METHODOLOGY FOR REFINEMENT AND OPTIMIZATION OF DYNAMIC MEMORY MANAGEMENT FOR EMBEDDED SYSTEMS IN MULTIMEDIA APPLICATIONS

*Marc Leeman**
Geert Deconinck, Vincenzo De Florio
KULeuven ESAT
Kasteelpark Arenberg 10
B3001 Leuven, Belgium

David Atienza†
Jose M. Mendias
DACYA/UCM
Avda. Complutense s/n
28040, Madrid, Spain

Chantal Ykman, Francky Catthoor,
Rudy Lauwereins
IMEC
Kapeldreef 75
3001 Heverlee, Belgium

ABSTRACT

In multimedia applications, run-time memory management support has to allow real-time memory de/allocation, retrieving and processing of data. Thus, its implementation must be designed to combine high speed, low power, large data storage capacity and a high memory bandwidth. In this paper, we assess the performance of our new system-level exploration methodology to optimize the memory management of typical multimedia applications in an extensively used 3D reconstruction image system [1, 2]. This methodology is based on an analysis of the number of memory accesses, normalized memory use [1] and energy estimations for the system studied. This results in an improvement of normalized memory footprint up to 44.2% and the estimated energy dissipation up to 22.6% over conventional static memory implementations in an optimized version of the driver application. Finally, our final version is able to scale perfectly the memory consumed in the system for a wide range of input parameters whereas the statically optimized version is unable to do this.

1. INTRODUCTION

The fast growth in the variety and complexity of multimedia applications and platforms has created the need for optimal algorithms on one hand, and the development of high storage capacity and efficient memory systems on the other hand. Good examples where they become necessary are archaeological site recording and reconstruction, architectural planning, augmented reality and film industry [2, 3, 4]. These systems depend upon dynamic data management, which constitutes one of the most difficult design challenges when mapping them on low-power and high-speed processors that are often not equipped with extensive hardware and system support for dynamic memory (DM). This dynamic memory management (DMM) must provide an efficient memory de/allocation, retrieving and processing of the data involved in the multimedia algorithms by combining speed, low power, large data storage and an optimal management of multiple Dynamic Data Types (DDTs). It has to take into account the fact that these DDTs have a limited lifetime and a variable behavior while the application is running. As a consequence, three factors influence the overall performance of the memory system: (1) the access pattern over time of the algorithm implemented, (2) the amount of memory accesses and (3) the mechanisms to access the data (as defined by the data structures of the system). Taking all these into account, it is clear that a systematic exploration at the system-level of the possible choices for memory management in multimedia applications is a necessity.

In this paper, we demonstrate the efficiency of our new system-level exploration methodology, which optimizes the DMM for typical 3D multimedia applications with the aforementioned behavior [1]. After applying the methodology, the results improve to a great extent the memory footprint, memory accesses and estimated energy dissipation compared to manually optimized implementations.

The remainder of this paper is organized as follows. In Section 2, the foundations of the methodology and design features of DMM for multimedia systems are presented. In Section 3, the specification of the 3D multimedia application used to apply our methodology is illustrated. In Section 4, we characterize the behavior of the relevant DDTs of the system. After that, in Section 5, we explain all the different steps to optimize the DMM of the system. In Section 6, the experimental results are presented. Finally, in Section 7, we draw our conclusions and present future extensions.

2. RELATED WORK

Conceptually, the basis of a good DMM is already well established for general-purpose systems [5]. Also, several implementations for DM managers exist in a general-purpose context to allow large data storage, real-time de/allocation

*This work is partially supported by the Fund for Scientific Research - Flanders (Belgium, F.W.O.) through project G.0036.99 and a Postdoctoral Fellowship for Geert Deconinck

†This work is partially supported by the Spanish Government Research Grant TIC2002/0750.

[1] The sum of the memory used at a time slice, multiplied by the time. This amount is then divided over one run of the algorithm

0-7803-7795-8/03/ $17.00 © 2003 IEEE

and frequent updates of the data structures [5, 6].

Due to the behavior of multimedia applications (with a high demand of data retrieval and storage), the access to the data must be highly optimized. Presently, research has been started to propose suitable access methods to DDT implementations [7].

For manual and automatic memory management in embedded systems, research is performed [8, 9]. In manual memory management work, DM is partitioned into blocks and tracked by single linked lists [9]. For general purpose manual memory managers, [6] describes a fast C++ template infrastructure to build custom memory allocators.

In a different field, telecom network applications, an approach that performs an exploration methodology driven by the number of memory accesses has been outlined in [10]. These applications have very specific behavior dominated by key accesses and lookups. Furthermore, they use one or few independent DDTs. These and other characteristics restrict the work to the telecom domain (see [10] for details).

System-level optimizations and techniques for general-purpose design to reduce power consumption are explained in [11]. However, optimizing the memory management at the system-level in multimedia applications with complex dynamic behavior has not been given much attention.

In this paper, we propose to use a fast, stepwise, cost-driven exploration and refinement for the DDTs in multimedia applications at the system-level, where the impact on memory performance is the most important part.

3. DEMONSTRATOR

The 3D image reconstruction algorithm used as case study in this paper is heavily characterized by intensive internal DM use. This metric 3D-reconstruction from video algorithm [1] allows the reconstruction of 3D scenes from images and requires no other information apart from multiple frames. This makes the code especially useful for situations where extensive 3D setup with sensitive equipment is extremely difficult, e.g. crowded streets or remote locations, or impossible as when the scene is no longer available [2, 3].

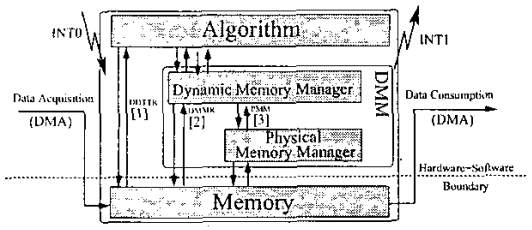

Fig. 1. Overview of the system design in the optimizations.

For quick on-site visualization and processing of more frames for a more detailed reconstruction, speeding up the application is necessary and demands extensive code transformations and optimizations. Moreover, energy consumption is paramount for hand-held visualization devices.

Within the application framework of our methodology, we depict a typical dedicated system for intensive numeric processing (see Figure 1). An instance of this can be a representative example of the platforms used for multimedia applications, where the data is immediately transferred into memory via DMA and the execution is triggered by a software interrupt. When processing is finished, another software interrupt is fired to handle the processed data.

The software module used as our driver application is one of the basic building blocks in many current 3D vision algorithms: *feature selection and matching*. The algorithm selects and matches features (corners) on different images and the relative offsets of these features define their spatial location. The number of generated candidate matches is highly dependent on a number of factors. Firstly, the *image properties* affect the generation of the matching candidates. Images with highly irregular or repetitive areas will generate a large number of localized candidates, while a low number will be detected in other parts of the image. Secondly, the *corner detection parameters* have a profound influence on the results (and, consequently, on the input to the subsequent matching algorithm) because they affect the sensitivity of the algorithm used to identify the interesting features in the images/frames [12]. Finally, the *corner matching parameters* that determine the matching phase have a profound influence and are changed at runtime (e.g. the acceptance and rejection criterion changes over time as more 3D information is retrieved from the scene).

Taking all the previous factors into account, the possible combinations of parameters in the system make an accurate estimation of the memory cost, memory accesses and energy dissipation at compile time very hard or nearly impossible. This unpredictable memory behavior can be observed in many state-of-the-art 3D vision algorithms [13] and multimedia algorithms because they use some sort of *candidate selection* followed by *a criterion evaluation*.

4. MEMORY PERFORMANCE

In this paper, we will not focus on the static data (images and detected points), but on the internal DDTs of the algorithm. The optimization of the transfers and accesses to the static data can be done by other techniques [14, 15]. The memory performance and behavior of the 3D module is characterized by the following DDTs: (1) `ImageMatches` (`IMatches`) is the list of pairs where, for every point in the first image, (new) matches on a second image are considered based on neighborhood or epipolar distance [1]. (2) `CandidateMatches` (`CMatches`) is the list of candidates that needs evaluation, e.g. normalized cross correlation of a a window around the points [13]. (3) `MultiMatches` (`MMatches`) is the list that stores the match pairs that pass

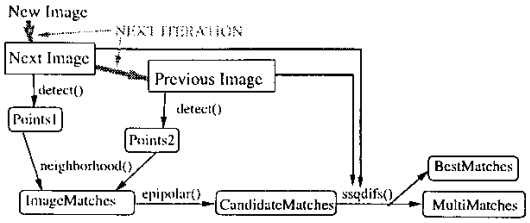

Fig. 2. The interaction of the dynamic variables

the evaluation criterion. In this list, one point can have still multiple counterparts on the other image. (4) `BestMatches` (`BMatches`) is a subset of `MMatches` and retains only the best match for a point, according to the criterion already mentioned (if the evaluation is satisfied).

All these DDTs were originally implemented using a double linked list and exhibit typical *data consumption* and *generation behavior*. Figure 2 shows the interaction of the DDTs in the algorithm code. From the images, corners are selected. On each corner of one image, a neighborhood search is done and the pairs that pass a threshold are stored in `IMatches`. Based on information of previous frames, these are accepted into `CMatches`. Every pair in `CMatches` is checked by a *normalized cross correlation* and stored in `MMatches` and `BMatches`. These results are passed to the next software module involved in the 3D reconstruction algorithm. It is important to mention that, although in this phase of the algorithm the image is still being accessed, these accesses are randomized. As such, classic optimizations like row dominated accesses versus column wise accesses and other image access optimizations are not relevant. Furthermore, this algorithm variant uses the variant that re-uses the intermediate data in later steps.

5. MEMORY MANAGEMENT

In order to optimize the DM performance, our methodology uses a layered approach. It starts from a high level specification and adds more system-specific information in each step. This new information is used to refine previously made estimations. This approach allows to obtain an early idea of representative implementation costs and requirements without going through the entire expensive design process.

Figure 1 shows the three most important steps to optimize the aforementioned DM in the methodology. Firstly, the approach starts with the premise that the algorithm interacts directly with the memory, this is the Dynamic Data Type Transformation and Refinement step (DDTTR). Secondly, memory allocators that handle and optimize the DM de/allocations are added to the design space evaluation in the Dynamic Memory Management Refinement step (DMMR). Finally, a physical memory manager layer is added to optimize further, e.g. solve bank conflicts and introduce paging; this is the Physical Memory Management step (PMM).

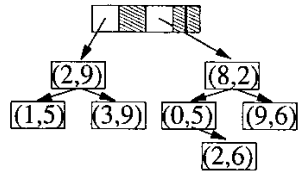

Fig. 3. Example of layered DDT

Because the last step tackles specific system and hardware information with all their complexities, it falls outside the scope of this paper.

5.1. Dynamic Data Type Transformation and Refinement

DDTs define the way in which the memory is allocated, accessed and free. Initially, simple DDTs are modeled by Data Types (DTs), e.g. lists, arrays or graphs, in combination with operations like `add`, `remove` and `get`. Then, these simple DDTs can be combined in layered structures that offer a compromise between flexibility, memory use and access time. For instance, a linked list solution is very flexible, but slow in access, while an array offers fast access, but it is hard to maintain and rigid in size. Highly optimized programs combine these simple data types in some sort, but these decisions are rarely taken in a methodological way and are left to the *experience* and *inspiration* of the programmer. Our methodology makes these decisions in a systematic way. In order to do this and identify the optimal DDT implementation for each variable, relatively detailed information of the DM behavior at run-time is required.

The DDTs are modeled in "low level" `C++`, which provides some Object Oriented (O.O.) conceptual benefits, while the code is still easy to convert to `C` (the code size overhead we have obtained due to our "low level" `C++` compared to `C` is really negligible), the target language of many SoC integration flows. The choice also allows minimal changes in the `C` algorithm code to integrate the DDTs.

The tool developed to support this step has two main building blocks. First, *profile objects* are added and supported by an O.O. profiling framework. This way, the use of combined layered DDTs is made transparent to the algorithm since all memory behavior (allocations, accesses,...) is contributed to the complex DDT, and not to the composing simple DDTs. For example, one possible realistic DDT implementation for `CMatches` can be seen in Figure 3. It is an array to do the first lookup. Then, each position in the array points to a tree, which points to an array that finally stores the data (pairs of related points in the frames). When the memory behavior of a DDT is evaluated, the developer is not only interested in the contribution of the array type in the DDT, but rather what the combined cost of this complex structure is. Finally, this profile framework, together with all the required code transformations and instrumentation are added automatically to the code.

The combination of multiple DTs in multiple layers and the use of multiple DDTs in an application results in an exponential search space. Currently, some simple heuristics are employed to automatically explore the search space, using a representative input data set. As a result of the exploration process, Pareto optimal solutions[2] are located, based on normalized memory use, memory accesses and energy estimates and the final solution depends on the restrictions of the designer and system. By modifying the heuristics, more details about (sub-optimal) DDTs can be obtained, at the cost of an increased exploration time.

5.2. Dynamic Memory Management Refinement

The purpose of DMMR is an efficient use of the the DM available in the system for certain constraints. An inattentive management of the DDTs of the application can lessen severely the performance of the whole system and increase the memory accesses and the energy dissipation. Therefore, a well-adjusted dynamic manager has to be selected for the multimedia application under study.

In order to select efficiently the DM manager, we take into consideration high-level strategic issues [5] as well as a full analysis of the DM behavior of the application. The DM manager has to maximize the performance of the application taking into account memory consumption and size among other constraints. The profiling information obtained in the previous steps of the methodology is used to characterize the evolution in time of the DDTs involved in the application (see Figure 5).

```
template <class SHeap> class LogLayer:\
     public SHeap, public _profile{
 public:
  inline void* malloc(size_t sz){
   EPRINT("(%d) malloc %d bytes\n", id, sz);
   return SupHeap::malloc(sz); }
  inline void free(void* ptr){
   EPRINT("(%d) free %d bytes\n",id,SHeap::getSize(ptr));
   SuperHeap::free(ptr); } };

class Kingsley KingsleyNC<SbrkHeap>{};
class KingsleyLogged KingsleyNC<LogLayer<SbrkHeap> >{};
```

Fig. 4. Definition and use of a LogLayer

The information about the DDTs is characterized in a search space defined to this end. It is based on a taxonomy of orthogonal decisions that tackles the different parts of any possible allocator [5] for manual memory management. The choices from our taxonomy are combined to eventually constitute global DM managers according to the constraints of the system.

After that, we explore the different memory managers defined by our search space using a fast infrastructure of C++ mixin layers [6], which allows to compose and create these managers with very complex de/allocation strategies. For example, first fit with address ordered for the free blocks and deferred coalescing allowed [5]. This allows fast changes in the features of DM managers (e.g. replacement policies or sizes of the pools). Due to mixin based template instantiation, localized changes for detailed profiling can be made by inserting additional layers. An example is shown in Figure 4, where a simplified definition of our *Loglayer* is added to the Kingsley allocator [6] to obtain information of what memory is requested from the system and recycled (fragmentation). The infrastructure of layers and profiling objects is used to heuristically explore the values in many characteristics of the DM managers designed with our search space and select the ideal for the application.

[2] A point is said to be Pareto optimal, if it is not longer possible to improve upon one cost factor without worsening any other.

Table 1. DM use from DDTs in the original implementation

Variable	memory accesses	memory footprint (B)	energy (nJ)
IMatches	1.201×10^{6}	0.340×10^{3}	2.162×10^{4}
CMatches	8.442×10^{5}	2.486×10^{5}	3.039×10^{5}
CMCStatic	9.140×10^{7}	6.247×10^{4}	1.695×10^{7}
MMatches	1.849×10^{4}	0.678×10^{3}	3.328×10^{3}
BMatches	1.664×10^{4}	0.623×10^{3}	2.996×10^{3}
Total	9.348×10^{7}	3.127×10^{5}	1.728×10^{7}

6. RESULTS

As we have already explained, optimizing the memory management at system-level in multimedia applications with complex dynamic behavior has not been given much attention. Therefore, our methodology will be used to refine the 3D image reconstruction system used as our case study.

In the first step, DDTTR, the representation of the four main DDTs described in Section 4 is optimized. After running the tool to get the profiling information from the original code, a memory use graph is generated (see left chart in Figure 5). In order to account for the varying memory use during the program run, normalized memory is used to get an estimation of the overall contribution of each DDT to the energy dissipation (see Table 1). This attributes more accurate contributions to energy cost estimates and avoid that e.g. DDTs with very short and high memory usage distort and hide the memory contribution from other DDTs. The model used in the results to obtain energy estimations is described in [16] for large SRAMs with .25 μm technology. This model depends on memory footprint factors (i.e. memory size, internal structure of banks and sub-banks, memory leaks, working time of the memory and technology) and energy consumption factors created by memory accesses (i.e. number of memory accesses, energy consumption in active mode, size of the memory and technology). In the tools, the memory model can be replaced by others in a modular way.

A first analysis of the run-time profiling information of Figure 1 shows the existence of a dynamic array copy of CMatches in the original code to optimize the accesses to

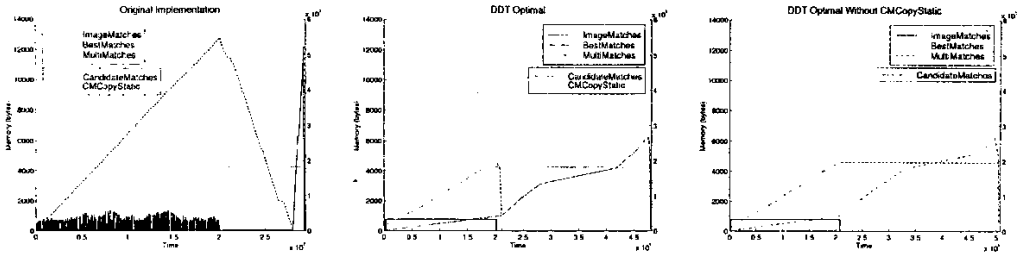

Fig. 5. Memory footprint over time. All plots mapped on the left axis, except `CandidateMatches` and `CMCopyStatic` (right one). Left original implementation, center Pareto implementation with `CMCopyStatic` and right after removing it.

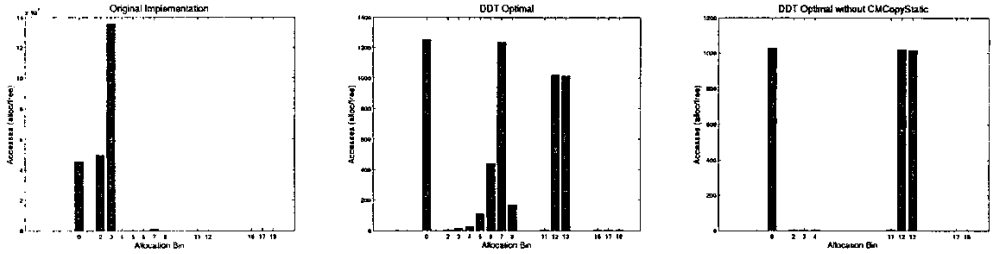

Fig. 6. The use of different block-size allocation for the original algorithm, left. Center, algorithm with optimized DDTs, but without removing `CMCopyStatic`. Right, with the removal of `CMCopyStatic`. Note the changes on the Y-axes values.

Table 2. DM use from relevant DDTs after DDTTR

Variable	memory accesses	memory footprint (B)	energy (nJ)
`IMatches`	4.020×10^5	0.624×10^3	7.243×10^4
`CMatches`	3.898×10^5	1.174×10^5	7.017×10^4
`MMatches`	7.684×10^3	1.391×10^3	1.383×10^3
`BMatches`	7.161×10^3	1.368×10^3	1.289×10^3
`Total`	8.066×10^5	1.208×10^5	1.452×10^5

this DDT. This copy is the DDT `CMCopyStatic (CMCStatic)`. Also three dynamic sets form clear bottlenecks. First, `CMatches` is the largest dynamic data structure in the application. Secondly, `IMatches` has a low normalized memory use, but it is accessed extensively. Finally, the `CMCStatic` *speed optimization dynamic array* accounts for the most important part of the memory accesses and consumes most of the energy in the application.

After the subsequent exploration step, our tool suggested an ideal solution for the different DDTs according to the constraints entered. We used minimal energy dissipation and two layered DDT structures (pointer-arrays to arrays) with array sizes of 756, 1024 and 16384 Bytes (B) were selected. As the second plot in Figure 5 depicts, the DDT refinement already attains a significant influence on performance and DM footprint.

Finally, because of the optimized DDTs found in the exploration, it is possible to remove `CMCStatic` and refine even more the algorithm combining it with `CMatches` (for more details, see [17]). After this, the figures in Table 2 and the right plot in Figure 5 are generated. They show that the accesses to `IMatches` are less than half of the original ones and the normalized memory use of `CMatches` is 43.9% less. The removal of `CMCStatic` influences the normalized memory footprint (and removed the short memory peak used while copying), but has little effect on the performance (speed) of the overall program. This shows the importance of DDTTR to speed up the application comparing the values of the X-axes from Figure 5 to match two images (time values of $\times10^7$ microseconds in the first plot, in the refined ones with our methodology only $\times10^5$ microseconds). Furthermore, DDTTR reduces memory footprint and power of the DDTs, and allow further global refinements.

In the next DMMR phase, an optimized DM manager is selected. After the exploration has been performed using the profiling information from the previous steps and our search space, the DM manager selected discerns the different behaviors of the DDTs in the case study, which are just a few as Figure 6 shows. It partitions the DM of the system in three different regions or pools with different sizes that suit the DDTs in the application. Inside every block allocated, the object size is recorded and, for every region, a double linked list is used to lessen the time required to traverse and access the data. Furthermore, the blocks are stored inside each sublist of sizes using a LIFO order. When a block is freed, the manager returns the deallocated memory to the appropriate region and it becomes available for block re-

Table 3. DM use from DDTs in the manually opt. version

Variable	memory accesses	memory footprint (B)	energy (nJ)
IMatches	4.020×10^{5}	0.857×10^{3}	7.243×10^{4}
CMatches	3.151×10^{5}	2.021×10^{5}	1.134×10^{5}
MMatches	5.856×10^{3}	6.876×10^{3}	1.054×10^{3}
BMatches	4.913×10^{3}	6.876×10^{3}	0.884×10^{3}
Total	7.282×10^{5}	2.167×10^{5}	1.878×10^{5}

quests inside the specific range of sizes allowed in this pool. It provides an excellent performance and fragmentation is minimized. It only adds approximately a 10% overhead in normalized memory use and 18% in energy consumption compared to the results of the bare DDTs shown in Table 2 due to internal management and fragmentation.

Finally, in addition to the original implementation with DM, we have created a manually optimized version of the algorithm. We consider that the designer, after manually profiling and extracting the necessary information from the original code (for the full code of the entire 3D algorithm with 1.75 million lines of high level C++, see [13]), was able to select an optimal static DT representation of the relevant DDTs. This static version achieves very good performance and the energy consumed is much lower than the original version, as Table 3 shows. However, since it is really optimized for a specific configuration of parameters, it does not scale for larger resolutions, different parameters or abnormal images features.

After both steps (DDTTR and DMMR), the total memory used in the application is greatly improved compared to the manually optimized version and the original one, as shown in Table 2. The DDTs require less DM than the original version (see Table 1 and Figure 5) and the manually optimized version (see Table 3). The energy consumed is also reduced significantly comparing with the best version, i.e. the manually optimized version (see Table 2 and Table 3). Finally, compared to the original version the memory accesses are reduced enormously, as Table 1 and Table 2 show. Summing up, with the whole methodology applied, compared to the manually optimized version, the memory footprint improves up to 44.2% and estimated energy dissipation up to 22.6%. In addition, the system can scale with extreme-cases of input parameters whereas the manually optimized version cannot do it.

7. CONCLUSIONS

One of the crucial and most difficult parts in multimedia applications is DMM. In this paper, we prove the effectiveness of a new system-level exploration methodology to optimize the aforementioned DMM for typical multimedia applications applying it to a relatively new 3D reconstruction algorithm. This methodology allows a structured analysis of the memory access patterns hidden in algorithms with complex DM use and can also help to solve fundamental algorithmic problems. It allows the system integrator to obtain a detailed and clear view of the DM behavior and optimize it. In a first phase, the way in which the data is stored for the algorithm studied is optimized. By doing this, the access patterns to memory are transformed and optimized, reducing power consumption and memory footprint. Since most embedded systems do not have complex memory management provided by a combination of hardware and system software, an approach is proposed to compose efficient specific memory managers in a modular way, using high-level C++ code. Even though this paper only presents one driver application, similar results have been obtained in game engine algorithms [18].

8. REFERENCES

[1] Marc Pollefeys et al, "Metric 3D surface recons. from uncalibrated images," in *Lect. Notes in Comp. Science*, Springer-Verlag, 1998.

[2] John Cosmas et al, "3D murale," 2002, `http://www.brunel.ac.uk/project/murale/home.html`.

[3] "Eyetronics 3d scanning," `http://www.eyetronics.com`.

[4] Robin Rowe, "Industrial light and magic," *Linux Journal*, 2002.

[5] Paul R. Wilson et al, "Dynamic storage allocation, a survey and critical review," in *Int. Workshop on Mem. Management*, UK, 1995.

[6] Emery D. Berger et al, "Composing high-performance memory allocators," in *Proc. of PLDI*, USA, 2001.

[7] E.G. Daylight et al, "Analyzing energy friendly steady phases of dyn. apps. in terms of sparse data structs," in *Proc. of ISLPED*, USA, 2002.

[8] Roger Henriksson, *Scheduling Garbage Collection in Embedded Systems*, Ph.D. thesis, Lund Institute of Tech., 1998.

[9] N. Murphy, "Safe mem. usage with DM allocation," *Embedded Systems*, 49 – 57, 2000.

[10] Sven Wuytack et al, "Mem. manag. for embedded network apps.," *IEEE Trans. on Computer-Aided Design*, 18(5):533–544, 1999.

[11] L. Benini et al, "System level power optimization techniques and tools," in *ACM Trans. TODAES*, 2000.

[12] C.J. Harris et al, "A combined corner and edge detector," in *4th Alvey Vision Conf.*, 147–151, 1998.

[13] "Target jr," 2002, `http://www.targetjr.org`.

[14] T. M. Chilimbi et al, "Cache-conscious structure layout," in *SIGPLAN Conf. on PLDI*, 1–12, 1999.

[15] P. Ranjan Panda et al, "Mem. organization for improved data cache performance in embedded processors," in *Proc. of ISSS*, 90–95, 1996.

[16] B. S. Amrutur et al, "Speed and Power Scaling of SRAM's," *IEEE Trans. on Solid-State Circuits*, 35(2), 2000.

[17] Marc Leeman et al, "Intermediate var. elimination in a global context for a 3d multimedia application," in *Proc. of ICME*, USA, 2003.

[18] Marc Leeman et al, "Power estimation approach of dynamic data storage on a hw sw boundary level," in *Proc. of PATMOS*, Italy, 2003.