Available online at www.sciencedirect.com

SCIENCE DIRECT®

ELSEVIER

INTEGRATION, the VLSI journal 39 (2006) 113–130

INTEGRATION the VLSI journal

www.elsevier.com/locate/vlsi

# Efficient system-level prototyping of power-aware dynamic memory managers for embedded systems[☆]

David Atienza[a,*], Stylianos Mamagkakis[b], Francesco Poletti[c], Jose M. Mendias[a], Francky Catthoor[d,1], Luca Benini[c], Dimitrios Soudris[b]

[a]*DACYA/UCM, Avda. Complutense s/n, c Juan del Rosal, 8, 28040 Madrid, Spain*
[b]*VLSI Design Center-Demokritus University, Thrace, 67100 Xanthi, Greece*
[c]*University of Bologna, DEIS, Viale Risorgimento 2, Bologna, Italy*
[d]*IMEC vzw, Kapeldreef 75, 3001 Heverlee, Belgium*

Received 1 September 2003; received in revised form 17 March 2004; accepted 8 August 2004

## Abstract

In the near future, portable embedded devices must run multimedia and wireless network applications with enormous computational performance (1-40*GOPS*) requirements at a low energy consumption (0.1–2 W). In these applications, the dynamic memory subsystem is currently one of the main sources of power consumption and its inappropriate management can severely affect the performance of the whole system. Within this context, the construction and power evaluation of custom memory managers is one of the most difficult parts for an efficient mapping of such dynamic applications on low-power embedded systems. In this paper, we present a new system-level approach to model complex dynamic memory managers integrating detailed power profiling information. This approach allows to obtain power consumption estimates, memory footprint and memory access values to refine the dynamic memory

[☆]This work is partially supported by the Spanish Government Research Grant TIC2002/0750 and the European founded program AMDREL IST-2001-34379. Part of this work has been submitted to COLP 2003.

[*]Corresponding author. Tel.: +34-91-394-76-14; fax: +34-91-394-75-27.

*E-mail addresses:* datienza@dacya.ucm.es (D. Atienza), smamagka@ee.duth.gr (S. Mamagkakis), fpoletti@deis.unibo.it (F. Poletti), mendias@dacya.ucm.es (J.M. Mendias), catthoor@imec.be (F. Catthoor), lbenini@deis.unibo.it (L. Benini), dsoudris@ee.duth.gr (D. Soudris).

[1]Also at ESAT/K.U.Leuven-Belgium.

0167-9260/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.vlsi.2004.08.003

management of the system in an early stage of the design flow and to easily explore the large search space of memory manager implementations.

*Keywords:* Dynamic memory management; Low-power design; System-level prototyping

## 1. Introduction

Over the last decade, the design gap between the top line digital signal processors (DSPs) and general-purpose processors has decreased. The innovations introduced in DSPs designed exclusively for performance are now implemented on DSPs targeted for hand-held devices where power consumption is a crucial design priority, both at the hardware and software design side. In the past, most implementations that were ported to these embedded platforms stayed mainly in the classic domain of signal processing and actively avoided algorithms that employ dynamic memory (DM from now on). 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 DM 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 [1].

In addition, power consumption has become a real issue in overall system design (both embedded and general-purpose) due to circuit reliability and packaging costs [2]. 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 implementation involved.

Current implementations of DM managers can provide a reasonable level of performance for general-purpose systems [3]. 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 currently face the need to manually optimize the implementations of the initial DM managers on a case-per-case basis. This has to happen 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 prove 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 (even if standardized languages such as C or C+ + offer considerable support).

In this paper, we present a new high-level programming and profiling approach (based on abstract derived classes or mixins [4] in C++) to implement complex custom DM managers from its basic parts (e.g. de/allocation strategies, order within pools, splitting, coalescing, etc.) [3] in a modular way 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. Moreover, the versatility of the C++ language allows to insert this new principle of extensive modular library inside real current embedded operating systems (OS), e.g. RTEMS [5], without the code size overhead found in other approaches that optimize managers in C code based on profiling [6,7]. This will be shown more in detail in Section 4 with our experimental results.

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 3.3 we explain the power estimation technique used within our high-level approach. 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

Currently the basis of an efficient DM management in a general-context are already well established. In the software community much literature is available about DM management implementations and policies to be used in general-purpose systems [3]. 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 [3]. Also, in recent real-time operating system synthesis approach for embedded systems [5], dynamic allocation is supported with custom DM managers based on region allocators [9] for the specific platform features.

Another recent method to refine the DM subsystem is to simulate the system with different customizable memory managers and infrastructures. In [7], a memory manager that allows to define multiple regions in memory with different disciplines for each of them is presented. However, this approach cannot be extended with new functionality and is limited to a small set of user-defined functions for memory de/allocation. Also, in [6], 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, [10] 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 [3]) for other metrics (e.g. power consumption).

Regarding profiling of the DM subsystem, work can be found to estimate power consumption based on software instead of at a circuit-level. Most of it uses an instruction-level analysis [11], but more recently research has been developed for assembly code and a higher abstraction level [2]. Nevertheless, current methods do not yet include run-time profiling analysis and hence are not sufficient for modern dynamic multimedia and network applications. In addition, due to large scale integration systems, several analytical and abstract power estimation models at the architecture-level have received more attention lately [12]. 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 modular low-power dynamic memory managers

The search space of DM manager implementations 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 [4] with template `C++` classes [13]. In the remainder of the text, we use the definition of mixins as used in [4]: a method of specifying extensions of a class without defining up-front which class exactly it extends. This approach allows easy and flexible modelling and refinement of layered DM managers, as we already mentioned in Section 2. This will be illustrated now more concretely.

In Fig. 1 we show the basic concepts used in this approach. In the first example of Fig. 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 Fig. 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

```
// 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++`.

approaches [3,6,10] this kind of modelling 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 implementation can be explored by the designer due to the time-consuming effort of reprogramming their global structures.

## *3.1. Constructing modular dynamic memory managers with profile support*

Using the previously explained concepts of abstract derived and template `C++` classes, we have redefined the library proposed in [10] and integrated our own profile framework (see Section 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 [1]. In Fig. 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 Fig. 2). Second, the two basic data structures to test within the manager, i.e. double linked lists (DLList) and binary trees

```
// Basic blocks for heap requests to the system
template<typename MyT>
        class BasicHeap: public TypeClass<MyT,mheap>;
// Data types of the DM manager
template<typename MyT, class SuperClass>
        class DLList {// 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 memory manager with profiling objects.

(BTTree) are implemented. Third, they are instantiated for the basic sizes to use in the DM manager. Then, the profile objects (see Section 3.2 for 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 [3], which are formed by different dynamic data structures inside (DLList or BTTree) and different allocation policies (best fit or first fit) [3].

As Fig. 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 [10]. 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 [10] 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 Fig. 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, Fig. 8 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 run-time profile framework

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

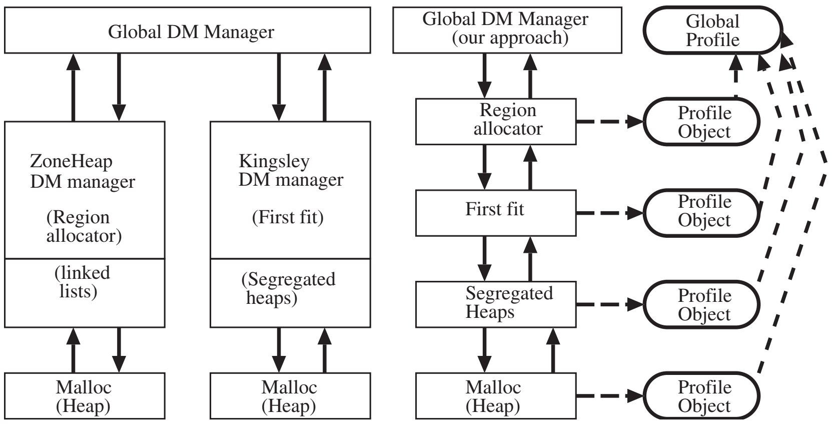


Fig. 3. Example of the structure of a custom DM manager. On the left, built with the approach proposed in [10] where the layers are really interdependent. On the right, our own approach.

profile their characteristics at different levels, e.g. memory accesses of each implementation layer in the internal structure, as Fig. 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 [14] for complex dynamic data types. As Figs. 3 and 2 (i.e. `_profile *prof1, *prof2, *profileGlobal`) depict, it consists of an objects-oriented profiling framework where independent objects are attached in a hierarchical way to profile each part of the DM managers. This way the profiling information is decoupled from the implementation class hierarchy of the DM managers, which provides more versatility to attach the profile objects to any subpart of the implementation of the managers and thus acquire accurate run-time information on memory accesses, memory footprint, timing information and method calls that can be stored. Finally, we can use this stored profiled information to obtain power model estimates for the DM managers using a realistic model of the underlying memory hierarchy in a post-execution phase. As a result of the whole process, 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 power estimation phase (see more details about it in Section 3.3).

### 3.3. Validated power model and post-execution power estimation phase

In the post-execution phase we estimate the power consumption of the system based on the stored profiling information, which contains the necessary information about memory accesses (i.e. read or write), maximal value of memory footprint consumed during execution and memory addresses of each access to the data types. We have observed that the most important part of the power consumed in DM management of the system (almost 80%) in these new multimedia applications really depends on the amount of memory accesses [1] and the total amount of memory footprint required by the system. In fact, the operations performed by the DM manager are very simple (e.g. check values or follow pointers) and the power consumption is mainly determined by memory accesses rather than the computation power. Thus, for this post-execution phase, in order to use the profiling information acquired by our profile framework (see Section 3.2 for more details), we have to accurately model the power consumed on the on-chip (i.e. SRAM) and off-chip (i.e. SDRAM) memories.

On the one hand, regarding the on-chip memories we use an updated version of the CACTI model [15], which is a complete energy/delay/area model for embedded SRAMs that depends on memory footprint factors (e.g. size, internal structure or leaks) and factors originated by memory accesses (e.g. number of accesses or technology node used). It has two main advantages. First, a clear hierarchy is present in the modelling of the different memory components

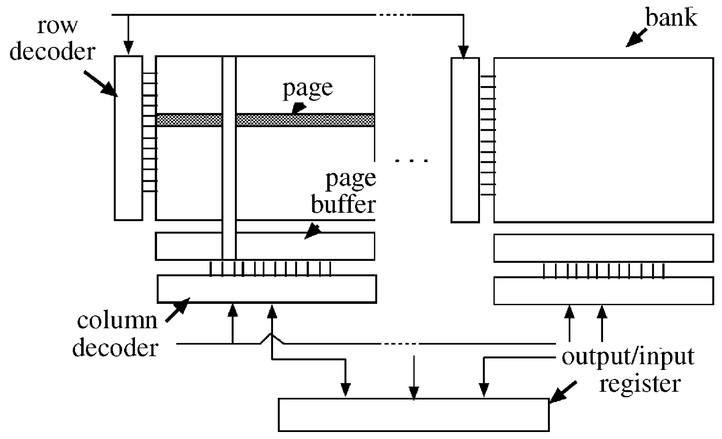


Fig. 4. Multi-banked SDRAM architecture.

Table 1
Energy consumption parameters for off-chip SDRAMs (.13 μm tech.)

| | |
|---|---|
| $E_{\mathrm{precharge/activate}}$ | 14000 pJ/miss |
| $E_{\mathrm{read/write}}$ | 2000 pJ/access |
| $P_{\mathrm{stby}}$ (for each bank) | 30 mW |

Table 2
Profiling values for different memory managers in the Deficit Round Robin application for a stream of 1,00,000 packets

| Memory manager | Memory accesses | Memory usage (B) | Energy (μJ) .13 μm tech | Execution time (s) |
|---|---|---|---|---|
| SegFitSLL FIFO | $2.00 \times 10^6$ | $2.09 \times 10^6$ | $13.28 \times 10^6$ | 115.04 |
| (on-chip values) | $0.25 \times 10^6$ | $16.38 \times 10^3$ | $49.60 \times 10^3$ | — |
| Total | $2.25 \times 10^6$ | $2.09 \times 10^6$ | $13.33 \times 10^6$ | 115.04 |
| Kings + LIFOSLL | $1.25 \times 10^6$ | $2.09 \times 10^6$ | $8.32 \times 10^6$ | 64.25 |
| (on-chip values) | $0.15 \times 10^6$ | $16.38 \times 10^3$ | $31.00 \times 10^3$ | — |
| Total | $1.40 \times 10^6$ | $2.09 \times 10^6$ | $8.32 \times 10^6$ | 64.25 |
| Kings + FIFODLL | $1.75 \times 10^6$ | $2.09 \times 10^6$ | $11.62 \times 10^6$ | 135.63 |
| (on-chip values) | $0.22 \times 10^6$ | $16.38 \times 10^3$ | $43.40 \times 10^3$ | — |
| Total | $1.97 \times 10^6$ | $2.09 \times 10^6$ | $11.66 \times 10^6$ | 135.63 |

at four different levels. The second main advantage of CACTI is the fact that it is scalable to different technology nodes. For the results shown in Figs. 5, 8, and Table 2, we use the .13 μm technology node.

On the other hand, regarding the off-chip memories, we use a simplified SDRAM memory model. A simplified view of a typical multi-banked SDRAM architecture is shown in Fig. 4. Fetching or storing data in an SDRAM involves three memory operations. An activation operation decodes the row address, selects the appropriate bank and moves a page/row to the page buffer of the corresponding bank. After a page is opened, a read/write operation moves data to/from the output pins of the SDRAM. Only one bank can use the output pins at the time. When the next read/write accesses hit in the same page, the memory controller does not need to activate the page again (a page hit). However, when another page is needed (a page miss), precharging the bank is needed first. Only thereafter the new page can be activated and the data can be read. All these previous features are included in our power estimation model and to this end the energy parameters have been derived from a power estimation tool of Micron 32 Mb/64 Mb mobile SDRAM [16]. They are shown in Table 1.

Using the previous explained energy consumption parameters about SDRAMs, our power model decomposes the energy consumption of off-chip memories in a static and a dynamic part, as indicated in the formulas below. First, the static energy consumption is due to the standby power of the SDRAM. In our model we consider that the memory is always in standby mode (i.e. $P_{\mathrm{stby}}$) when it is not accessing data. Therefore, we calculate a worst-case estimation by considering that 4 banks exist (average value in most SDRAMS [16] of the sizes we consider, e.g. 64 Mbits) and it is multiplied by the total execution time spent by the application (i.e. $t_{\mathrm{stby}}$). Second, the dynamic energy consumption is calculated considering the amount of memory accesses captured by our profile framework to the scratchpad memories and multiplied by the energy indicated in Table 1 for read/write operations ($E_{\mathrm{read/write}}$) in case of a page hit. In case it is a page miss, which is decided by checking in our stored profiling the addresses of each two consecutive memory accesses, we add the energy value of precharging/activating the new bank ($E_{\mathrm{precharge/activate}}$) to the energy spent in the access/write of the page buffer ($E_{\mathrm{read/write}}$). Thus, the energy consumption of the SDRAM is computed with the following formula:

$$\begin{aligned}
E_{\mathrm{totalSDRAM}} &= E_{\mathrm{static}} + E_{\mathrm{dynamic}},\\
E_{\mathrm{static}} &= 4.P_{\mathrm{stby}}.t_{\mathrm{stby}},\\
E_{\mathrm{dynamic}} &= N_{\mathrm{pa}}.E_{\mathrm{pa}} + N_{\mathrm{rw}}.E_{\mathrm{rw}},
\end{aligned}$$

where $E_{\mathrm{pa}}$ is the energy of a precharge/activation, $E_{\mathrm{rw}}$, the energy of a read/write, $t_{\mathrm{stby}}$, the execution time (or standby) of application, $N_{\mathrm{pa}}$, the number of precharge and activations of banks, $N_{\mathrm{rw}}$, the number of reads and writes in application.

Note that any other additional mode of SDRAM memories (i.e. clock-suspend or power down mode [16]) could be added later to the model (and static formula) thanks to our timing information of the stored profiling in case we need to have a finer-grained power consumption estimation. Moreover, any other model for a specific memory hierarchy can be used just by replacing the aforementioned power modules in the tools. Finally, in order to validate the results (e.g. memory accesses or memory footprint) of our high-level approach, we have used a complete cycle-accurate simulation platform [17]. This simulation platform is briefly described in Section 4, together with our experimental results.

## 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 with variations of less than 2%.

### *4.1. Method applied to a new 3D image reconstruction system*

The first case study forms one of the sub-algorithms of a 3D reconstruction algorithm [18] that works like 3D perception in living beings, where the relative displacement between several 2D projections is used to reconstruct the third dimension. The software module used as our driver application heavily uses DM and 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 [19] 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. This implementation matches corners [18] detected in two subsequent frames. The operations done on the images are particularly memory intensive, e.g. each image with a resolution of $640 \times 480$ uses over 1 Mb, and the accesses of the algorithm (in the order of millions of accesses) to the images are randomized. Thus, classic image access optimizations as row-dominated accesses versus column-wise accesses are not relevant to reduce the memory footprint, 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 [3] (`KingsLayered` in Fig. 5). But it has a considerable fragmentation due to its use of power-of-two segregated-fit lists [3]. A graphical representation of its implementation structure with our layered-approach is shown in Fig. 6. As Fig. 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 [3] frequently found in current embedded systems, i.e. RegAlloc in Fig. 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) [3], due to the limited range of data type sizes used in the application [14]. 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 Fig. 6 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 Fig. 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 Fig. 6 indicates at 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 Fig. 6 and marked as `KHierarchy` in Fig. 5.

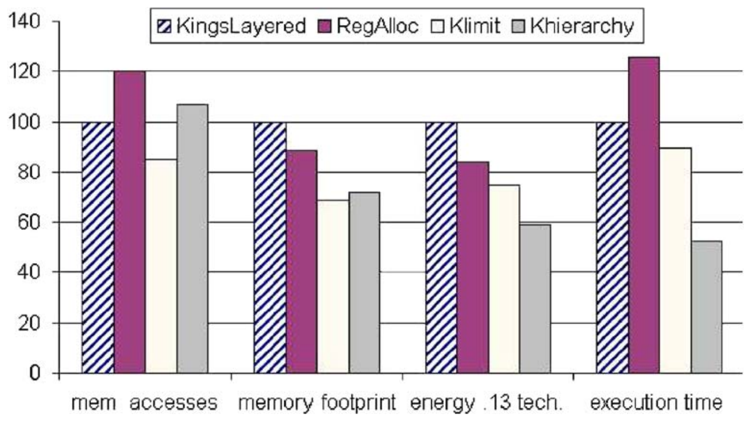


Fig. 5. Profiling results of different DM managers (normalized to Kingsley, i.e. `KingsLayered`) in the 3D Image Reconstruction System per each matching process of two frames.

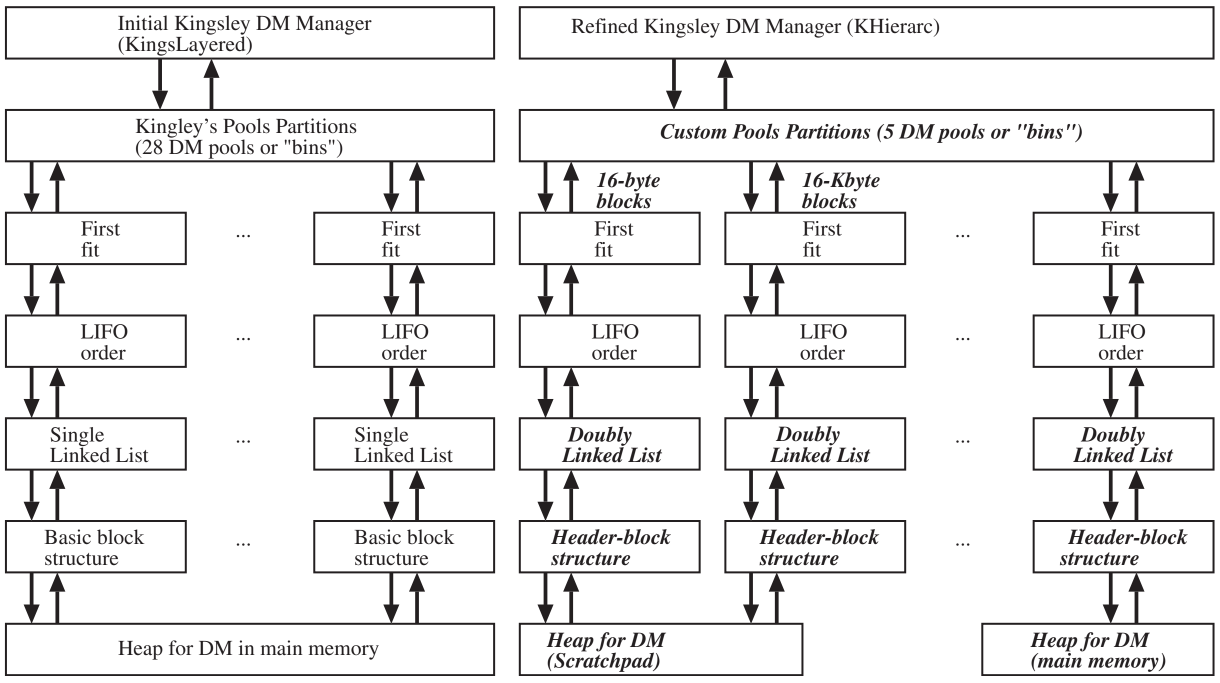

Initial Kingsley DM Manager (KingsLayered)
Kingley's Pools Partitions (28 DM pools or "bins")
First fit
LIFO order
Single Linked List
Basic block structure
Heap for DM in main memory
Refined Kingsley DM Manager (KHierarc)
Custom Pools Partitions (5 DM pools or "bins")
16-byte blocks
16-Kbyte blocks
First fit
LIFO order
Doubly Linked List
Header-block structure
Heap for DM (Scratchpad)
Heap for DM (main memory)

Fig. 6. 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`, with the main changes indicated in bold.

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 Fig. 5.

### *4.2. Method applied to a network scheduling application*

The third case study presented is the Deficit Round Robin (DRR) application taken from the NetBench benchmarking suite [20]. It is a scheduling algorithm implemented in many routers today. In the DRR algorithm, the scheduler visits each internal non-empty 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, 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. The DRR application was profiled in our results for an input trace of 1,00,000 packets.

For this case study, using our approach to evaluate different versions of the same basic structure in the memory manager, we have implemented and profiled managers of the basic segregated fit scheme, both general-purpose and custom DM managers. All of them are power-of-two segregated-fit lists [3], without coalescing or splitting services due to the hard real-time constraints of this network application. As in the previous case studies, for speed purposes we have considered as our basis the structure of the general-purpose Kingsley manager, but we have implemented two different variations of it. One uses a LIFO single linked freelist (Kings+LIFOSLL in Table 2) and the other one a FIFO double linked freelist (Kings+LIFODLL in Table 2). Finally, we have also designed a faster custom memory manager than Kingsley with FIFO double linked lists as a FIFO single linked list structure with segregated fit algorithm (SegFitSLL FIFO in Table 2). It shows 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 2. Note that Table 2 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.

In addition, we have tested several region [9] and stack-like managers [3], but they are not appropriate for this specific application since they would require huge amounts of memory (over 200 Mbytes) due to the special dynamic nature of the DRR algorithm (being executed for a long period with allocated packages that can remain alive for a long time). Thus, these kind of managers that require to know an upper bound value for the initial request of allocated memory for each region cannot be really used.

As Table 2 indicates, the memory footprint is the same for the managers because they are all power-of-two segregated-fit lists with the same internal organization. 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 is achieved by the manager with Kingsley basis and a LIFO single linked list structure. It is faster than the custom FIFO SLL segregated fit because it combines both organization schemes. Also, it is better than the LIFO structure because it can be updated faster than the FIFO organization, and in fact this pattern is less used by the router. Finally, it has also the lowest power consumption, because it has the least memory accesses. This is mainly 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 of the router, more packets are likely to arrive in a short period of time to the same queue (and very often 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.

As we have already mentioned in Section 3.2, we validate the system-level values obtained with our approach (i.e. Table 2) by integrating and simulating our C++ library in a complete System-On-Chip cycle-accurate simulation platform [17]. This simulation platform is described in SystemC [21], which provides the advantage of describing both hardware and software in a common language, namely C++. From the hardware point of view, this simulation platform first uses an instruction set processor that is a cycle-accurate ARM core written in C++ called SWARM [22]. It is integrated in the platform using a SystemC wrapper that creates a standard interface between this processor and the memory hierarchy. Instead of the ARM also other instruction set simulators could have been used after the appropriate integration work. Then, the simulation platform includes two memory hierarchies: an on-chip software controlled or scratchpad memory, and an off-chip main memory also written in SystemC. A global description of the whole simulation platform is shown in Fig. 7. From the software point of view, this simulation platform includes a complete port of an embedded OS, RTEMS [5] that allows to integrate in it our C++ library of layers to create custom DM managers. Hence, we can execute the DRR application on this simulation platform to verify that the values obtained with our system-level approach are accurate enough.

After the simulation with the ported version of our library in the cycle-accurate simulation platform, the results obtained only slightly vary from our high-level estimations: less than 6% on

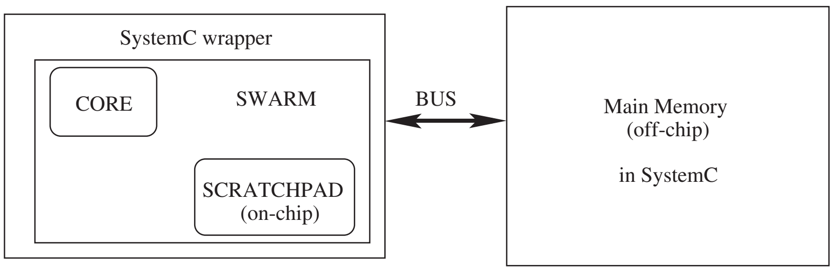


Fig. 7. Overview of the simulation platform used to validate our high-level approach.

average after a set of 10 simulations for each of the managers shown in Table 2. Moreover, the values obtained in the simulation platform scale in the same way for all the DM managers. Thus, the relative ranking and the conclusions achieved using our high-level approach about the features of each memory manager are perfectly valid, and the designer can estimate their behavior without a very time-consuming effort in tedious cycle-accurate simulations. In this example, our high-level system simulations and power consumption estimations took us 45 min in total whereas the cycle-accurate simulations 22 hours in a Pentium III at 800 MHz with 265 Mb and running GNU/Linux 2.4.20.

Finally, to evaluate the speed up of the implementation and refinement process of DM managers, we want to remark that the DM managers for this application constitute around 400 lines of C++ code each and took us approximately one week to build them, obtain detailed profiling of their basic components and eventually refine their implementations. Each allocator is composed out of 5 layers and because all of them are variations of segregated fit algorithms, 2 layers were reused in each implementation.

### *4.3. Method applied to a 3D rendering system*

The third case study is the 3D rendering module [23] of a whole 3D video system application. This module belongs to the new category of 3D algorithms with scalable meshes [24] to adapt the quality of each object displayed on the screen according to the position of the user watching at them at each moment on time to (e.g. Quality of Service systems [24]). For simplification purposes, we consider the scenario where only one object must be rendered on the screen while the user is moving the camera around it. This object is 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. First, those vertices are traversed in the first three phases of the whole visualization process, i.e. modelview transformation, lighting process and canonical view transformation [23]. Finally, the system processes the faces of the objects in the next three phases (i.e. clipping, viewport mapping and rasterization [23]) of the visualization process to show the final object with the appropriate resolution on the screen.

In this case, due to the variable memory sizes of the system, we have implemented and tested with our approach one of the best general-purpose DM managers (in terms of the combination of speed and memory footprint) [3,10], i.e. Lea Allocator v2.7.2 [3]. This is a hybrid allocator for general-purpose software design with different behavior for different object sizes. For small objects it uses some kind of quick lists [3], for medium-sized objects it performs approximate best-fit allocation [3] and for large objects it uses dedicated memory (allocated directly with the `mmap()` function). Apart from it, we have also used Kingsley [3] to compare both in memory footprint, memory accesses and total energy consumption figures. In addition, we have also implemented and profiled a well-known custom DM manager optimized for a stack-like DM behavior, i.e. Obstacks [3]. As Fig. 8 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 the complex maintenance structure of this manager. In addition, Fig. 8 indicates that Kingsley suffers from high fragmentation penalty, but produces a lot less accesses. As a result, although they show completely different characteristics, both managers are close in their final figures for power consumption. Also, due

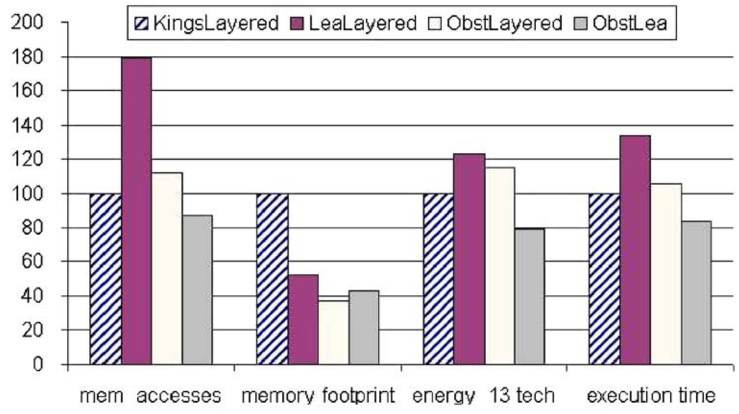


Fig. 8. Profiling results of different DM managers (normalized to Kingsley, i.e. `KingsLayered`) in the 3D Rendering System for one object in one frame.

to the stack-like application behavior with the triangles, Obstacks has a lower amount of accesses but its behavior 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 high penalty in memory accesses and energy dissipation per frame in these last three phases of the algorithm. Therefore, its final values are not as good as expected.

Nevertheless, these results suggests the convenience of a custom DM manager that combines the behaviour of Obstacks with Lea in the last three phases. We have used our approach to build it in a fast way (3 or 4 weeks, approximately) and it is marked as `ObstLea` in Fig. 8. This figure shows that this new manager accomplishes very good overall results.

In addition, our own DM manager designs have a similar execution time (differences of less than 10% on average in total execution time) compared to the original (manually designed) versions of Obstacks and Lea, but we observe a clear improvement in design complexity on our side. We have around 700 lines of C++ code for our version of the Lea Allocator instead of more than 20,000 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 library implementation.

## 5. Conclusions

Embedded devices have improved their capabilities in the last years, making it feasible to map very complex and dynamic applications (e.g. multimedia and wireless network) in portable devices. Such applications have grown lately in complexity- and demand-intensive DM requirements that must be heavily optimized (i.e. memory footprint, power and memory use) for an efficient mapping on current low-power embedded devices. System-level exploration and refinement methodologies have started to be proposed to consistently perform that refinement. Within this context, 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 a new system-level approach to characterize custom DM managers with an integrated power and memory footprint profiling method. This approach largely simplifies the complex engineering process of designing and profiling several implementation candidates, allowing the developers to cover a vast part of the implementation space (e.g. different strategies of the heap, internal blocks of the allocators, etc.) with a minimal programming and modelling effort. Furthermore, we have shown in our case studies that the profiling results obtained for power consumption, memory accesses and memory footprint are close to those obtained with real values obtained using much more time-consuming cycle-accurate simulations. They show the same relative ranking. Finally, it has been also shown how our approach can be integrated in a real current embedded operating system and that the instantiation process of this easy-to-compose approach leaves more freedom to new compilers for an efficient binary optimization, e.g. method in-lining. This results in managers that run as fast as their equivalent monolithic manually refined managers, but without the difficulties for maintenance that these last ones imply due to their complex code.

## References

[1] D. Atienza, S. Mamagkakis, F. Catthoor, J. Manual Mendias, D. Soudris, Dynamic memory management design methodology for reduced memory footprint in multimedia and wireless network applications, in: Proceedings of DATE '04, France, 2004.

[2] N. Vijaykrishnan, M. Kandemir, M.-J. Irwin, H. Suk Kim, W. Yw, D. Duarte, Evaluating integrated HW-SW optimization using a unified energy estimation framework, IEEE Trans. Comput., 2003.

[3] P.R. Wilson, M.S. Johnstone, M. Neely, D. Bowles, Dynamic storage allocation, a critical review, in: Workshop on Memory Management, UK, 1995.

[4] Y. Smaragdakis, D. Batory, Mixin layers: an object-oriented implementation technique for refinement and collaboration-based designs, Trans. SW Eng. Meth. 11 (2) (2002).

[5] Rtems, open-source real-time OS, 2002, http://www.rtems.com/.

[6] G. Attardi, T. Flagella, P. Iglio, A customizable memory management framework for c++, Software Practice Experience 28 (11) (1998).

[7] K.-P. Vo, Vmalloc: a general and efficient memory allocator, in: Software Practice and Experience, vol. 26, 1996.

[8] N. Murphy, Safe memory usage with dynamic memory allocation, Embedded Systems, 2000.

[9] D. Gay, A. Aiken, Mem. manag. with explicit regions, in: Proceedings of Programming Language Design and Implementation, USA, 2001.

[10] E.D. Berger, B.G. Zorn, K.S. McKinley, Composing high-performance memory allocators, in: Proceedings of Programming Language Design and Implementation, 2001.

[11] D. Sarta, D. Trifone, G. Ascia, A data dependent approach to instruction level power estimation, in: Proceedings of Workshop on Low-Power Design, Italy, 1999.

[12] R.Y. Chen, M.J. Irwin, Speed and Power Scaling of SRAM's, ACM Trans. Design Automation Electron. Systems 6 (1) (2001).

[13] D. Vandevoorde, N.M. Josuttis, C++ Templates, The Complete Guide, Addison-Wesley, UK, 2003.

[14] M. Leeman, D. Atienza, F. Catthoor, G. Deconinck, J. Manual Mendias, V. De Floria, R. Lauwereins, Power estimation approach of dynamic data storage on a HW SW boundary level, in: Proceedings of PATMOS, Italy, 2003.

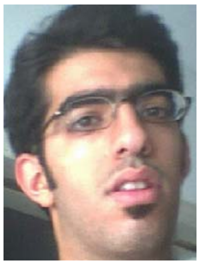
[15] N. Jouppi, Western research lab., cacti, 2002, http://research.compaq.com/wrl/people/jouppi/CACTI.html.

[16] Micron Technology, SRAM and SDRAM Products, http://www.Micron.com.

[17] F. Poletti, D. Bertozzi, L. Benini, A. Bogliolo, Performance analysis of arbitration policies for SoC communication architectures, Integration, VLSI J. 2003.

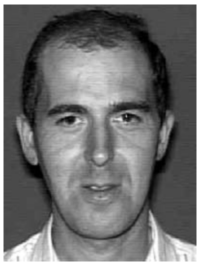
[18] M. Pollefeys, R. Koch, M. Vergauwen, L. Van Gool, Metric 3D surface reconstruction from uncalibrated image sequence, in: Lecture Notes in Computer Science, Springer, Berlin, 1998.
[19] Target jr., 2002, **http://www.targetjr.org**.
[20] G. Memik, W.H. Mangione-Smith, W. Hu, Netbench: a benchmarking suite for network processors, CARES Technical Report 2001-2-01, 2001.
[21] Systemc community, 2003, **http://www.systemc.org**.
[22] M. Dales, Swarm: software arm, 2003, **http://www.dcs.gla.ac.uk/michael/phd/swarm.html**.
[23] M. Woo, J. Neider, T. Davis, OpenGL Programming Guide, second ed., Silicon Graphics, 1997.
[24] D. Luebke, M. Reddy, J. Cohen, A. Varshney, B. Watson, R. Huebner, Level of Detail for 3D Graphics, Morgan-Kaufmann, Los Altos, CA, 2002.

DAVID ATIENZA: received the M.Sc. degree in Computer Sciences from the Complutense University of Madrid, Spain in 2001. Since then he has joined the Department of Computer Architecture and Automation of Complutense University of Madrid as a Ph.D. student. His research interests include optimization of dynamic memory management on multimedia applications for low power and high performance, computer architecture and high-level design automation.



STYLIANOS MAMAGKAKIS: received his Diploma in Electrical and Computer Engineering from the Democritus University of Thrace, Greece, in 2002. He is currently a post graduate student in the VLSI Design and Testing Center in the Demokritus University of Thrace. His research interests include optimization of dynamic memory management on wireless network applications for low power and high performance, embedded systems and high-level design optimizations.

FRANCESCO POLETTI: was born in Imola, Italy, on September 7, 1977. On March 2003 he received the degree in Informatical Engineering from the University of Bologna, Italy, with a thesis on "Software Developing for a Multiprocessor System on Chip". His research interests include low-power applications and design of portable systems, with a particular focus on multiprocessor System-On-chip and memory management.



FRANCKY CATTHOOR: is a fellow at IMEC, Heverlee, Belgium. He received the Eng. degree and a Ph.D. in El. Eng. from the K.U.Leuven, Belgium in 1982 and 1987 respectively. Since 1987, he has headed research domains in the area of architectural and system-level synthesis methodologies, within the DESICS (formerly VSDM) division at IMEC. His current research activities belong to the field of architecture design methods and system-level exploration for power and memory footprint within real-time constraints, oriented towards data storage management, global data transfer optimization and concurrency exploitation. Platforms that contain both customizable/configurable architectures and (parallel) programmable instruction-set processors are targeted.

JOSE M. MENDIAS: received the M.Sc. and Ph.D. degrees in physics from the Complutense University of Madrid in 1992 and 1998, respectively. He joined the Department of Computer Architecture and Systems Engineering, Complutense University in 1992 as a lecturer, and became an associate professor in 2001. Since 2002, he is Vice-dean of the Computer Science Faculty at the same University. His current research interests include design automation, computer architecture and formal methods.

LUCA BENINI: Luca Benini is an Associate Professor at the Department of Electrical Engineering and Computer Science (DEIS) of the University of Bologna. He received a Ph.D. degree in electrical engineering from Stanford University in 1997. He also holds visiting researcher positions at Stanford University and the Hewlett-Packard Laboratories, Palo Alto, CA. Dr. Benini's research interests are in all aspects of computer-aided design of digital circuits, with special emphasis on low-power applications, and in the design of portable systems. On these topics he has published more than 200 papers in international journals and conferences and two books. He is a member of the organizing committee of the International Symposium on Low Power Design and of the Design Automation and Test in Europe Conference. He is a member of the technical program committee of several technical conferences, including the Design Automation Conference, International Symposium on Low Power Design, the Symposium on Hardware-Software Codesign.

DIMITRIOS SOUDRIS: received his Diploma in Electrical Engineering from the University of Patras, Greece, in 1987. He received the Ph.D. Degree from in Electrical Engineering, from the University of Patras in 1992. He is currently working as Assistant Professor in Dept. of Electrical and Computer Engineering, Democritus University of Thrace, Greece. His research interests include low power design, Memory Management for Multimedia Applications, Parallel Architectures, Computer Arithmetic, and vlsi signal processing.He was the General Chair of the Ninth International Workshop Power and Timing, Modeling, Optimization, Simulation (PATMOS '99). Recently, he received an award from Intel and IBM for the Low Power Design project LPGD ESPRIT IV #25256. He is also a member of IEEE and ACM.