in the same global manager, using [2] we would need to create both DM managers and combine them later as independent heaps because a great part of the structure of each DM manager is fixed. On the contrary, our approach allows to create a global DM manager using just a single heap. This global manager would by composed by several intermediate layers that define a very customized and flexible implementation structure including the two different allocation strategies in the same heap. This example is depicted in Figure 3. Thus, we can merge the two allocation heaps saving memory footprint because the memory can be reused for both. Also, our final structure is simpler to compose because parts of the maintenance data structures can be shared and accessed simultaneously (e.g. pointers of the memory blocks). Hence, the number of memory accesses and eventual power consumption of the DM manager are reduced (as shown in Section 4, Table 6 with `ObstLea`). Finally, note that any modification in the implementation structure of the heap only requires to substitute a very limited number of layers. Therefore, the programming effort to do it is reduced heavily.

3.2 Structured Profile Framework and Power Model

Apart from simplifying the effort of exhaustively covering the implementation space of DM management, the presence of multiple layers in the DM managers also gives a lot

of flexibility to profile their characteristics at different levels, e.g. memory accesses of each implementation layer in the internal structure, as Figure 3 indicates. This detailed profiling is required for a suitable optimization (e.g. for power consumption) of the DM manager since small changes in the implementation of some layers can completely change the global results of the DM manager in the system, even if most of the implementation structure remains the same. For example, as we explain in our case studies in Section 4, a LIFO reuse strategy of the blocks can produce completely different results compared to a FIFO reuse strategy. However, this detailed profiling at the level of the individual layers in the DM manager requires a new system-level profiling framework that is flexible enough to handle all kinds of combinations between the layers. Since more than one layer can constitute the part of the manager to measure, the profiling information must be grouped and cannot be collected at one layer only. Therefore, we have integrated a similar approach to the one proposed in [5] for complex dynamic data types. As Figure 3 depicts, it consists of an object-oriented profiling framework that decouples this information from the class hierarchy of the DM managers, providing accurate run time information on memory accesses, memory footprint, timing information and method calls. Then, we can use this information to obtain power model estimates for the DM managers using a realistic model of the underlying memory hierarchy in a post-execution phase. Thus, the application runs at its normal speed and the total evaluation time for one DM manager is reduced from several hours of simulation in typical cycle-accurate simulations to few minutes including the post-execution phase.

For this post-execution phase, we use an updated version of the CACTI model [4], which is a complete energy/delay/area model for embedded SRAMs that depends on memory footprint factors (e.g. size or leaks) and factors originated by memory accesses (e.g. number of accesses or technology node). The main advantage of CACTI is that it is scalable to different technology nodes. For the results shown in Figure 5, Figure 6 and Table 1, we use the $.13\mu$m technology node. Note that any other model for a specific memory hierarchy can be used just by replacing this power module in the tools.

4 Case Studies and Experimental Results

We have applied the proposed method to three case studies that represent different modern multimedia and network application domains: the first case study is part of a new 3D image reconstruction system, the second one is a 3D rendering system based on scalable meshes and the third one is a scheduling algorithm from the network domain. All the results shown are average values after a set of 10 simulations for each application and DM manager implementation. The obtained results (e.g. execution time, power consumption estimations) were all very similar (variations of less than 2%).

The first case study is a 3D vision reconstruction application [10] (see [12] for the full code of the algorithm with more than 1 million lines of high level `C++`). It heavily uses DM due to the variable features of input images. This implementation reconstructs 3D images by matching corners [10] detected in 2 subsequent frames. The operations done 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

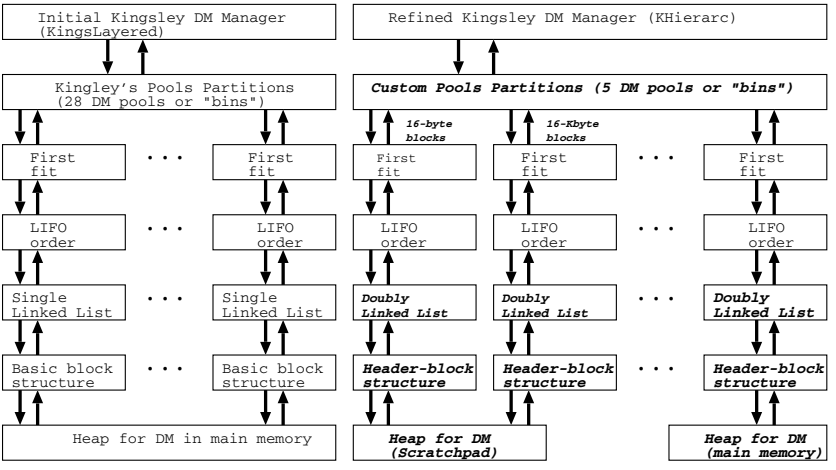

Fig. 4. On the left, initial implementation structure of Kingsley DM manager with our approach. On the right, our final refined version of it, i.e. `KHierarc`, main changes in bold

access optimizations as row-dominated accesses versus column-wise accesses cannot be applied to reduce the memory accesses and power consumption values.

For this case study, we have implemented and profiled several DM managers starting from a general-purpose one and refining its implementation using our approach. First of all, we have implemented one of the fastest general-purpose managers, i.e. Kingsley DM manager [15] (`KingsLayered` in Figure 5). But it has a considerable fragmentation due to its use of power-of-two segregated-fit lists [15]. A graphical representation of its implementation structure with our layered-approach is shown in Figure 4. As Figure 5 shows, its memory footprint is larger than any other DM manager in our experiments, but its total execution time is faster than the new region-semantic managers [15] frequently found in current embedded systems, i.e. RegAlloc in Figure 5.

After implementing and profiling these two generic DM managers, we have observed that most of the accesses in Kingsley occur in just few of the "bins" (or memory pools of the heap) [15], due to the limited range of data type sizes used in the application [5]. Therefore, we try to reduce its memory waste by modifying its design with our layers and by limiting the number of bins to the actual sizes used in the application (5 main sizes), as Figure 4 shows at the top in its right graph. This variation is the most significant change in its internal structure and allows to define the custom manager marked as `KLimit` in Figure 5. We can see that its improvement is already significant in energy dissipated per matching process of two frames. Then, we try to improve its structure even further with our layered approach. Thus, the bins that produce most of the accesses (the bins for allocation sizes of 16 bytes with the maintenance information of the manager and the data types of blocks of 16 Kbytes) are easily separated using our infrastructure of layers from the global heap used in the manager. They are now handled in a different and small heap (57 Kbytes) that is placed permanently in the scratchpad, as Figure 4 indicates at

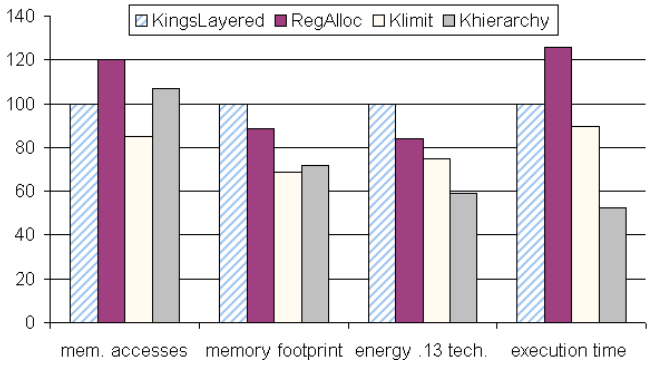

Fig. 5. Profiling results of different DM managers (normalized to Kingsley, i.e.`KingsLayered`) in the 3D Image Reconstruction System per each matching process of two frames

the bottom in its right graph. This custom DM manager, which is optimized according to the final memory hierarchy, is depicted on the right side of Figure 4 and marked as `KHierarchy` in Figure 5.

The latter figure shows that `KHierarchy` DM manager has increased its total amount of memory accesses and total memory footprint compared to `KLimit`, but most of the accesses of the manager are now in the on-chip scratchpad memory (i.e. 95%). Also note that the increase in memory footprint is mainly due to data copied between the different levels of the memory hierarchy and this increase is not really significant comparing it with the accesses saved to the off-chip memory. Hence, we can observe that the total energy dissipation and execution time of this custom memory manager have decreased enormously compared to the other ones in Figure 5.

The second case study is a realistic example of new 3D rendering applications where scalable meshes [6] are used to adapt the quality of each object displayed on the screen according to the position of the user watching at them at each moment. In this case we have implemented with our approach one of the best general-purpose DM managers (in terms of the combination of speed and memory footprint) [15,2], i.e. Lea Allocator v2.7.2 [15]. Apart from it, we have used Kingsley [15] to compare both in memory footprint, memory acesses and total energy consumption figures. Also, we have tested a well-known custom DM manager optimized for a stack-like DM behavior, i.e. Obstacks [15]. As Figure 6 shows, the Lea Allocator (`LeaLeayered`) obtains average values for a certain trade-off in performance and memory footprint. However, its energy dissipation is very high due to the additional accesses for its complex maintenance structure. Also, Figure 6 indicates that Kingsley suffers from high fragmentation, but produces a lot less accesses. Thus, with completely different characteristics, both managers are close in their final figures for power consumption. Also, Obstacks has few accesses during the first 3 phases of the rendering process due to their partial stack-like behavior, but suffers from high penalty in memory accesses and energy dissipation per frame in these last three phases. Hence, its final values are not as good as expected.

These results suggest the convenience of a custom DM manager that combines the behaviour of Obstacks with Lea in the last three phases. We have built it with our

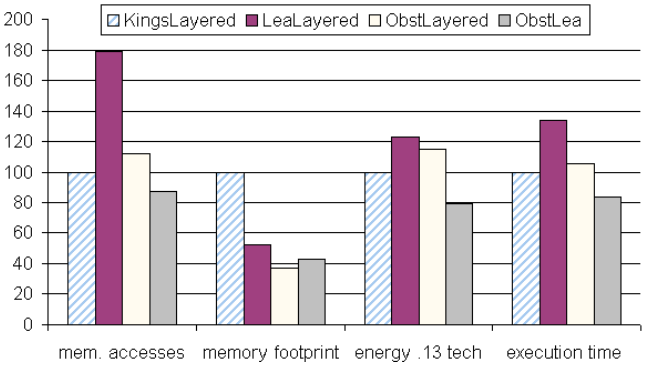

Fig. 6. Profiling results of different DM managers (normalized to Kingsley, i.e.`KingsLayered`) in the 3D Rendering System for one object in one frame

approach (3 weeks) and it is marked as `ObstLea` in Figure 6. This figure shows that this new manager accomplishes very good overall results. Also, our DM manager designs have a similar execution time (differences of less than 6% in execution time) compared to the original (manually-designed) versions of Obstacks and Lea, but with a clear improvement in design complexity on our side. Our version of the Lea Allocator has around 700 lines of C++ code instead of more than 20000 lines of C code as in the original Lea implementation, and 400 lines of C++ code for our version of Obstacks compared to 2500 lines approximately of its state-of-the-art implementation.

The third case study presented is the Deficit Round Robin (DRR) scheduling application taken from the NetBench benchmarking suite [7]. It is a fair scheduling algorithm implemented in many routers today where the scheduler visits each internal queue and forwards the corresponding packets according to their size and priority. The DRR application was profiled in our results for realistic input traces of 100000 packets.

For this case study, we have implemented and profiled using our approach different versions of the same basic structure in the DM manager, i.e. Power-of-two segregated-fit lists [15], without coalescing or splitting services. For speed requirements, we have started from the structure of the general-purpose Kingsley manager and have implemented two variations of it. One uses a LIFO single linked freelist (`Kings+LIFOSLL` in Table 1) and the other one a FIFO double linked freelist (`Kings+LIFODLL` in Table 1). Finally, we have also designed a custom DM manager with FIFO single linked list structure using a segregated fit algorithm (`SegFitSLL FIFO` in Table 1). Our results show that not only the global policy of the manager is important, but also a careful study of the ideal structure of reuse, data types, etc. inside the managers. The results obtained are shown in Table 1. Note that Table 1 is divided in the energy contribution of the off-chip memories and on-chip memories (i.e. lines labelled as `on-chip values`) for each DM manager to the total. We consider in this case that an on-chip scratchpad memory of 16 KBytes is available for all the DM managers.

As Table 1 indicates, the memory footprint is the same for the managers because they are all power-of-two segregated-fit lists with the same internal data organization.

Table 1. Profiling values of DM managers in the DRR application for streams of 100000 packets

memory manager	memory accesses	memory usage (B)	power (μnW) .13μm tech	execution time (secs)
SegFitSLL FIFO	2.00×10^6	2.09×10^6	13.28×10^6	115.04
(on-chip values)	0.25×10^6	16.38×10^3	49.60×10^3	—
Total:	2.25×10^6	2.09×10^6	13.33×10^6	115,04
Kings+LIFOSLL	1.25×10^6	2.09×10^6	83.01×10^6	64.25
(on-chip values)	0.15×10^6	16.38×10^3	31.00×10^3	—
Total:	1.40×10^6	2.09×10^6	8.33×10^6	64,25
Kings+FIFODLL	1.75×10^6	2.09×10^6	11.62×10^6	135.63
(on-chip values)	0.22×10^6	16.38×10^3	43.40×10^3	—
Total:	1.97×10^6	2.09×10^6	11.66×10^6	135,63

However, as we have previously mentioned, it can be seen that by implementing a different allocation reuse scheme (e.g. FIFO and LIFO) we can gain considerably on performance and memory power consumption. The best performance and lowest power consumption figures are achieved by the manager with a Kingsley basis and a LIFO single linked list structure. This is due to the fact that its data structures can be updated using less memory accesses than the others considering the run-time access pattern observed with our profiling. In fact, when one packet has arrived to a certain queue, more packets are likely to arrive in a short period of time to the same queue and with the same size. Thus, the FIFO implementation achieves the best results in power consumption by increasing locality in memory references more than the any other solution.

Finally, to evaluate the speed up of the refinement process, remark that the DM managers for this application constitute around 400 lines of C++ code each and took us one week to build them, profile their components and refine their implementation. Each allocator has 5 layers and since all are variations of segregated fit algorithms, 2 layers were reused in each implementation. Also, remark that our approach is not limited to any specific memory hierarchy, in each case the final DM manager is optimized for the specific memory hierarchy of the system.

5 Conclusions

Consumer applications (e.g. multimedia) have grown lately in complexity and demand intensive DM requirements that must be heavily optimized (e.g. power, memory footprint) for an efficient mapping on current low-power embedded devices. System-level exploration methodologies have started to be proposed to consistently perform that refinement. Within them, the manual exploration and optimization of the DM manager implementation is one of the most time-consuming and programming intensive parts. In this paper we have presented and shown in realistic examples the applicability of a new system-level approach that allows developers to implement DM managers with high maintainability. At the same, it allows to acquire detailed profiling information (e.g. power consumption) of the basic implementation structures of DM managers that can be used to refine their initial implementations.

References

1. G. Attardi, et al. A customiz. mem. manag. framework for C++. *Sw. Pract. and Exp.*, 1998.
2. E. D. Berger, et al. Composing high-performance mem. allocators. In *Proc. of PLDI.*, 2001.
3. R. Y. Chen, et al. Speed and Power Scaling of SRAM's. *ACM TODAES*, 6(1), 2001.
4. V. Agarwal, et al. Effect of technology scaling on microarchitectural structures. TR2000-02, USA(2002).
5. M. Leeman, et al. Power estimation approach of dyn. data storage on a HW SW boundary level. In *Proc. of PATMOS*, Italy(2003).
6. D. Luebke, et al. *Level of Detail for 3D Graphics*. Morgan-Kaufman (2002).
7. G. Memik, et al. Netbench: A benchmarking suite for network processors, 2001.
8. N. Murphy. Safe mem. use with dynamic alloc. *Embedded Systems* (2000).
9. Rtems, Open-Source Real-Time OS. (2002) `http://www.rtems.com/`.
10. M. Pollefeys, et al. Metric 3D surface reconst. from uncalibrated images. In *Lecture Notes in Computer Science*, Springer-Verlag (1998).
11. Y. Smaragdakis, et al. Mixin layers: Object-Oriented implementation techn. for refinement and collaboration-based designs. *Trans. on SW Engineering and Methodology* (2002).
12. Target jr. `http://computing.ee.ethz.ch/sepp/targetjr-5.0b-mo.html` .
13. D. Vandevoorde, et al. *C++ Templates, The Complete Guide*. Addison Wesley, UK(2003).
14. N. Vijaykrishnan, et al. Evaluating integrated HW-SW optimizations using a unified energy estimation framework. *IEEE Trans. on Computers* (2003).
15. P. R. Wilson, et al. Dynamic storage allocation. In *Worksh. on Mem. Manag.*, UK(1995).
16. D. Atienza, et al., DM manag. design method. for reduced mem. footprint in multim. and n etwork apps., in *Proc. of DATE* (2004).