# Power Estimation Approach of Dynamic Data Storage on a Hardware Software Boundary Level*

Marc Leeman[1], David Atienza[2,3], Francky Catthoor[3], V. De Florio[1], G. Deconinck[1], J.M. Mendias[2], and R. Lauwereins[3]

[1] ESAT/K.U.LEUVEN, Kasteelpark Arenberg 10, 3001 – Leuven, Belgium;
[2] DACYA/U.C.M., Avenida Complutense s/n, 28040 – Madrid, Spain;
[3] IMEC vzw, Kapeldreef 75, 3000 Leuven, Belgium.

**Abstract.** In current multimedia applications like 3D graphical processing or games, the run-time memory management support has to allow real-time memory de/allocation, retrieving and data processing. The implementations of these algorithms for embedded platforms require high speed, low power and large data storage capacity. Due to the large hardware/software co-design space, high-level implementation cost estimates are required to avoid expensive design modifications late in the implementation. In this paper, we present an approach designed to do that. Based on memory accesses, normalised memory usage[1] and power estimates, the algorithm code is refined. Furthermore, optimal implementations for the dynamic data types involved can be selected with a considerable power contribution reduction.

## 1 Introduction

The fast growth in the variety, complexity and functionality of multimedia applications and platforms impose a high demand of memory and performance. This results in high cost and power consumption systems while current markets demand low power consumption ones. In addition, most of the new multimedia algorithms rely heavily on the use of dynamic memory due to the unpredictability of the input data at compile-time. This, combined with the memory hungry nature of certain parts of the algorithms, makes the dynamic memory subsystem one of the main sources of power consumption.

With the aforementioned characteristics, classical hardware (HW) design improvements, like voltage or technology scaling can only partially compensate for the growing HW/software (SW) gap [10]. In the last years, the boundary of what is considered to be critical design improvements for very large scale integration systems has been shifting consistently towards the SW side.

Even though a lot of current SW dominated transformations [3] result in platform independent improvements, they are critical for embedded devices due to the more constrained HW specification and especially of the memory hierarchy. Next to performance

* This work is partially supported by the Spanish Government Research Grant TIC2002/0750, the Fund for Scientific Research - Flanders (Belgium, F.W.O.) through project G.0036.99 and a Postdoctoral Fellowship for Geert Deconinck.

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

J.J. Chico and E. Macii (Eds.): PATMOS 2003, LNCS 2799, pp. 289–298, 2003.
© Springer-Verlag Berlin Heidelberg 2003

related techniques, power consumption is of paramount importance for hand-held devices. Even in devices that are not dependent on batteries, energy has become an issue due to the circuit reliability and packaging costs [15]. As a result, optimisation for embedded systems has three optimisation goals that cannot be seen independently from each other: memory usage, power consumptions and performance.

In order to optimise embedded system designs, detailed power consumption profiling must be available at an early stage of the design flow. Unfortunately, they do not exist presently at this level for the dynamically (de)allocated data type (further called Dynamic Data Type or DDT) implementations. In order to evaluate this accurately today, simulations would be necessary at a much closer level to the final implementation on a certain platform, e.g. at instruction (ISA) or cycle accurate HW level. Since each implementation of a DDT defines how the memory is accessed and allocated, they form an important factor for power consumption.

In this paper we explain how a high-level (i.e. from `C++` code) profiling approach is able to analyse and extract the necessary information for a power-aware refinement of the DDT implementations involved in multimedia applications at run-time. To this end, the three important factors that influence the power consumption and overall performance of the memory subsystem are studied, i.e. the memory usage pattern over time, the amount of memory accesses and the data access mechanisms. This power analysis approach is a crucial enabling step to allow subsequent optimisations and refinements. In the latter stages, the DDT implementations can be optimised based on relative power contribution estimates early on during system integration, enabling large savings in the design-time of the system.

The remainder of this paper is organised as follows. In Section 2, we describe some related work. In Section 3, we explain the high-level profiling phase and further refinements that it allows. In Section 4, we describe our drivers and present the experimental results. Finally, in Section 5, we state our conclusions.

## 2 Related Work

A large body of SW power estimation techniques have been proposed at lower abstraction levels, starting from code that is already executable on the final platform. One of the first papers is [14] and many contributions have added to that. But none of these has explicitly modelled the contribution of the dynamically allocated data types in the memory hierarchy of the platform. Work to obtain accurate figures on a higher level is more recent (e.g. [2,15]).

Although the level of such estimations has been extended to the assembly code and also `C` code, they are based on an analysis and design space without run-time analysis. This is not sufficient to deal with dynamic memory applications. In fact, in algorithms governed by dynamic memory accesses and storage (such as multimedia applications) the control flow and accesses to the DDTs are unknown at compile-time, and the aforementioned run-time analysis becomes necessary.

Most of the power estimation systems focus on obtaining accurate absolute values for HW-SW systems. In the framework of DDT optimisations and refinements, this is an

overkill and we are mainly interested in the *power contribution* of the dynamic memory sub-system. As such, accurate relative figures are more important.

Several analytical and abstract power estimation models at the architecture-level have received more attention recently [4] since they are needed for high-level power analysis in very large scale integration systems. However, they do not focus on the dynamic memory hierarchy of the systems and they are not able to analyse the power consumption from the DDTs at the SW level for dynamic data-dominated applications like multimedia applications.

To solve this gap in the power analysis context with respect to global dynamic memory profiling, the approach we propose is inspired partially from [17], but it is clearly different in a number of important parts. Basically, this reference handles applications from the network routing domain, where only one simple DDT is used and the main focus is in the multiple tasks running on the system. In fact, once the system has been initialised, the usage of the DDTs in that application domain does not vary much and averages around the same values. A snapshot of the memory footprint at any time during execution gives a reasonably good image of the memory behaviour. Therefore, in [17], a detailed profiling is not performed at run-time for DDTs and the memory footprint is used to determine the memory contribution in the power estimations.

In addition, according to the characteristics of certain parts of multimedia applications, several transformations for DDTs [16] and design methodologies [3] have made significant headway for static data profiling and optimisations taking into account static memory access patterns to physical memories. Also, the access to the data at a SW level has to take power consumption into account and research has been started to propose suitable power-aware data structure transformations at this level for embedded systems [5].

# 3 Description of the Approach

## 3.1 General Framework

The main objective of our high-level profiling is to provide the necessary run-time information of the DDTs used by an algorithm in a very early stage of the development flow. To do this, the algorithm that needs to be ported and optimised must go through a number of phases. First of all, the source code is analysed (including its structure in classes) and the profiling code to extract accurate information of the DDTs at run-time is inserted. Secondly, it is executed and the profiling information is stored. Finally, this information is processed to get the necessary power estimations and memory accesses reports in a post-processing phase.

After the analysis, the detailed power and timing representation of the DDTs is used to analyse and optimise their interactions in the global source code flow (i.e. intermediate variable elimination transformation [9]). Finally, the refinement of the DDTs can be performed with an exploration that uses the same profiling framework to evaluate possible trade-offs between power consumption, memory footprint and performance. Figure 1 gives an schematic overview of the overall optimisation approach that we propose, which is not the topic of this paper but it is summarised here to show the context of our power analysis approach. For a more in-depth discussion, see [8].

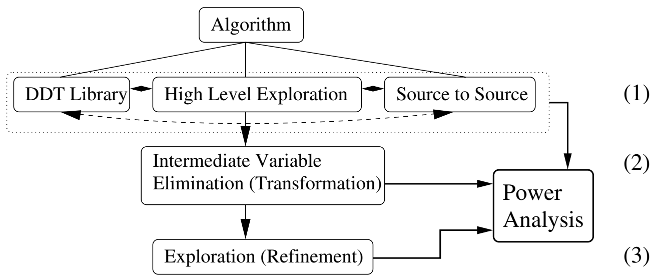


**Fig. 1.** Overview of the used system-level refinement approach

One of the most important characteristics of this approach is that it is based on a phase-wise exploration and refinement. Every phase is ideally self-contained and once an algorithm is optimised at that level, it can be handed down to the next phase in the design flow, i.e. a more HW oriented optimisation. As a result, the development team still has sufficient freedom to make (significant) changes without expensive re-iterations through the entire development flow.

The main features of our power analysis approach to support the optimisation and refinement steps are outlined in the following:
**multiple and complex dynamic data types:** All the considered multimedia applications employ a number of complex dynamic data types with very different behaviour.
**automatic instrumentation and insertion of profile objects:** Instead of tedious manual code transformations, an automatic tool has been developed to support our approach. It analyses the dynamic memory accesses in the application and modify the custom DDTs sources from developers to include all the information required for profiling.
**structured reports generation:** In applications which multiple DDTs, an analysis of the hierarchy of classes and structure of the source code must be done. Therefore, a profile framework has been developed to collect run-time profile information analysing where memory is used in the different parts of the DDTs, i.e. allocations blocks, intermediate layers, etc.
**detailed power and timing information acquisition:** In multimedia applications, during an algorithm run, data sets can be dominant and non-existent in other parts. As such, memory can be re-used and dynamic memory usage can vary a lot. Normalised memory usage is used to give a better representation of a DDTs impact. Profile runs gather detailed memory access patterns and memory usage for power consumption estimates.
**layered DDT library:** The implementation of complex DDTs can be considered as layered implementations of basic DDTs[2] [13]. A library provides standardised interfaces to the most commonly used $N$-layered implementations.
**source to source transformations are possible:** Using structured profiling reports, a global source code optimisation is able to eliminate temporary buffers that introduce memory movements between intra-algorithmic phases without any useful processing of data [9].

[2] All DDTs are a combination of a list, array and tree

### 3.2 Profiling Phase

As pointed out previously, it is mandatory to obtain run-time information about the DDTs to optimise the system. The developer has several choices to modify and explore the DDT search space. When the developer wants to evaluate the internal data structures provided with the algorithm or has his own library in C++. In that case, the *automatic insertion of profile collectors* modifies and instruments the sources including the profile framework. A second option is linking the algorithm sources with the provided library of multi-layered DDTs. The library provides standardised interfaces that need to be integrated in the algorithm sources. Finally, a third option is to explore DDTs not yet included in his custom sources or in the DDT library implementations. In that case, a modular approach for composing multi-layered DDTs is provided, based on mixins [12].

The search space for DDTs is then explored with an heuristic or exhaustive fashion, depending on the complexity of the program. The profile runs enable the framework to extract detailed run-time information. Finally, an automated post processing extracts timing information and power estimates of each DDT combination used in the exploration run. In Figure 2, the timing visualisation obtained for the multimedia drivers are shown.

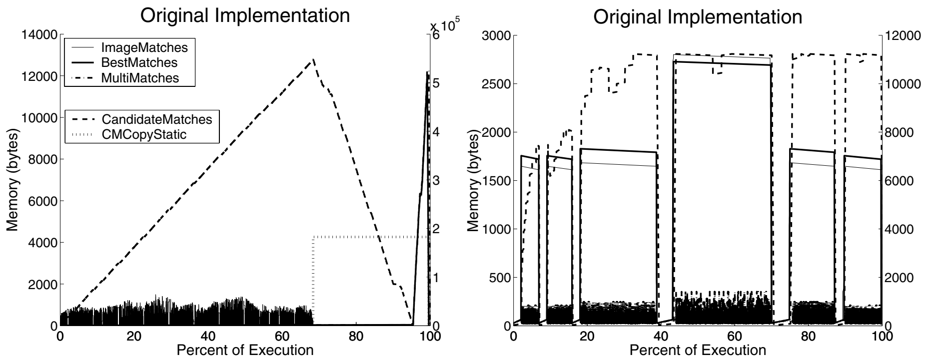


**Fig. 2.** On the left, memory behaviour of the matching algorithm between 2 frames (only *CandidateMatches* and *CMCopyStatic* plotted on the right axis). On the right, a similar plot for processing 6 frames in the game engine. The dashed line should be plotted on the right axis.

### 3.3 Memory Power Models

As it has been explained, for the profiling phase a realistic model for the dynamic memory subsystem is necessary. We have used initially the CACTI model [7], which is a complete energy/delay/area model for embedded SRAMs. It has two main advantages. First, a clear hierarchy in the modelling of the different memory components at four different levels. The first level includes modelling of transistors and interconnect wires. The second level is where these devices are combined to form the memory circuits, i.e. address decoder, SRAM cell, etc. For the delay, the Horowitz approximation [6] is used,

while the energy consumption depends only on the equivalent circuit capacitance, supply voltage and swing voltage. The last level consists of an exploration phase that returns (among other results) the least power consumption values for an optimal partitioning scheme for the specific memory. The second main advantage of CACTI is the fact that it is scalable to different technology nodes.

With the aforementioned model, we have represented the different sizes of memories required and compared with real data-sheet values from Trimedia for the on-chip memories caches with a size of 32 KB and an SRAM of 1 MB at 166 MHz with $.18\ \mu$m technology. We also compared to a very recent model for large SRAMS [1].

From this, it became clear that the main drawbacks of this CACTI model are the outdated circuits and the old technology parameters to scale under $.18\ \mu$m technology. It was built originally developed considering the $.8\ \mu$m technology node. Consequently, we are also using a variation of it where we try to make more accurate the energy and delay contribution of the sense amplifiers and the address decoder using a certain percentage of the total sub-bank energy and delay (instead of a constant as in the original CACTI model). Therefore, the sense amplifiers contribute 20% in memory and the address decoder contributes 30% in memory energy consumption and 20% in delay. This way the results are more accurate since the delay and energy of these components depend on the memory size.

### 3.4 Global Source Code Transformations Phase

When the timing information has been produced in the profiling phase, global source code transformations taking into account the DDTs interactions are viable. This will be illustrated based on an application demonstrator. The details of the optimisation approach itself are given in [9]. It allows to analyse the behaviour of the DDTs of the application, as Figure 2 shows. In it, the small vertical lines in the lower part represent small temporary buffers used to evaluate the DDT `ImageMatches (IMatches)` and generate the DDT `CandidateMatches (CMatches)` in a first step. Later, it is used to build the fast DDT `CMCopyStatic (CMCStatic)`. In a final step, it allows the creation of `BestMatches (BMatches)` and `MultiMatches (MMatches)`.

In fact, most of the multimedia applications are written in this previously explained *data production and consumption* fashion. The algorithm can be subdivided in smaller components (often functions or method calls) with internal and specifically designed DDTs, which receive an input buffer, do some processing and produce the output.

As such, the dynamic data structures passed between functionality components become critical bottlenecks (comparable to physical memory bandwidth congestions). When these intermediate DDTs are not used for any other purpose and there is an injective relation in the data-flow, they can be removed [9]. Figure 3 shows the results obtained after these transformations for our multimedia drivers.

### 3.5 Dynamic Data Type Refinement Using High-Level Profiling

In the final phase of the approach proposed, alternative complex DDT implementations from our library are evaluated to refine the original implementations of the DDTs. For each implementation in the library, the same high-level profile framework explained in

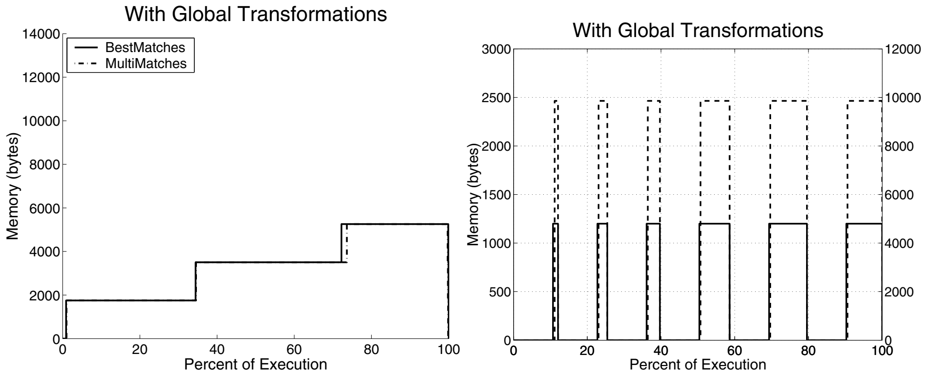


**Fig. 3.** Global source code transformations allowed the removal of 3 DDTs in the matching algorithm (left) and 2 DDTs in the game engine (right), saving memory footprint and power. The final code runs up to $10\times$ faster.

Section 3.2 is employed. In this refinement, optimal solutions are determined by a combination of the objectives pursued (i.e. power, memory accesses and normalised memory usage). As a result, a number of *Pareto optimal* [3] points, which represent different DDT implementations, are obtained. Then, the developer decides according to his constraints and requirements. An example of the Pareto points with the power models used for one of our multimedia drivers is shown in Figure 4.

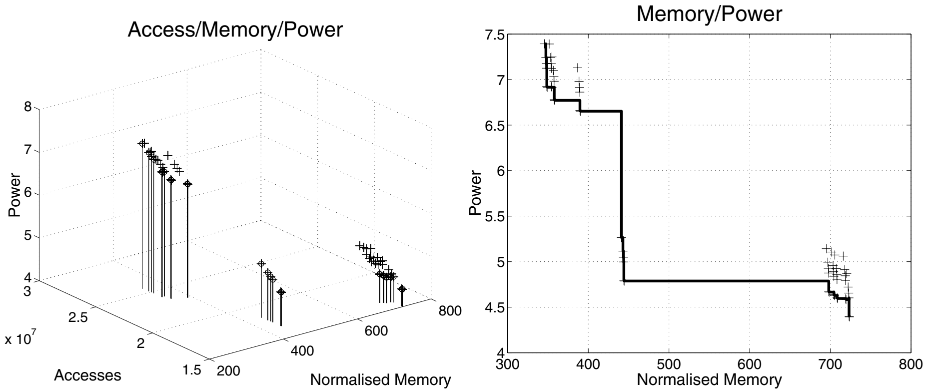


**Fig. 4.** The left figure shows the combination of pareto optimal solutions in the 3D game. The global pareto points are projected in the Memory/Accesses plain. They form a pareto curve. The left shows a projection of this 3D space in the Power/Memory plane. These figures are obtained with a Cacti based model for .18 $\mu$m technology.

[3] A point is called Pareto optimal, when it is impossible to improve one objective without worsening any other objective.

## 4 Multimedia Drivers and Results

To illustrate the approach presented in this paper, we have applied it to two applications. They represent two different multimedia application domains: the first application is part of a new image processing system, while the second one has been developed as a *game technology demo*.

The first application forms one of the corner-stones of a 3D reconstruction algorithm [11] and works like 3D perception in living beings, where the relative displacement between several 2D projections is used to reconstruct the $3^{rd}$ dimension. The global algorithm is subdivided in smaller building blocks (sub-algorithms).

The sub-algorithm under study forms the bridge between images or related frames and the mathematical abstraction that is used in subsequent phases. This implementation matches corners [11] detected in 2 subsequent frames (images) and the operations on images are particularly memory intensive (a $640 \times 480$ image consumes over 1 MB). This algorithm uses internally several DDTs that, due to the partial image-dependency related data, do not fit in internal memory of current embedded processors. The size of these DDTs is fixed by a number of factors (e.g. structure and textures in the images) determined outside the algorithm and are uncertain at compile-time. In this phase, the accesses to the images are randomised and classic image access optimisations like row dominated accesses versus column wise accesses are not relevant.

The second application where the methodology has been applied is a 3D simulation game driven by a frame grabbing device. In a frame, obstacles are detected in the scene. In the free-space area, balls are rendered. These balls can move according to 3 degrees of freedom (up/down, left/right, front/back). When the ball reaches a wall, it either bounces off the obstacle or gets stuck to the obstacle. In this case study, the uncertainty that determines the dynamic memory are the position of the obstacles in the input frames and the position, speed and direction of the generated balls.

Following the approach as sketched in Section 3, the source to source tool is used to add the detailed instrumentation and profile framework to the source code of the DDTs in both applications. One of the results of this initial profiling is detailed timing information as shown in Figure 2, there it can be seen that intra-algorithm data dependencies with small but extremely accessed buffers exist in both cases. Next, this information is used to apply global transformations [9]. This results in the removal of 3 DDTs in the matching algorithm and 2 DDTs in the game engine, as Table 1 and Figure 3 show.

On these *refined* versions of the algorithms, DDTs exploration is performed on representative input. For these results, the effect of 5 runs is considered to avoid random operating system behaviour[4]. For the explored DDT combinations, figures are obtained for memory usage, accesses and power. Figure 4 shows a subset of the obtained figures. From a designers point of view, only the pareto optimal points are interesting (circled). For both applications, the results for the least power consumption are shown in Table 1.

Even though the figures of the memory model change, the relative values are similar and let us select the same "optimal" DDTs. For `BMatches` and for `MMatches`, the final DDT implementation consist of a 2-layered array structure, with an external dynamic array of 10 positions, then each position consist of another array of 146 basic positions

[4] The results of the profiling runs was very similar, only minor variations were observed

of 3 floats each. For the game engine also the DDTs are 2-layered array structures, the DDTs that contain the walls are now implemented as a dynamic array in the first level of 10 positions. Then, another one of 56 basic elements for the vertical walls and 26 for the horizontal ones. In both cases, the basic elements consist of 6 floats. Finally, the highly accessed balls are implemented as a first dynamic array of 10 positions where dynamic arrays of 179 basic elements of 1 float each are stored.

In the end, as Table 1 shows, an improvement of normalised memory footprint up to 99.97% and power consumption up to 99.99% compared to the original implementation of the corner matching algorithm. Similarly, 26.6% and 45.5% (or 79.9% depending on the technology used) for the 3D simulation game. In addition, there was a final speedup (when the DDTs were refined in the last phase of the approach proposed) of almost 2 orders of magnitude for the matching algorithm and 1 order of magnitude for the 3D simulation game.

**Table 1.** Refinement results of the DDTs for both driver applications. Between parenthesis the percentage saved in power consumption and memory footprint (fprint) are given. The initial DDTs removed in the final version are marked as RM.

| DDTs | orig. mem. fprint (B) | orig. power .18$\mu$m ($\mu$J) | orig. power .13$\mu$m ($\mu$J) | final mem. fprint (B) | final power .18$\mu$m ($\mu$J) | final power .13$\mu$m ($\mu$J) |
|---|---|---|---|---|---|---|
| `IMatches` | $5.14\times10^2$ | $0.30\times10^3$ | $0.18\times10^3$ | RM | RM | RM |
| `CMatches` | $2.75\times10^5$ | $3.03\times10^3$ | $3.03\times10^3$ | RM | RM | RM |
| `CMCStatic` | $1.08\times10^5$ | $3.92\times10^5$ | $4.48\times10^4$ | RM | RM | RM |
| `MMatches` | $3.62\times10^2$ | $0.03\times10^2$ | $0.02\times10^1$ | $3.81\times10^3$ | $0.02\times10^2$ | $0.02\times10^1$ |
| `BMatches` | $3.07\times10^2$ | $0.04\times10^2$ | $0.02\times10^1$ | $3.81\times10^3$ | $0.03\times10^2$ | $0.02\times10^1$ |
| `Total: matching` | $3.85\times10^5$ | $3.95\times10^5$ | $4.80\times10^4$ | $7.63\times10^3$ (99.97%) | $0.05\times10^2$ (99.99%) | $0.04\times10^1$ (99.99%) |
| `VerWalls` | $1.72\times10^3$ | $2.96\times10^3$ | $2.94\times10^3$ | $8.30\times10^2$ | $2.01\times10^3$ | $1.52\times10^2$ |
| `VWallsBump` | $5.53\times10^1$ | $0.16\times10^3$ | $0.04\times10^2$ | RM | RM | RM |
| `HorWalls` | $1.64\times10^3$ | $0.15\times10^3$ | $0.28\times10^3$ | $6.99\times10^2$ | $1.77\times10^3$ | $1.28\times10^2$ |
| `HWallsBump` | $6.23\times10^1$ | $1.82\times10^3$ | $0.04\times10^2$ | RM | RM | RM |
| `Balls` | $8.42\times10^3$ | $9.33\times10^3$ | $1.91\times10^3$ | $7.20\times10^3$ | $4.18\times10^3$ | $7.54\times10^2$ |
| `Total: 3D game` | $1.19\times10^4$ | $1.46\times10^4$ | $5.14\times10^3$ | $8.73\times10^3$ (26.6%) | $7.96\times10^3$ (45.5%) | $1.03\times10^3$ (79.9%) |

## 5 Conclusions

Power and energy estimations at an early phase of system implementation has become an increasingly important concern. Power and timing estimations at a very high-level, i.e. SW level, early in the system design process for the DDTs present in modern multimedia applications are not available yet. At the same time, accurate estimations from lower-level models, e.g. RTL-level or gate-level, suffer from unacceptable long computing times and capture information that is not (yet) relevant. Moreover, they come too late in the global design flow of the final system, which implies a very costly set of iterations through

the design flow for any change in the first phases. In this paper, a fast and consistent system-level profiling approach that can be used to overcome the previous limitations is presented. It allows the designers to profile, analyse and refine the implementations and effects of the dynamic data types from their applications in a very early stage of the design flow taking into account power consumption, memory footprint and performance. Its effectiveness is illustrated using two different and complex multimedia examples. Finally, we have also explained how different power models can be used to obtain accurate results for a specific final platform if it is needed.

## References

1. B. S. Amrutur et al. Speed and Power Scaling of SRAM's. *IEEE Trans. on Solid-State Circuits*, 35(2) (2000)
2. L. Benin et al. A power modeling and estimation framework for vliw-based embedded systems. In *Proc. of PATMOS* , Yverdon Les Bains, Switzerland (2001) 2.1.1–2.1.10
3. F. Catthoor et al. *Custom Memory Management Methodology – Exploration of Memory Organisation for Embedded Multimedia System Design*. Kluwer Academic Publishers, Boston, USA (1998)
4. R. Y. Chen et al. Speed and Power Scaling of SRAM's. *ACM Trans. on Design Automation of Electronic Systems*, 6(1) (2001)
5. E. G. Daylight et al. Incorporating energy efficient data structures into modular software implementations for internet-based embedded systems. In *Proc. of wrkshp on Software Performance* (2002)
6. M. A. Horowitz. Timing models for mos circuits. Technical report, Technical Report SEL83-003, Integrated Circuits Lab. Stanford Univ. (1983)
7. N. Jouppi. Western research laboratory, cacti, (2002) `http://research.compaq.com/wrl/people/jouppi/CACTI.html`.
8. M. Leeman et al. Methodology for refinement and optimisation of DM management for embedded systems in multimedia applications. In *Proc. of SiPS*, Seoul, Korea (2003)
9. M. Leeman et al. Intermediate variable elimination in a global context for a 3d multimedia application. In *Proc. of ICME* , Baltimore, MD (2003)
10. T. Mudge. Power: A first class architectural design constraint. *IEEE Computer*, 34(4):52–58, (2001)
11. M. Pollefeys et al. Metric 3D surface reconstruction from uncalibrated image sequences. In *Lecture Notes in Computer Science*, volume 1506, Proc. SMILE Wrkshp (post-ECCV'98), Springer-Verlag (1998) 139–153.
12. Y. Smaragdakis et al. Implementing layered designs with mixin layers. *Lecture Notes in Computer Science*, 1445:550 (1998)
13. B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, Inc., Harlow, England (1997)
14. V. Tiwari et al. Power analysis of embedded software: A first step towards software power minimization. In *Proc. of ICCAD*, San Jose, California, USA (1994)
15. N. Vijaykrishnan et al. Evaluating integrated hardware-software optimizations using a unified energy estimation framework. *IEEE Transactions on Computers* (2003) 52(1):59–75
16. S. Wuytack et al. Global communication and memory optimizing transformations for low power systems. In *IEEE Intnl. wrkshp on Low Power Design*, Napa CA, (1994) 203–208.
17. C. Ykman et al. Dynamic Memory Management Methodology Applied to Embedded Telecom Network Systems. *IEEE Transactions on VLSI Systems* (2002)