CUSTOM DESIGN OF MULTI-LEVEL DYNAMIC MEMORY MANAGEMENT SUBSYSTEM FOR EMBEDDED SYSTEMS

S. Mamagkakis[†], *D. Atienza*[⋆], *C. Poucet*[‡], *F. Catthoor*[‡], *D. Soudris*[†], *J. M. Mendias*[⋆*]

[⋆] DACYA/UCM Avda. Complutense s/n, 28040, Madrid, Spain
[†] VLSI Center-Demokritus Univ., Thrace, 67100 Xanthi, Greece
[‡] IMEC, Kapeldreef 75, 3001 Heverlee, Belgium

ABSTRACT

Modern embedded systems have to run new dynamic wireless network and multimedia applications. As a result, these systems must provide run-time memory management support to allow real-time memory de/allocation, retrieving and processing of data while very limited power supply is available. Thus, its implementation must be designed to combine high speed access, low power and large data storage capacity. This is only possible by an efficient use of the memory hierarchy available in the embedded systems. In this paper, we propose a new approach to design convenient dynamic memory management subsystems making profit of the multiple memory levels. It analyzes the logical phases involved in modern dynamic applications to effectively distribute the dynamically allocated data among the multi-level memory hierarchies present in embedded devices. We assess the effectiveness of the proposed approach for three representative real-life case studies of the new dynamic application domains (i.e., network and 3D rendering applications) ported to embedded systems. The results accomplished with our approach show a very significant reduction in energy consumption (up to 40%) over state-of-the-art solutions for dynamic memory management on embedded systems with typical cache-main memory architectures while respecting the real-time requirements of these applications.

1. INTRODUCTION

Over the last few years, the main focus of the design of embedded systems has been to provide good performance and at the same time achieve low power consumption. To achieve optimal results, a good coordination between hardware and software is required. Therefore, memory-intensive applications running on embedded platforms (e.g., multimedia) must be closely linked to the underlying OS and hardware. Furthermore, new embedded applications heavily rely on dynamically allocated data due to their variable input (e.g., stream of arriving packets). This constitutes one of the most difficult design challenges when mapping them on low-power and high-speed embedded processors that are often not equipped with extensive hardware and OS support for Dynamic Memory (DM from now on) management. However, this DM management subsystem must provide efficient memory de/allocation and retrieving of the dynamically allocated data in embedded systems by combining speed, low power and large data storage. Therefore, the design and implementation of the DM management subsystem in wireless network and multimedia applications is a major design bottleneck, which requires highly customised solutions at the OS level handling the DM subsystem (i.e., DM managers) to achieve the required memory footprint, low power consumption and real-time requirements [1] .

Additionally, it is common practice to develop embedded platforms with extensive use of on-chip memory subsystems (i.e., caches and scratchpads) to improve the performance of new demanding applications. However, the existing solutions for the implementation of the DM management mechanisms [2] only consider the main memory organization for reducing memory-accesses. They do not consider their effect on the underlying on-chip memory subsystem and thus do not profit from it. As a result, most optimization work for embedded systems is focused on the effect of static data allocation (decided at compile-time [3, 4]), completely ignoring the dynamic nature of the latest embedded applications, which include extensive run-time changes (e.g., internet traffic, user behavior).

In this paper, we propose a new approach to suitably design and make use of the available on-chip memory subsystem for DM management mechanisms in modern dynamic applications (e.g., networking and multimedia) considering two representative memory architectures for current embedded systems (i.e., cache+main memory and scratchpad+main memory+DMA). Our method takes into account the typical dynamic behavior of new dynamic embedded applications to efficiently use these memory architectures for DM management and achieve significant gains in energy consumption while respecting the required level of performance.

*Work 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.

The remainder of this paper is organized as follows. In Section 2, we give an overview of related work. In Section 3, we present the memory architectures that are available for the DM subsystem and compared in our experiments. In Section 4, we present the proposed approach to analyze the dynamic behavior of new embedded applications and how DM management can efficiently use the on-chip memory subsystem. In Section 5, we describe how the energy consumption of the memory subsystem is modelled in our approach. In Section 6, our real-life case studies and the experimental results using our proposed approach are presented. Finally, in Section 7, we draw our conclusions.

2. RELATED WORK

In the embedded systems domain, DM management techniques [2] have not received much attention, and mainly methods for optimizing static allocation of data among different cache memories and non-cached scratchpad memories have been proposed [3, 4]. In [5], an algorithm based on profiling information is presented to find which segments of the linked executable should be mapped in the scratchpad. [6] describes how to partition large data structures. The tiles of the original data structures can then be mapped onto the scratchpad memory, but require data transfers between the scratchpad and the main memory.

The main limitation of the above design-time techniques is that they cannot cope with dynamic applications where it is only known at run-time which data needs to be assigned. [7] decides at design-time which global and stack variables could be assigned to scratchpad memories and inserts the necessary code in the application to explicitly transfer them from main memory to the scratchpad at run-time. However, this work explicitly mentions that heap data is not considered since its lifetime is unpredictable at compile-time.

As opposed to the previous compiler-based techniques we are not aware of any related work that evaluates and makes use of the detailed run-time information available in DM managers to avoid adding additional code and to efficiently make use of the entire memory hierarchy (e.g., caches, scratchpad) to allocate heap data (i.e., dynamically allocated data in the heap) at run-time. Our own previous work that comes closest to the current is [8], but scratchpads were only considered for handling the same global pools that were defined in the main memory. In this paper we extend our previous work by analyzing the main dynamic phases of each Dynamic Data Types (or DDTs from now on) and by redefining the initial main memory pools in smaller pools that can be mapped onto the on-chip memory hierarchy of the system.

3. MEMORY ARCHITECTURES

In this section, we describe the two memory architectures we consider for the DM management of embedded systems.

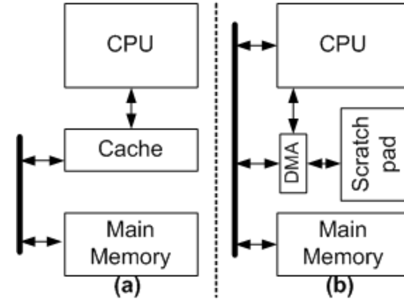

Fig. 1. Studied memory architectures for the DM subsystem of embedded systems. (a) Typical cache-main memory architecture and (b) scratchpad-main memory architecture

3.1. Typical Processor-Cache Memory Architecture

In modern embedded systems, due to their energy-efficient nature, the most typical memory architecture available consists of several memory levels (i.e., on-chip and off-chip memories) to try to make use of the locality of memory accesses in real-life applications. As a result, hardware-controlled caches try to map all of the most frequently accessed locations of main memories in smaller on-chip memories very close to the processor. The energy cost in accessing and communicating with these cache memories is much smaller than that required for fetching/storing information from/into large background memories. In the first memory architecture (marked as option `a` in Figure 1), we consider one level of on-chip hardware-controlled cache (i.e., L1) and the main memory for our DM management subsystem. With this architecture, all the accesses to the data pass through the cache. Hence, when the microprocessor requests the access to certain dynamic data (both the dynamic data generated in the application and the maintenance dynamic data structures of the DM managers), the hardware takes care of transparently moving the data from the main memory to the cache. After this, the processor accesses the data from the cache memory and any later access to the same data is fetched from the hardware-controlled cache.

3.2. Processor-Scratchpad Memory Architecture

In the second type of architecture (labeled as `b` in Figure 1), we have software controlled on-chip SRAM memories (i.e., scratchpad memories) for the DM management subsystem instead of on-chip hardware controlled caches. To efficiently transfer data between main memory and the scratchpad, which is rather costly in just a line-based copy scheme, we consider extra available hardware, i.e., a Direct Memory Access controller (DMA) [8] (see Figure 1), to be able to transfer also bursts of data (i.e., several lines in the same copy process). If the DMA is not programmed for a certain transfer of data, then the microprocessor directly fetches the dynamic data from the main memory using the global bus of the system (i.e., bypassing the scratchpad).

3.3. Possible Trade-Offs Between Both Architectures

Initially, we will consider only data that is read multiple times. The replacement of hardware-controlled caches by scratchpad memories into the memory hierarchy comes from the observation that caches do not represent the most energy-efficient hardware choice for a "local" memory. Hardware-controlled caches heavily simplify the exploitation of the locality of memory accesses present in actual programs by automatically controlling the complex process of moving data between main memories and cache memories. However, this automatic handling of access locality exploitation has a major energy cost compared to scratchpad memories. When accessing a cache line, we need to access and compare its tag to the incoming address. These common operations consume non-negligible energy in standard hardware-controlled cache architecture.

On the other hand, even if the scratchpad memories include the extra hardware (i.e., DMA) to allow to transfer their stored data in bursts as we propose in the second memory architecture, the cache+main memory architecture could still consume less energy than the scratchpad+main memory architecture. In the case that very complex and variable sizes of data need to be transferred at run-time, scratchpads will transfer a larger amount of data compared to hardware-controlled caches. This is due to the fact that the analysis to decide how much data to fetch in each DMA transfer for modern embedded applications is not perfectly predictable at compile-time and then the precise amount of dynamic data cannot be always transferred. Therefore, two measures must be taken to overcome this lack of precision in the transferred data. The first measure states that a larger amount of data must be transferred by the DMA compared to the cache and thus more energy is consumed in the scratchpad. In the second measure, a lower bound of amount of the real needed data must be fetched. In this case, the additionally required dynamic data needs to be fetched from the main memory using the global bus (see Figure 1, part `b`) instead of using the scratchpad and the local bus, which is much more expensive than fetching the data from the cache. As a consequence, the total consumed energy can be higher than using the cache+main memory architecture.

As a result, trade-offs concerning the overall energy consumption of the system can be envisaged between the two types of memory architectures available for the DM management subsystem due to the initially unpredictable dynamic behaviour of new embedded applications (e.g., wireless protocols and 3D image processing).

4. EFFICIENT USE OF THE MEMORY HIERARCHY FOR DM MANAGEMENT

On the one hand, static embedded applications usually have a good spatial and temporal locality, which can be exploited by processor-cache based systems [9]. This is because the static data is usually allocated within a certain scope (spatial locality) at the beginning of the application. It is later consumed at run-time (temporal locality).

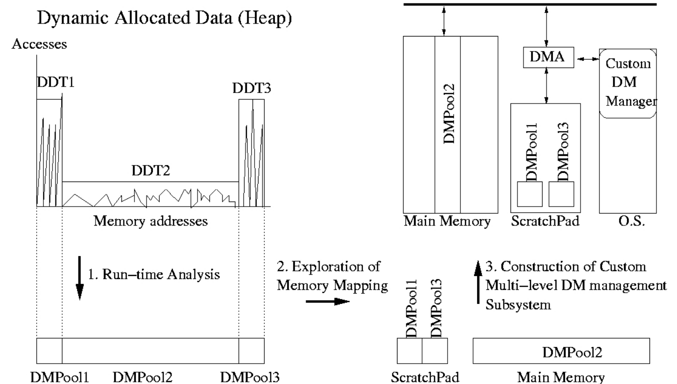

Fig. 2. Overview of the three main phases of our approach.

On the other hand, as we show in our experimental results (Section 6), dynamic embedded applications allocate their data in irregular time intervals and with more complex patterns. This partially destroys the available locality and thus reduces efficient exploitation of the cache memory hierarchy. In modern dynamic embedded applications, different DDTs are allocated at run-time. Then, before the embedded processor needs to access the data stored in the cache memory for a second time, a good chance exists that the data needs to be replaced by other dynamic data, thereby constituting a cache miss. Hence, the main memory has to be accessed again and the data re-fetched to the cache memory. The abundance of cache misses in new dynamic applications causes the advantages of cache memories for the dynamic data to be lost.

To overcome the negative effect of the dynamic nature of the applications in traditional memory hierarchies (i.e., caches+main memories) we propose a new approach that looks beyond the superficial randomness of the total amount of allocated data and accesses to the DDTs. It consists of three main phases as depicted in Figure 2. The first phase of our approach (number 1 in Figure 2) is an analysis of the run-time de/allocation and access behavior (i.e., reads and writes) of each DDT in the application under study. This is possible thanks to the use of our previous work, namely profiling tools for DDTs [10] and the exploration of custom DM managers [1]. The analysis of the first phase of our approach reveals the spatial (also temporal) locality of the DDTs in the dynamic application, which are mainly linked to the logical phases of the algorithms involved in the application (see Section 6 for real examples). It also allows us to discern how the de/allocation operations of the DM managers are interlaced with the usual access to DDTs and how the operations of DM managers can affect the locality of the accesses to the DDTs.

The previous analysis enables our approach to define

smaller memory pools for specific DDTs according to their degree of activity at each instant of the execution (number 2 in Figure 2). In our approach the level of activity in each pool is measured by dividing the amount of accesses yielding in the pool by the size of this pool.

Finally in the third phase of our approach, we select the most promising candidates of the previous exploration for memory mapping of DDTs onto on-chip memory pools (number 3 in Figure 2). The main parameter to consider within this memory mapping of smaller pools is the set of sizes (e.g., 4KB, 8KB) available of the on-chip memories of the embedded platforms. In our approach, as we show in Section 6, to be able to perform that on-chip mapping we need to use the scratchpad+main memory architecture presented in Section 3. The multi-level custom DM manager then controls the transfers of the selected smaller pools of DDTs between the scratchpad and the main memory (see Figure 2). In this way, the DDTs of the other memory pools not copied using the DMA to the scratchpad (by the DM manager), bypass the scratchpad and are fetched from main memory, so that they do not spoil the locality present in the dynamic data of on-chip memories.

5. ENERGY CONSUMPTION MODELING IN MEMORY ARCHITECTURES

To estimate the energy consumed by the dynamic data management system in the two different platforms, we combine a number of profiling tools and simulators in the global simulation environment for our approach. First, we compute the read and write accesses to the cache, scratchpad and main memory with the profiling data of: (i) the DDTs and (ii) the dynamic memory management mechanism as we explained in our previous work, i.e., [10] and [1] respectively. Next, to evaluate the cache hits and misses we use the very well known Dinero IV cache simulator [11]. After this, we estimate the energy per read and write access for the different required cache, scratchpad and main memory sizes of each case study using an updated version of the Cacti model [12]. This 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). For our calculations we use the $.13\mu$m technology node. Finally, we estimate the total energy consumed in the memory system for each execution and type of memory hierarchy based on the write and read accesses obtained in the previous simulation phases for each of the parts of the memory subsystem (e.g., main memory, scratchpad, caches). This model takes into account the switching, but not the static leakage of the memories involved.

Note that in our final results (see Figure 5) we present two different sets of energy figures. First, using Cacti we consider that scratchpad, cache and main memories are on-chip memories of different sizes using SRAMs memories. Second, because these on-chip memories are far too big (i.e., more than 1MB) we also provide the data when the main memory is off-chip. Since energy has become an issue due to the circuit reliability and packaging even in devices that are not dependent on batteries [13], we then also show the remaining on-chip energy without the main-memory (see Section 5).

6. CASE STUDIES AND EXPERIMENTAL RESULTS

We have evaluated the proposed approach in three real-life case studies that represent typical domains of modern multimedia and network application domains: a scalable 3D rendering system, a scheduling algorithm from the network domain and a new 3D image reconstruction application. All the results shown are average values after a set of 10 simulations for each application and memory hierarchy where all these results were very similar, with variations of less than 3%. Although, we could not evaluate the latency of each DM manager, our approach of creating custom DM managers does not affect the execution time of the application itself.

Our first case study is a new 3D video rendering application based on scalable meshes [14] that adapts the quality of each object displayed on the screen according to the position of the user. The objects are internally represented by vertices and faces (or triangles). They need to be dynamically managed in the meshing algorithm and corresponding complex data structure due to the uncertainty at compile time of the features of the objects to render. This complex dynamic data structure consists of a dynamically created tree where vertices and faces are stored separately. This data structure needs to be traversed according to different access patterns (i.e., the different rendering phases [14]) to render them onto the screen. First, the vertices are traversed during the first three phases of the whole visualization process. Then, the faces are processed in the final three phases [14] of the rendering process to show the objects with the appropriate resolution on the screen.

First we apply our approach to obtain a detailed analysis of the logical phases inside the application, the accesses (i.e., reads and writes) in each of them to the DDTs (i.e., vertices and faces) and their de/allocation access pattern. As a result of our analysis, we observe that in each phase of the rendering process the vertices and phases are accessed in small groups (never more than 15 points) and the accesses to these vertices are interlaced with the allocation requests to the DM manager. Therefore, although we define a custom DM manager that divides the heap in smaller subpools perfectly adjusted for each of the main allocated sizes (i.e., 6 sizes) to minimize the traversal inside the heap [1] and the memory fragmentation, the results obtained for energy consumption with the more traditional cache+main memory are not as good as expected (see Figure 3 and Figure 5).

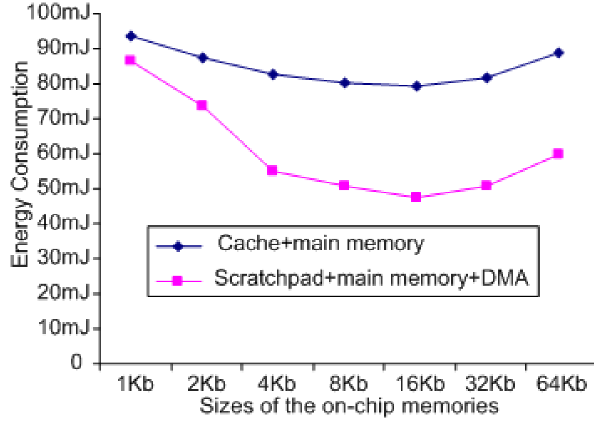

Fig. 3. Energy comparisons in the 3D rendering application for different on-chip memory sizes between the cache-based architecture and our scratchpad+main memory approach

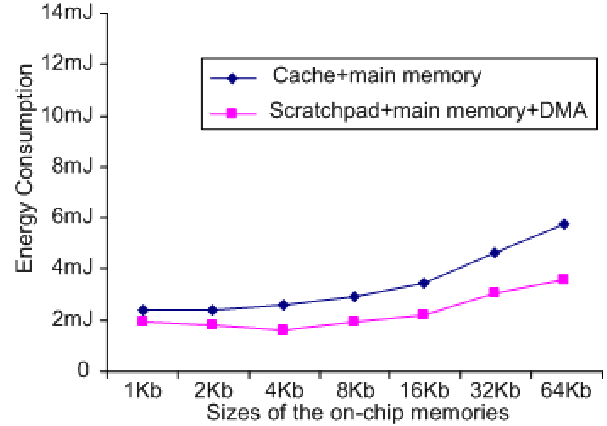

Fig. 4. Energy comparisons in the DRR applications for different on-chip memory sizes of the cache-based architecture and our scratchpad+main memory approach

The main reason for this is the interlaced accesses to the data, which are present if the cache is shared between the internal maintenance structures of the DM manager, and the real dynamic data generated in the application. The maintenance structures (up to 16KB in the worst case) include tables of pointers in memory to each of the subpools, the headers of the allocated/free memory blocks, etc. The application dynamic data include new vertices and results of the rendering process. Thus, not enough data access reuse is present due to the interlaced accesses (i.e., DM manager and access to DDTs) to successfully exploit the cache+main memory architecture.

Next, we evaluate the scratchpad+DMA+main memory combination. In this case, we permanently map the internal maintenance structures of our custom DM manager in the on-chip scratchpad memory available in the system. In addition, we use the knowledge of the DM manager about the position of the dynamic data in the pools to program the DMA and move the necessary set of vertices that are going to be process next (i.e., always less than 8 KB) to the available space in the scratchpad memory.

As a result, with this custom multi-layered DM manager (correctly supported by the use of a scratchpad+DMA+main memory architecture), the final energy consumed in the system improves by up to 40% (see Figure 3 and Figure 5) compared to the best results obtained with a custom DM manager and a more traditional memory architecture (i.e., cache+main memory).

The second case study presented is the Deficit Round Robin (DRR) application taken from the NetBench benchmarking suite [15]. It is a buffering and scheduling algorithm implemented in many wireless network routers today. Using the DRR algorithm the router tries to accomplish a fair scheduling by allowing the same amount of data to be passed and sent from each internal queue. It requires the use of DM because the input can vary enormously depending on the network traffic. The DRR application was profiled in our results for an input trace of 10,000 packets. A buffering-scheduling algorithm was chosen as a case study because these algorithms have the biggest demands in terms of power consumption in the networking domain [16].

The DRR algorithm has 3 types of dynamic data: the internet packets, the packet headers and the list that accommodates the internal queues. It works in 3 phases when it is receiving packets. First, it checks the packet header; secondly it traverses the list of internal queues and, finally, it traverses the correct internal queue and stores the internet packet in a FIFO order. Then, it forwards the packets in 3 additional phases. First, it traverses the list of internal queues, secondly, it picks up the first internet packet and, finally, checks the packet header to forward it.

We have run the algorithm in both types of memory hierarchies (see Figure 4). Our results show that the cache-based memory architecture does not achieve good results for energy due to the lack of locality in accesses to the DDTs and the de/allocation requests to the DM manager. Therefore, after analyzing the sub-phases of DDTs and DM manager with our approach, in the scratchpad+DMA-based memory architecture we decide to map the list of internal queues on the scratchpad and bypass it when accessing the other 2 DDTs from the main memory. This choice was based on the dynamic sub phases found with our approach in the algorithm, which indicate that the list of internal queues has 85% of the total accesses on average and that this list is small enough to fit inside a small-sized scratchpad (i.e., 4 KB or 8 KB). Then, after comparing the energy consumption for different sizes of caches and scratchpads (see Figure 4), we have come to the conclusion that a 17% lower energy consumption can be achieved with our scratchpad-based platform and a custom multi-level DM manager. This reduction in energy consumption can be obtained with the

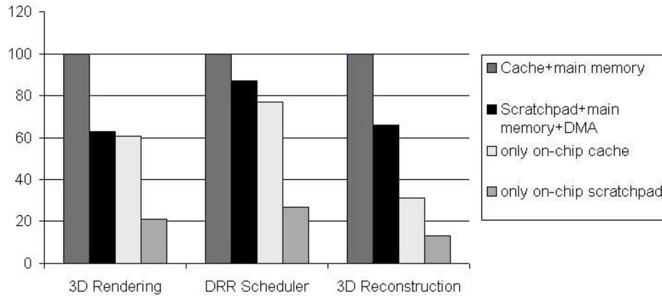

Fig. 5. Energy results in our real-life case studies. Values normalized to the cache+main memory figures

use of the 4 KB-sized-scratchpad (or 8 KB in the extreme cases of input streams of packets), which is exactly the size needed to accommodate the whole list of internal queues.

A third case study was used to extend our method to the new domain of 3D image reconstruction applications [17] for modern embedded systems, and similar results were obtained. This metric 3D-reconstruction from video [17] allows the reconstruction of 3D scenes from images and is characterized by intensive internal DM use. After applying our approach as in the other case studies, a similar energy-reduction-rate (34%) was achieved (See Figure 5).

In summary, as Figure 5 indicates, our approach (multi-level DM manager with scratchpad-based memory architecture) can achieve significant total energy consumption reductions (gains up to 40%) compared to the more usual cache+main memory architecture for state-of-the-art DM management in embedded systems. Furthermore, the used Cacti model overestimates the contribution of large memories [18], so in reality our total gains are even larger, namely closer to the on-chip gains. These on-chip gains for each case study, shown in Figure 5, are the results of comparing the bars related to only on-chip cache and only scratchpad memory values. They show that with our approach using scratchpad memories we achieve on-chip energy reductions of a factor up to 3 compared to the energy consumed in the cache. In this paper, we did not consider the energy consumption of the data bus and the CPU. We also did not explore architectures with multiple caches or multiple scratchpads, which can exist in single or multiple processor embedded systems. In the future we plan to extend our approach to address these issues and improve the total amount of energy consumed by modern applications.

7. CONCLUSIONS

Embedded devices have improved their capabilities in the last years making it feasible to map dynamic applications (e.g., multimedia) in portable devices. In this paper, we have shown that cache-based approaches are not well-suited to the inherent dynamic nature of these applications. We have presented a new approach that analyzes the initially disordered dynamic behavior of these applications and designs a suitable multi-level custom DM manager that can achieve significant energy savings using a scratchpad+DMA-based memory architecture. Our experimental results in real life case studies show a reduction of up to 40% in overall energy consumption of the dynamic memory subsystem when all memories in the system are on-chip, and up to 3 times in the on-chip memory subsystem when only caches and scratchpads are on-chip and the main memory is off-chip.

8. REFERENCES

[1] D. Atienza, et al., "DM management design methodology for reduced mem. footprint in multimedia and wireless network apps," in *Proc. of DATE '04*, 2004.

[2] Paul R. Wilson, et al. "Dynamic storage allocation, a survey and critical review," in *Springer Verlang LNCS*, 1995.

[3] O. Avissar, et al., "Heterogeneous mem. management for embedded systems," in *Proc. of CASES*, 2001.

[4] L. Benini, et al., "Increasing energy efficiency of emb. systems by app.-specific mem. hierarchy generation," *D&T Computers*, 2000.

[5] F. Angiolini, et al., "Polynomial-time algorithm for on-chip scratch-pad mem. partitioning," in *Proc. of CASES*, 2003.

[6] M. Kandemir, et al., "Dynamic manag. of scratch-pad memory space," in *Proc. of DAC*, 2001.

[7] S. Udayakumaran et al., "Compiler-decided DM allocation for scratch-pad based embedded systems," in *Proc. of CASES*, 2003.

[8] F. Poletti, et al., "An integrated HW/SW approach for run-time scratch-management," in *Accepted for Proc. of DAC*, June 2004.

[9] F. Catthoor, et al., *Custom Mem. Management Methodology – Exploration of Memory Organisation for Embedded Multimedia System Design*, Kluwer Academic Publishers, USA, 1998.

[10] M. Leeman, et al., "Power estimation approach of dynamic data storage on a HW-SW boundary level," in *Proc. of PATMOS*, 2003.

[11] J. Edler et al., DineroIV trace-driven cache simulator, `http://www.cs.wisc.edu/~markhill/DineroIV/`.

[12] N. Jouppi, Western Res. Lab., Cacti, 2002, `http://research.compaq.com/wrl/people/jouppi/CACTI.html`.

[13] N. Vijaykrishnan, et al., "Evaluating integrated HW-SW optimizations using a unified energy estimation framework," *IEEE Trans. Computers*, 2003.

[14] D. Luebke, et al., *Level of Detail for 3D Graphics*, Morgan-Kaufmann Publishers, 2002.

[15] G. Memik, et al., "Netbench: A benchmarking suite for network processors," CARES Tech Report 2001-2-01, 2001.

[16] S. Mamagkakis et al., "D9: Data-types, control and data flow structures of telecom network applications," `http://easy.intranet.gr/Public_deliverables.htm`.

[17] M. Pollefeys, et al., "Metric 3D surface reconstruction from uncalibrated image sequences," in *Springer-Verlag LNCS*, 1998.

[18] B. S. Amrutur et al., "Speed and Power Scaling of SRAM's," *IEEE Trans. on Solid-State Circuits*, February 2000.