Power-Aware Tuning of Dynamic Memory Management for Embedded Real-Time Multimedia Applications

***Abstract*— In the near future, portable embedded devices must run multimedia applications with enormous computational requirements at low energy consumption. These applications demand extensive memory footprint and must rely on dynamic memory due to the unpredictability of input data (e.g. 3D streams features) and system behaviour (e.g. variable number of applications running concurrently). Within this context, the dynamic memory subsystem is one of the main sources of power consumption and embedded systems have very limited batteries to provide efficient general-purpose dynamic memory management. As a result, consistent design methodologies that can tackle efficiently the complex dynamic memory behaviour of these new applications for low power embedded systems are in great need. In this paper we propose a step-wise system-level approach that allows the design of platform-specific dynamic memory management mechanisms with low power consumption for such kind of dynamic applications. The experimental results in real-life case studies show that our approach improves power consumption up to 89% over current state-of-the-art dynamic memory managers for complex applications.**

***Index Terms*— real-time systems and embedded systems, dynamic storage management, multimedia applications**

I. Introduction

Nowadays, complex applications (e.g. MPEG21) implemented before in devices designed exclusively for performance can be ported on embedded devices where the power consumption is a crucial design priority, both at the hardware and software design side. Traditionally, embedded processing was limited to relatively simple algorithms executed on static data blocks. It actively avoided algorithms that employ Dynamic Memory (DM from now on). Recently, with the emerging market of new portable devices that integrate multiple multimedia services (e.g. 3D games or 3D graphical processing [14]), the need to efficiently use DM in embedded low-power systems has arisen.

New multimedia applications (e.g. MPEG4) demand a variable amount of memory footprint at run-time due to their unpredictable input data (e.g. 3D streams features). Thus, in these applications the classical processor-memory bottleneck becomes much more important in terms of power consumption and memory bandwidth (resulting usually in a reduced overall system performance). Designing these new embedded multimedia systems in pure classical hardware systems imply the use of worst-case static allocation of all dynamically (de)allocated data while their associated application exists in the system, because a flexible remapping is not possible after the design time phase. Consequently, since this dynamic data has a limited lifetime and a variable behaviour while the application is running, this would lead to a very high overhead both in power consumption and memory footprint for embedded systems. Even if average values of these worst-case solutions are used, DM solutions will require less memory footprint (i.e. 22% less) than static solutions [1]. Moreover, these intermediate static solutions will not work in extreme cases of input data, whereas DM solutions will continue to work in almost any case, because they only allocate memory for the current present data chunks. Thus, DM management should be used in cost- and failure-sensitive realisations of such embedded applications, which is currently included in the software part of these embedded systems, i.e. within the real-time operating system.

However, in full SW systems (with a general-purpose orientation) too much overhead is present (e.g. power consumption) in DM management related issues to be used directly by current embedded operating systems (e.g. RTEMS [12]). For example, sophisticated DM managers, e.g. Lea Allocator or Kingsley, possess complex structures that are suitable for performance [18], but not for low power. In fact, heavily customized and simplified versions of such DM managers in specific-purpose embedded systems can be very close to their initial implementation in performance, but with large savings in power consumption (more than 30%, see Section V for more examples). Hence, specifically designed (or custom) DM managers must be used in embedded systems according to the underlying memory hierarchy and the kind of applications that will run on them. Unfortunately, nowadays when custom DM managers are used, their designs have to be manually optimized by the developer, considering only a limited amount of design and implementation alternatives, since no methodologies exist to explore the DM management design space.

Taking all the previous issues into account, in this paper we present a new step-wise system-level approach to suitably handle the DM behaviour of new embedded multimedia applications (e.g. [14]) and to design custom DM management mechanisms with the required low power consumption requirements of embedded systems. The rest of the paper is organized in the following way. In Section II, we summarize some related work. In Section III, we intuitively explain the limitations of current DM management for low power systems. Next, in Section IV we briefly explain the internal steps that conform our approach to design custom DM management of new dynamic embedded multimedia applications. In Section V, we shortly introduce our case studies and show the experimental results obtained with our approach. Finally, in Section VI, we draw our conclusions.

II. Related Work

In the embedded system engineering community, low power is a main objective in the dynamic power management domain (see [3] for a good tutorial overview) and in the dynamic voltage scheduling domain [8]. However, the DM subsystem related issues are hardly covered.

The foundations of an efficient DM subsystem in a general-context are already well established, and techniques for building-application specific custom DM managers that try to improve performance (but not power consumption) have received extensive attention lately [18]. Also, new methods have been proposed to refine the DM subsystem by evaluating it empirically with customizable DM managers. In [6], a DM manager that allows to define multiple memory regions with different disciplines 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 [4], the abstraction level of customizable DM managers has been extended to C++. It proposes a framework that provides support for garbage collection and partially different (de)allocation features (e.g. block sizes) for memory regions. Nevertheless, power consumption issues or a structured exploration of the DM design space for low power embedded systems have not received special attention.

Recent real-time OSs for embedded systems (e.g. [12]) support dynamic allocation via platform-specific (custom) DM managers based on simple region allocators [7] with a reasonable performance. Finally, in new embedded systems where the range of applications to be executed is very wide (e.g. new consumer devices), variations of state-of-the-art general-purpose DM managers are frequently used. For instance, Linux-based systems use as their basis the Lea Allocator [18], a best-fit approximation DM manager. Windows-based systems include in their foundations the Kingsley manager [18], which is a power-of-two segregated fits DM manager.

Regarding profiling and simulation of the memory subsystem, work can be found aiming at estimating power consumption of software using assembly code and a higher abstraction level [16]. Also, for System-On-Chip design, new techniques that evaluate power consumption using concurrent power estimators for each of their parts (e.g. buses, processing units) have been proposed [10]. In addition, a large amount of research has been performed at system-level based on execution traces [15]. However, none of these techniques focus on power consumption of DM managers at system-level and the influence of the memory hierarchy in their design.

III. Dynamic Memory Management for Current Dynamic Embedded Systems

System DM management basically consists of two separate tasks, i.e. allocation and deallocation. Allocation is the mechanism that searches for a block big enough to satisfy the request of a given application and deallocation is the mechanism that returns this block to the available memory of the system in order to be reused later. In real dynamic multimedia applications, the blocks are requested and returned in any order, thus creating ”holes” among used blocks. These holes are known as memory fragmentation [18]. Therefore, apart from handling the memory de/allocation requests, the DM manager must deal with fragmentation issues at the same time. This is done by splitting and merging free blocks to limit actual memory fragmentation in the memory subsystem.

In many current embedded operating systems, initially state-of-the-art general-purpose DM managers (e.g. region managers [18]) are usually considered as starting point for further refinement [12] due to time-to-market (and design complexity) constraints. Then, manually defined simplifications of these complex managers are applied to create very specialized DM managers. For example, DM managers (i.e. Partition manager [12]) where only one fixed size is allowed in the whole DM space for performance purposes. In addition, when the range of applications that can use the system is much broader, more complex DM managers need to be created. In this case, DM managers usually include similar features to a general-purpose DM manager (e.g. immediate splitting and coalescing mechanisms to limit fragmentation [12]), which have to be manually added if the applications that will run on the final embedded system seem to benefit from it. For example, if a variable number of allocation sizes can be requested within one application, these additional mechanisms (e.g. splitting or coalescing) could be useful to limit the amount of memory fragmentation in the system. However, this complex engineering process of partially customizing DM managers for the specific platform features (and the range of dynamic applications that will run on it) can usually take several months [12] because everything is based on manual profiling and testing according to the programming style and inspiration of each developer to apply convenient transformation in his own code. Moreover, if the purpose of the customized DM manager (e.g. maximizing performance) is slightly changed (e.g. power reduction is also added as system constraint in the new design), the new custom DM manager needs to be redesigned from the beginning to respect the new requirements. In our approach, the DM manager is built in a systematic way instead of mainly relying on the experience and knowledge of the designer.

IV. Approach Description

The proposed approach is characterized by optimizations only at the level of O.S. DM managers, i.e. once the dynamically allocated data of the applications and the features of the platform have already been defined and cannot be changed. Techniques and optimizations that can be applied to optimize the dynamically allocated data structures implementations according to specific embedded system constraints are explained in [5] and yield out of the scope of this paper. In this section, we explain the stages needed in our approach to create custom DM management mechanisms in a structured way for new dynamic embedded applications (e.g. 3D rendering) considering their necessary low power consumption requirements. They are outlined in Figure 1.

As Figure 1 shows, to construct a suitable custom DM management scheme for a certain application, in a first stage (first rounded-edged shape in Figure 1) detailed information is obtained for the application under study to characterize its real run-time DM behaviour. This information is obtained via simulation using our own `C++` profiling library [2] that replaces the usual calls of the system allocator (e.g. `malloc()`, `free()`, etc.) and needs to be integrated in the application under study. For real-life applications (see Section V for more details), this integration process takes no longer than one week. The acquired profiling provides us information about the most important characteristics for each specific multimedia application:

- Ranges of different (de)allocation block sizes.
- Main (de)allocation phases (that usually correspond to each logical phase from the algorithmic point of view) existing in each multimedia application.
- The particular (de)allocation pattern (and timing behaviour) of each allocation data structure (e.g. short bursts, allocation ramps, etc. [18]).

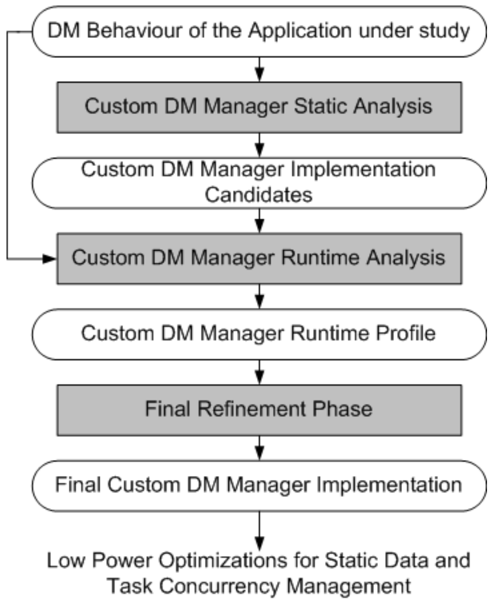
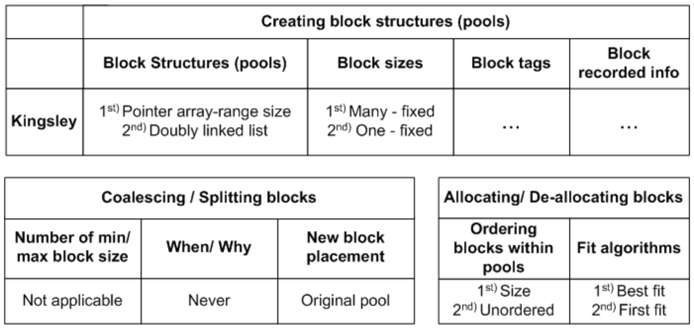

DM Behaviour of the Application under study
Custom DM Manager Static Analysis
Custom DM Manager Implementation Candidates
Custom DM Manager Runtime Analysis
Custom DM Manager Runtime Profile
Final Refinement Phase
Final Custom DM Manager Implementation
Low Power Optimizations for Static Data and Task Concurrency Management

Fig. 1. Flow of the proposed system DM management refinement approach for new embedded real-time multimedia applications

Then, taking the acquired profiling information into account, we perform a static analysis (marked in Figure 1 as the first rectangle) of each (de)allocation phase of the application under study using our own design space of possible decisions for DM management in embedded systems [1]. We have classified all the important design choices to compose DM managers in different decision categories [1]. First, *Creating block structures* to handle the way block data structures are created and used to satisfy memory requests. Second, *De/Allocating blocks* to deal with the actions required in DM management to satisfy memory requests and to free the used blocks. Finally, *Coalescing/Splitting blocks* to determine the strategies to ensure a low percentage of memory fragmentation [18]. These are orthogonal categories, which means that any taken decision in any category can be combined with any decision in another. Then, the result should be a potentially valid combination or DM manager (which does not necessarily mean that it already meets all timing and cost constraints). An example of how a typical state-of-the-art DM manager (i.e. Kingsley) can be represented within our design space is shown in Figure 2. The DM manager is divided in its basic layers and thus in some of these categories two levels exist. For example, its basic block structure in Figure 2 is composed by a first layer (1st) with a pointer array that includes the header of each pool of blocks (e.g. information about sizes allowed in the memory pool, space available). This pointer array has the number of



	Creating block structures (pools)			
	Block Structures (pools)	Block sizes	Block tags	Block recorded info
Kingsley	1st) Pointer array-range size 2nd) Doubly linked list	1st) Many - fixed 2nd) One - fixed	…	…

Coalescing / Splitting blocks		
Number of min/ max block size	When/ Why	New block placement
Not applicable	Never	Original pool

Allocating/ De-allocating blocks	
Ordering blocks within pools	Fit algorithms
1st) Size 2nd) Unordered	1st) Best fit 2nd) First fit

Fig. 2. Example of Kingsley represented within our design space

elements (e.g. memory pools) needed to cover the range of allocations sizes allowed in Kingsley. Then, inside each pool a second layer (2nd) is present, where the used dynamic data type is a doubly linked list [18]. Similarly, the different characteristics of these two layers for all the other parts regarding DM management are defined within our categories (e.g. information of block tags, fit algorithm to choose a block, etc. [18]).

As a result of this previous static analysis of DM managers implementations, refinements can be performed systematically considering the different alternatives for each of their basic components [1]. For example, using the original Kingsley structure in an application with a very small range of possible allocation sizes will produce a significant waste of memory within the DM pools, thus we would have to change the field *Number of min/max block sizes* of Figure 2 to fit the specific sizes of the application under study. Furthermore, if the application uses a very specific access pattern to its DM blocks, then we need to modify the way the blocks are stored in each pool to avoid unnecessary accesses in the second layer, i.e. 2nd layer of field *Ordering blocks within pools* and adjust it to the application (e.g. using *LIFO* instead of *Unordered* as in usual Kingsley). Finally, note that in the Coalescing/Splitting blocks category, Kingsley does not use any of these mechanisms (i.e. option `Never` by default). Its original design is optimized for performance, thus all the necessary accesses for coalescing and splitting to reduce memory fragmentation are removed hoping to improve performance. But this choice can result in severe power consumption penalties and memory fragmentation wastage for certain DM (de)allocation patterns (e.g. sizes not multiples of powers of two as the sizes of the pools are) [2], [18]. Then, if the studied dynamic multimedia application falls inside these unsuitable patterns for Kingsley, using our static analysis it would be possible to modify its behaviour in this aspect to achieve the pursued low power requirements.

After this static analysis and exploration phase following the suitable order we propose for low power exploration [2], promising custom DM managers candidates (second rounded-edged shape in Figure 1) can be defined. Then, the actual run-time behaviour of these potential candidates is evaluated via simulation (second rectangle in Figure 1) with a `C++` library that we have developed to implement the decisions in our DM design space [2]. This library is based on a combination of `C++` templates and inheritance to allow fast changes in the implementation of DM managers, e.g., replacement policies or size of the pools [18]. Moreover, they include profiling `C++` objects to store the information about the memory taken from the system and the memory accesses performed by DM managers. In the same way as for the first profiling phase, our library of DM managers includes a whole set of common ANSI-C standard functions for (de)allocation operations (e.g. `malloc()`, `free()`), which allow to integrate it in any real-life embedded multimedia application without a time consuming effort (up to 2 weeks in our case studies). As a result, this complex infrastructure of layers and objects provide the necessary run-time profiling of the ideal values in many characteristics of the final custom DM manager candidates (third rounded-edged shape in Figure 1). This final profiling allows us to determine in a post-execution refinement phase (third rectangle in Figure 1) the ideal implementation for our custom DM manager (e.g. number of memory pools) for the multimedia application under study (forth rounded-edged shape in Figure 1). This final post-execution refinement phase is not performed at run-time to preserve the actual run-time behaviour of the system when the profiling is acquired.

Finally, after the custom DM manager is integrated in the code of the application under study, low power memory optimizations for concurrency management in embedded systems could be used [17] (lower part of Figure 1). They will also map the DM pools together with other statically allocated arrays or scalars to the physical memory organisation, including SDRAMs, shared scratchpads (if not specially dedicated scratchpads are used for DM) and caches.

V. Case Studies and Results

Our approach is illustrated using two real applications from different modern multimedia application domains: the first case study is a 3D rendering system where the objects are represented as scalable meshes, the second one is part of a new 3D image reconstruction system. For the power consumption estimations, an updated version of the memory model described in [9] is used. It is a complete energy/delay/area model for embedded SRAMs that is able to scale to different technology nodes (we use the $.13\ \mu$ technology node for the results).

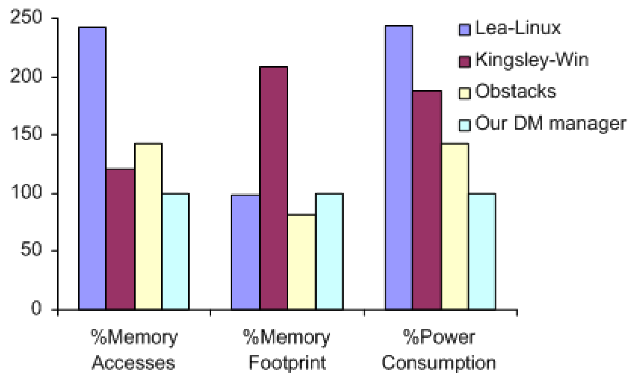


Fig. 3. Comparison results between different DM managers in the 3D rendering system for a 3D image of 6713 vertices and 13406 faces. Values normalized to our custom DM manager

This model depends on the technology node, memory footprint factors (e.g. super-logarithmic on the size or leakage) and especially on memory accesses (linearly).

Our first case study is a new 3D video rendering application based on scalable meshes [11] that adapt 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 [11]) 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 [11] of the rendering process to show the objects with the appropriate resolution on the screen.

After applying our approach to obtain profiling information of the de/allocation access pattern of the vertices and faces in the application, a custom DM manager trying to minimize power consumption is designed (i.e. CUSDMMRENDER in Table I). In this case, we have compared our custom DM manager with Lea, Kingsley and, due to its stack-like allocation behaviour in the phases with vertices, we have also tested the application with Obstacks [18], a well-known custom DM manager optimized for such behaviour. The results obtained are depicted in Table I and Figure 3. They show that Kingsley accomplishes better results in power consumption (29.3% less) than Lea because it does not perform any maintenance operation (i.e. splitting or coalescing blocks) to limit the fragmentation of the system. However, the power results of Kingsley are not as good as expected because its used technique (i.e. avoiding completely coalescing and splitting by using 30 pools of separated allocation sizes) produces an enormous overhead in memory footprint (see Table I). Thus, bigger memories are needed in the system (and then more power is consumed in each memory access) to allow the system to work correctly. In contrast, after the detailed study of the application with the proposed approach, our custom DM manager only contains two separated memory pools, for the three allocation block-sizes used in this specific application, i.e. 40, 44 (share the same pool) and 84 Bytes. Then, we can also observe that Obstacks improves the results of Kingsley for power consumption (27.8%) thanks to its optimizations for the stack-like behaviour of the first three phases of the rendering process, which reduces its number of DM accesses (e.g. deallocating at one shot a number of consecutive memory blocks). Finally, our custom manager improves further the values for power consumption of Obstacks (by 29.8%), because the latter cannot exploit its stack-like optimizations in the three final phases of the rendering process. In these phases, the faces are used all independently in a disordered pattern and they are required to be freed separately. Thus, Obstacks suffers from a high penalty in memory accesses and energy dissipation per frame in these last three phases of the algorithm due to its internal block structure organization (optimized for deallocating in one shot several blocks, but not each block separately). On the contrary, our custom DM manager with a more elaborated internal maintenance structure for both types of deallocation behaviour does not suffer from it. As a result, after applying our approach to the 3D Rendering system, significant gains in power consumption are achieved (see Table I and Figure 3), and the new optimized system can also perfectly fulfil the real-time requirements needed by this application.

TABLE I
RESULTS IN THE 3D RENDERING SYSTEM

Dyn. Mem. managers	memory accesses	memory footprint (B)	power (μW) .13μm tech.
Lea-Linux	7.60×10^6	1.86×10^6	5.05×10^4
Kingsley-Win	3.80×10^6	3.96×10^6	3.89×10^4
Obstacks	4.45×10^6	1.55×10^6	2.95×10^4
cusDMMrender	3.13×10^6	1.89×10^6	2.07×10^4

Our second case study is a 3D image reconstruction application [14] that matches corners from two subsequent frames in a frame stream to reconstruct 3D objects. The operations done on the images are memory intensive, e.g. each comparison process uses over 1.5Mb, and the accesses in it are randomized. Thus, typical image access optimizations (as row-dominated accesses versus column-wise accesses) cannot be used.

We have optimized this application with our systematic approach to demonstrate the applicability of our approach to design a DM manager that suitably employs the memory hierarchy (e.g. main memory, scratchpad memories). First, an optimized DM manager is designed using our approach trying to minimize power consumption, i.e.`cusDMMrec1` in Table II. It is compared in Table II with two typical DM managers for this kind of embedded systems, i.e. Kingsley [18] and an optimized version of the new region-semantic managers [18] (RegAlloc in Table II). We can observe that `cusDMMrec1` improves the power consumption results of these DM managers by adjusting its memory pools structure to the limited range of allocation sizes used in the optimized dynamic data types of the application (i.e. 2, 4.2, 7.56, 10, 16, 119 and 124 KB). Then, the number of memory accesses due to memory fragmentation is minimized by splitting and coalescing block mechanisms [18] in those pools that include more than one block-size (i.e. 119 and 124 KB). Furthermore, `cusDMMrec1` makes use of the memory hierarchy we suppose available for the DM managers in the final embedded system to test our approach with a multi-level memory subsystem, i.e. on-chip SW-controlled scratchpad of 64 KB and main off-chip memory. The accesses to each level of the memory hierarchy are distinguished in Table II as `on-chip` and `off-chip values` respectively. Thus, in `cusDMMrec1` the memory pool that produces most of the accesses (those for allocation sizes of 16 bytes with the maintenance information of the DM manager and the dynamic data types of blocks of 16 KB) are separated from the global heap used in the manager. They are now handled in a different and small one (57 KB) that is placed permanently in the scratchpad. Therefore, Table II shows that `cusDMMrec1` reduces the power consumption needed compared to Kingsley (89.04%) and new region managers (81.32%). This kind of suitable use of the memory hierarchy is not envisaged in Kingsley or new region managers. As a consequence, their designs include memory pools of much bigger size (e.g. 256 KB) than the 64 KB of the scratchpad which cannot be placed there. Furthermore, a redefinition of their internal pool structure to make use of this small scratchpad memory is a very time-consuming (and error-prone) task since the whole implementation structures of these DM managers depend on these big sizes of their pools. Thus, we can conclude that the use of a multi-level memory subsystem for the DM managers of embedded systems allows to achieve even better results using our approach because we can make profit of it in the whole design and implementation process of the DM managers.

In summary, after applying our approach, the total power consumption of the 3D reconstruction application is reduced significantly, i.e. almost one order of magnitude (see Table II), compared to current manual optimization solutions and respects the real-time requirements of this application.

TABLE II
RESULTS IN THE 3D IMAGE RECONSTRUCTION SYSTEM

Dyn. Mem. managers	memory accesses	memory footprint (B)	power (μW) .13μm tech.
Kingsley	4.25×10^6	2.26×10^6	4.30×10^4
RegionDM	4.68×10^6	2.08×10^6	3.11×10^4
cusDMMrec1 total:	4.64×10^6	1.56×10^6	3.94×10^3
(off-chip values)	2.57×10^5	1.52×10^6	1.71×10^3
(on-chip values)	4.39×10^6	6.55×10^4	2.23×10^3

VI. CONCLUSIONS

Currently, DM management tuning is one of the crucial and most difficult parts to optimize current embedded multimedia applications due to the unpredictability of their input data and events (e.g. images and user movements). In this paper, we have presented a new system-level design approach that is able to obtain a detailed view of the dynamic behaviour, i.e. (de)allocation pattern, of new multimedia application and optimize it using a step-wise refinement flow. The results achieved in real applications show significant power consumption reductions over current implementations of these systems, which allow porting them to actual embedded systems.

REFERENCES

[1] Removed for blind review purposes.
[2] Removed for blind review purposes.
[3] L. Benini and G. De Micheli. *Dynamic power management design techniques and CAD Tools*. Kluwer Publishers, 1998.
[4] G. Attardi, et al. A customizable memory management framework for c++. *Software Practice and Experience*, 1998.
[5] Removed for blind review purposes.
[6] K.-P. Vo. Vmalloc: A general and efficient mem. allocator. *Sw. Practice and Experience*, 1996.
[7] D. Gay and A. Aiken. Memory management for with explicit regions. In *Proc. of PLDI '01*, 2001.
[8] N. K. Jha. Low power system scheduling and synthesis. In *Proc. of IEEE Int. Conf. on Computer-Aided Design*, 2001.
[9] N. Jouppi. CACTI, 2002. `http://research.compaq.com/wrl/people/jouppi/CACTI.html`.
[10] M. Lajolo, et al. Cosimulation-based power estimation for system-on-chip design. *IEEE Trans. VLSI Systems*, 2002.
[11] D. Luebke, M. Reddy, et al. *Level of Detail for 3D Graphics*. Morgan-Kaufmann Publishers, 2002.
[12] Rtems, Open-Source RTOS. `http://www.rtems.org`.
[13] P. R. Panda, et al. Data and memory optimizations for embedded systems. *ACM TODAES*, April 2001.
[14] M. Pollefeys, et al. Metric 3D surface reconstruction from uncalibrated image sequences. In *LNCS*, Springer-Verlag, 1998.
[15] R. A. Uhlig and T. N. Mudge. Trace-driven memory simulation: a survey. *ACM Comput. Surv.*, 1997.
[16] N. Vijaykrishnan, et al. Evaluating hw-sw optim. using a unified energy estimation framework. *IEEE Trans. on Computers*, 2003.
[17] Removed for blind review purposes.
[18] P. R. Wilson, et al. Dynamic storage allocation, a survey and critical review. In *Int. Workshop on Mem. Management*, 1995.