Iason Filippopoulos

Exploration of energy efficient memory organizations exploiting data variable based system scenarios

Doctoral thesis
for the degree of philosophiae doctor

September 2015

Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Electronics and Telecommunications

Catholic University of Leuven
Department of Electrical Engineering

NTNU
Innovation and Creativity

KU LEUVEN

NTNU

Norwegian University of Science and Technology

Doctoral thesis
for the degree of philosophiae doctor

Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Electronics and Telecommunications

Catholic University of Leuven
Department of Electrical Engineering

ISBN (printed version)
ISBN (electronic version)
ISSN 1503-8181

Doctoral theses at NTNU,

Printed by NTNU-trykk

Abstract

Modern embedded systems are capable of performing a wide range of tasks and their popularity is increasing in many different application domains. The recent progress on the semiconductor processing technology greatly improves the performance of the embedded systems, due to the increased number of transistors on a single chip. The continuous performance improvement increases the possibilities for new embedded system designs. However, many embedded systems rely on a battery source, which puts a significant limitation on their lifetime and usage.

In this thesis, we focus on the design of energy efficient memory architectures for embedded systems. A hardware/software co-design methodology is proposed for the reduction of the energy consumption on the memory subsystem. The methodology exploits variations in memory needs during the lifetime of an application in order to optimize energy usage. The different resource requirements, that change dynamically at run-time, are organized into groups to efficiently handle a very large exploration space. Apart from the development of the methodology, an extended memory model is included in this work. The memory models is based on existing state-of-the-art memories, available from industry and academia. In addition, the impact of the technology scaling is studied and the effectiveness of the proposed methodology is analyzed for the future memory architectures.

We also investigate the combination of the developed methodology with known code transformation techniques, specifically data interleaving. The proposed design methodology aims to be compatible with the already available code optimization techniques. We further extend the evaluation of the memory design methodology using a test-case wireless system. The proposed reconfigurable memory subsystem is studied in a dynamic platform with several reconfiguration options that combine the memory and the processing elements.

Preface

This doctoral thesis was submitted to the Norwegian University of Science and Technology (NTNU) in partial fulfillment of the requirements for the degree philosophiae doctor (PhD). The thesis is a part of a dual PhD program between NTNU and the Catholic University of Leuven (K. U. Leuven), in cooperation with Interuniversity Microelectronics Center (IMEC). The work herein was performed at the Department of Electronics and Telecommunications, NTNU and the Department of Electrical Engineering, KU Leuven. The work was performed under the supervision of Professor Per Gunnar Kjeldsberg and Professor Francky Catthoor.

Acknowledgements

I would like to thank my supervisors Prof. Per Gunnar Kjeldsberg and Prof. Francky Catthoor for their support and advise through the long process that eventually became this thesis. I also extend my gratitude to my co-supervisor Prof. Sverre Hendseth for providing encouragement and positive attitude all these years.

I would also like to thank Mladen for sharing both his direct and thought-provoking opinions and office space with me. Furthermore, I thank Elena and Yahya for our great co-operation and their patience on the evaluation meetings. I was lucky to meet great co-workers at NTNU and IMEC that offered their help.

Finally, I would like to thank my family and friends for their support.

Iason Filippopoulos
September 2015

Contents

List of Tables

List of Figures

List of Symbols

Chapter 1

Introduction

1.1 Embedded Systems and Energy Consumption - what is our general goal

1.2 Data Intensive Applications - what is our general goal

1.3 Brief summary of current way of tackling and what has not been addressed here

1.4 Problem Statement

1.5 Proposed Solution

1.6 Thesis Outline - what is covered in which paper

Chapter 2

Background

2.1 Scratch-pad Memory Architectures - related work incl

2.2 System Scenarios - all the literature with focus on the ones related

- vs. use case - pareto

2.3 DTSE

-

Chapter 3

Methodology

3.1 System Scenario Platform

3.1.1 Target memory platform architecture

3.1.2 Memory models

3.1.3 Technology scaling

3.2 Data variable based memory-aware system scenario methodology

3.2.1 Interleaving exploration based on data variables

3.2.2 Design-time profiling based on data variables

3.2.3 Design-time system scenario identification based on data variables

3.2.4 Run-time system scenario detection and switching based on data variables

Chapter 4

Research Results and Contributions

4.1 Contribution A: Memory-Aware Methodology Development

4.2 Contribution B: System Scenarios on Memory and PEs

4.3 Contribution C: Integrated Interleaving and Data-to-Memory Mapping

4.4 Contribution D: Interconnection Cost Scaling

4.5 Paper A.I - Abstract - My contribution

4.6 Paper A.II

4.7 Paper A.III

4.8 Paper B.I

4.9 Paper C.I

4.10 Paper D.I

Chapter 5

Conclusions

Appendix A

Energy Impact of Memory-Aware System Scenario Approach

Iason Filippopoulos, Francky Catthoor, Per Gunnar Kjeldsberg,
Elena Hammari and Jos Huisken
IEEE NORCHIP conference
2012

Abstract

System scenario methodologies propose the use of different scenarios, e.g., different platform configurations, in order to exploit variations in computational and memory needs during the lifetime of an application. In this paper several extensions are proposed for a system scenario based methodology with a focus on improving memory organisation. The conventional methodology targets mostly execution time while this work aims at including memory costs into the exploration. The effectiveness of the proposed extensions is demonstrated and tested using two real applications, which are dynamic and suitable for execution on modern embedded systems. Reductions in memory energy consumption of 40 to 70% is shown.

A.1 Introduction

Modern embedded systems are becoming more and more powerful as the semiconductor processing technique keep increasing the number of transistors on a single chip. Consequentially, demanding applications, such as medical signal processing and streaming applications, can be executed on these devices [14]. On the other hand, the desired performance has to be delivered with the minimum power consumption due to limited amount of power offered in mobile devices [12]. System scenario methodologies propose the use of different platform configurations in order to exploit variations in computational and memory needs often seen during the lifetime of such applications [12]. A platform can, e.g., be configured through frequency/voltage scaling or turning certain processing units on or off. In this work a reconfigurable memory platform is employed in order to study the effectiveness of a memory-aware system scenario methodology.

As shown in [8] memory contributes around 40% to the overall power consumption in general purpose systems. Especially for embedded systems, the memory subsystem accounts for up to 50% of the overall energy consumption [5] and the cycle-accurate simulator presented in [22] estimates that the energy expenditures in the memory subsystem range from 35% up to 65% for different architectures. According to [12], conventional allocation and mapping of data done by regular compilers is suboptimal. Performance loss is caused by stalls for fetching data and data conflicts for different tasks, due to the limited size of memory and the competition between tasks. In addition, modern applications exhibit more and more dynamism. This gives a strong motivation for study and optimization of memory organisation in embedded devices with strongly dynamic application behaviour.

This paper is organized as follows. Section A.2 surveys related work on system level exploration and on system scenario methodologies. Section A.3 presents the chosen methodology and our novel extensions to make it applicable in a memory organisation study. In Section A.4 the target platform is described while the demonstrator applications are presented in Section A.5. Results of applying the described methodology to the targeted applications are shown in Section A.6, while conclusions are drawn in Section A.7. The main contribution of the current work is the proposal of extensions to the existing system scenario methodology, in order to take into account memory costs while performing the design exploration.

A.2 Related Work and Contribution Discussion

Many papers have focused on memory related optimisations, also in the presence of a partitioned and distributed memory organisation with memory blocks of different sizes (e.g. [3], [2], [13], [19]). However, they do not incorporate sufficient support for very dynamically behaving application codes. System scenarios allow to alleviate this bottleneck and to handle such dynamic behaviour. An overview of work on system scenario methodologies and their application are presented in [7]. So far memory organisation is rarely considered and not fully analysed. Furthermore, the majority of the published work focus on control variables for scenario prediction and selection. They can take a relatively small set of different values that can be fully explored. However, the use of data variables [9] is required for these tasks by many dynamic systems with value ranges that make full exploration impossible.

Authors in [17] present a technique to optimise memory accesses for input data dependent applications by duplicating and optimising the code for different execution paths of a control flow graph (CFG). One path or a group of paths in a CFG form a scenario and its memory accesses are optimized using global loop transformations (GLT). Apart from if-statement evaluations that define different execution paths, they extend their technique to include while loops with variable trip count in [18]. A heuristic to perform efficient grouping of execution paths for scenario creation is analysed in [16]. However, our work extends the existing solutions towards exploiting the presence of a distributed memory organisation with reconfiguration possibilities.

Reconfigurable hardware for embedded systems, including the memory architecture, is a topic of active research. An extensive overview of current approaches is found in [6]. The approach presented in this paper differentiates by focusing on the data-to-memory partitioning aspects in the presence of a platform with dynamically configurable memory blocks.

A.3 Extended System Scenario Methodology

The system scenario methodology is based on the observation that the workload of most systems varies significantly during their lifetime due to dynamic variation of computational and memory needs in the application code. Most of the existing design methods define the worst case execution time (WCET) of the most demanding task and tune the system in order to meet its needs [12]. Obviously, this approach leads to wasted time and memory area for tasks with lower execution time and memory requirements, since those tasks could meet their needs using fewer resources and consequentially consuming less energy.

In contrast, designing with scenarios is workload adaptive and offers different configurations of the platform and the freedom of switching to the most efficient scenario at run-time. In contrast to use case scenario approaches in which scenarios are generated based on a user's behaviour, the system scenario methodology focuses on behaviour of the system to generate scenarios. A system scenario is a configuration of the system that combines similar run-time situations (RTSs). An RTS consists of a running instance of a task and its corresponding cost (e.g. energy consumption) and one complete run of the application on the target platform represents a sequence of RTSs [9]. The system is configured to meet the cost requirements of an RTS by choosing the appropriate scenario, which is the one that satisfies the requirements using minimal power.

In the following subsections, the different steps of the system scenario methodology is outlined, with emphasis on the new extensions that make it possible to take memory into account.

A.3.1 General description of system scenario methodology

The scenario methodology follows a two stage exploration, namely design-time and run-time stages, as described in [7]. The two stage exploration is chosen because it reduces run-time overhead while preserving an important degree of freedom for run-time configuration [12]. In more detail, the application is analysed at design-time and different execution paths and variations in processing and memory demands are identified. This procedure, which is time consuming and as a result can be performed only during the design phase, will result in a grey-box model representation of the application. The grey-box model hides all static and deterministic parts of the application, by providing only related costs for those, and keeps parts of the application code that are non-deterministic available to the system designer [10]. This way, more focus can be given to parts of the application that have impact on the cost variations so that different execution decisions can be studied. Those different options are made available to the system designer and each decision could be either fixed at design time or forwarded for dynamic evaluation at run-time.

A.3.2 Design-time Profiling

Application profiling is performed at design-time and consists of an analysis of the target application during its lifetime and for a wide range of inputs. The analysis focuses on execution time for a reference processor in the conventional system scenario methodology, but in our case the cost will be memory size. This cost metric is chosen, because several applications allocate and deallocate memory space during their execution. Also, data reuse behavior and decisions made by the system designer about the size of sets of data to be copied to different units in the memory hierarchy (i.e., copy-candidates [4]) strongly influences the memory size. Energy consumption and access time of a memory unit is affected by its size [20]. Together with access pattern information, the memory size is hence an important metric.

The flow of the profiling stage is depicted in Fig. A.1 and consists of running the application code with suitable input data often found in a database, in order to produce profiling results. This reveals parts of the application code with high memory activity and with varying memory access intensity, which possibly depends on input data. Because of this behaviour, a static study of the application code alone is insufficient since the target applications for this methodology have non-deterministic behaviour that is driven by input. Choosing an extensive and accurate database is vital and will heavily influence and steer the designer's decisions in later steps.

Given code and database as inputs, profiling will show memory usage during execution time by running the application using the whole database as an input. Results provided to the designer include complete information about allocated memory size values together with the number of occurrences and duration for each of these memory size values. Moreover, correlation between input data values and the resulting memory behaviour can possibly be observed. This information is forwarded to the next stage.

There are several ways that profiling can be performed. Current program monitoring software cannot offer the needed level of detail. Debuggers and memory tools, such as Valgrind [15], or direct hard-coded profiling are preferable methods, because of their higher accuracy.

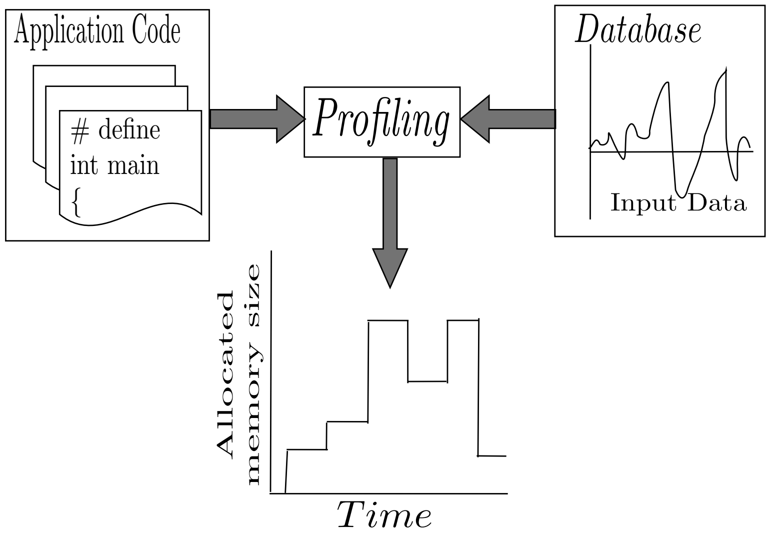


Figure A.1: Profiling results based on application code and input data.

A.3.3 Design-time Scenario Identification and Prediction

Scenario generation is also performed during design time and is the procedure of clustering profiled information into groups with similar characteristics. More precisely, cost points are clustered in groups based on their distance in cost axis and the frequency of their occurrence. Clustering is necessary, because it will be extremely costly to have a different scenario for every possible situation. Switching cost, which is the cost of switching platform to another configuration, for example by turning a memory to retention state, grows with the number of scenarios, because more scenarios result in more frequent switching. In addition, the runtime manager becomes more complex with an increasing number of scenarios.

As shown in Fig. A.2, using the information available after application profiling, situations with similar platform needs will be organised in a common scenario. This is known as clustering of RTSs [7]. This is a rational choice, because two instances with similar platform needs have similar memory energy consumption. The energy gain of having two dedicated scenarios is small and normally outweighed by switching costs if organized in different scenarios. In this example, clustering results in three different scenario areas (Fig. A.2) and the execution scenario sequence is 1(lower), 2(middle), 3(top), 2, 3 and 1. Also, the frequency of occurrence of each instance is an

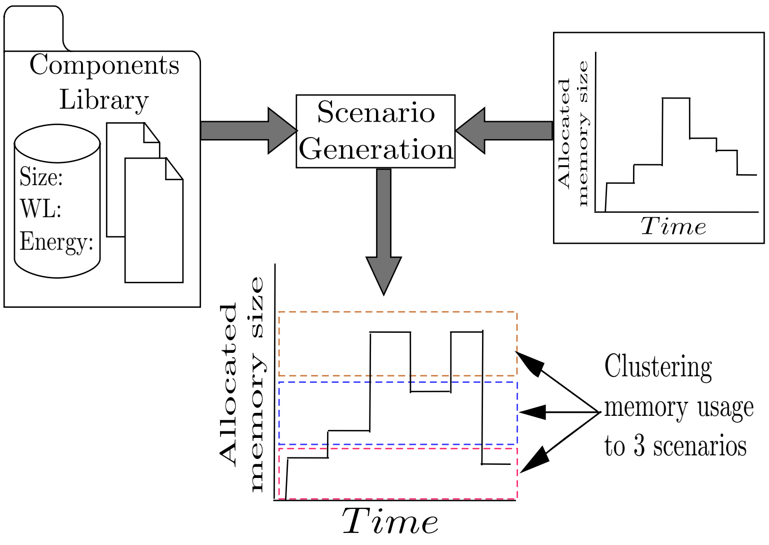


Figure A.2: Scenario generation based on profiling information and memory models.

important factor in scenario generation. Instances with higher frequencies normally have their dedicated scenario, to allow higher optimization, while less common instances can be clustered together in a less optimal scenario.

In order to generate scenarios that can be implemented in realistic memory organisations, a detailed library with memory components is needed. The library should contain a variety of memory models, including different technologies and sizes, and is used to calculate scenario costs. In this work the component library is based on memory models published in [1].

The design-time scenario prediction phase consists of determination of the variables that define the active scenario. This can be achieved by careful study of the application code, combined with the application's data input. In our case the grey-box model reveals only the code parts that will influence memory usage, so that variables deciding memory space changes can be identified. An example of this is a non static variable that influences the number of iterations for a loop that performs one memory allocation at each iteration. Moreover, the designer should look for a correlation between input values and the corresponding cost. This information will be useful in the following steps of the methodology [12].

A.3.4 Run-time Identification, Detection, and Switching

After profiling and scenario generation at design-time, all the necessary information needed for run-time management is available. The run-time manager monitors the current values of prediction variables selected in the previous design time step. Based on this, prediction of the next active scenario is performed [12].

Switching decisions will be taken at run-time by the run-time manager. The switching phase consists of all platform configuration decisions that can be made at run-time, e.g., frequency/voltage scaling, turning on/off a memory unit, and remapping of data on memory units. Switching takes place when the switching cost is lower than the energy gains achieved by switching. In more detail, the run-time manager compares the memory energy consumption of executing the next task in the current active scenario with the energy consumption of execution with the optimal scenario. If the difference is greater than the switching cost, then scenario switching is performed [12]. Switching costs are defined by the platform and include all memory energy penalties for run-time reconfigurations of the platform, e.g., extra energy needed to change state of a memory unit.

A.4 Target Platform

Selection of target platform is an important aspect of the memory-aware system scenario methodology. The key feature needed in the platform architecture is the ability to realize different scenarios generated by the methodology. Execution of different scenarios then leads to different energy costs, as each configuration of the platform results in a specific memory energy consumption. In the conventional system scenario methodology several platform reconfiguration options have been studied [7] with dynamic voltage/frequency scaling (DVFS) most used [21]. By using DVFS techniques, one can change the frequency of the processing element and its supply voltage and energy consumption accordingly. For a memory aware scenario methodology there are multiple ways that memory reconfiguration can be performed. E.g., data can be mapped to different memory units according to size and access frequency, memory units can be turned on or off, or DVFS can be used to allow exactly the access frequency currently needed.

In this paper we have selected a relatively conventional reconfigurable

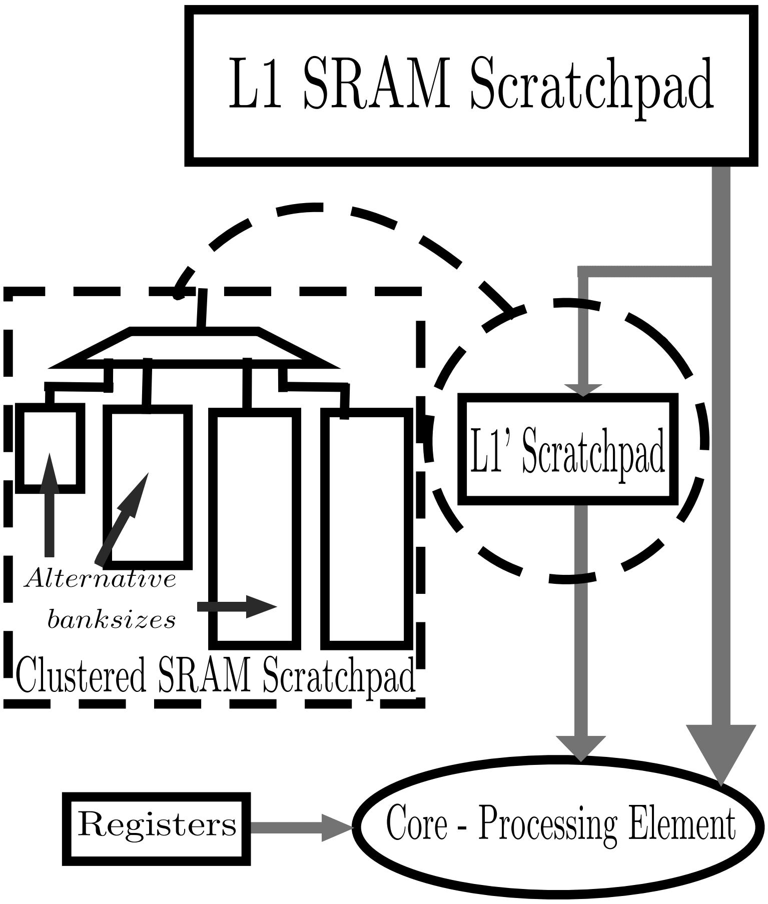


Figure A.3: Target platform with focus on memory organisation.

architecture template that is suitable for implementing system scenarios as shown in Fig. A.3. This dynamic memory organisation is based on memory prototypes presented in [1], and consists of two software controlled SRAM scratchpad memories, L1 and L1', and a processing element with its registers. For simplicity it is assumed that all data needed during execution is available in the L1 scratchpad memory without any time penalties even when large background memories are used. This assumption is reasonable for the kind of applications that are generally executed in embedded systems and can, e.g., be achieved through prefetching. The methodology is in general not restricted to this assumption, however, and can handle more complex hierarchical memory architectures, also including regular caches. Registers are used to save currently used elements, and between the reg-

Table A.1: Energy costs per second and access for a reference memory bank (size: 120 bytes, technology: 90nm) in clustered L1' memory

Modes	Leakage [J]	Wake up [J]	Access [J]
On	$351,86 \times 10^{-6}$	-	-
Off	$4,73 \times 10^{-6}$	$2,77 \times 10^{-12}$	-
Retention	$97,33 \times 10^{-6}$	$2,2 \times 10^{-12}$	-
DFF synch	-	-	0.227×10^{-6}
DFF asynch	-	-	2.18×10^{-6}
Nand2	-	-	0.334×10^{-6}

isters and the L1 scratchpad a much smaller L1' scratchpad is introduced. This is a clustered memory that consists of four memory banks that support three different states (on, off and retention modes) [1].

The memory energy consumption for every access in the clustered scratchpad memory is calculated based on the current situation of the platform as shown in Fig. A.3. The leakage energy estimation is also provided in [1] and is included in our study. In Tab. A.1 the leakage and wake up costs for all different operation modes are presented. The leakage in L1' is small compared to L1 and constant, thus the effect in the results is minimal.

The memory energy consumption per read access is given by Equation A.1, which is provided by the memory models in [1].

$$\begin{aligned} Read\,energy &= (size\,of\,bank) \times (DFF\,sync) + \\ &+ \frac{bank\,lines}{log2} \times DFF\,async + 4 \times size \times Nand2 \end{aligned} \tag{A.1}$$

The cost of accessing the clustered scratchpad organisation is a function of the overall size of the cluster and the size of the specific bank being accessed. In addition, there is a contribution in energy consumption per access added by the flip-flops and the gates needed for the correct operation of the L1' memory organisation.

A.5 Application Benchmarks

The extended methodology will be tested on two representative real life applications that differentiate significantly from each other and cover different domains of applications. The first one is a computational intensive application, in which the code is dominated by loops with dynamic bounds determined by results calculated inside the loop. The second has a more static execution path, but its memory footprint is dynamically defined by the input data. Ideal applications, that can most benefit from memory-aware system scenario methodology, are applications that have dynamic behaviour in memory organisation utilization during their execution. That required dynamism could be produced by several code characteristics, covering a wide range of potential application domains for the proposed methodology.

A.5.1 Epileptic Seizure Predictor

An epileptic seizure predictor algorithm developed at Arizona State University (ASU) [11] is chosen as a benchmark, along with a database with measurements performed on real patients, also provided by ASU. Up to now the algorithm has only been used in clinical environments using PCs, although it would be very helpful for patients as an embedded device, as part of outpatient care. Dynamic input data dependent behaviour is identified in the algorithm and memory traces for execution of three different input samples are presented in Fig. A.4. For each data sample a loop is iterated 168 times. Depending on the input data sample, different memory elements are accessed in each iteration. Note that even though the element accessing for all three samples are drawn together in the figure, only one set of memory elements are accessed for each data sample and only one data sample is processed at a time. Because of the nature of the EEG signal used for the epileptic seizure prediction the benchmark has variables with a very wide range of potential values. The memory usage is heavily influenced by these input values. For demonstration purposes our study focuses on the parts of the prediction algorithm that is most dynamic. In [9] the algorithm is split in thread nodes and dynamic behaviour affecting execution time is identified. The data reuse size for each sample in Fig. A.4 contains all the elements with data reuse factor greater than 1, i.e., are read more than once. They should be saved in L1'. For example, elements in index range from roughly 3300 to 7200 for sample 1 form an approximately 3900×(element size bytes) L1' memory size requirement. These elements are read for the first time between loop iteration 0 and 50 and are also re-read later during the sample's lifetime between loop iteration 120 and 170. In Fig. A.4 these memory elements are included in the red rectangle. The exact L1' sizes needed for processing of sample 1, 2, and 3 are 3899, 2646, and 3780,

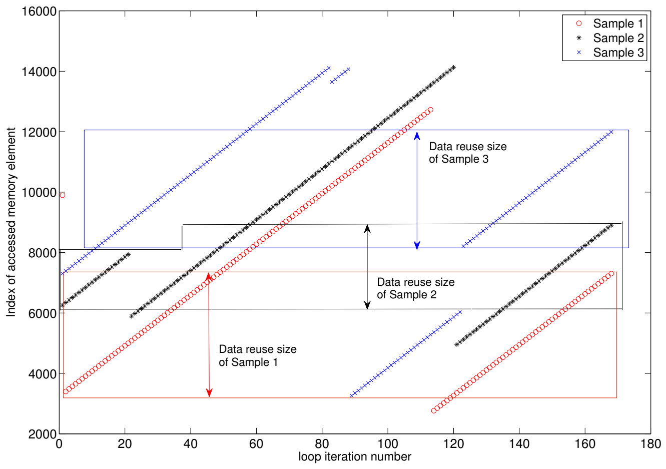


Figure A.4: Memory access pattern of epilepsy predictor. Profiling of data reuse size for 3 samples from a given ([11]) database. The number of elements accessed is input dependent. Only those of the 16K elements accessed multiple times should be saved in L1'. The rest are accessed from L1.

respectively.

Based on profiling results, scenario generation can be performed. Two possible clustering solutions are used in this paper, both of them consisting of five scenarios. In the first clustering, the range of memory indexes is split in equally sized partitions. In the second clustering, the range is split based on occurrence frequency, so that each scenario has almost equal number of occurrences. The reconfigurable memory platform is instantiated accordingly in our target template outlined in Section A.4. It contains four equal memory banks of size 975 in the first case and four banks of increasing size (195, 585, 1170, and 1950) in the second case, since smaller sizes are used more often.

A.5.2 Viterbi Algorithm Encoder

The Viterbi algorithm [24] is widely used, both in the industry and academia, as an encoding/decoding algorithm for convolutional codes. As part of the encoding of the transmitted signal, redundant bits are added, which are later used by the decoder for correction of transmission errors. The output

Table A.2: Constraint length for SNR levels on the channel (BER $\leqslant 10^{-5}$)

SNR (dB)	K	L1' size (B)	SNR (dB)	K	L1' size (B)
6,5 - 6,1	5	40	3,9 - 3,1	9	72
6,1 - 5,5	6	48	3,1 - 2,8	12	96
5,5 - 3,9	8	64	2,8 - 2,5	14	112

of an encoder is a function of current input bit(s) and K earlier inputs, where K is the constraint length of the algorithm. Profiling shows that the constraint length dominates the memory cost. In Tab. A.2 [23] constraint length values to achieve 10^{-5} BER are presented for different noise levels in the channel. Due to its limited range of values, the constraint length is classified as a control variable. The scenario generation is based on profiling and aims at achieving a consistent bit error rate with the minimum memory usage. If SNR decreases, another memory bank is activated while reduction in channel noise leads to the opposite effect. The L1' sizes can be found in Tab. A.2. Clustering has been performed for K-values 5 and 6, with a resulting L1' size of 48B, and between K-values 8 and 9, with a resulting L1' size of 72B

A.6 Results

The memory aware system scenario methodology is applied to both of our benchmark applications to study its effectiveness. Memory energy consumption is calculated based on [1] and is the sum of (energy per access) $\times$ (number of accesses) and energy costs for all transitions between memory modes. The energy for each access is defined by the type and the size of the accessed memory. Based on profiling and scenario generation results, four different memory organizations are compared for the epileptic seizure predictor. The first one is a scenario strategy without memory awareness and use a single memory bank large enough to satisfy the most demanding sample. That approach is the worst case for the memory size and statically allocates the highest value of the memory space demanded during the lifetime of the application, i.e., 3900. However, the number of accesses to the memory will be determined by each sample and no worst case assumption is made for number of accesses.

The second approach assumes that L1' has four banks of the same size (975) that can be turned on and off. The third one assumes an L1' with

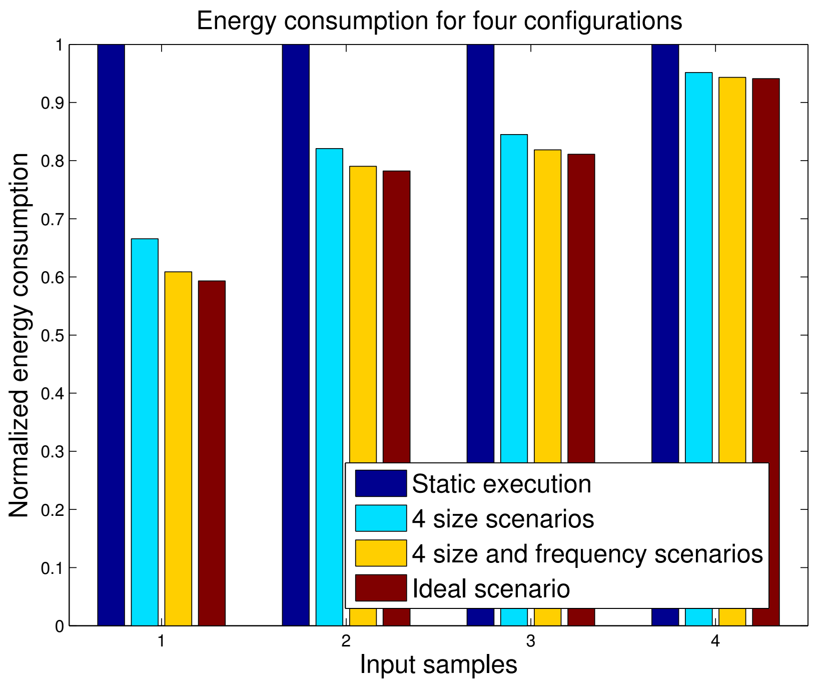


Figure A.5: Memory-aware scenario gains - Epileptic seizure predictor

four banks with different sizes (195, 585, 1170, and 1950), in order to better exploit RTSs with small memory footprints. Finally, a theoretical lower bound assumes an unlimited number of memory banks in all sizes to optimally exploit every situation. Comparison results are shown in Fig. A.5 and memory energy gains up to 40% are achieved with the scenario methodology for dynamic input samples. The high quality of our results is substantiated by the fact that our energy consumption is very close to the theoretical lower bound, even though the latter is impossible due to limited area in embedded devices.

Even higher gains are found for the Viterbi encoder (Fig. A.6) compared to a worst-case assumption of the constraint length being equal to 14. That value of length is used to achieve acceptable BER over very noisy channels. Using the extended system scenario methodology, L1' is split into four smaller banks that can be turned off when the noise of the channel is reduced. Again, the theoretical lower bound assumes a memory organisation with bank sizes optimized for each SNR level. Its hardware implementation would need 24 memory banks instead of 4 in our case, which leads to an

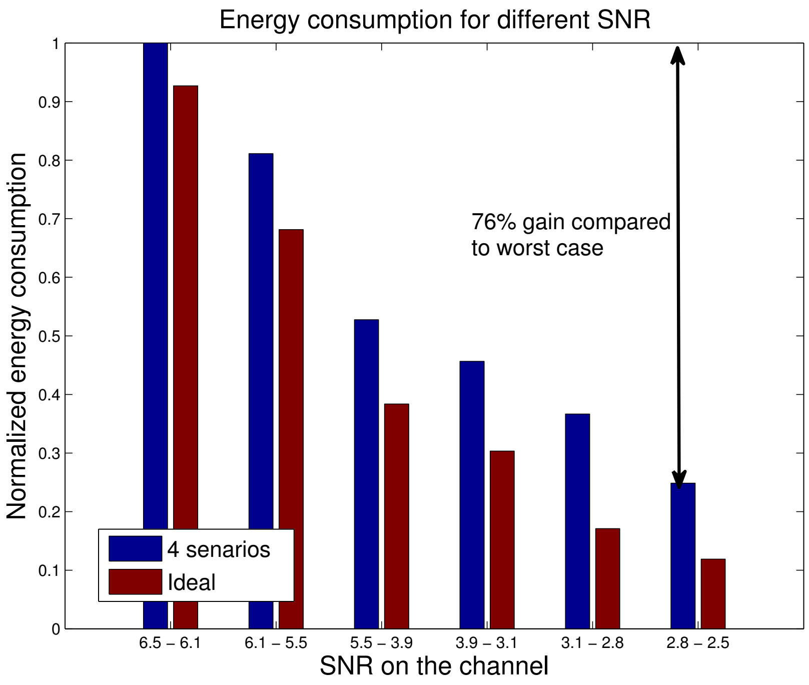


Figure A.6: Memory-aware scenario gains - Viterbi encoder

unacceptable overhead. The proposed reconfigurable memory organisation can lead to more than 70% reduction in memory energy consumption in situations with low noise, compared to a static architecture that is tuned to handle SNR levels down to 2.5 dB.

A.7 Conclusion

The scope of this work is to extend the existing system scenario techniques to better take memory into account. The memory-aware system scenario methodology has been described and tested on two real applications from bioengineering and wireless communications domains. Results justify the effectiveness of the methodology in reduction of memory energy consumption, which is of great importance in embedded devices. Since memory size requirements are still met in all situations, performance is not reduced. The memory-aware system scenario methodology is suited for applications that experience dynamic behaviour with respect to memory organisation utilization during their execution.

Bibliography

[1] A. Artes et al. Run-time self-tuning banked loop buffer architecture for power optimization of dynamic workload applications. In *VLSI and System-on-Chip (VLSI-SoC), 2011 IEEE/IFIP 19th International Conference on*, pages 136 –141, oct. 2011.

[2] L. Benini et al. Increasing energy efficiency of embedded systems by application-specific memory hierarchy generation. *Design Test of Computers, IEEE*, 17(2):74 –85, apr-jun 2000.

[3] L. Benini, A. Macii, and M. Poncino. A recursive algorithm for low-power memory partitioning. In *Low Power Electronics and Design, 2000. ISLPED '00. Proceedings of the 2000 International Symposium on*, pages 78 – 83, 2000.

[4] Francky Catthoor, Sven Wuytack, G.E. de Greef, Florin Banica, Lode Nachtergaele, and Arnout Vandecappelle. *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*. Springer, 1998.

[5] E. Cheung, H. Hsieh, and F. Balarin. Memory subsystem simulation in software tlm/t models. In *Design Automation Conference, 2009. ASP-DAC 2009. Asia and South Pacific*, pages 811 –816, jan. 2009.

[6] Philip Garcia, Katherine Compton, Michael Schulte, Emily Blem, and Wenyin Fu. An overview of reconfigurable hardware in embedded systems. *EURASIP J. Embedded Syst.*, 2006(1):13–13, January 2006.

[7] Stefan Valentin Gheorghita, , et al. System-scenario-based design of dynamic embedded systems. *ACM Trans. Des. Autom. Electron. Syst.*, 14(1):3:1–3:45, January 2009.

[8] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *Solid-State Circuits, IEEE Journal of*, 31(9):1277 – 1284, sep 1996.

[9] E. Hammari, F. Catthoor, J. Huisken, and P G Kjeldsberg. Application of medium-grain multiprocessor mapping methodology to epileptic seizure predictor. In *NORCHIP, 2010*, pages 1 –6, nov. 2010.

[10] Stefaan Himpe, Geert Deconinck, Francky Catthoor, and J van Meerbergen. Mtg* and grey-box: modelling dynamic multimedia applications with concurrency and non-determinism. In *System Specification and Design Languages: Best of FDL02*, 2002.

[11] L.D. Iasemidis et al. Long-term prospective on-line real-time seizure prediction. *Clinical Neurophysiology*, 116(3):532–544, 2005.

[12] Zhe Ma et al. *Systematic Methodology for Real-Time Cost-Effective Mapping of Dynamic Concurrent Task-Based Systems on Heterogenous Platforms.* Springer Publishing Company, Incorporated, 1st edition, 2007.

[13] A. Macii, L. Benini, and M. Poncino. *Memory Design Techniques for Low-Energy Embedded Systems.* Kluwer Academic Publishers, 2002.

[14] Narasinga Rao Miniskar. *System Scenario Based Resource Management of Processing Elements on MPSoC.* PhD thesis, Katholieke Universiteit Leuven, 2012.

[15] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *Proceedings of the Third International ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, 2007.

[16] M. Palkovic, H. Corporaal, and F. Catthoor. Heuristics for scenario creation to enable general loop transformations. In *System-on-Chip, 2007 International Symposium on*, pages 1 –4, nov. 2007.

[17] Martin Palkovic, Francky Catthoor, and Henk Corporaal. Dealing with variable trip count loops in system level exploration. In *ODES: 4th Workshop on Optimizations for DSP and Embedded Systems*, 2006.

[18] Martin Palkovic et al. Systematic preprocessing of data dependent constructs for embedded systems. *Journal of Low Power Electronics, Volume 2, Number 1*, 2006.

[19] P. R. Panda et al. Data and memory optimization techniques for embedded systems. *ACM Trans. Des. Autom. Electron. Syst.*, 6(2):149–206, April 2001.

[20] D.A. Patterson and J.L. Hennessy. *Exploiting Memory Hierarchy in Computer Organization and Design the HW/SW Intelface.* Morgan Kaufmann, 1994.

[21] A. Portero et al. Dynamic voltage scaling for power efficient mpeg4-sp implementation. In *Application-specific Systems, Architectures and Processors, 2006. ASAP '06. International Conference on*, pages 257 –260, sept. 2006.

[22] T. Simunic, L. Benini, and G. De Micheli. Cycle-accurate simulation of energy consumption in embedded systems. In *Design Automation Conference, 1999. Proceedings. 36th*, pages 867 –872, 1999.

[23] Sriram Swaminathan et al. A dynamically reconfigurable adaptive viterbi decoder. In *Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, FPGA '02, pages 227–236, New York, NY, USA, 2002. ACM.

[24] A. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *Information Theory, IEEE Transactions on*, 13(2):260 –269, april 1967.

Appendix B

Exploration of energy efficient memory organisations for dynamic multimedia applications using system scenarios

Iason Filippopoulos, Francky Catthoor and Per Gunnar Kjeldsberg
Memory Architecture and Organization Workshop
Embedded Systems Week
2013

Abstract

We propose a memory-aware system scenario approach that exploits variations in memory needs during the lifetime of an application in order to optimize energy usage. Different system scenarios capture the application's different resource requirements which change dynamically at run-time. In addition to computational resources, the many possible memory platform configurations and data-to-memory assignments are important system scenario parameters. Here we present an extended memory model that includes existing state-of-the-art memories, available in the industry and academia, and show how it is employed during the system design exploration phase. Both commercial SRAM and standard cell based memory models are explored in this study. The effectiveness of the proposed methodology is demonstrated and tested using a large set of multimedia benchmarks published in the Polybench, Mibench and Mediabench suites. Reduction in energy consumption in the memory subsystem ranges from 35% to 55 % for the chosen set of benchmarks.

B.1 Introduction

Modern embedded systems are becoming more and more powerful as the semiconductor processing techniques keep increasing the number of transistors on a single chip. Consequentially, demanding applications, e.g., in the signal processing and multimedia domains, can be executed on these devices [17]. On the other hand, the desired performance has to be delivered with minimum power consumption due to the limited energy available in mobile devices [12]. System scenario methodologies propose the use of different platform configurations in order to exploit run-time variations in computational and memory needs often seen in such applications [12].

Platform reconfiguration is performed through tuning of different system parameters, also called system knobs. For the memory-aware system scenario methodology, a platform can be reconfigured through a number of potential knobs, each resulting in different performance and power consumption in the memory subsystem. Foremost, modern memories support different energy states, e.g., through power gating techniques and by switching to lower power modes when not accessed. The second platform knob is the assignment of data to the available memory banks. The data assignment decisions affect both the energy per access for the mapped data, the data conflicts as a result of suboptimal assignment, and the number of active banks. In this work a reconfigurable memory platform is constructed using detailed memory models. This is followed by experiments with dynamic multimedia applications in order to study the effectiveness of the methodology.

The main contribution of the current work is the development of data variable based system scenarios. Previous control variable based system scenarios are unable to handle the fine-grain behaviour of the studied multimedia applications due to their significant variation under different execution situations. Furthermore, compared with use case scenario approaches in which scenarios are generated based on a user's behaviour, the system scenario methodology focuses on the behaviour of the system to generate scenarios and can, therefore, fully exploit the detailed platform mapping information. Rather than focusing on the processing cores, this work analyses the application of system scenarios on the memory organisation. Other contributions are for the purpose sufficiently detailed and accurate memory models used for the system design exploration, an extensive number of benchmark applications on which the methodology is applied, and a categorisation of applications based on their dynamic characteristics. For the multimedia domain, the current work presents a comprehensive methodology for optimising energy consumption in the memory subsystem.

B.2 Motivation and Related Work

A large number of papers have demonstrated the importance of the memory organization to the overall system energy consumption. Especially for embedded systems, the memory subsystem accounts for up to 50% of the overall energy consumption [4] and the cycle-accurate simulator presented in [20] estimates that the energy expenditures in the memory subsystem range from 35% up to 65% for different architectures. According to [12], conventional allocation and assignment of data done by regular compilers is suboptimal. Performance loss is caused by stalls for fetching data and data conflicts for different tasks, due to the limited size of memory and the competition between tasks.The significant contribution that the memory subsystem has to the overall energy consumption of a system and the dynamic nature of many applications offer a strong motivation for the study and optimization of the memory organisation in modern embedded devices.

Many papers have focused on memory related optimisations, also in the presence of a partitioned and distributed memory organisation with memory blocks of different sizes. In [2] authors present a methodology for automatic memory hierarchy generation that exploits memory access locality, while in [1] they propose an algorithm for the automatic partitioning of on-chip SRAM in multiple banks. Several design techniques for designing energy efficient memory architectures for embedded systems are presented in [13]. The current work differentiates by employing a platform that is reconfigurable during run-time. In [18] a large number of data and memory optimisation techniques, that could be dependent or independent of a target platform, are discussed. Again, reconfigurable platforms are not considered.

Energy-aware assignment of data to memory banks for several task-sets based on the MediaBench suit of benchmarks is presented in [14]. Low energy multimedia applications are discussed also in [5] with focus on processing rather than the memory platform. Furthermore, both [14] and [5] base their analysis on use case situations and do not incorporate sufficient support for very dynamically behaving application codes. System scenarios alleviate this bottleneck and enable handling of such dynamic behaviour. In addition, the current work explores the assignment of data to the memory and the effect of different assignment decisions on the overall energy consumption.

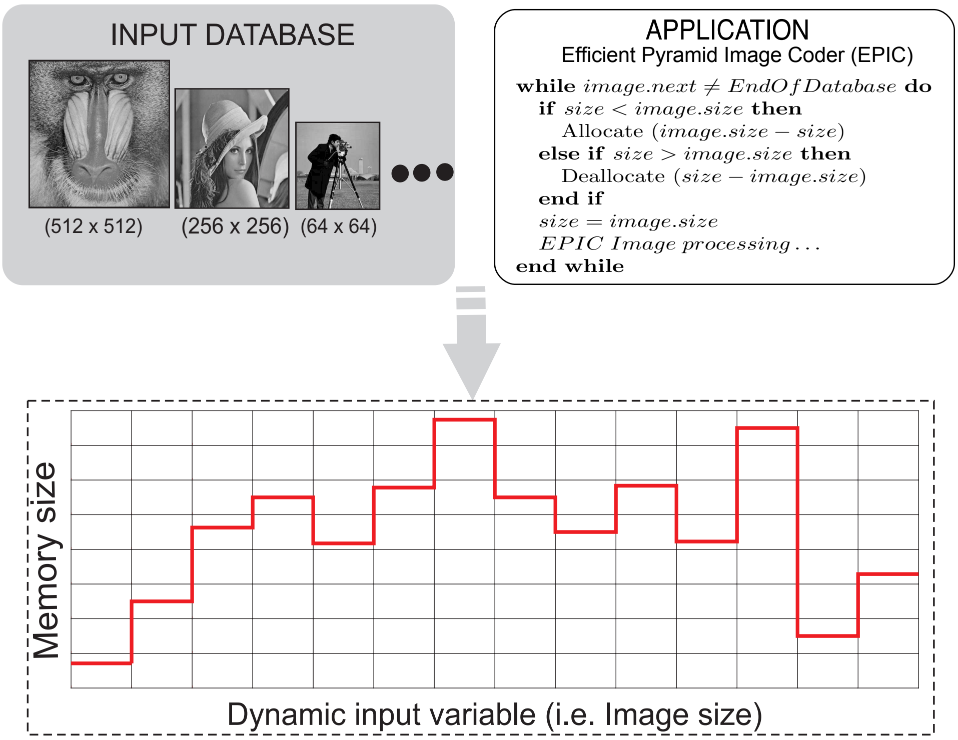
B.3 Data Variable Based Memory-Aware System Scenario Methodology

Designing with system scenarios is workload adaptive and offers different configurations of the platform and the freedom of switching to the most efficient scenario at run-time. A system scenario is a configuration of the system that combines similar run-time situations (RTSs). An RTS consists of a running instance of a task and its corresponding cost (e.g., energy consumption) and one complete run of the application on the target platform represents a sequence of RTSs [8]. The system is configured to meet the cost requirements of an RTS by choosing the appropriate system scenario, which is the one that satisfies the requirements using minimal power. In the following subsections, the different steps of the memory-aware system scenario methodology are outlined.

The general system scenario methodology follows a two stage exploration, namely design-time and run-time stages, as described in [6]. This splitting is also employed in the memory-aware extension of the methodology. The two stage exploration is chosen because it reduces run-time overhead while preserving an important degree of freedom for run-time configuration [12]. The application is analysed at design-time and different execution paths causing variations in memory demands are identified. This procedure, which is time consuming and as a result can be performed only during the design phase, will result in a grey-box model representation of the application. The grey-box model hides all static and deterministic parts of the application, by providing only related memory costs for those, and keeps parts of the application code that are non-deterministic in terms of memory usage available to the system designer [9].

B.3.1 Design-time Profiling Based on Data Variables

Application profiling is performed at design-time for a wide range of inputs. The analysis focuses on the allocated memory size during execution and on access pattern variations. Techniques described in [10] are, e.g., used in order to extract the access scheme through analysis of array iteration


INPUT DATABASE
(512 x 512)
(256 x 256)
(64 x 64)
APPLICATION
Efficient Pyramid Image Coder (EPIC)
Memory size
Dynamic input variable (i.e. Image size)

```
while $image.next \neq EndOfDatabase$ do
  if $size < image.size$ then
    Allocate $(image.size - size)$
  else if $size > image.size$ then
    Deallocate $(size - image.size)$
  end if
  $size = image.size$
  $EPIC\ Image\ processing\ldots$
end while
```

Figure B.1: Profiling results based on application code and input data

spaces.

The profiling stage is depicted in Fig. B.1 and consists of running the application code with suitable input data often found in a database, in order to produce profiling results. The results shown here are limited for demonstrational purposes. A real application would have thousands or millions of profiling samples. The profiling reveals parts of the application code with high memory activity and with varying memory access intensity, which possibly depends on input data variables. Because of this behaviour, a static study of the application code alone is insufficient since the target applications for this methodology have non-deterministic behaviour that is driven by input.

In Fig. B.1 the profiled applications are two image related multimedia benchmarks and the input database should consist of a variety of images. The memory requirements in each case are driven by the current input image size, which is classified as a data variable due to the wide range of its possible values. Depending on the application the whole image or a region of

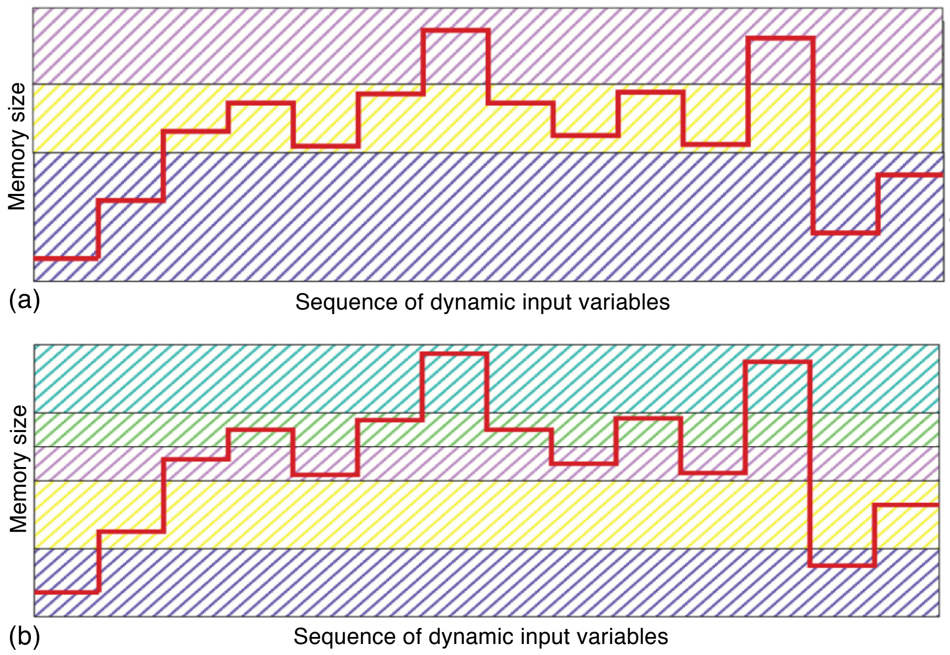


Figure B.2: Clustering of profiling results into three (a) or five (b) system scenarios

interest is processed. Other applications have other input variables deciding the memory requirement dynamism, e.g., the SNR level on the channel in the case of an encoding/decoding application.

B.3.2 Design-time System Scenario Identification and Prediction Based on Data Variables

The next step is the clustering of the profiled memory sizes into groups with similar characteristics. This is referred to as system scenario identification. Clustering is necessary, because it will be extremely costly to have a different scenario for every possible size, due to the number of memories needed. Clustering neighbouring RTSs is a rational choice, because two instances with similar memory needs have similar energy consumption. In Fig. B.2 the clustering of the previously profiled information is presented. The clustering of RTSs is based both on their distance on the memory size axis and the frequency of their occurrence. Consequently, the memory size is split unevenly with more frequent RTSs having a shorter memory size range. This is better than even splitting because the energy cost of each system scenario is defined by the upper size limit, as each scenario should support all RTSs within its range. With more scenarios, e.g., five instead of 3, the aggregated RTS running overhead is reduced. Still the number of scenarios should be limited due to overhead of a complex memory platform and of frequent switching between scenarios.

The design-time system scenario prediction phase consists of determination of the data variables that define the active system scenario. This can be achieved by careful study of the application code, combined with the application's data input. In our case the grey-box model reveals only the code parts that will influence memory usage, so that data variables deciding memory space changes can be identified. An example of this is a non static variable that influences the number of iterations for a loop that performs one memory allocation at each iteration. In the depicted example the system scenario prediction data variable is the input image height and width values. Moreover, the designer should look for a correlation between input values and the corresponding cost. This information will be useful in the following steps of the methodology [12].

B.3.3 Run-time System Scenario Detection and Switching Based on Data Variables

Switching decisions are taken at run-time by the run-time manager. The switching phase consists of all platform configuration decisions that can be made at run-time, e.g., frequency/voltage scaling, changing the power mode of memory units, including turning them off, and reassignment of data on memory units. Switching takes place when the switching cost is lower than the energy gains achieved by switching. In more detail, the run-time manager compares the memory energy consumption of executing the next task in the current active system scenario with the energy consumption of execution with the optimal system scenario. If the difference is greater than the switching cost, then scenario switching is performed [12]. Switching costs are defined by the platform and include all memory energy penalties for run-time reconfigurations of the platform, e.g., extra energy needed to change state of a memory unit.

In Fig. B.3 an example of the run-time phase of the methodology is depicted. The run-time manager identifies the size of the image that will

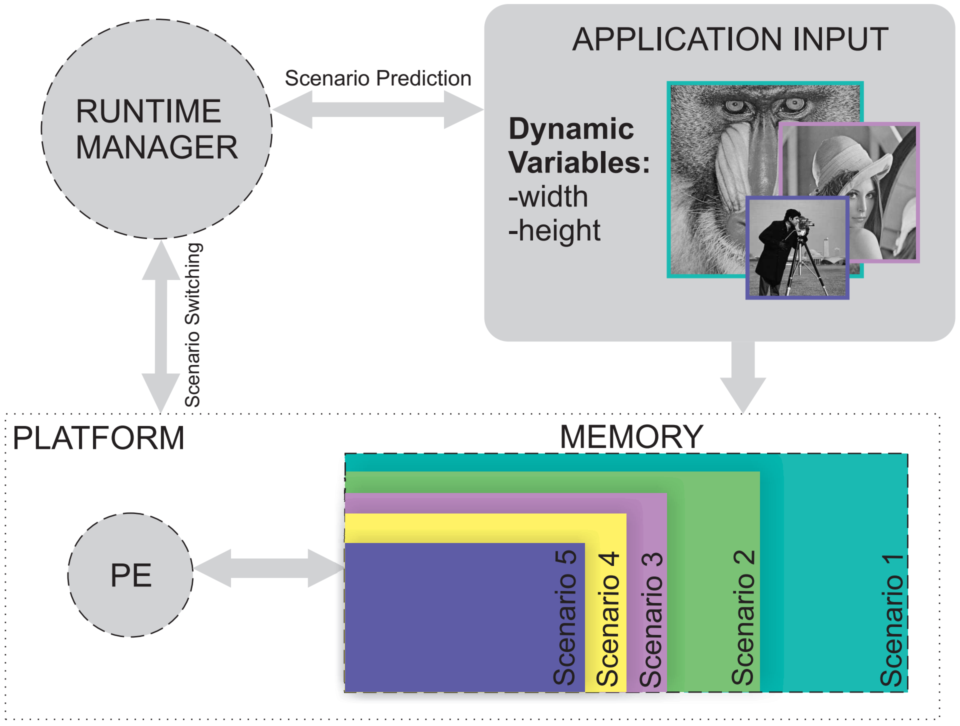


Figure B.3: Run-time system scenario prediction and switching based on the current input

be processed and reconfigures the memory subsystem on the platform, if needed, by increasing or decreasing the available memory size. The reconfiguration options are effected by platform hardware limitations. The image size is the data variable monitored in order to detect the system scenario and the need for switching.

B.4 Target Platform and energy models

Selection of target platform is an important aspect of the memory-aware system scenario methodology. The key feature needed in the platform architecture is the ability to efficiently support different memory sizes that correspond to the system scenarios generated by the methodology. The dynamic memory platform is achieved by organising the memory area in a varying number of banks that can be switched between different energy

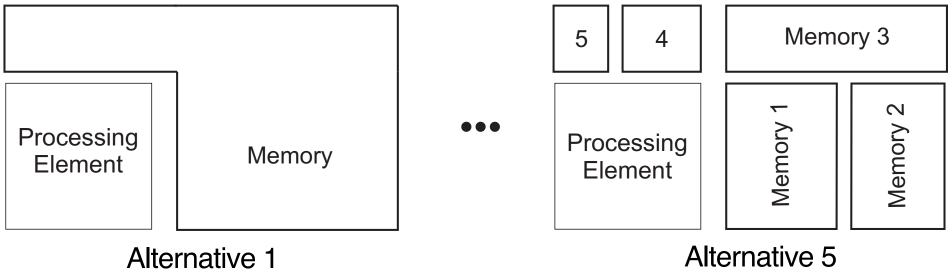


Figure B.4: Alternative memory platforms with varying number of banks

states. In this work, a clustered memory organisation with up to five memory banks of varying sizes is explored. Some examples of alternative memory platforms that can be used for exploration is shown in Fig. B.4.

B.4.1 Models of Different Memory Types

The dynamic memory organisation is constructed using commercially available SRAM memory models (MM). In addition, experimental standard cell-based memories (SCMEM) [16] are considered for smaller memories due to their energy and area efficiency for reasonably small storage capacities, as argued in [15]. Both MMs and SCMEMs can operate under a wide range of supply voltages, thus support different operating modes that provide an important exploration space. In the active mode the memory can be accessed at the maximum supported speed and the supply voltage is set at 1.1V. While data are not accessed for a period of time the light/deep sleep or shut down mode should be considered. In light sleep mode the supply voltage is lowered with values around 0.7V, while on deep sleep mode the supply voltage is set to the lowest possible value that can be used without loss of data. This voltage threshold is expected to be lower for SCMEMs than MM models and can be as low as 0.3V. The shut down mode uses power-gating techniques to achieve near zero leakage power, but stored data is lost. The time and the energy required for switching from these low leakage modes to the active state differs and all the necessary energy/power information is available to the system designer.

B.4.2 Energy consumption calculation

The overall energy consumption for each configuration is calculated using a detailed formula, as can be seen in (B.1).

$$\begin{aligned}
E \quad = & \sum_{memories}^{all} (N_{rd} \times E_{Read} + N_{wr} \times E_{Write} \\
+ & \ (T - T_{LSleep} - T_{DSleep} - T_{ShutDown}) \times P_{leak_{Active}} \\
+ & \ T_{LSleep} \times P_{leak_{LSleep}} + T_{DSleep} \times P_{leak_{DSleep}} \\
+ & \ T_{ShutDown} \times P_{leak_{ShutDown}} \\
+ & \ N_{SW\,Light} \times E_{LSleep\,to\,Active} \\
+ & \ N_{SW\,Deep} \times E_{DSleep\,to\,Active} \\
+ & \ N_{SW\,ShutDown} \times E_{ShutDown\,to\,Active})
\end{aligned} \tag{B.1}$$

All the important transactions on the platform that contribute to the overall energy are included, in order to achieve as accurate results as possible. In particular:

- N_{rd} is the number of read accesses
- E_{Read} is the energy per read
- N_{wr} is the number of write accesses
- E_{Write} is the energy per write
- T is the execution time of the application
- T_{LSleep}, T_{DSleep} and $T_{ShutDown}$ are the times spent in light sleep, deep sleep and shut down states respectively
- $P_{leak_{Active}}$ is the leakage power in active mode
- $P_{leak_{LSleep}}$, $P_{leak_{DSleep}}$ and $P_{leak_{Shutdown}}$ are the leakage power values in light sleep, deep sleep and shut down modes with different values corresponding to each mode
- $N_{SW\,Light}$, $N_{SW\,Deep}$ and $N_{SW\,ShutDown}$ are the number of transitions from each retention state to active state
- $E_{LSleep\,to\,Active}$, $E_{DSleep\,to\,Active}$ and $E_{ShutDown\,to\,Active}$ are the energy penalties for each transition respectively.

Algorithm 1 Memory organisation exploration steps

```
1: $RTSset \leftarrow$ storage requirement for each RTS
2: $Database \leftarrow$ memory database
3: $N \leftarrow$ number of scenarios (up to 5 in this work)
4: for $i = 1 \rightarrow N$ do
5:     for all combinations of i banks in database do
6:         if $\sum_1^i size(bank) \geq size(max(RTS))$ then
7:             Keep configuration
8:         end if
9:         Select configuration that minimizes Eq.B.1 for $RTSset$
10:    end for
11: end for
```

The overall energy consumption is given after calculating the energy for each memory bank. The execution time of the application is needed to calculate the leak time. It can be found by executing the application on a reference embedded processor. The simulator described in [3] is chosen to calculate execution time for the chosen applications in this work. The processor is assumed to be running continuously, accepting new input data as soon as computations on the previous data set has been finished. Memory sleep times are hence only caused by data dependent dynamic behaviour.

B.4.3 Architecture Exploration

The exploration of alternative memory platforms is performed using the steps described in Alg. 1. All potentially energy efficient configurations are tested for a given number of scenarios and the sequence of RTSs of the application. First, all possible configurations for a given number of memory banks are constructed. The only requirement in order to keep a configuration for further investigation is that the combined size of all banks should satisfy the storage requirements of the most demanding RTS. Then, each configuration is tested for the sequence of RTSs and the one that minimizes Eq.B.1 is chosen as the most energy efficient for this number of scenarios (i.e., number of banks).

B.5 Application Benchmarks

The applications that benefit most from the memory-aware system scenario methodology are characterised by having dynamic utilization of the memory organisation during their execution. Multimedia applications often exhibit such dynamicity and are consequentially suitable candidates for the presented methodology. The effectiveness is demonstrated and tested using a variety of open multimedia benchmarks, which can be found in the Polybench [19], Mibench [7] and Mediabench [11] benchmark suites.

An overview of the benchmark applications that were tested is presented in Tab. B.1. Two key parameters under consideration are the dynamic data variable of each application and the variation in the memory requirement it causes. The dynamic data variable is the variable that results in different system scenarios due to its range of values. Examples of such a variable are an input image of varying size or data dependent loop bound values. For each application an appropriate input database is constructed with realistic RTS cases. The memory size limits are defined as the minimum and maximum storage requirement occurred during testing of an application. The dynamic characteristics that are used to categorize the applications are the dynamism in the memory size bounds and the variance of cases within the memory size limits.

The memory size bounds correspond to the minimum and maximum memory size values profiled over all possible cases. In general, larger distances between upper and lower bounds increase the possibilities for energy gains. This is a result of using larger and more energy hungry memories in order to support the memory requirements for the worst case even when only small memories are required. Large energy gain is expected when large parts of the memory subsystem can be switched into retention for a long time.

Table B.1: Benchmark applications overview

Application	**Source**	**Data variables used for scenario prediction**	**Dynamic Characteristics**		
			Memory Variation(B)	Number of cases	Distribution of cases
Epic image compression	MediaBench	Image size	4257 - 34609	Average	good
Motion Estimation	MediaBench	Image size	4800 - 52800	High	average
Blowfish decoder	MiBench	Input file size	256 - 5120	Low	poor
Jacobi 1D Decomposition	Polybench	Number of steps	502 - 32002	Low	good
Mesa 3D	MediaBench	Loop bound	5 - 50000	High	average
JPEG DCT	MediaBench	Block size	10239 - 61439	High	average
PGP encryption	MediaBench	Encryption length	3073 - 49153	High	average
Viterbi encoder	Open	Constraint length	5121 - 14337	Low	good

Another metric used for identification of different kinds of dynamism is the memory requirement variation. The variation takes into consideration both the number of different cases that are present within the memory requirement limits and the distribution of those cases between minimum and maximum memory size. Applications with a limited number of different cases are expected to have most of its possible gain obtained with a few platform supported system scenarios and much smaller energy gains from additional system scenarios. After this point most of the cases are already fitting one of the platform configurations and adding new configurations have a minimal impact. The opposite is seen for applications that feature a wide range of well distributed cases.

B.6 Results

The memory aware system scenario methodology is applied to all the presented benchmark applications to study its effectiveness. The profiling phase is based on different input for the data variables shown in Tab. B.1 and is followed by the clustering phase. The execution and sleep times needed in Eq.B.1 are found through the profiling but are also reflected by the dynamic characteristics in Tab. B.1. Data variables are the variables used by the run-time manager in order to predict the next active scenario. The clustering is performed with one to five system scenarios. All potentially energy efficient configurations are tested for a given number of scenarios using the steps described in Alg. 1. For example, in the case of 2 scenarios all possible memory platforms with 2 memory banks that fulfil the memory size requirement of the worst case are generated and tested. The same procedure is performed for 3, 4 and 5 scenarios. The exploration includes memories of different sizes, technologies and varying word lengths. The energy gain percentages are presented in Fig. B.5. Energy gains are compared to the case of a fixed non-re-configurable platform, i.e., a static platform configuration with only 1 scenario. This corresponds to zero percentage gain in Fig. B.5.

The introduction of a second system scenario results in energy gains between 15% and 40% for the tested applications. Depending on the application's dynamism the maximum reported energy gains range from around 35% to 55%. As expected according to the categorisation presented in subsection B.5, higher energy gains are achieved for applications with more dynamic memory requirements, i.e., bigger difference between the minimum and maximum allocated size. The maximum gains for JPEG, motion

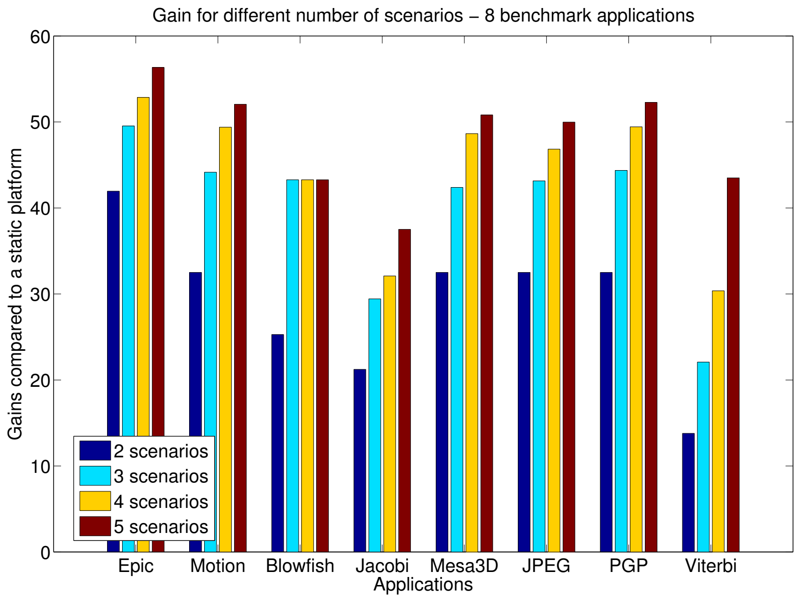


Figure B.5: Energy gain for increasing number of system scenarios - Static platform corresponds to 0%

estimator, mesa 3D and PGP are around 50% while blowfish, jacobi, and Viterbi decoders are around 40%.

As the number of system scenarios that are implemented on the memory subsystem increases, the energy gains improve since variations in memory requirements can be better exploited with more configurations. The switching cost also increases for an increasing number of system scenarios due to the increasing frequency of platform reconfiguration. This overhead reduces the achieved gain, but for up to 5 scenarios we still see improvements for all but one of our benchmarks. The switching cost is below 2% even for a platform with 5 memory banks in all cases. The most efficient of the tested organisations for each benchmark are presented in Fig. B.6, where each memory bank is depicted with a different colour and each length is proportional to the memory bank size. The blowfish decoder is the only benchmark that has only 3 banks in its most efficient memory organisation.

Comparative results from applying a use case scenario approach as a reference are presented in Fig. B.7. Reported energy gains for both use case scenarios and system scenarios are given assuming a static platform as a base (0%). Use case scenarios are generated based on a higher abstraction

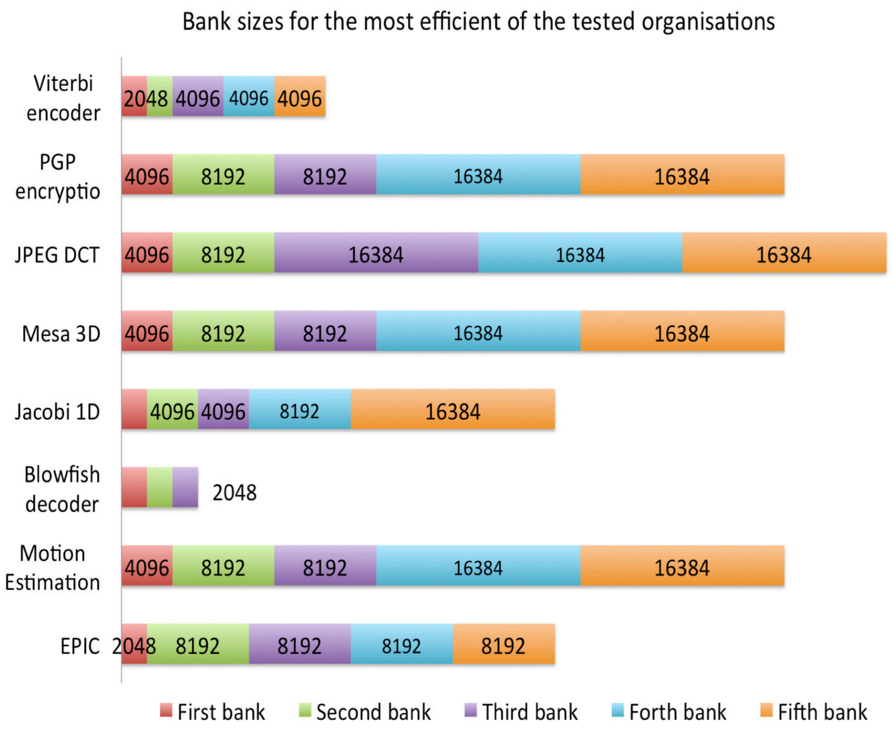


Figure B.6: Bank sizes for the most efficient of the tested organisations for each benchmark

level that is visible as a user's behaviour. For example, use case scenarios for image processing applications generate three scenarios, if large, medium and small are the image sizes identified by the user. In general, use case scenario identification can be seen as more coarse compared to identification on the detailed system implementation level.

B.7 Conclusion

The scope of this work is to apply the memory-aware system scenario methodology to a wide range of multimedia application and test its effectiveness based on an extensive memory energy model. The results demonstrate the effectiveness of the methodology reducing the memory energy consump-

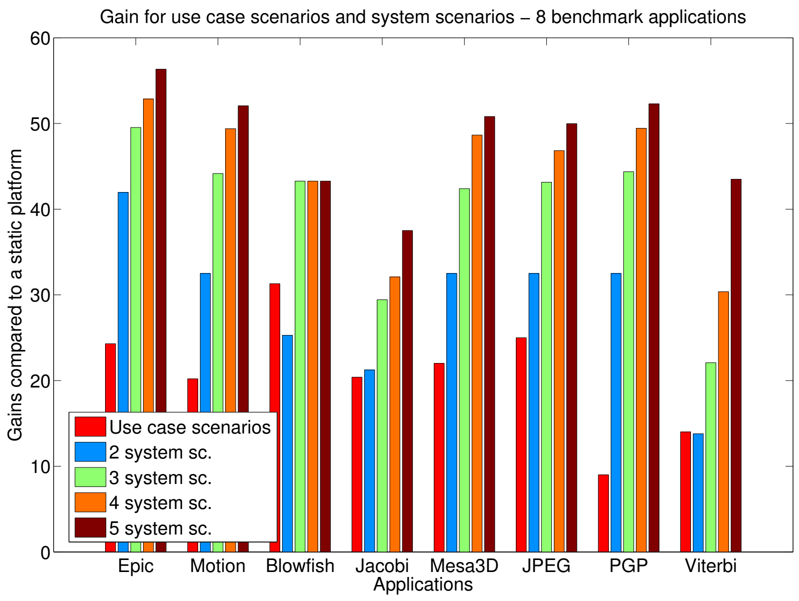


Figure B.7: Energy gain for use case scenarios and system scenarios

tion between 35% and 55%.

Bibliography

[1] L. Benini et al. Increasing energy efficiency of embedded systems by application-specific memory hierarchy generation. *Design Test of Computers, IEEE*, 17(2):74 –85, apr-jun 2000.

[2] L. Benini, A. Macii, and M. Poncino. A recursive algorithm for low-power memory partitioning. In *Proceedings of the 2000 intr symp on Low Power Electronics and Design*, pages 78 – 83, 2000.

[3] Nathan Binkert et al. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[4] E. Cheung et al. Memory subsystem simulation in software tlm/t models. In *Proceedings of Asia and South Pacific Design Automation Conference, 2009.*, pages 811 –816, jan. 2009.

[5] Eui-Young Chung et al. Contents provider-assisted dynamic voltage scaling for low energy multimedia applications. In *Proc. of the 2002 intr symp on Low Power Electronics and Design*, pages 42–47, 2002.

[6] Stefan Valentin Gheorghita et al. System-scenario-based design of dynamic embedded systems. *ACM Trans. Des. Autom. Electron. Syst.*, 14(1):3:1–3:45, January 2009.

[7] M.R. Guthaus et al. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE Int. Workshop on*, pages 3–14. IEEE, 2001.

[8] E. Hammari et al. Application of medium-grain multiprocessor mapping methodology to epileptic seizure predictor. In *NORCHIP*, 2010.

[9] Stefaan Himpe et al. MTG* and grey-box: modeling dynamic multimedia applications with concurrency and non-determinism. In *System Specification and Design Languages: Best of FDL02*, 2002.

[10] A. Kritikakou et al. Near-optimal and scalable intra-signal in-place for non-overlapping and irregular access scheme. *ACM Trans. Design Automation of Electronic Systems (TODAES)*, accepted, 2013.

[11] C. Lee et al. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335. IEEE Computer Society, 1997.

[12] Zhe Ma et al. *Systematic Methodology for Real-Time Cost-Effective Mapping of Dynamic Concurrent Task-Based Systems on Heterogenous Platforms.* Springer Publishing Company, Incorporated, 1st edition, 2007.

[13] A. Macii, L. Benini, and M. Poncino. *Memory Design Techniques for Low-Energy Embedded Systems.* Kluwer Academic Publishers, 2002.

[14] P. Marchal et al. SDRAM energy-aware memory allocation for dynamic multi-media applications on multi-processor platforms. In *Design, Automation and Test in Europe*, pages 516–521, 2003.

[15] P Meinerzhagen, C Roth, and A Burg. Towards generic low-power area-efficient standard cell based memory architectures. In *Circuits and Systems (MWSCAS), 53rd IEEE intr symp on.* IEEE, 2010.

[16] Pascal Meinerzhagen et al. Benchmarking of standard-cell based memories in the sub-vt domain in 65-nm cmos technology. *IEEE Transactions on Emerging and Selected Topics in Circuits and Systems*, 2011.

[17] Narasinga Rao Miniskar. *System Scenario Based Resource Management of Processing Elements on MPSoC.* PhD thesis, KU Leuven, 2012.

[18] P. R. Panda et al. Data and memory optimization techniques for embedded systems. *ACM Trans. Des. Autom. Electron. Syst.*, 6(2):149–206, April 2001.

[19] L.N. Pouchet. Polybench: The polyhedral benchmark suite.

[20] T. Simunic et al. Cycle-accurate simulation of energy consumption in embedded systems. In *Proceedings of the 36th Design Automation Conference, 1999.*, pages 867 –872, 1999.

Appendix C

Exploration of energy efficient memory organisations for dynamic multimedia applications using system scenarios

Iason Filippopoulos, Francky Catthoor and Per Gunnar Kjeldsberg
Design Automation for Embedded Systems
Springer
2014

Abstract

We propose a memory-aware system scenario approach that exploits variations in memory needs during the lifetime of an application in order to optimize energy usage. Different system scenarios capture the application's different resource requirements that change dynamically at run-time. In addition to computational resources, the many possible memory platform configurations and data-to-memory assignments are important system scenario parameters. In this work we focus on clustering of different memory requirements into groups and presenting the system scenario generation in detail. The clustering is a non-trivial problem due to the many different memory requirements, which leads to a very large exploration space. An extended memory model is used as a practical enabler, in order to evaluate the methodology. The memory models include existing state-of-the-art memories, available from industry and academia, and we show how they are employed during the system design exploration phase. Both commercial SRAM and standard cell based memory models are explored in this study. The effectiveness of the proposed methodology is demonstrated and tested using a large set of multimedia benchmarks published in the Polybench, Mibench and Mediabench suites, representative for the domain of multimedia applications. Reduction in energy consumption in the memory subsystem ranges from 35% to 55% for the chosen set of benchmarks.

C.1 Introduction

Modern embedded systems are becoming more and more powerful as the semiconductor processing techniques keep increasing the number of transistors on a single chip. Consequently, demanding applications, e.g., in the signal processing and multimedia domains, can be executed on these devices [31]. On the other hand, the desired performance has to be delivered with minimum power consumption due to the limited energy available in mobile devices [26]. System scenario methodologies propose the use of different platform configurations in order to exploit run-time variations in computational and memory needs often seen in such applications [26].

Platform reconfiguration is performed through tuning of different system parameters, also called system knobs. For the memory-aware system scenario methodology, a platform can be reconfigured through a number of potential knobs, each resulting in different performance and power consumption in the memory subsystem. Foremost, modern memories support different energy states, e.g., through power gating techniques and by switching to lower power modes when not accessed. The second platform knob is the assignment of data to the available memory banks. The data assignment decisions affect both the energy per access for the mapped data, the data conflicts as a result of suboptimal assignment, and the number of active banks. In this work a reconfigurable memory platform is constructed using detailed memory models. This is followed by experiments with dynamic multimedia applications in order to study the effectiveness of the methodology.

The main contribution of the current work is the development of data variable [16] based system scenarios. Previous control variable based system scenarios [10] are unable to handle the fine-grain behaviour of the studied multimedia applications due to their significant variation under different execution situations. Furthermore, compared with use case scenario approaches in which scenarios are generated based on a user's behaviour [14], the system scenario methodology focuses on the behaviour of the system to generate scenarios and can, therefore, fully exploit the detailed platform mapping information. Compared with previous work on system scenarios that has focused on the processing cores, the current work analyses the use of system scenarios on the memory organisation. More specifically, this work focuses on the system scenario identification phase of the methodology. The wide range of memory requirements, the amount of different cases, and the different frequency in which each case occurs, results in a very large exploration space. Therefore, there is a need for developing an algorithmic approach that can efficiently tackle this problem.

Another significant contribution is the extensive number of benchmark applications on which the methodology is applied. The chosen set is representative for the domain of multimedia applications. Furthermore, we present a categorisation of applications based on their dynamic characteristics, also applicable to the entire multimedia domain. For the experimental needs of this work we present for the purpose sufficiently detailed and accurate memory models, which are used for the system design exploration. For the multimedia domain, the current work presents a comprehensive methodology for optimising energy consumption in the memory subsystem.

This article is organized as follows. Section C.2 motivates the study of optimization of the memory organisation. Section C.3 surveys related work on system level memory exploration and on system scenario methodologies and compares it with the current work. Section C.4 presents the chosen methodology with main focus on the memory organisation study. In Section C.5 the target platform is described accompanied by a detailed description of the employed memory models, while the multimedia benchmarks and their characteristics are analysed in Section C.6. Results of applying the described methodology to the targeted applications are shown in Section C.7, while conclusions are drawn in Section C.8.

C.2 Motivational Example

A large number of papers have demonstrated the importance of the memory organization to the overall system energy consumption. As shown in [11] memory contributes around 40% to the overall power consumption in general purpose systems. Especially for embedded systems, the memory subsystem accounts for up to 50% of the overall energy consumption [6] and the cycle-accurate simulator presented in [41] estimates that the energy expenditures in the memory subsystem range from 35% up to 65% for different architectures. The breakdown of power consumption for a recently implemented embedded system presented in [18] shows that the memory subsystem consumes more than 40% of the leakage power on the platform. According to [26], conventional allocation and assignment of data done by regular compilers is suboptimal. Performance loss is caused by stalls for fetching data and data conflicts for different tasks, due to the limited size of memory and the competition between tasks.

Algorithm 2 Motivational example of dynamic memory usage

```
1: while $image \neq EndOfDatabase$ do
2:   $height \leftarrow height(image)$
3:   $width \leftarrow width(image)$
4:   $store(image[height][width])$
5:   for $i = 0 \rightarrow height$ do
6:     for $j = 0 \rightarrow width$ do
7:       $array[i][j] \leftarrow func1(image[i][j])$
8:     end for
9:   end for
10:  $image \leftarrow new.image$
11: end while
```

In addition, modern applications exhibit more and more dynamic behaviour, which is reflected also in fluctuating memory requirements [26]. Techniques have been developed in order to estimate the memory size requirements of applications in a systematic way [23]. The significant contribution that the memory subsystem has to the overall energy consumption of a system and the dynamic nature of many applications offer a strong motivation for the study and optimization of the memory organisation in modern embedded devices.

To illustrate the sub-optimal conventional allocation and assignment of data, the simple example of Alg. 2 is used. The kernel code of an image processing application continuously reads a sequence of images, saves each image in memory and performs function *func1* on each pixel of the image. Arrays are typically used for storing the intermediate calculations in image processing applications, exemplified with the *array* variable in the motivation example. The memory size used for the storage of each initial *image* and the computed *array* are determined by the dimensions of the input image and can be different for a series of input images. In a conventional assignment the highest values of *height* and *width* are identified and a static compiling results in allocation of the worst-case area for the *array* variable. However, only a part of the allocated space is accessed during processing of smaller images.

Assume for instance that we have two different image sizes, ImgA with L=H=1 and ImgB with L=H=2. That is, the size of ImgB is 4 $\times$ ImgA. Each pixel in each image is accessed once giving rise to N accesses to each ImgA and 4 $\times$ N accesses to each ImgB. Furthermore, in the input stream of images there are four times as many ImgA as ImgB. In our pool of alternative memories we have three memories with Size1 = ImgA, Size2 = 3 × ImgA, and Size 3 = 4 × ImgA (i.e., the size of ImgB). The energy cost of accessing a Size1 memory is 1E, while the average leakage energy in the time between the start of two accesses is 0.3E. The corresponding access/leakage numbers for Size2 and Size3 memories are 1.3E/0.9E and 1.5E/1.2E, respectively. The numbers reflects the fact that as a first order approximation access energy increases sub linearly with increased memory size, while leakage increases linearly with memory size. The total energy (access + leakage) during computations on four ImgA and one ImgB using only one memory of Size3 is:

$$4 \times N \times 1.5E + 4 \times N \times 1.5E + 8 \times N \times 1.2E = 21.6NE$$

The same calculation using one memory of Size1 and one of Size2, in total the size of ImgB, is:

$$\begin{aligned}&4 \times N \times 1.0E + 1 \times N \times 1.0E + 3 \times N \times 1.3E \\ &+ 4 \times N \times 0.3E + 4 \times N \times (0.3E + 0.9E) = 14.9NE\end{aligned}$$

giving a reduction in energy consumption of 31%. These calculations are done with simplified assumptions regarding input data and memory models. The results in Section C.7 show even larger gain with realistic dynamic applications, memory models and data.

C.3 Related Work and Contribution Discussion

The memory allocation problem has been studied before. However, we extend state of the art by proposing a more generic approach, which is also suitable for applications with input driven dynamic behaviour. The authors in [3] present a methodology to generate a static application-specific memory hierarchy. Later, they extend their work in [2] to a reconfigurable platform with multiple memory banks. However, our work differentiates by proposing a more generic and application agnostic methodology and employing the use of system scenarios, in order to efficiently handle a wider range of dynamic application characteristics.

Several techniques for designing energy efficient memory architectures for embedded systems are presented in [27]. The current work differentiates by employing a platform that is reconfigurable at run-time. In [36] a large number of data and memory optimisation techniques, that could be dependent or independent of a target platform, are discussed. Again, reconfigurable platforms are not considered.

Energy-aware assignment of data to memory banks for several task-sets based on the MediaBench suit of benchmarks is presented in [28]. Low energy multimedia applications are discussed also in [7] with focus on processing rather than the memory platform. Furthermore, both [28] and [7] base their analysis on use case situations and do not incorporate sufficient support for very dynamically behaving application codes. System scenarios alleviate this bottleneck and enable handling of such dynamic behaviour. In addition, the current work explores the assignment of data to the memory and the effect of different assignment decisions on the overall energy consumption.

The authors in [1], [19] and [25] present methodologies for designing memory hierarchies. Design methods with main focus on the traffic and latencies in the memory architecture are presented in [5], [12], [20] and [37]. Improving memory energy efficiency based on a study of access patterns is discussed in [21]. Application specific memory design is a research topic in [40], while memory design for multimedia applications is presented in [32]. The current work differentiates by introducing the concept of system scenarios that supports the dynamic handling of application's requirements, although the data mapping is static inside each scenario.

An overview of work on system scenario methodologies and their application are presented in [10]. In [8] extensions towards a memory-aware system scenario methodology are presented and demonstrated using theoretical memory models and two target applications. This work is an extension both in complexity and accuracy of the considered memory library and on the number of target applications.

Furthermore, the majority of the published work focus on control variables for system scenario prediction and selection. Control variables can take a relatively small set of different values and thus can be fully explored. However, the use of data variables [15] is required by many dynamic systems including the majority of multimedia applications. The range of possible values for data variables is wider and makes full exploration impossible.

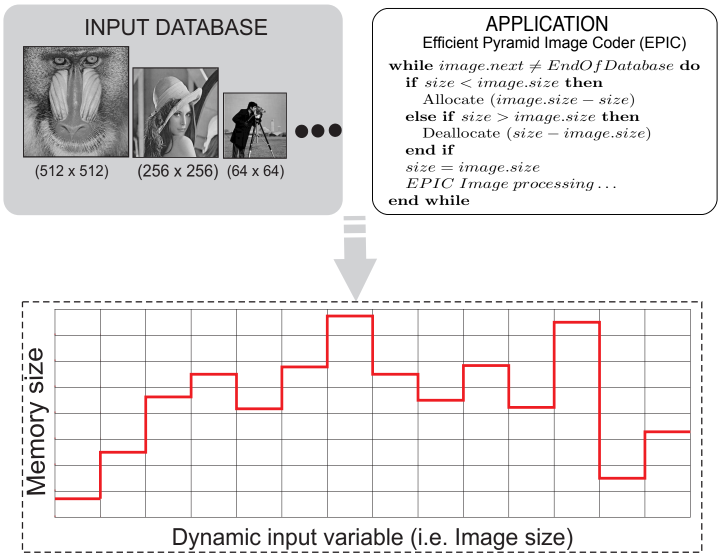
Authors in [34] present a technique to optimise memory accesses for input data dependent applications by duplicating and optimising the code for different execution paths of a control flow graph (CFG). One path or a group of paths in a CFG form a scenario and its memory accesses are optimised using global loop transformations (GLT). Apart from if-statement evaluations that define different execution paths, they extend their technique to include while loops with variable trip count in [35]. A heuristic to perform efficient grouping of execution paths for scenario creation is analysed in [33]. Our work extends the existing solutions towards exploiting the presence of a distributed memory organisation with reconfiguration possibilities.

Reconfigurable hardware for embedded systems, including the memory architecture, is a topic of active research. An extensive overview of current approaches is found in [9]. The approach presented in this paper differentiates by focusing on the data-to-memory assignment aspects in the presence of a platform with dynamically configurable memory blocks. Moreover, many methods for source code transformations, and especially loop transformations, have been proposed in the memory management context. These methods are fully complementary to our focus on data-to-memory assignment and should be performed prior to our step.

C.4 Data Variable Based Memory-Aware System Scenario Methodology

The memory-aware system scenario methodology is based on the observation that the memory subsystem requirements at run-time vary significantly due to dynamic variations of memory needs in the application code. Most existing design methodologies define the memory requirements as that of the most demanding task and tune the system in order to meet its needs [26]. Obviously, this approach leads to unused memory area for tasks with lower memory requirements, since those tasks could meet their needs using fewer resources and consequently consuming less energy. Another source of energy waste in the memory system is caused by data conflicts due to misplaced data. Replacement of old data and fetching of new data is both time and energy consuming and should therefore be avoided. Handling of data conflicts is also part of a memory-aware system scenario methodology.

Designing with system scenarios is workload adaptive and offers different configurations of the platform and the freedom of switching to the most efficient scenario at run-time. A system scenario is a configuration of the system that combines similar run-time situations (RTSs). An RTS consists of a running instance of a task and its corresponding cost (e.g., energy consumption) and one complete run of the application on the target platform represents a sequence of RTSs [15]. The system is configured to meet the cost requirements of an RTS by choosing the appropriate system scenario,


INPUT DATABASE
(512 x 512)
(256 x 256)
(64 x 64)
APPLICATION
Efficient Pyramid Image Coder (EPIC)
Memory size
Dynamic input variable (i.e. Image size)

```
while $image.next \neq EndOfDatabase$ do
    if $size < image.size$ then
        Allocate ($image.size - size$)
    else if $size > image.size$ then
        Deallocate ($size - image.size$)
    end if
    $size = image.size$
    $EPIC\ Image\ processing\ldots$
end while
```

Figure C.1: Profiling results based on application code and input data

which is the one that satisfies the requirements using minimal power. In the following subsections, the different steps of the memory-aware system scenario methodology are outlined.

The system scenario methodology follows a two stage exploration, namely design-time and run-time stages, as described in [10]. This splitting is also employed in the memory-aware extension of the methodology. The two stage exploration is chosen because it reduces run-time overhead while preserving an important degree of freedom for run-time configuration [26]. The application is analysed at design-time and different execution paths causing variations in memory demands are identified. This procedure, which is time consuming and as a result can be performed only during the design phase, will result in a grey-box model representation of the application. The grey-box model hides all static and deterministic parts of the application, instead only providing their related memory costs to the system designer. Parts of the application code that are non-deterministic in terms of memory usage are directly available to the system designer [17].

C.4.1 Design-time Profiling Based on Data Variables

Application profiling is performed at design-time for a wide range of inputs. The analysis focuses on the allocated memory size during execution and on access pattern variations. Techniques described in [22] are, e.g., used in order to extract the access scheme through analysis of array iteration spaces.

The profiling stage is depicted in Fig. C.1 and consists of running the application code with suitable input data often found in a database, in order to produce profiling results. The results shown here are limited for demonstrational purposes. A real application would have thousands or millions of profiling samples. The profiling reveals parts of the application code with high memory activity and with varying memory access intensity, which possibly depends on input data variables. Because of this behaviour, a static study of the application code alone is insufficient since the target applications for this methodology have non-deterministic behaviour that is driven by input.

Profiling results provided to the designer include complete information about allocated memory size values together with the number of occurrences and duration for each of these memory size values. Moreover, correlation between input data variable values and the resulting memory behaviour can possibly be observed. This information is useful for the clustering step that follows. Profiling also reveals the worst case memory usage for a given set of inputs. The memory usage is measured using techniques presented in [22], in which authors compute the minimum amount of memory resources required to store the elements of an application.

In Fig. C.1 the profiled applications are two image related multimedia benchmarks and the input database should consist of a variety of images. The memory requirements in each case are driven by the current input image size, which is classified as a data variable due to the wide range of its possible values. Depending on the application the whole image or a region of interest is processed. Other applications have other input variables deciding the memory requirement dynamism, e.g., the channel SNR level in the case of an encoding/decoding application.

The input data used for profiling are generated based on realistic assumptions for each of the chosen benchmarks. Each application is studied and based on its functionality, a range of rational inputs are developed. In addition, example inputs are available for most of the studied benchmark applications. The choice of common and open-source benchmarks provides

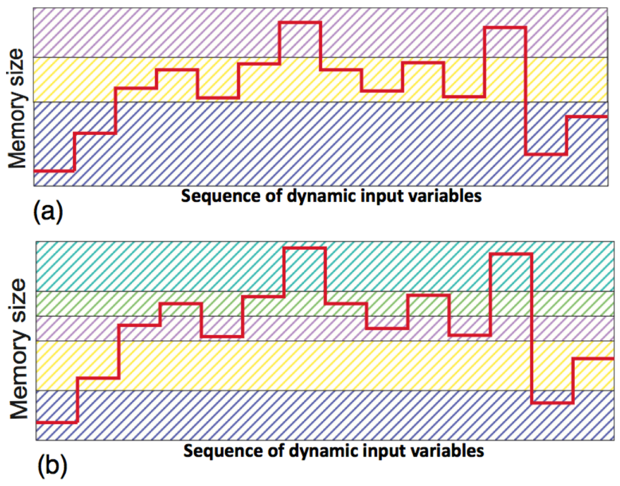


Figure C.2: Clustering of profiling results into three (a) or five (b) system scenarios

us the opportunity to find inputs publicly available. Based on the collected information, we define the range of realistic inputs and we generate a random set on inputs within these limits. For example, the bibliography provides information for the real SNR levels and the constraint length of the Viterbi encoder for each level, in order to achieve a successful communication. For profiling, we generate randomly a set of SNR levels that cover this whole range. Similar technique is used for the applications that use an image as an input. First, we explore the sizes of images commonly used to define the minimum and the maximum size and then we randomly generate a set of image sizes within the size limits. For the random generation of inputs, we assume the same probability for each situation, because there is no information available regarding the frequency of each input. As a result the frequency of each input is random.

C.4.2 Design-time System Scenario Identification Based on Data Variables

The next step is the clustering of the profiled memory sizes into groups with similar characteristics. This is referred to as system scenario identification. Clustering is necessary, because it will be extremely costly to have a different scenario for every possible size, due to the number of memories needed. Clustering neighbouring RTSs is a rational choice, because two instances with similar memory needs have similar energy consumption.

In Fig. C.2 the clustering of the previously profiled information is presented. The clustering of RTSs is based both on their distance on the memory size axis and the frequency of their occurrence. Consequently, the memory size is split unevenly with more frequent RTSs having a shorter memory size range. In the case of a clustering to three system scenarios the space is divided in the three differently coloured hashed areas depicted in Fig. C.2(a). Due to the higher frequency of RTSs in the yellow hashed area, that system scenario has a shorter range compared with its neighbouring scenarios. Such clustering is better than an even splitting because the energy cost of each system scenario is defined by the upper size limit, as each scenario should support all RTSs within its range. Consequently the overhead for the RTSs in the yellow area is lower compared to the overhead in the two other areas.

The same principle applies also when the number of system scenarios is increased to five, as depicted in Fig. C.2(b). The frequency sensitive clustering results in two short system scenarios that contain four RTSs each and three wider system scenarios with lower numbers of RTSs. The number of system scenarios should be limited mainly due to two factors. First, implementation of a high number of system scenarios in a memory platform is more difficult and complex. Second, the switching between the different scenarios involves an energy penalty that could become significant when the switching takes place frequently.

The memory size and the frequency of each RTS are not the only two parameters that should be taken into consideration during the system scenario identification. The memory size of each RTS results in a different energy cost depending on the way it is mapped into memory. The impact of the different assignment possibilities is included into the clustering by introduction of energy as a cost metric. The energy cost for each RTS is calculated using a reference platform with one to N memory banks. Increasing the number of memory banks results in lower energy per access since the most accessed elements can be assigned to smaller and more energy efficient banks. Unused banks can be switched off.

In the system scenario methodology, Pareto curves are used to capture alternative system configurations within a scenario [26]. In our work, a Pareto space is used for clustering that also includes the energy cost metric. For each RTS all different assignment options on alternative platform config-

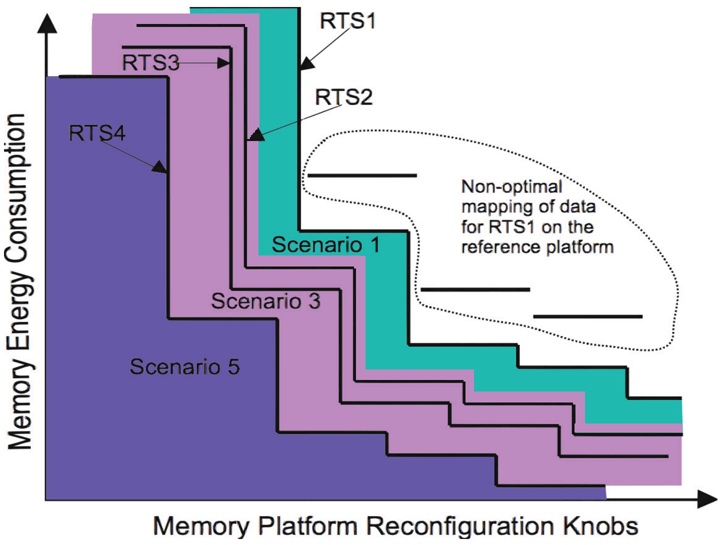

RTS1
RTS3
RTS2
RTS4
Non-optimal mapping of data for RTS1 on the reference platform
Scenario 1
Scenario 3
Scenario 5
Memory Energy Consumption
Memory Platform Reconfiguration Knobs

Figure C.3: Clustering of Pareto curves

urations are studied. Memory platform knobs are different sets of memory banks that are turned on and off. A Pareto curve is constructed for each RTS that contains the optimal assignment for each platform configuration. Hence, suboptimal assignments and assignments that result in conflicts are not included in the Pareto curve. In Fig. C.3 four Pareto curves, each corresponding to a different RTS, are shown together with energy cost levels corresponding to different platform configuration and data-to-memory assignment decisions. Three non-optimal mappings are also shown in Fig. C.3 for illustration. They are not part of the Pareto curve and consequently not included in the generation of scenarios. Pareto curves are clustered into three different system scenarios based again both on their memory size differences and frequency of occurrence. Clustering of RTSs using Pareto curves is more accurate compared to the clustering depicted in Fig. C.2, as it includes data-to-memory assignment options in the exploration.

The system scenario identification step includes the selection of the data variables that determine the active system scenario. This can be achieved by careful study of the application code, combined with the application's data input. The variable selection is done before clustering of RTSs into scenarios. For the choice of identification variables, there is a trade-off between the complexity and the accuracy of the scenario detection step. On one hand, if the identification is done using a group of complex variables and their correlation, there is a number of calculations needed in order to predict the active scenario. On the other hand, if the value of a single variable is monitored for scenario identification, the scenario detection is straightforward. Obviously, the accuracy of the scenario detection is higher on the first case, while the computational needs for scenario detection are lower on the second case. In other words, the more accurate scenario detection, the more resources are used by the run-time manager for detection. In our case the grey-box model reveals only the code parts that will influence memory usage, so that data variables deciding memory space changes can be identified. An example of this is a non static variable that influences the number of iterations for a loop that performs one memory allocation at each iteration. In the depicted example the system scenario detection data variable is the input image height and width values. Moreover, the designer should look for a correlation between input values and the corresponding cost. This information will be useful in the following steps of the methodology [26].

C.4.3 Run-time System Scenario Detection and Switching Based on Data Variables

Switching decisions are taken at run-time by the run-time manager. In this work, we use a simple and straightforward switching approach. Memory models provide the necessary information for the switching decision, namely the energy and the time penalty for switching between stages. The switching step consists of all platform configuration decisions that can be made at run-time, e.g., frequency/voltage scaling, changing the power mode of memory units, including turning them off, and reassignment of data to memory units. Switching takes place when the switching cost is lower than the energy gains achieved by switching.

In more detail, the switching mechanism implemented by the run-time manager includes the following actions:

1. Calculation of the energy consumption by processing the next input on the currently active scenario (E1).

2. Calculation of the energy consumption by processing the next input on its most energy efficient scenario (E2).

3. Calculation of the energy penalty for switching the needed memory banks to the configuration of the most efficient scenario (E3).

4. Evaluation of the expression: E1 >E2 + E3. If the energy cost of the current configuration (E1) is greater than the combined cost of the new configuration and the required switching (E2 + E3), then the decision is to switch. Otherwise, the switching decision is negative and the system stays on the currently running configuration

5. Switching of the platform to its new configuration. The memory banks switch to the appropriate state, which is defined by the chosen scenario.

Switching costs are defined by the platform and include all memory energy penalties for run-time reconfigurations of the platform, e.g., extra energy needed to change state of a memory unit.

The run-time manager is minimal, resulting in a very small overhead, and is complementary to an operating system (OS), if such is available on the platform. In the presence of an embedded OS, the OS needs to start the runtime manager and grant it access to system reconfiguration calls. In both cases, when the run-time manager is active, it performs a few simple steps with minimal system performance overhead. In more detail, the run-time manages checks the current state of the monitored identification variable(s), determine the next scenario based on this value and decides whether switching should be performed. The corresponding scenario for a given value of the identification variable is a simple look-up, because all the analysis has been performed at design-time. The hardware configuration for the active scenario is also explored at design-time and the run-time manager is aware of the required changes. In our approach no need is present for modifications of the application code. Instead, we add the above mentioned changes in the middleware layer. That has the additional advantage that the run-time manager is reusable across several tasks/processes running at the application layer on top of this shared middleware. The application data are stored and accessed the same way at the application level. The difference with the proposed approach is the size and the state of the memory bank that the data are stored in. The application is unaware of the exact way the data is stored and accessed and the methodology ensures that the accessed addresses in the application code always corresponds to an active bank.

In Fig. C.4 an example of the run-time phase of the methodology is depicted. The run-time manager identifies the size of the image that will be processed and reconfigures the memory subsystem on the platform, if needed, by increasing or decreasing the available memory size. The reconfiguration options are effected by platform hardware limitations. The image

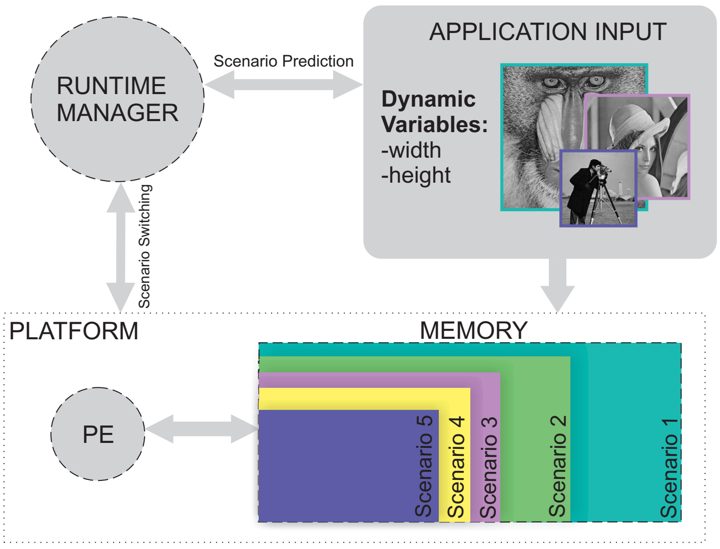


Figure C.4: Run-time system scenario detection and switching based on the current input

size is in this case the data variable monitored in order to detect the system scenario and the need for switching.

C.5 Target Platform and Energy Models

Selection of target platform is an important aspect of the memory-aware system scenario methodology. The key feature needed in the platform architecture is the ability to efficiently support different memory sizes that correspond to the system scenarios generated by the methodology. Execution of different system scenarios then leads to different energy costs, as each configuration of the platform results in a specific memory energy consumption. The dynamic memory platform is achieved by organising the memory area in a varying number of banks that can be switched between different energy states.

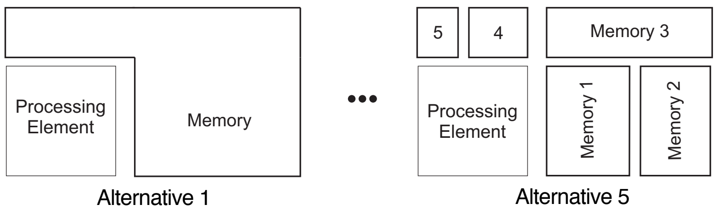


Figure C.5: Alternative memory platforms with varying number of banks

C.5.1 Target Memory Platform Architecture

In this work, a clustered memory organisation with up to five memory banks of varying sizes is explored. The limitation in the number of memory banks is necessary in order to keep the interconnection cost between the processing element (PE) and the memories constant through exploration of different architectures.

For more complex architectures the interconnection cost should be considered and analysed separately for accurate results. Although power gating can be applied to the bus when only a part of a longer bus is needed, an accurate model of the memory wrapper and interconnection must developed, which is beyond the scope of the current work.

Some examples of alternative memory platforms that can be used for exploration is shown in Fig. C.5. Point-to-point connections with negligible interconnect costs between elements are assumed for up to five memory banks. The decision to use memory banks with varying sizes on the clustered memory organization increases the reconfiguration options and consequently the potential energy gains. In general, smaller memories are more energy efficient compared to larger memories banks. However, in some cases large memory banks are needed in order to fit the application data without the need for too many small memories causing complex interconnects. The goal is to use the most energy efficient banks to store the most frequently used data. The calculations needed for enabling the minimum number of banks is simple, given the application requirements for the current input and the sizes of the five memory banks.

C.5.2 Models of Different Memory Types

The dynamic memory organisation is constructed using commercially available SRAM memory models (MM). For those models delay and energy numbers are derived from a commercial memory compiler. In addition, experimental standard cell-based memories (SCMEM) [30] are considered for smaller memories due to their energy and area efficiency for reasonably small storage capacities, as argued in [29]. The standard cell-based memories are synthesized using Cadence RTL compiler for TSMC 40nm standard library. Afterwords, power simulations on the synthesized design are carried out using Synopsys PrimeTime, in order to obtain energy numbers. Both MMs and SCMEMs can operate under a wide range of supply voltages, thus support different operating modes that provide an important exploration space.

- Active mode: The normal operation mode, in which the memory can be accessed at the maximum supported speed. The supply voltage is 1.1V. The dynamic and leakage power are higher compared to the other modes. Only on active mode the data are accessible without time penalties, in contrast to light and deep sleep modes. In this work all the memory accesses are performed in active mode.

- Light sleep mode: The supply voltage in this mode is lower than active with values around 0.7V. The access time of the memory is significantly higher than the access time in active mode. Switching to active mode can be performed with a negligible energy penalty and a small time penalty of a few clock cycles (less than 10). Data is retained.

- Deep sleep mode: The supply voltage is set to the lowest possible value that can be used without loss of data. This voltage threshold is expected to be lower for SCMEMs than MM models and can be as low as 0.3V. The number of clock cycles needed for switching to active mode is higher compared to sleep mode, typically in the range of 20 to 50 clock cycles depending on the clock speed. Consequently, the speed of the PE and the real-time constrains of the applications has to be taken into consideration when choosing light or deep sleep mode at a specific time.

- Shut down mode: Power-gating techniques are used to achieve near zero leakage power. Stored data is lost. The switch to active mode requires substantially more energy and time. However, switching unused memories to this mode, providing that their data are not needed in the future, results in substantial energy savings.

The exploration includes memories with 4 energy modes in contrast to a more conservative approach that assumes only on and off states. This is in line with is modern energy efficient memories that tend to support an increasing number of energy modes as a feature. The methodology is still applicable to a memory organization that supports only two modes, but the intention of this work is to explore the state of the art memory technologies. The policy for switching between the modes depends on the reuse of the stored data, which depends on the nature of the target application. The switching policy is determined by examining the following condition for the stored data in a memory bank:

- If the data is currently under processing, then the only acceptable mode is the active mode.
- If the data is not accessed, but will be needed in the near future, then the light or deep sleep mode should be chosen.
- If the processing of the data is completed and the application is not accessing them again, the shut-down mode is the optimal solution.

Applications that perform calculations on a set of input data to produce a result and never re-access the initial data, normally only use two modes. The need for more energy modes arises from the fact that many applications re-access the initial data or some intermediate results. Thus, the runtime manager chooses a sleep mode that reduces the leakage power, but retains the data for future use.

The necessary energy/power information is available to the system designer and relative values for a subset of the used sizes in the current work are presented in Tab. C.1 and in Tab. C.2. It shows that the choice of memory units has an important impact on the energy consumption. Moreover, different decisions have to be made based on the dominance of dynamic or leakage energy in a specific application. In the current work memory architectures with 1 to 5 memory units of different sizes are explored and the optimal configuration is chosen. The methodology is in general not restricted to specific memory types or benchmarks and can handle more complex hierarchical memory architectures and applications. However, in this study the chosen applications have a relatively small memory space requirement limited to around 100KB, which is the case for many applications run on modern embedded systems.

Table C.1: Relative dynamic energy for a range of memories with varying capacity and type

Type	**Lines x wordlength**	**Dynamic Energy [J]**		Switching to Active from	
		Read	Write	Deep[uJ]	Light[uJ]
MM	32 x 8	4.18×10^{-8}	3.24×10^{-8}	0.223	0.031
MM	32 x 16	6.79×10^{-8}	5.89×10^{-8}	0.223	0.031
MM	32 x 128	4.33×10^{-7}	4.31×10^{-7}	1.42	0.168
MM	256 x 128	4.48×10^{-7}	4.60×10^{-7}	1.70	0.171
MM	1024 x 128	5.11×10^{-7}	5.75×10^{-7}	2.81	0.179
MM	4096 x 128	9.60×10^{-7}	4.57×10^{-7}	9.01	0.457
SCMEM	128 x 128	2.5×10^{-7}	0.8×10^{-8}	1.51	0.045
SCMEM	1024 x 8	1.7×10^{-8}	0.6×10^{-8}	0.325	0.021

Table C.2: Relative static power for a range of memories with varying capacity and type

Type	**Lines x wordlength**	**Static Leakage Power per Mode[W]**			
		Active	Light-sleep	Deep-sleep	Shut-down
MM	32 x 8	0.132	0.125	0.063	0.0016
MM	32 x 16	0.134	0.127	0.064	0.0022
MM	32 x 128	0.171	0.160	0.083	0.0112
MM	256 x 128	0.207	0.184	0.104	0.0293
MM	1024 x 128	0.349	0.283	0.189	0.102
MM	4096 x 128	0.95	0.708	0.544	0.396
SCMEM	128 x 128	0.083	0.057	0.027	0.0022
SCMEM	1024 x 8	0.042	0.028	0.014	0.0011

C.5.3 Total Energy Consumption Calculation

Both the dynamic and the static energy consumed in the memory subsystem is included in the calculations. The overall energy consumption for each configuration is calculated using a detailed formula, as can be seen in Eq.C.1. All the important transactions on the platform that contribute to the overall energy are included, in order to achieve as accurate results as possible. In particular:

- N_{rd} is the number of read accesses
- E_{Read} is the energy per read
- N_{wr} is the number of write accesses
- E_{Write} is the energy per write
- T is the execution time of the application
- $T_{LightSleep}$, $T_{DeepSleep}$ and $T_{ShutDown}$ are the times spent in light sleep, deep sleep and shut down states respectively
- $P_{leak_{Active}}$ is the leakage power in active mode
- $P_{leak_{LightSleep}}$, $P_{leak_{DeepSleep}}$ and $P_{leak_{Shutdown}}$ are the leakage power values in light sleep, deep sleep and shut down modes with different values corresponding to each mode
- $N_{SWLight}$, N_{SWDeep} and $N_{SWShutDown}$ are the number of transitions from each retention state to active state
- $E_{LightSleep\,to\,Active}$, $E_{DeepSleep\,to\,Active}$ and $E_{ShutDown\,to\,Active}$ are the energy penalties for each transition respectively.

$$
\begin{aligned}
E = \sum_{memories}^{all} & (N_{rd} \times E_{Read} \\
& + N_{wr} \times E_{Write} \\
& + (T - T_{LightSleep} - T_{DeepSleep} - T_{ShutDown}) \times P_{leak_{Active}} \\
& + T_{LightSleep} \times P_{leak_{LightSleep}} \\
& + T_{DeepSleep} \times P_{leak_{DeepSleep}} \\
& + T_{ShutDown} \times P_{leak_{ShutDown}} \\
& + N_{SW\,Light} \times E_{LightSleep\,to\,Active} \\
& + N_{SW\,Deep} \times E_{DeepSleep\,to\,Active} \\
& + N_{SW\,ShutDown} \times E_{ShutDown\,to\,Active})
\end{aligned}
\tag{C.1}
$$

The overall energy consumption is given after calculating the energy for each memory bank. The execution time of the application is needed to calculate the leakage time. It can be found by executing the application on a reference embedded processor. The simulator described in [4] is chosen to calculate execution time for the chosen applications in this work. The processor is assumed to be running continuously, accepting new input data as soon as computations on the previous data set has been finished. Memory sleep times are hence only caused by data dependent dynamic behaviour.

C.5.4 Memory Architecture Exploration

The exploration of alternative memory platforms is performed using the steps described in Alg. 3. The exploration is performed at system scenario identification phase, after the profiling of RTSs. All potentially energy efficient configurations are tested for a given number of scenarios and the sequence of RTSs of the application. First, a database with all the memory models that are available to the system designer is imported. The memory database can afterwards be pruned to reduce the complexity of the exploration, as explained in the following paragraph. After the optional pruning, all possible configurations for a given number of memory banks are constructed. The only requirement in order to keep a configuration for further investigation is that the combined size of all banks should satisfy the storage requirements of the most demanding RTS. Then, each configuration is tested for the sequence of RTSs and the one that minimizes Eq.B.1 is chosen as the most energy efficient for this number of scenarios (i.e., number of banks).

Algorithm 3 Memory organisation exploration steps

```
 1: $RTSset \leftarrow$ storage requirement for each RTS
 2: $Database \leftarrow$ extensive memory database
 3: //Database pruning:
 4: for all relevant memory sizes do
 5:     pick memory models from $Database$ according to application characteristics
 6: end for
 7: $m \leftarrow$ number of memory models in pruned $Database$
 8: $N \leftarrow$ number of scenarios (up to 5 in this work)
 9: //Exploration of memory organisations:
10: for $n = 1 \rightarrow N$ do
11:     $k \leftarrow$ combination of $n$ banks out of a set of $m$ memories
12:     for all generated k combinations do
13:         if $\sum_1^n size(bank) \geq size(max(RTS))$ then
14:             //Select configuration that minimizes Eq.B.1
15:             $Ecurrent \leftarrow$ Energy given by Eq.B.1 for current combination
16:             if $Eminimum > Ecurrent$ then
17:                 $Eminimum \leftarrow Ecurrent$
18:             end if
19:         end if
20:     end for
21: end for
```

The exploration for the most energy efficient memory organization is a computational intensive task and can only be performed at design-time. At run-time the search for the optimal configuration is very simple, because the set of the few possible configurations is available. The exhaustive search for the most energy efficient configuration for up to five banks is performed in a reasonable time at design-time. The number of possible configurations is given by the number of the memory banks and the number of memories in the database, as shown in Alg. 3. In general, if n is the number of banks and m is the number of memory models, there are m^n possible combinations.

Assuming a range of sizes from 1KB to 64KB, there are 7 different sizes (1KB, 2KB, 4KB, 8KB, 16KB, 32KB, 64KB). In every size, there is at most one memory model with the minimum energy per read access, one with the minimum energy per write access and one with the minimum leakage power. Depending on the worst-case memory size given by application profiling, only a number of combinations between the 7 memory sizes fulfils the requirements. For example, 5 memory banks of 1KB are not sufficient for any of the studied applications. The code of the application may reveal if there is dominance of read/write accesses for an array or it is not accessed frequently so that leakage will dominate. This way some of the memory models can be eliminated for each size. Let us assume an application with a worst case memory requirement of 100KB that is dominated by sequences of read accesses. In this case, we have only 21 possible combinations (all combinations of 5 out of a set of 7 with repetitions and unimportant order [39]). Then, we have to explore only the combinations that are greater than 100KB using bank models with the minimum energy per read.

More formally, at design-time we generate all the combinations with repetitions from a database of m memory models taken out n at a time. We are looking for sets with repetitions, which means that we can choose the same memory model more than once. This is important, in order to also include more homogeneous organizations into the exploration. However, the order of memories has no effect on our exploration. This means that the combination of memories S1 = (8KB, 8KB, 16KB, 32KB) is equivalent with S2 = (8KB, 16KB, 8KB, 32KB) and only one of them is included in our exploration. The number of possible configurations is given by the following general formula:

$$k = \binom{m}{n} = \frac{m(m-1)...(m-n+1)}{n(n-1)...1}$$

In our case, the typical values for m and n are 15 and 5 respectively. The complexity of the exploration algorithm is $\mathcal{O}(k)$, where k is the number of possible memory combinations. Therefore, the whole exploration for up to 5 memory banks can be performed during the design phase. At run-time the system can chose the appropriate configuration, without the need for exploration.

C.6 Application Benchmarks

The applications that benefit most from the memory-aware system scenario methodology are characterised by having dynamic utilization of the memory organisation during their execution. Multimedia applications often exhibit such a dynamic variation in memory requirements during their lifetime and consequently are suitable candidates for the presented methodology. The

Table C.3: Benchmark applications overview

Name	Source	Scenario detection variable
Epic image compression	MediaBench	Image size
Motion Estimation	MediaBench	Image size
Blowfish decoder	MiBench	Input file size
Jacobi 1D Decomposition	Polybench	Number of steps
Mesa 3D	MediaBench	Loop bound
JPEG DCT	MediaBench	Block size
PGP encryption	MediaBench	Encryption length
Viterbi encoder	Open	Constraint length

effectiveness is demonstrated and tested using a variety of open multimedia benchmarks, which can be found in the Polybench [38], Mibench [13] and Mediabench [24] benchmark suites. The broad set of multimedia benchmarks under exploration is representative for the entire domain of multimedia applications.

C.6.1 Benchmark Applications and Corresponding Input Databases

An overview of the benchmark applications that were tested is presented in Tab. C.3.

Two key parameters under consideration are the dynamic data variable of each application and the variation in the memory requirement it causes. The dynamic data variable is the variable that results in different system scenarios due to its range of values. Examples of such a variable are an input image of varying size or data dependent loop bound values. For each application an appropriate set of realistic RTS cases is constructed. The memory size limits are defined as the minimum and maximum storage requirement occurring during the profiling of an application.

EPIC (Efficient Pyramid Image Coder) image compression can compress all possible sizes of images. The size of the input image has an effect on memory requirements during compression and several images were given as inputs. *Motion estimation* is another media application in which image size is the dynamic data variable. In this case the image defines the area that has to be explored to determine the motion vectors and different images are tested. The set of input data is constructed using publicly available images commonly used for testing this algorithm.

The dynamism in the *blowfish decoder* benchmark is a result of variations in the input file that is decoded. Again, the methodology explores the behaviour for several input files in order to identify system scenarios. The *Jacobi 1D decomposition* algorithm can be executed using a varying number of steps with a direct effect on memory usage and is hence another suitable benchmark for the system scenario methodology. The number of steps is increased on every next iteration, in order to generate a set of different memory requirements. The set of input data is a random sequence of input signals and each of them corresponds to a different number of steps. *Mesa 3D* is an open graphics library with a dynamic loop bound in its kernel that provides the desired dynamic behaviour.

The discrete cosine transformation (DCT) block used in the *JPEG compression* algorithm has a memory footprint that is heavily influenced by the block size. The input database consists of an ascending size sequence of blocks. For the *PGP encryption* algorithm the encryption length parameter has an important impact on memory size, which can be exploited using system scenarios. Thus, we create a database starting from the lowest encryption length value of 384 and gradually increasing it up to 2048. The effect of the channel SNR level on the constraint length value of the *Viterbi encoder* algorithm is discussed in [8]. Increasing noise on the channel demands a more complex encoding in order to maintain a constant bit error rate (BER), which consequently increases the memory requirements during execution. The memory size variation is given for execution under different SNR levels.

C.6.2 Classification of Applications Based on Dynamic Characteristics

The required dynamism for applying the memory-aware system scenario methodology can be produced by several code characteristics, covering a wide range of potential applications, as discussed in the previous subsection. In this subsection dynamic characteristics are outlined that can assist the system designer in the employment of the methodology and reveal the expected behaviour prior to experimentation with an application. The dynamic characteristics that are used to categorize the applications are the dynamism in the memory size bounds and the variance of cases within the memory size limits. The characterization of the benchmark applications based on those key parameters is presented in Tab. C.4.

Table C.4: Characterization of benchmark applications (See Tab. C.3 for index

Name	**Dynamic Characteristics**	
	Memory Variation(B)	Shape of histogram
Epic image compression	4257 - 34609	Right skewed
Motion Estimation	4800 - 52800	Gaussian-like
Blowfish decoder	256 - 5120	Left skewed
Jacobi 1D Decomposition	502 - 32002	Right skewed
Mesa 3D	5 - 50000	Gaussian-like
JPEG DCT	10239 - 61439	Gaussian-like
PGP encryption	3073 - 49153	Gaussian-like
Viterbi encoder	5121 - 14337	Right skewed

The profiling information can be organised into a histogram in order to be easily comprehensive. The horizontal axis depicts the memory requirements for the RTSs and the vertical axis the number of occurrences for its RTS. In this way the system designer can quickly identify the expected gains of the system scenario methodology by identifying the width and the shape of the histogram. The width of the histogram gives an overview of the memory size bounds, while the shape reveals the variation of the RTSs for the studied application.

The memory size bounds correspond to the minimum and maximum memory size values profiled over all possible cases. In general, larger distances between upper and lower bounds increase the possibilities for energy gains. This is a result of using larger and more energy hungry memories in order to support the memory requirements for the worst case even when only small memories are required. Large energy gain is expected when large parts of the memory subsystem can be switched into retention for a long time. For several of the benchmarks the difference between maximum and minimum memory size is close to 50KB. This includes JPEG, motion estimator, mesa 3D, and PGP, where large gains can be expected. On the other hand, the system designer should expect lower energy gains for applications that show a relatively less dynamic behaviour with regard to their memory size limits. Examples here are the blowfish and viterbi algorithms.

The variation takes into consideration both the number of different cases that are present within the memory requirement limits and the distribution of those cases between minimum and maximum memory size. This variation corresponds to the shape of the histogram of the application. Applications with a limited number of different cases are expected to have most of its possible gain obtained with a few platform supported system scenarios and much smaller energy gains from additional system scenarios. After this point most of the cases are already fitting one of the platform configurations and adding new configurations have a minimal impact. The opposite is seen for applications that feature a wide range of well distributed cases.

Based on the analysis above, applications can be classified into four main categories. The first category has a histogram with a wide range of RTS memory sizes and most RTSs placed to the left. This category is defined as a right skewed distribution, according to the direction of the tail. In this category the RTSs corresponding to large memories are rarely used, so high gains are expected by applying the methodology. The second case is when there is a close to Gaussian RTS distribution in the histogram. The expected gains are here lower than for the first category. The opposite of the first category is when the histogram is left skewed, meaning that the RTSs with the higher memory requirements are dominant. In this category, system designer should expect the smallest gains. The forth case is when all the RTSs have the same memory requirements and there is no distribution on the histogram, which results in no energy gains for the methodology.

C.7 Results

The memory aware system scenario methodology is applied to all the presented benchmark applications to study its effectiveness. The profiling phase is based on different input for the data variables shown in Tab. C.3 and is followed by the clustering phase. Different stimuli are used for the profiling phase and the run-time calculations. The two different sets of input are generated using the same random distribution function. The two input sets have different sequences of inputs and the occurrence of each RTS is random. The probability of each RTS is kept the same in the two input sets. The execution and sleep times needed in Eq.B.1 are found through the profiling but are also reflected by the dynamic characteristics in Tab. C.4. Data variables are the variables used by the run-time manager in order to predict the next active scenario. The clustering is performed with one to five system scenarios. All potentially energy efficient configurations are tested for a given number of scenarios using the steps described in Alg. 3. For example, in the case of 2 scenarios all possible memory platforms with 2 memory banks that fulfil the memory size requirement of the worst case are generated and tested. The same procedure is performed for 3, 4 and 5 scenarios. The exploration includes memories of different sizes, technologies and varying word lengths.

The proposed methodology provides the same performance on the memory subsystem and only reduces energy, if possible. The access time is the same both in the static and the dynamic memory architecture. The goal of the methodology is to always meet the memory requirements of the applications, using the minimum amount of active memory banks. A worst case scenario is activated any time that the input is outside of the range of inputs. The worst case configuration activates all the available memory banks. In this case the system achieves its highest capability, but the energy consumption is the highest. The active scenario always provides enough space to fit the active data for an input within the profiled range of inputs. The energy gains are a result of switching unnecessary banks to a low-power mode when it is possible. The active banks have the same performance and are independent of any inactive banks on the clustered architecture.

The energy gain percentages are presented in Fig. C.6. Energy gains are compared to the case of a fixed non-re-configurable platform, i.e., a static platform configuration with only 1 scenario. This corresponds to zero percentage gain in Fig. C.6. The most efficient of the tested organisations for each benchmark are presented in Fig. B.6, where each memory bank is depicted with a different colour and each length is proportional to the memory bank size. The blowfish decoder is the only benchmark that has only 3 banks in its most efficient memory organisation. In Tab. C.5 the minimum and maximum energy gains for each benchmark application are shown.

C.7.1 Classification of the Applications

The introduction of a second system scenario results in energy gains between 15% and 40% for the tested applications. Depending on the application's dynamism the maximum reported energy gains range from around 35% to 55%. As expected according to the categorisation presented in subsection C.6.2, higher energy gains are achieved for applications with more dynamic memory requirements, i.e., bigger difference between the minimum and maximum allocated size. The maximum gains for JPEG, motion estimator, mesa 3D and PGP are around 50% while blowfish, jacobi, and Viterbi decoders are around 40%.

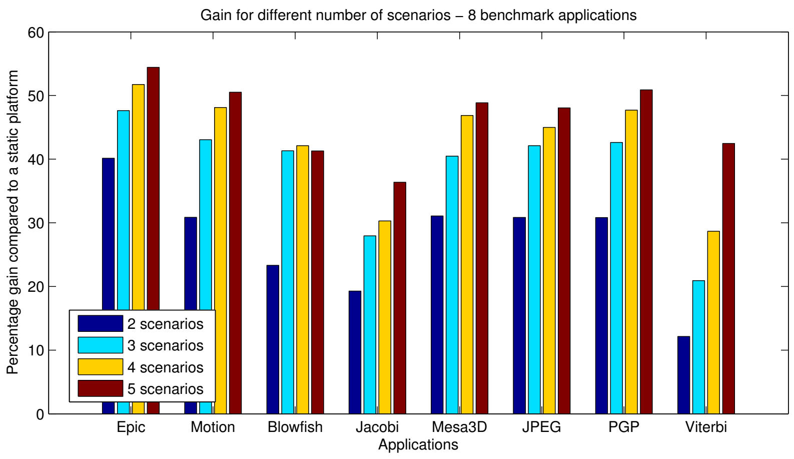


Figure C.6: Energy gain for increasing number of system scenarios - Static platform corresponds to 0%

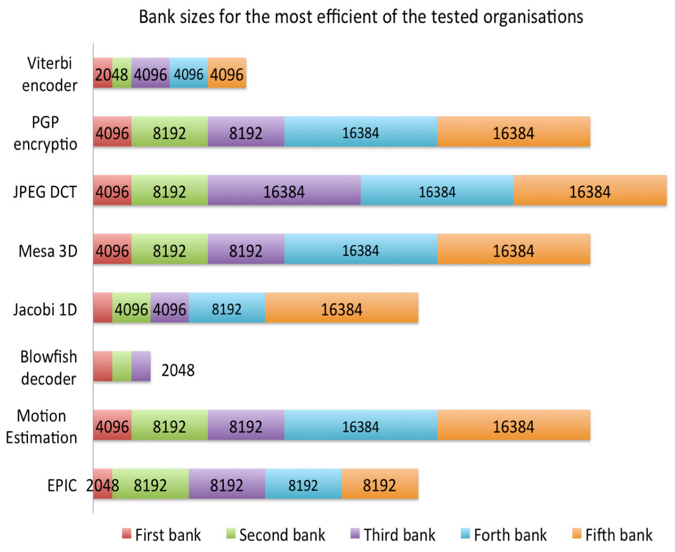


Figure C.7: Bank sizes for the most efficient of the tested organisations for each benchmark

Table C.5: Range of energy gains on the memory subsystem

EPIC		**Motion**		**Blowfish**		**Jacobi**	
Min	Max	Min	Max	Min	Max	Min	Max
40.6%	55.1%	31.2%	50.1%	23.5%	42.0%	19.8%	35.9%
Mesa3D		**JPEG**		**PGP**		**Viterbi**	
Min	Max	Min	Max	Min	Max	Min	Max
31.3%	49.2%	31.1%	48.9%	30.6%	51.2%	12.5%	42.4%

As the number of system scenarios that are implemented on the memory subsystem increases, the energy gains improve since variations in memory requirements can be better exploited with more configurations. However, the improvement with increasing numbers of system scenarios differ depending on the kind of dynamism present in each application. The application with the highest variation in distribution of memory requirements is the Viterbi encoder/decoder and gains around 10% is seen for every new memory bank added, even for a platform growing from four to five banks. In contrast, the application with the lowest number of different cases, blowfish, cannot further exploit a platform with more than three banks. Another case in which smaller energy gains are achieved, after a certain number of platform supported system scenarios have been reached, is the PGP encryption algorithm. In this benchmark the introduction of more scenarios has an energy impact of less than 5% after the limit of three system scenarios has been reached.

C.7.2 Switching Overhead

The switching cost increases for an increasing number of system scenarios due to the increasing frequency of platform reconfiguration. This overhead reduces the achieved gain, but for up to 5 scenarios we still see improvements for all but one of our benchmarks. The switching cost is below 2% even for a platform with 5 memory banks in all cases. Apart from the number of scenarios, the switching cost depends on the sensitivity of the variable used for scenario identification. A change of value on the identification variable indicates potentially a new scenario. For an increasing frequency of changes, the switching cost increases.

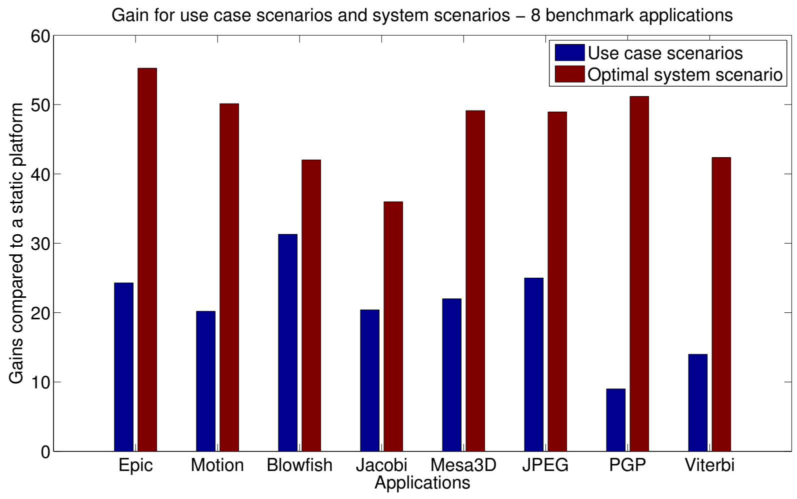


Figure C.8: Energy gain for use case scenarios and system scenarios

C.7.3 Comparison with Use Case Scenario

Comparative results from applying a use case scenario approach as a reference are presented in Fig. C.8. Reported energy gains for both use case scenarios and the most efficient case of the system scenarios are given assuming a static platform as a base (0%). Use case scenarios are generated based on a higher abstraction level that is visible as a user's behaviour. For example, use case scenarios for image processing applications generate three scenarios, if large, medium and small are the image sizes identified by the user. Similarly, use case scenarios for JPEG compression identify only low and high compression as options and motion estimation is performed on I, P and B video frames, without exploring fine grain differences inside a frame. In general, use case scenario identification can be seen as more coarse compared to identification on the detailed system implementation level. As seen in Fig. C.8 the use case gains are superior only to a static platform.

C.7.4 Run-Time Overhead

The reported energy gains are for the memory subsystem. As motivated in Section C.2 this has previously been shown to be a major contributor to the total energy consumption. An additional energy overhead from the system scenario approach can be found in the processor performing the run-time system scenario detection and switching. This overhead is partly incorporated in $E_{SleepActive}$, in particular if traditional system scenarios are already implemented so that the only overhead is the addition of memory-awareness. The run-time overhead is kept low, because the run-time manager is active for less than 1% of the time needed for the execution of the application.

C.8 Conclusions

The scope of this work is to apply the memory-aware system scenario methodology to a wide range of multimedia application and test its effectiveness based on an extensive memory energy model. A wide range of applications is studied that allow us to draw conclusions about different kinds of dynamic behaviour and their effect on the energy gains achieved using the methodology. The results demonstrate the effectiveness of the methodology reducing the memory energy consumption with between 35% and 55%. Since memory size requirements are still met in all situations, performance is not reduced. The memory-aware system scenario methodology is suited for applications that experience dynamic behaviour with respect to memory organisation utilization during their execution.

Bibliography

[1] Santosh G Abraham, Scott Mahlke, et al. Automatic and efficient evaluation of memory hierarchies for embedded systems. In *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on*, pages 114–125. IEEE, 1999.

[2] L. Benini et al. Increasing energy efficiency of embedded systems by application-specific memory hierarchy generation. *Design Test of Computers, IEEE*, 17(2):74 –85, apr-jun 2000.

[3] L. Benini, A. Macii, and M. Poncino. A recursive algorithm for low-power memory partitioning. In *Low Power Electronics and Design, 2000. ISLPED '00. Proceedings of the 2000 International Symposium on*, pages 78 – 83, 2000.

[4] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[5] Fei Chen and Edwin Hsing-Mean Sha. Loop scheduling and partitions for hiding memory latencies. In *Proceedings of the 12th international symposium on System synthesis*, page 64. IEEE Computer Society, 1999.

[6] E. Cheung, H. Hsieh, and F. Balarin. Memory subsystem simulation in software tlm/t models. In *Design Automation Conference, 2009. ASP-DAC 2009. Asia and South Pacific*, pages 811 –816, jan. 2009.

[7] Eui-Young Chung, Giovanni De Micheli, and Luca Benini. Contents provider-assisted dynamic voltage scaling for low energy multimedia

applications. In *Proceedings of the 2002 international symposium on Low power electronics and design*, ISLPED '02, pages 42–47, 2002.

[8] Iason Filippopoulos, Francky Catthoor, Per Gunnar Kjeldsberg, Elena Hammari, and Jos Huisken. Memory-aware system scenario approach energy impact. In *NORCHIP, 2012*, pages 1 –6, nov. 2012.

[9] Philip Garcia, Katherine Compton, Michael Schulte, Emily Blem, and Wenyin Fu. An overview of reconfigurable hardware in embedded systems. *EURASIP J. Embedded Syst.*, 2006(1):13–13, January 2006.

[10] Stefan Valentin Gheorghita, , et al. System-scenario-based design of dynamic embedded systems. *ACM Trans. Des. Autom. Electron. Syst.*, 14(1):3:1–3:45, January 2009.

[11] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *Solid-State Circuits, IEEE Journal of*, 31(9):1277 –1284, sep 1996.

[12] Peter Grun, Nikil Dutt, and Alex Nicolau. Mist: An algorithm for memory miss traffic management. In *Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design*, pages 431–438. IEEE Press, 2000.

[13] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14. IEEE, 2001.

[14] Javier J Gutierrez, Maria J Escalona, Manuel Mejias, Jesus Torres, and Arturo H Centeno. A case study for generating test cases from use cases. In *Research Challenges in Information Science, 2008. RCIS 2008. Second International Conference on*, pages 209–214. IEEE, 2008.

[15] E. Hammari, F. Catthoor, J. Huisken, and P G Kjeldsberg. Application of medium-grain multiprocessor mapping methodology to epileptic seizure predictor. In *NORCHIP, 2010*, pages 1 –6, nov. 2010.

[16] Elena Hammari, Francky Catthoor, Per Gunnar Kjeldsberg, Jos Huisken, K Tsakalis, and L Iasemidis. Identifying data-dependent system scenarios in a dynamic embedded system. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, page 1. The Steering Committee of The World

Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2012.

[17] Stefaan Himpe, Francky Catthoor, G De Coninck, and J Van Meerbergen. Mtg and grey-box: modeling dynamic multimedia applications with concurrency and non-determinism. 2002.

[18] J. Hulzink, M. Konijnenburg, M. Ashouei, A. Breeschoten, T. Berset, J. Huisken, J. Stuyt, H. de Groot, F. Barat, J. David, et al. An ultra low energy biomedical signal processing system operating at near-threshold. *Biomedical Circuits and Systems, IEEE Transactions on*, 5(6):546–554, 2011.

[19] Bruce L Jacob, Peter M Chen, Seth R Silverman, and Trevor N Mudge. An analytical model for designing memory hierarchies. *Computers, IEEE Transactions on*, 45(10):1180–1194, 1996.

[20] Axel Jantsch, Peeter Ellervee, Ahmed Hemani, Johnny Öberg, and Hannu Tenhunen. Hardware/software partitioning and minimizing memory interface traffic. In *Proceedings of the conference on European design automation*, pages 226–231. IEEE Computer Society Press, 1994.

[21] Mahmut Kandemir, Ugur Sezer, and Victor Delaluz. Improving memory energy using access pattern classification. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 201–206. IEEE Press, 2001.

[22] A. Kritikakou, F. Catthoor, V. Kelefouras, and C. Goutis. Near-optimal and scalable intra-signal in-place for non-overlapping and irregular access scheme. *ACM Trans. Design Automation of Electronic Systems (TODAES)*, conditionally accepted, 2013.

[23] A. Kritikakou, F. Catthoor, V. Kelefouras, and C. Goutis. A scalable and near-optimal representation for storage size management. *ACM Trans. Architecture and Code Optimization*, conditionally accepted, 2013.

[24] C. Lee, M. Potkonjak, and W.H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335. IEEE Computer Society, 1997.

[25] Yanbing Li and Wayne H Wolf. Hardware/software co-synthesis with memory hierarchies. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 18(10):1405–1417, 1999.

[26] Zhe Ma et al. *Systematic Methodology for Real-Time Cost-Effective Mapping of Dynamic Concurrent Task-Based Systems on Heterogenous Platforms.* Springer Publishing Company, Incorporated, 1st edition, 2007.

[27] A. Macii, L. Benini, and M. Poncino. *Memory Design Techniques for Low-Energy Embedded Systems.* Kluwer Academic Publishers, 2002.

[28] P. Marchal, D. Bruni, J.I. Gomez, L. Benini, L. Pinuel, F. Catthoor, and H. Corporaal. Sdram-energy-aware memory allocation for dynamic multi-media applications on multi-processor platforms. In *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 516–521, 2003.

[29] P Meinerzhagen, C Roth, and A Burg. Towards generic low-power area-efficient standard cell based memory architectures. In *Circuits and Systems (MWSCAS), 2010 53rd IEEE International Midwest Symposium on*, pages 129–132. IEEE, 2010.

[30] Pascal Meinerzhagen, SM Yasser Sherazi, Andreas Burg, and Joachim Neves Rodrigues. Benchmarking of standard-cell based memories in the sub-vt domain in 65-nm cmos technology. *IEEE Transactions on Emerging and Selected Topics in Circuits and Systems*, 1(2), 2011.

[31] Narasinga Rao Miniskar. *System Scenario Based Resource Management of Processing Elements on MPSoC.* PhD thesis, Katholieke Universiteit Leuven, 2012.

[32] Yuji Oshima, Kukuh Nirmala, Jyoji Go, Yoshiko Yokota, Jiro Koyama, Nobuyoshi Imada, Tsuneo Honjo, and Kunio Kobayashi. High accumulation of tributyltin in blood among the tissues of fish and applicability to environmental monitoring. *Environmental toxicology and chemistry*, 16(7):1515–1517, 1997.

[33] M. Palkovic, H. Corporaal, and F. Catthoor. Heuristics for scenario creation to enable general loop transformations. In *System-on-Chip, 2007 International Symposium on*, pages 1 –4, nov. 2007.

[34] Martin Palkovic, Francky Catthoor, and Henk Corporaal. In *Proc. of the 4th Workshop on Optimizations for DSP and Embedded Systems*, pages 21–30. IEEE and ACM SIGMICRO, 2006.

[35] Martin Palkovic et al. Systematic preprocessing of data dependent constructs for embedded systems. *Journal of Low Power Electronics, Volume 2, Number 1*, 2006.

[36] P. R. Panda et al. Data and memory optimization techniques for embedded systems. *ACM Trans. Des. Autom. Electron. Syst.*, 6(2):149–206, April 2001.

[37] NL Passes, Edwin HM Sha, and Liang-Fang Chao. Multi-dimensional interleaving for time-and-memory design optimization. In *Computer Design: VLSI in Computers and Processors, 1995. ICCD'95. Proceedings., 1995 IEEE International Conference on*, pages 440–445. IEEE, 1995.

[38] L.N. Pouchet. Polybench: The polyhedral benchmark suite.

[39] Herbert John Ryser. *Combinatorial mathematics.* MAA Washington, 1963.

[40] Herman Schmit and Donald E Thomas. Synthesis of application-specific memory designs. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 5(1):101–111, 1997.

[41] T. Simunic, L. Benini, and G. De Micheli. Cycle-accurate simulation of energy consumption in embedded systems. In *Design Automation Conference, 1999. Proceedings. 36th*, pages 867 –872, 1999.

Appendix D

Integrated Exploration Methodology for Data Interleaving and Data-to-Memory Mapping on SIMD architectures

Iason Filippopoulos, Namita Sharma, Francky Catthoor,
Per Gunnar Kjeldsberg and Preeti Ranjan Panda
ACM Transactions on Embedded Computing Systems
Submitted in 2015

Abstract

This work presents a methodology for efficient exploration of data interleaving and data-to-memory mapping options for SIMD (Single Instruction Multiple Data) platform architectures. The system architecture consists of a reconfigurable clustered scratch-pad memory and a SIMD functional unit, which performs the same operation on multiple input data in parallel. The memory accesses contribute substantially to the overall energy consumption of an embedded system executing a data intensive task. The scope of this work is the reduction of the overall energy consumption by increasing the utilization of the functional units and decreasing the number of memory accesses. The presented methodology is tested using a number of benchmark applications with irregularities in their access scheme. Potential gains are calculated based on the energy models both for the processing and the memory part of the system. The reduction in energy consumption after efficient interleaving and mapping of data is between 40% and 80% for the complete system and the studied benchmarks.

D.1 Introduction

Energy is an important limiting factor for modern embedded systems. New novel hardware solutions have been proposed to reduce the energy consumption. On the processing side, SIMD architectures offer new possibilities for potential improvements on the performance and the energy consumption. On the memory side, modern memory architectures provide different energy modes and clustered scratch-pad memory banks that can operate independently, offering more options for reducing energy consumption. Dynamic and data intensive algorithms are implemented on embedded systems. Data intensive applications have often irregular memory accesses, meaning that the data in memory are not always accessed in order. The lack of spatial locality and the presence of redundant data results in lower utilization of the hardware resources, given that a conventional approach is employed for the handling of data. This problem motivates the development of a methodology to improve the management of the data.

In order to tackle this problem we propose an interleaving exploration that aims to increase spatial locality and reduce the memory accesses on redundant data. The goal of this work is to improve the energy consumption for data intensive applications without any loss on the application's performance. We focus on single instruction multiple data (SIMD) architectures and explore applications that have irregularities in their access scheme. SIMD architectures can potentially increase the performance of an application, providing that the utilization of them is high. However, applications with irregular access patterns do not provide compact sequences of data that are suitable for high utilization. Hence the performance is lower than expected and the number of memory accesses is high due to the accessing of redundant data. In order to reduce the number of memory accesses and achieve higher utilization of the system architecture a systematic exploration of the interleaving options and memory mapping for an application's data is needed.

A large number of papers have demonstrated the importance of the memory organization to the overall system energy consumption. As shown in [11] memory contributes around 40% to the overall power consumption in general purpose systems. Especially for embedded systems, the memory subsystem accounts for up to 50% of the overall energy consumption [8] and the cycle-accurate simulator presented in [38] estimates that the energy expenditures in the memory subsystem range from 35% up to 65% for different architectures. The breakdown of power consumption for a recently implemented embedded system presented in [14] shows that the memory subsystem consumes more than 40% of the leakage power on the platform. According to [24], conventional allocation and assignment of data done by regular compilers is suboptimal. Performance loss is caused by stalls for fetching data and data conflicts for different tasks, due to the limited size of memory and the competition between tasks.

The energy consumption can be divided into two parts, namely the processing and the memory subsystem consumption. The energy needed for processing depends mainly on the utilization of the FUs and any potential stalls, if the memory cannot provide data at the needed rate. The interleaving exploration can increase the utilization of the processing subsystem and reduce time penalties for data loading. The energy consumption in the memory subsystem is affected by the number of memory accesses and the energy per access. Again, the memory architecture and the data-to-memory mapping decisions have great impact on both the number of accesses and the energy per access.

This article is organized as follows. Section D.2 motivates the study of developing a methodology for optimization of the data interleaving exploration and the data-to-memory mapping. Section D.3 surveys related work on both the interleaving and memory mapping exploration. Section D.5 presents the general work-flow and the sequence of the methodology steps. In Section D.4 the target platform is described accompanied by a detailed description of the employed SIMD architecture and the memory models, while the set of benchmarks and their characteristics are analyzed in Section D.6. Results of applying the described exploration methodology to the targeted applications are shown in Section D.7, while conclusions are drawn in Section D.8.

D.2 Motivational Example

Many different techniques have been proposed already to deal with the memory accesses and the potential memory bottlenecks. Authors in [26] propose a low power cache memory for embedded systems that can optimize memory accesses based on application's requirements. The goal in [13] is to effectively utilize the scratch-pad memory of an embedded system in order to improve the memory accesses. A recent survey of the developed techniques for improving cache power efficiency is presented in [30]. Reconfigurable hardware for embedded systems, including the memory architecture, is a topic of active research. An extensive overview of current approaches is found in [10]. The current work is complementary to the existing ones and focuses on the areas that has not been explored in detail as discussed in D.3.

The approach presented in this paper differentiates by exploring both the data transformations and data-to-memory assignment aspects in the presence of a platform with dynamically configurable memory blocks and bus widths. Moreover, many methods for source code transformations, and especially loop transformations, have been proposed in the memory management context. These methods are fully complementary to our focus on data-to-memory assignment and should be performed prior to our step. The contribution of the proposed work is the presentation of a combined approach that investigates the interleaving and memory mapping options for a reconfigurable SIMD architecture.

To illustrate the sub-optimal utilization of SIMD architectures using conventional allocation and assignment of data, the simple example of Alg. 4 is used. In this example, we assume that the desired result is always the sum of 4 elements from arrays $A,\ B,\ C\ and\ D$. The access pattern shows an irregularity, as a result of the iteration index. For every group of four sequential array elements, there is only one used for the calculation of the result and the other three are skipped. An intuitive interleaving optimization is the interleaving of the arrays $A,\ B,\ C\ and\ D$, in order to generate sequences of elements that are all useful on the calculation of the *result* variable. A full interleaving exploration could reveal several options to produce larger sequences of array elements that are needed during the execution of Alg. 4. For example, the interleaving within the combined array $(A|B|C|D)$ can result in a sequence of 8 accessed elements. The sequence of 8 elements is achieved by combining every forth line of the combined array, i.e. line 0 and line 4 are placed next to each other resulting in a sequence of 8 useful elements.

Algorithm 4 Motivational Example Algorithm

```
1: $N \leftarrow 100$
2: for $i = 0 \rightarrow N$ do
3:     $result(i) \leftarrow A(i) + B(i) + C(i) + D(i)$
4:     $i \leftarrow i + 4$
5: end for
```

The initial data representation for the arrays $A,\ B,\ C\ and\ D$ is shown in Fig.D.1(a). The array elements that are accessed during the execution of the motivational example are marked with colors, in contrast to the elements that are not accessed. The array elements are drawn in 25 lines with 4 elements on each line only for a better visual representation. At this point there is no mapping to the memory architecture and no constraint regarding the number of lines or their width in the memory design. The interleaving of the arrays *A, B, C and D* is shown in Fig.D.1(c). Each line of the new interleaved array contains one elements from the initials arrays. For example, the first line consists of the elements *A(0), B(0), C(0) and D(0)*. The result of the interleaving is the construction of blocks consisting of 4 colored elements followed by blocks with 12 redundant elements.

Let us now consider possible data-to-memory mappings for the initial and interleaved array elements. We assume here that he memory architecture either consists of one single memory large enough to fit all four arrays or of four memory banks with a combined memory size large enough to fit the four arrays. The conventional approach with the initial set of data maps each array after each other in the single memory or maps each array in a separate bank. A possible data-to-memory mapping for the initial set of data using four banks is presented in Fig. D.1(b). The data-to-memory mapping for the constructed interleaved array is presented in Fig. D.1(d). The array is split into four parts and stored in the four different banks using the same memory architecture. Each of the lines of the interleaved array is stored in the bank corresponding to modulo 4 of the array line. For example, line x is stored in bank x mod 4. As a result only one forth of array A can be found in the first bank in contrast to the non-interleaving case, in which the whole array A is mapped in the first bank.

A quick estimation for the difference in the number of memory accesses between the two approaches presented in Fig. D.1 can be made. The target architecture for the current work consists of a reconfigurable clustered memory architecture, a processing element that supports SIMD FUs and a wide bus between them. The architecture is presented in detail in Sec. D.4. The motivational example uses a system architecture with an SIMD ADD FU that performs operations over 4 array elements at a time and a memory to processor path that has a width of 4 elements. Each array element is assumed to have the size of one word. The register file at the processor level can only store the iteration variable i and 5 elements, which are the variables *result, A, B, C and D*. Without interleaving of data the number of memory accesses for each loop iteration is 4, because the elements *A(i), B(i), C(i) and D(i)* are stored in different lines and different memory banks

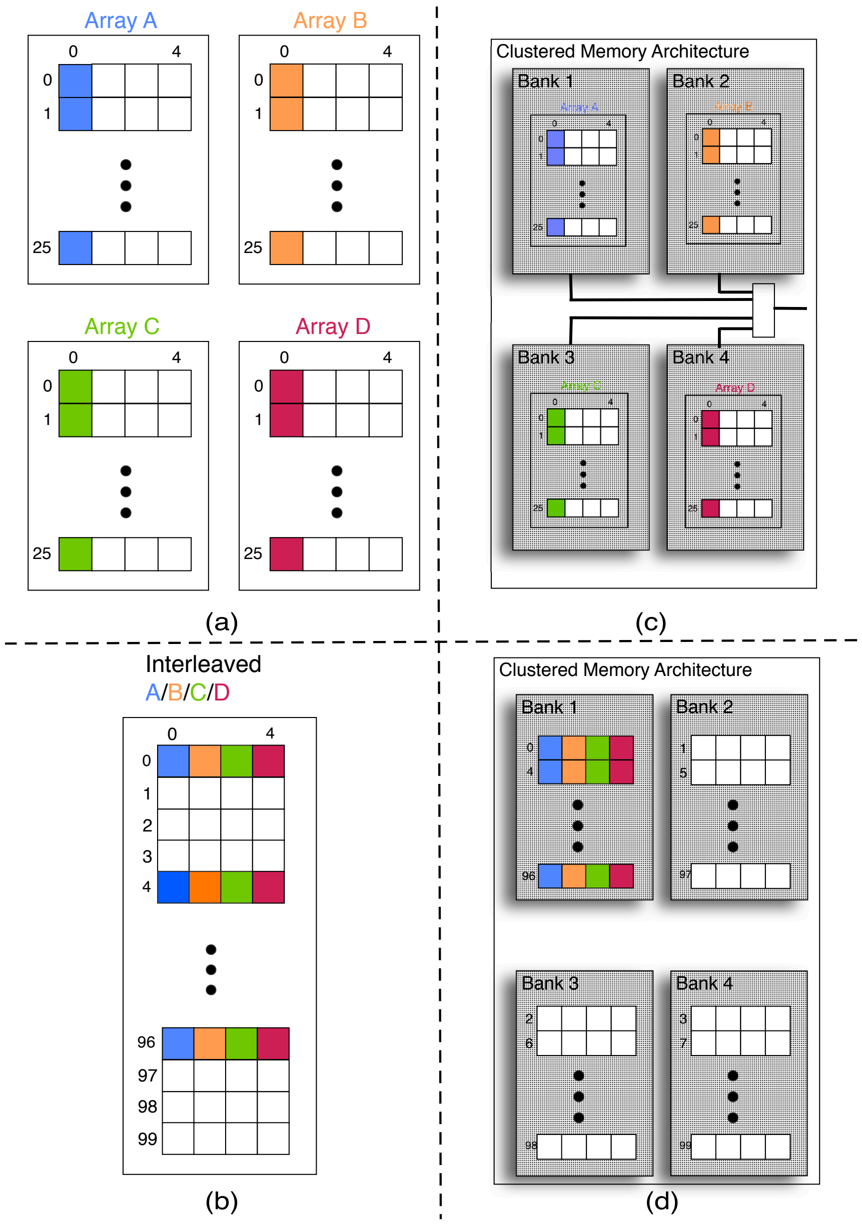


Figure D.1: Motivational Example. Representation of the initial set of data (a) and the interleaved set of data (b). Data-to-memory mapping for the initial (c) and the interleaved data (d) on a clustered scratch-pad memory architecture.

or one larger bank. The overall number of memory accesses is:

$$\textit{25 loop iterations} \times \textit{4 accesses per iterations} = \textit{100 memory accesses.} \tag{D.1}$$

After performing the data interleaving exploration there is only one memory access for each loop iteration, because the elements *A(i), B(i), C(i) and D(i)* are stored in one memory line in the same bank. The overall number of memory accesses is:

$$\textit{25 loop iterations} \times \textit{1 accesses per iteration} = \textit{25 memory accesses.} \tag{D.2}$$

A quick estimation for the difference in the energy consumption between the approaches presented in Fig. D.1 can be calculated using a simple energy model for the memory banks. We assume that the energy cost of accessing a memory bank the size of one array is 1E, while the average leakage energy in the time needed for the execution of one loop iteration is 0.3E. The corresponding access and leakage numbers for a four times larger memory that can fit all the data together is 3E and 1.2E, respectively. The numbers reflects the fact that as a first order approximation access energy increases sub-linearly with increased memory size, while leakage increases linearly with memory size. The energy consumption for a large memory with non-interleaved data is:

$$100 \times 3E + 100\times\ 1.2E = 420E. \tag{D.3}$$

In this case there is a performance loss and an increased leakage energy, because all the accesses has to be done sequentially on the same memory. The energy consumption for a four bank memory architecture with non-interleaved data is:

$$100 \times 1E + 4\times\ 100\times\ 0.3E = 220E. \tag{D.4}$$

The energy consumption for a large memory with interleaved data is:

$$25 \times 3E + 25\times\ 1.2E = 105E. \tag{D.5}$$

Finally, the energy consumption for a four bank memory architecture with interleaved data is:

$$25 \times 1E + 1\times\ 25\times\ 0.3E = 32.5E. \tag{D.6}$$

It should be noted that the leakage energy in the last case is considered only for the one bank that is always accesses. The memory banks that store redundant data are assumed switched to retention mode to minimize leakage. The simple estimations presented in equations (1)-(6) shows that a pure interleaving optimization results in a 75% reduction in the number of accesses. A memory optimization based on an energy efficient four bank clustered memory architecture can also give an important energy reduction. The energy gains are even greater when interleaved data are mapped to an efficient memory architecture. The observations for this simple example motivates the further exploration of the data interleaving and the data-to-memory mapping for applications with irregularities in their accesses.

D.3 Related work

Many studies have been published in the area of data layout transformations and memory management. This work differentiates by presenting an integrated approach that combines the two areas. The combination is performed in a systematic way and the final result is better than the sum of the two individual optimizations. The data transformations on the application level are studied in combination with the data-to-memory mapping on the hardware level. An overview of the related work on those two areas is presented.

Data layout optimizations aim to arrange data in memory with the objective of improving system performance and energy through means such as reduced memory access count or reduced cache misses. Several generic data layout techniques have been explored by researchers at various levels in the memory hierarchy. In [27], cache partitioning, a layout technique for arrays, maps each array to different cache partitions to reduce conflicts. In [21] authors address the problem for cache miss reduction by evaluating the tiling size for arrays and merging the arrays appropriately for each loop nest. Memory Hierarchy Layer Assignment (MHLA), an optimization methodology for data memory hierarchy presented in [4], determines for each data set an appropriate layer in the hierarchy and type of memory (single/dual port) taking data re-use into account. The strategy in [33] partitions the variables to scratch-pad memory and DRAM to minimize interference between variables. The cache-oriented techniques proposed earlier by researchers in [34] do not find a straightforward application in our context, where the target system hardware consists of a scratch-pad memory and a SIMD functional unit. The current work explores the assignment of data to the memory and the effect of different interleaving decisions on the overall energy consumption.

Several techniques for designing energy efficient memory architectures for embedded systems are presented in [25]. The current work differentiates by employing a platform that is reconfigurable at run-time. In [32] a large number of data and memory optimisation techniques, that could be dependent or independent of a target platform, are discussed. Again, reconfigurable platforms are not considered. The authors in [3] present a methodology to generate a static application-specific memory hierarchy. Later, they extend their work in [2] to a reconfigurable platform with multiple memory banks.

The authors in [1], [15] and [23] present methodologies for designing memory hierarchies. Design methods with main focus on the traffic and latencies in the memory architecture are presented in [7], [12], [16] and [35]. Improving memory energy efficiency based on a study of access patterns is discussed in [18]. Application specific memory design is a research topic in [36], while memory design for multimedia applications is presented in [31]. The current work differentiates by presenting both a reconfigurable platform and the necessary data interleaving to better exploit the platform.

The current work combines and expands in a non-trivial way, the interleaving exploration presented in [37] and the data to memory mapping methodology presented in [9]. In [37] a memory dominant application is chosen, which normally suffers from data memory organization bottlenecks while implemented on conventional architectures. A data interleaving approach is employed in order to improve memory energy by reducing memory accesses. Significant reduction in data memory energy consumption can be achieved if array data are interleaved, with no performance overhead. In [9] a memory-aware system scenario methodology is presented. The variations in memory needs during the lifetime of an application are studied in order to optimize energy usage. Different system scenarios capture the application's different resource requirements that change dynamically at run-time. Many possible memory platform configurations and data-to-memory assignments are studied as system scenario parameters, which lead to a large exploration space.

D.4 Target Architecture and Energy Models

Selection of target platform is an important aspect of the exploration. The motivational example in the previous section shows that the available choices on the memory models have a great impact on the overall energy consumption. The key feature needed in the platform architecture is the ability to efficiently support different memory sizes that are suitable for different data structures. A generic architecture is presented in Fig.D.2. The scratch-pad memory consists of up to four memory banks. For more complex architectures the interconnection cost should be considered and analyzed separately for accurate results. Although power gating can be applied to the bus when only a part of a longer bus is needed, an accurate model of the memory wrapper and interconnection must developed, which is beyond the scope of the current work. The SIMD vector FU is assumed to perform instructions on multiple data.

The methodology explores different interleaving and data to memory mapping options for the reconfigurable architecture. The optimal sizes are found based on the sizes of the data after the interleaving exploration. The memory banks can operate independently and have different sizes. The explored data lengths for the vector FU and the connections are 4, 8 and 16 elements. The exploration is based on the available system models and results in the final system architecture. Thus, the accuracy of the models influence the decisions taken during the design phase.

D.4.1 Memory Models

The dynamic memory organization is constructed using commercially available SRAM memory models (MM). For those models delay and energy numbers are derived from a commercial memory compiler. In addition, experimental standard cell-based memories (SCMEM) [29] are considered for smaller memories due to their energy and area efficiency for reasonably small storage capacities, as argued in [28]. The standard cell-based memories are synthesized using Cadence RTL compiler for TSMC 40nm standard library. Afterwards, power simulations on the synthesized design are carried out using Synopsys PrimeTime, in order to obtain energy numbers. Both MMs and SCMEMs can operate under a wide range of supply voltages, thus support different operating modes that provide an important exploration space.

- Active mode: The normal operation mode, in which the memory can be accessed at the maximum supported speed. The supply voltage is 1.1V. The dynamic and leakage power are higher compared to the other modes. Only in active mode the data are accessible without

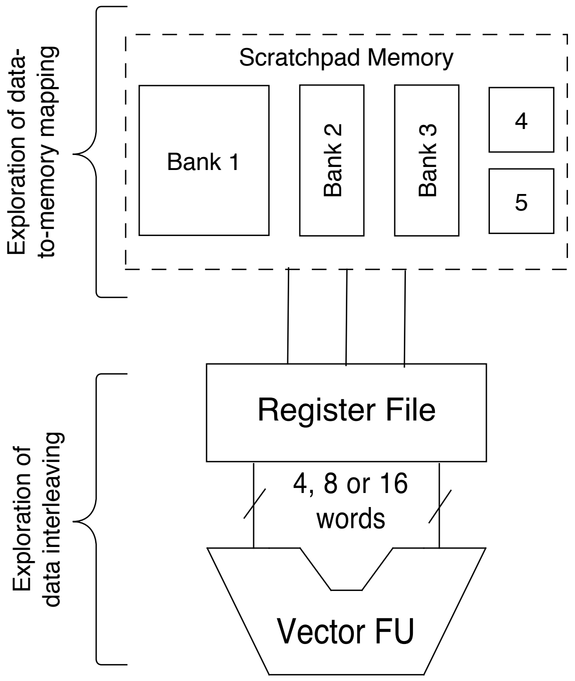


Figure D.2: Exploration options and system knobs depending on a general platform architecture

time penalties, in contrast to light and deep sleep modes. In this work all the memory accesses are performed in active mode.

- Light sleep mode: The supply voltage in this mode is lower than active with values around 0.7V. The access time of the memory is significantly higher than the access time in active mode. Switching to active mode can be performed with a negligible energy penalty and a small time penalty of a few clock cycles (less than 10). Data is retained.
- Deep sleep mode: The supply voltage is set to the lowest possible value that can be used without loss of data. This voltage threshold is expected to be lower for SCMEMs than MM models and can be as low as 0.3V. The number of clock cycles needed for switching to active mode is higher compared to sleep mode, typically in the range of 20 to 50 clock cycles depending on the clock speed. Consequently, the speed of the PE and the real-time constraints of the applications have to be taken into consideration when choosing light or deep sleep mode at a specific time.
- Shut down mode: Power-gating techniques are used to achieve near zero leakage power. Stored data is lost. The switch to active mode requires substantially more energy and time. However, switching unused memories to this mode, providing that their data are not needed in the future, results in substantial energy savings.

The necessary energy and power information is available for the memory models. Relative values for a subset of them is presented in Table D.1. It is clearly shown that the choice of memory units has an important impact on the energy consumption.

D.4.2 Functional Unit Models

The processing of the interleaved data is performed in the part of the architecture consisting of the SIMD FU and the central vector register file, as shown in Fig.D.2. For efficient utilization of the vector FU, the register file has a wide interface (256 bit wide) with the scratchpad memory and allows testing for word length of 4, 8 and 16 elements, assuming that each array element is 16 bit wide. This processor belongs to the class of coarse-grain reconfigurable array (CGRA) processors and is described in more detail in

Table D.1: Relative dynamic energy for a range of memories with varying capacity and type

Type	**Lines x wordlength**	**Dynamic Energy [J]**		Switching to Active from	
		Read	Write	Deep[uJ]	Light[uJ]
MM	32 x 8	4.18×10^{-8}	3.24×10^{-8}	0.223	0.031
MM	32 x 16	6.79×10^{-8}	5.89×10^{-8}	0.223	0.031
MM	32 x 128	4.33×10^{-7}	4.31×10^{-7}	1.42	0.168
MM	256 x 128	4.48×10^{-7}	4.60×10^{-7}	1.70	0.171
MM	1024 x 128	5.11×10^{-7}	5.75×10^{-7}	2.81	0.179
MM	4096 x 128	9.60×10^{-7}	4.57×10^{-7}	9.01	0.457
SCMEM	128 x 128	2.5×10^{-7}	0.8×10^{-8}	1.51	0.045
SCMEM	1024 x 8	1.7×10^{-8}	0.6×10^{-8}	0.325	0.021

[22]. The HDL model of the processor is synthesized using Cadence RTL compiler [5] and the energy numbers are extracted using Synopsys Primepower [17].

For efficient utilization of the vector FU, the register file has a wide interface with the clustered scratch-pad memory. Since the target architecture is a configurable processor that can be customized in various ways, the standard evaluation and execution mechanism is to run the programs on a processor simulator. An XML based language is used to describe the architecture, and a cycle-accurate simulator of the processor is used to simulate the generated code on the architecture. The energy consumption information for different types of instructions and different widths is available for the FU models. The width of the FU corresponds to the number of pairs of array elements on which the specific instruction is performed. Relative values for a subset of them is presented in Table D.2. It is clearly shown that the wider FUs have a significantly higher energy consumption compared to FUs that operate on only two element. Thus, it is important to achieve high utilization of wider FUs to achieve both performance and energy gains.

D.5 System Design Exploration Work-flow

As motivated in the previous sections a systematic exploration of the interleaving options and the data-to-memory mapping possibilities is necessary. The input to the work-flow is the application code and the output of the exploration is an efficient solution for the interleaving of the data and a map-

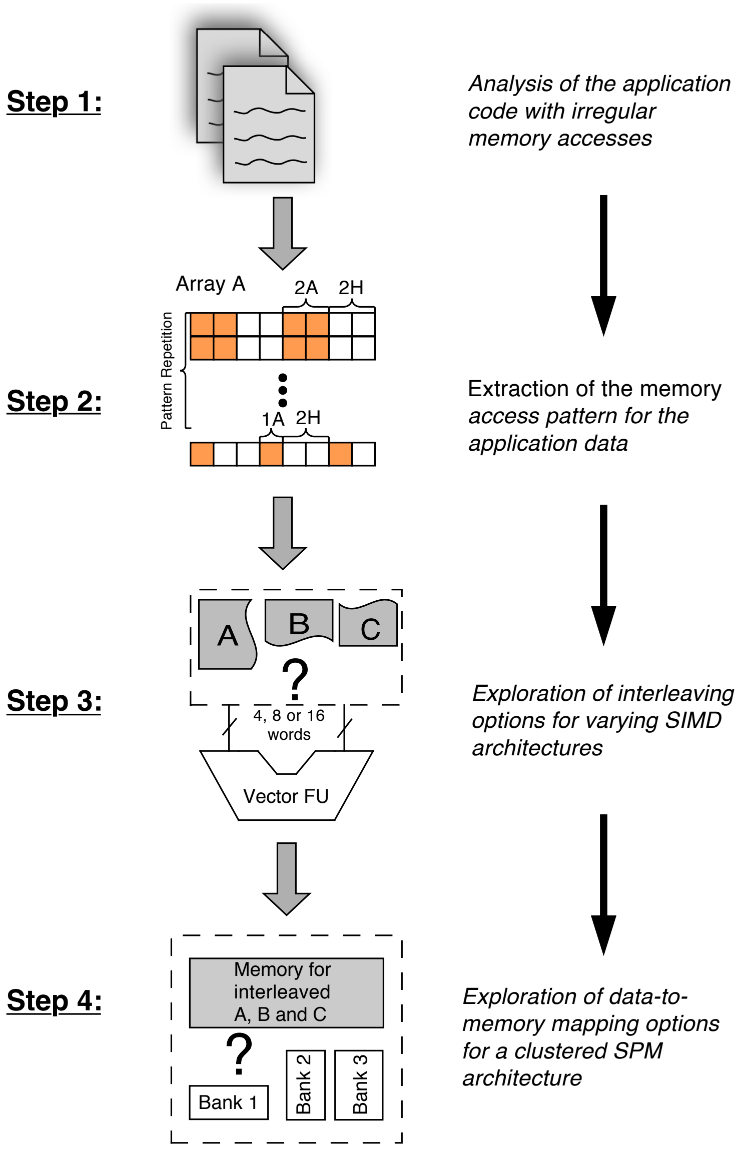

Step 1:
Analysis of the application code with irregular memory accesses
Array A
2A
2H
Pattern Repetition
1A
2H
Step 2:
Extraction of the memory access pattern for the application data
A
B
C
?
4, 8 or 16 words
Vector FU
Step 3:
Exploration of interleaving options for varying SIMD architectures
Memory for interleaved A, B and C
?
Bank 1
Bank 2
Bank 3
Step 4:
Exploration of data-to-memory mapping options for a clustered SPM architecture

Figure D.3: Methodology steps

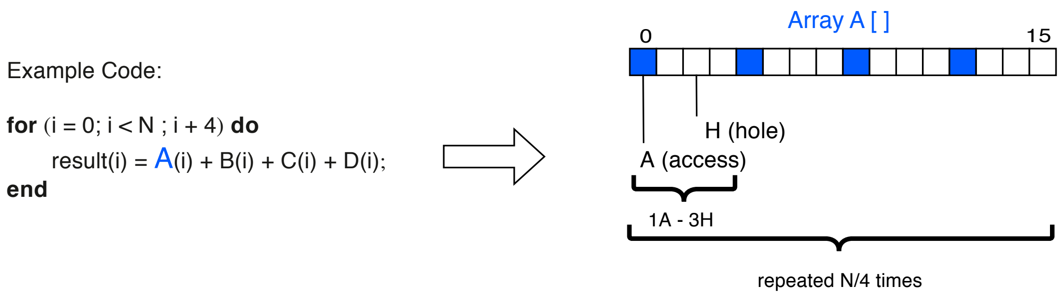
Table D.2: Relative dynamic energy for different FU models

Type of FU instruction	Width	Dynamic Energy [J]
ADD	1	5.70E-08
ADD	4	2.28E-07
MULT	1	2.48E-07
MULT complex	1	5.33E-07
MULT	4	1.03E-06
MULT complex	4	1.95E-06

ping to a reconfigurable system architecture based on the available models presented in Sec.D.4. The exploration space consists of the potential combinations between the different arrays and the different scratch-pad memory architectures, which combine memories of different types and sizes.

The overall work-flow is presented in Fig.D.3. The first step of the methodology is the analysis of the application code. The methodology is applicable to any application. However the applications that can benefit more from the proposed methodology are the ones with irregular access patterns. The second step is the extraction of the access pattern for each data structure present on the application code. The third step is the interleaving exploration, which explores all the different options for the re-arrangement of the data. The aim of this step is the construction of compact sets of data by using defined operations between the available access patterns. The goal is to better utilize the different width of the vector FUs. The result of the interleaving is a set of data with a reduced number of holes. The forth step is the mapping of the interleaved data set to the clustered memory architecture.

Some significant changes are needed in order to combine the different techniques in a functional work-flow. The interleaving and the data-to-memory mapping exploration steps have to be modified and a simple concatenation is not sufficient. The results of the interleaving exploration are given as constraints to the mapping exploration. The constraints stir the exploration in a potentially different solution compared to the result of the data-to-memory mapping exploration without the interleaving constraints. The interleaving exploration finds solutions for the different bus widths and then all the different solutions are propagated to the mapping exploration. The mapping exploration takes all the interleaving constraints into consideration. Thus, the data-to-memory mapping solutions using the constraints are in general different from the solutions provided in a conventional ap-



Example Code:

```
for (i = 0; i < N ; i + 4) do
    result(i) = A(i) + B(i) + C(i) + D(i);
end
```

Figure D.4: Extraction of access pattern from application code

proach. The propagation of the constraints is explained in more detail later.

D.5.1 Formal Model Representation of Access Patterns

A representation model for the memory access patterns is employed in order to formally present each step of the methodology. The model presented in [19] is a generic model suitable for irregular iteration spaces on arrays. The irregularities are created by the application code access statements in a conditional loop structure.

When an array element is accessed during the code execution, it is represented with an A(Access). Otherwise, there is a hole in the access pattern represented with an H(Hole) as shown in Fig.D.4. The sequence of accesses and holes is usually repeated periodically, because normally the loop conditions are not totally random. Thus we can define the frequency of each access pattern. The analysis of the application code results in the access patterns and their corresponding frequencies, which is the necessary input for the next step in the work-flow.

D.5.2 Data Interleaving Exploration

Interleaving is a data layout transformation for combining the storage of multiple arrays, so that blocks of data from different arrays are stored contiguously, in order to achieve spatial locality in memory accesses. By interleaving we are able to group the data to be accessed and thus reduce the number of memory accesses for accessing them. The basic principles of the performed interleaving exploration are presented in [37].

The employed model of access pattern representation is accompanied with a suitable algebra presented in [20]. The operations defined help the designer explore different interleaving options. The goal of the process is to rearrange the data in an efficient way to increase the number of sequential accesses. A simple example is presented in Fig.D.5. Arrays A and B are interleaved by interchangeably storing an element of each array in the new array. The access patterns of arrays A and B are combined in a new access pattern, which has two accessed elements placed consecutively. The use of access patterns and the theory for calculating the access patterns of different combinations enables an extensive exploration of the possible data interleaving options.

The impact of the data interleaving exploration on the number of memory accesses is significant. When the accesses are irregular and the data are organized in index order, each memory access results in a small amount of useful data due to the presence of holes. In contrast the re-organization of the data provides a sequence of useful data without many holes between them. Thus, a single access to the memory results in a higher number of useful elements. The overall number of memory accesses is reduced, as each access has a higher utilization.

This idea is illustrated in Fig.D.5. The number of memory accesses for each case can be calculated by assuming that each memory access loads four array elements, i.e. the word length for the memory and bus architecture is four elements. Each time an element from the array A or B is needed, the memory access returns four elements, from which one is the useful and the other three are not used by the running code. In the case of the interleaved array, each memory access returns four elements, from which two are useful and two redundant. Thus, the overall number of memory access in the second case is reduced by half compared to the first case.

D.5.3 Data-to-Memory Mapping Exploration

Given the input from the previous step, we explore the mapping of the interleaved data to the memory architecture. A clustered memory organisation with up to five memory banks of varying sizes is explored. The limitation in the number of memory banks is necessary in order to keep the interconnection cost between the processing element (PE) and the memo-

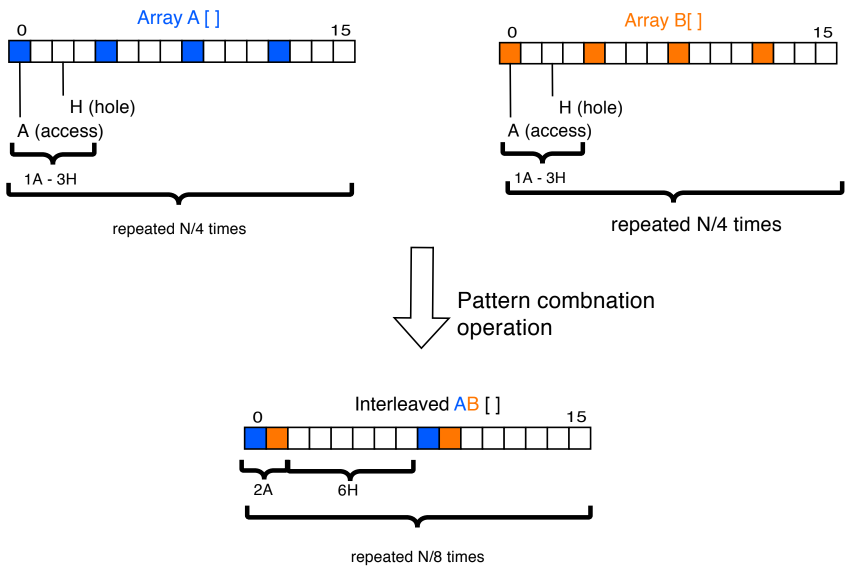


Figure D.5: Example of combination between two arrays and their access patterns

ries constant through exploration of different architectures. The decision to use memory banks with varying sizes on the clustered memory organization increases the reconfiguration options and consequently the potential energy gains. In general, smaller memories are more energy efficient compared to larger memories banks. However, in some cases large memory banks are needed in order to fit the application data without the need for too many small memories causing complex interconnects. The goal is to use the most energy efficient banks to store the interleaved data.

The exploration space consists of different sizes and types of memory banks. The goal is to select the optimal set of memory banks and to make the optimal decision regarding the mapping of the data to the different memory banks. The parts of the interleaved data that consist mostly of useful elements are mapped into memory banks with low energy per access but at the same time with the necessary access time. The parts of the interleaved data that consist of access holes and rarely accessed elements are optimally mapped into memory banks with energy efficient retention states. In both cases the size of the memory banks should be adequate to fit the stored data but at the same time as small as possible to avoid area and energy penalties.

D.5.4 One way constraint propagation

The interleaving decisions influence the data-to-memory mapping decisions and vice versa. For example, the decision to interleave two arrays A *and* B have an impact on the freedom of mapping the interleaved array $A|B$ on the memory banks, because of the difference in size and structure. The data-to-memory mapping decisions affect the interleaving decisions in a similar way. Assuming that the mapping is performed first using the initial data the following interleaving options are reduced. For example, the decision to map two arrays on different memory banks removes the option to interleave them later. Optimizing both the interleaving and the memory mapping at the same time results in a large and inefficient loop of constrain propagations between the two exploration phases.

We choose to perform the interleaving exploration step first and then propagate the most efficient interleaving options to the data-to-memory mapping step. The code and data transformations should be performed earlier to avoid omitting possible solutions as justified in [6]. Thus, the interleaving decisions are propagated as constraints on the mapping exploration phase. The constrains consist of the arrays that are interleaved, the new access pattern and the width of SIMD architecture for which the specific interleaving solution is optimal. The mapping step is performed based on these constrains.

D.6 Applications

The applications that benefit most from the proposed methodology are characterized by having irregular access patterns with holes. Applications with irregular accesses can be found on several application domains. The current study includes representative candidates from three different domains, namely the multimedia, the wireless and the equation solver domains. Suitable applications can also be found on other areas. An overview of the tested benchmark applications is presented below:

1. Motivational example is the very simple example presented in Sec. D.2. It uses four arrays that all have the same irregular access pattern, namely 1A-3H. One element from each array is used to calculate an intermediate result on every loop iteration. The interleaving is easy because the four arrays have the same pattern. Further interleaving

within the array can result in even longer sequences of useful elements. The access pattern is repeated for every four elements (1A-3H), so the scaling for the different word lengths (4, 8 and 16) is expected to be good.

2. Successive Over Relaxation (SOR) is a method for evaluating partial differential equations or solving a linear system of equations. The SOR benchmark has a more irregular access pattern. The interleaving exploration provides sequences of three or six sequential useful elements. Thus, the utilization on the SIMD architecture is expected to be lower. This application is an representative example from the equation solver domain.

3. FFT benchmark has an irregular pattern during the access of the pilot matrices. The interleaving exploration for FFT is presented in [37]. The number of matrices is higher and more interleaving options are present. However, the interleaving cannot provide an acceptable solution for 16 sequential useful elements. This application is an representative example from the wireless domain, because FFT is widely used on wireless applications.

4. Motion estimation benchmark is a dynamic algorithm that results in different access patterns based on the identification of the moving objects. The static parts are not accessed resulting in holes in the accesses and the interleaving aims to minimize those parts. The interleaving exploration provides alternatives for all the tested word lengths. This application is an representative example from the multimedia domain.

D.7 Results

The design exploration is applied to the chosen application benchmarks and energy numbers are derived based on the described target platform. The energy numbers are calculated both for the memory and the SIMD architecture presented in Sec.D.4. Four approaches are explored and the corresponding energy consumption for each of them is calculated.

- *No optimization.* In this case there is no interleaving exploration and the memory architecture consists of one large memory bank. All the data are mapped on the memory bank without any optimization.

- *Memory Size Optimization.* In this case there is no interleaving exploration and the memory architecture consists of five memory banks. The optimal size for the memory banks and the optimal mapping of the non-interleaved data on them is explored. The number of memory accesses is the same as in the previous approach. However, the data is mapped on an efficient clustered memory architecture.

- *Interleaving optimization.* In this case the interleaving exploration step is performed and the memory architecture consists of one large memory bank. The optimal interleaving decision is found and applied to the data, so the locality of useful data is increased. However, all the data are mapped on one large memory bank. The number of accesses is significantly reduced by the interleaving step, but the energy per access is kept high due to lack of data mapping to an efficient clustered architecture.

- *Integrated co-exploration.* In this case the co-exploration of both the interleaving and the data-to-memory mapping optimizations is performed. Both he number of memory accesses and the energy per access are reduced.

D.7.1 Motivational Example

The normalized energy consumption for the motivational example is presented in Fig.D.6. The four different approaches are normalized using the monolithic approach without any optimization. Energy results for this benchmark are presented for a bus width of 4, 8 and 16 elements, which means that every memory access loads/stores 4,8 or 16 elements. The interleaving exploration proposes three different solutions for an SIMD architecture width of 4, 8 and 16 elements respectively. The data access patterns of the three interleaving decisions are propagated as a set of constrains to the mapping step. The mapping step explores all the different data-to-memory mapping options and proposes the most energy efficient memory organization for each of the three interleaved data solutions.

The application code is perfectly suited for interleaving as discussed in Sec.D.2. Thus the interleaving exploration has a greater impact than the memory size optimization. However, the integrated approach optimizes the energy consumption even further. The application is suitable for higher bus widths and there are important gains while moving from 4 to 8 and

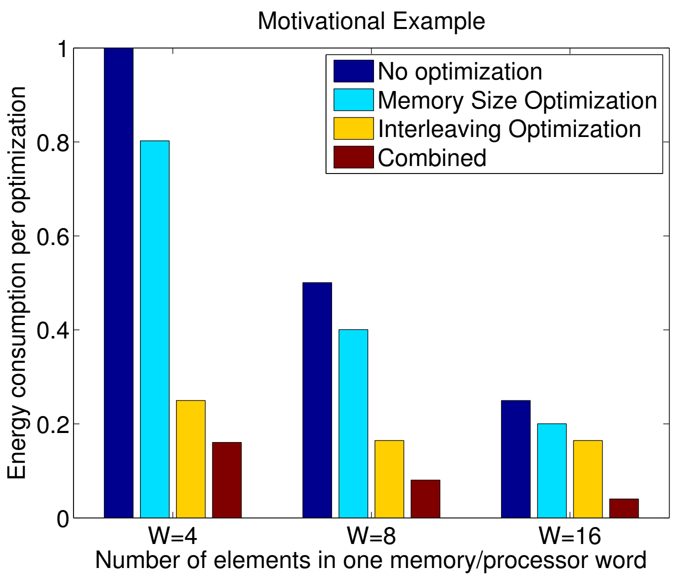


Figure D.6: Motivational Example

16. The increase in the architecture width result in significant gains even without any optimization. The gains are approximately 50 and 70% for a width of 8 and 16 compared to a width of 4. This is explained by the fact that the accessed array elements are close enough even without data transformations and the existence of holes in the access pattern. By fetching 8 and 16 elements from the memory, the number of useful elements is two and four times more. Studying the motivational example in Fig.D.6(a) for a width of 16, each access results in 4 lines with one colored element each.

The interleaving optimization results in greater improvements compared to the memory optimization. This is explained by the nature of the application that offers good interleaving options for larger words, i.e. higher values of bus width. The memory optimization is around 20% lower than the monolithic approach for any width. The interleaving optimization results in more than 70% lower energy for a width of 4 and more than 80% for a width of 8 and 16. The interleaving of the arrays provides perfectly compact sets of data as shown in Fig.D.6 and the interleaving cannot improve more for higher values of width. The great impact of the combined approach is better illustrated for a width of 16. In this case the two optimizations alone report an energy gain around 20% and 35% compared to the non optimized case for width of 16. The combined approach results in an energy gain of

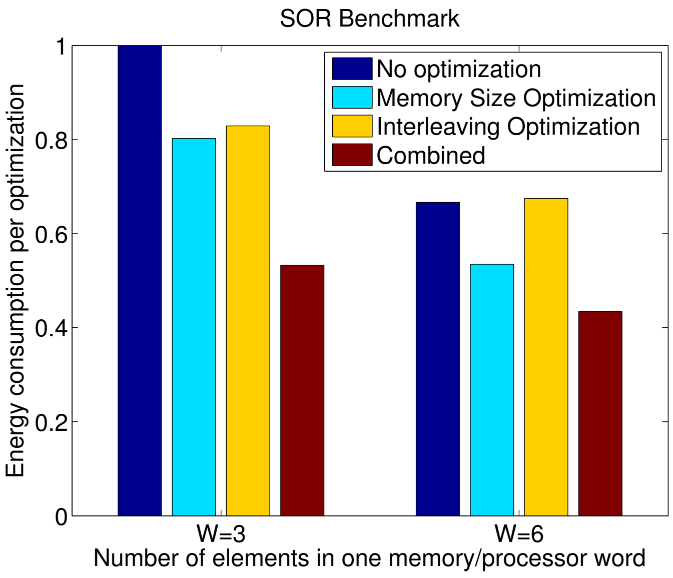


Figure D.7: SOR Benchmark

84% on the same case.

D.7.2 SOR Benchmark

Algorithm 5 Code snippet from the SOR benchmark

```
1: ...
2: $N \leftarrow 100$
3: **for** $j = 0 \rightarrow N$ **do**
4:     $resid \leftarrow c(i)(j) \times u(i)(j-1) + d(i)(j) \times u(i)(j) + e(i)(j) \times u(i)(j+1)$
5:     $j \leftarrow j + 2$
6: **end for**
7: ...
```

The normalized energy consumption for the SOR benchmark is presented in Fig.D.7. The four different approaches are normalized using the energy consumption of a system without any optimization as the base. Energy results for this benchmark are presented for a bus width of 3 and 6, although the architecture supports bus widths of the power of 2. This means that every memory access loads or stores 4 or 8 elements, but only up to 3 or 6 array elements are used in the algorithm. This limitation is due to the nature of the application code that adds elements from 3 arrays to 3 sequential elements of another array. Alg. 5 is a code snippet that demonstrates the considered data structures. The arrays *c, d and e* are potential candidates for the interleaving exploration. The interleaving exploration on the application code concludes that it is only possible to make sequential sets of 3 and 6 elements by interleaving the arrays *c, d and e*. The elements that are multiplied with them are already sequential.

The interleaving exploration has a small impact on the reduction of energy consumption for a bus width of 3 elements. This is due to the addition of a redundant element, in order to have a width of four that is supported by the architecture. The results are even worse for a bus width of 6 elements, because there is a need for two redundant elements to comply with a set width of eight. The mapping of the initial data to a clustered memory results in a reduction of around 20%, which is slightly better than the interleaving optimization. The reason is that the mapping of the initial data demands all the memory banks active and the memory is heavily accessed during the execution of the benchmark. The integrated approach exploit on a better way the few interleaving options and provides a better mapping of the interleaved data to the memory architecture. The final gain for a width of three is 47%, compared to 17% and 20% for the interleaving and the memory optimization respectively.

D.7.3 FFT Benchmark

The normalized energy consumption for the FFT benchmark is presented in Fig.D.8. The four different approaches are normalized using the monolithic approach without any optimization. Energy results for this benchmark are presented for a bus width of 4 and 8 elements, which are the two viable options provided by the interleaving exploration. Again, the integrated approach results in the lower energy consumption. The energy gains for a bus width of 8 are significant, because blocks of 8 elements can be constructed by interleaving of application's arrays.

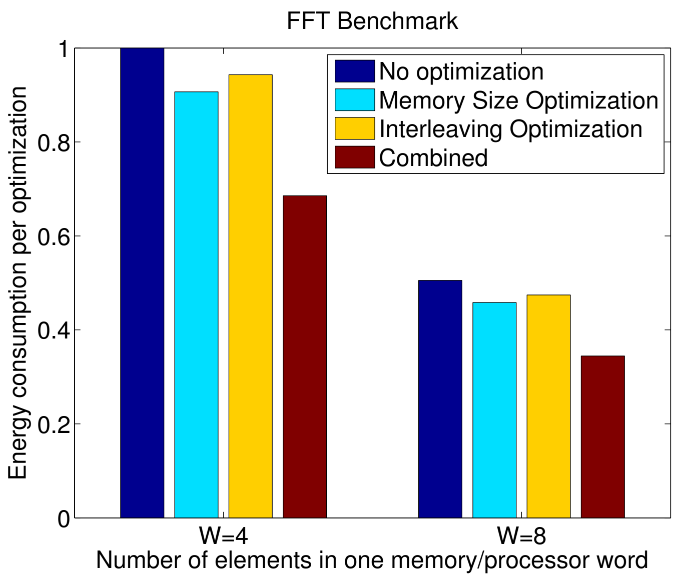


Figure D.8: FFT Benchmark

D.7.4 Motion Estimation Benchmark

The normalized energy consumption for the motion estimation benchmark is presented in Fig.D.9. The four different approaches are normalized using the monolithic approach without any optimization. Energy results for this benchmark are presented for a bus width of 4, 8 and 16 elements. The application code provides good possibilities for interleaving and consequently the interleaving exploration has a greater impact than the memory size optimization. However, the integrated approach optimizes the energy consumption even further. In this case the energy gains for increasing size of bus width are minimal when only one optimization is applied. This is explained by the nature of the application, which has large parts of redundant data spread out through the memory. Thus, the whole memory is active when the mapping is not performed on the compact interleaved data. The combined approach offers significant energy gains even for the highest bus width.

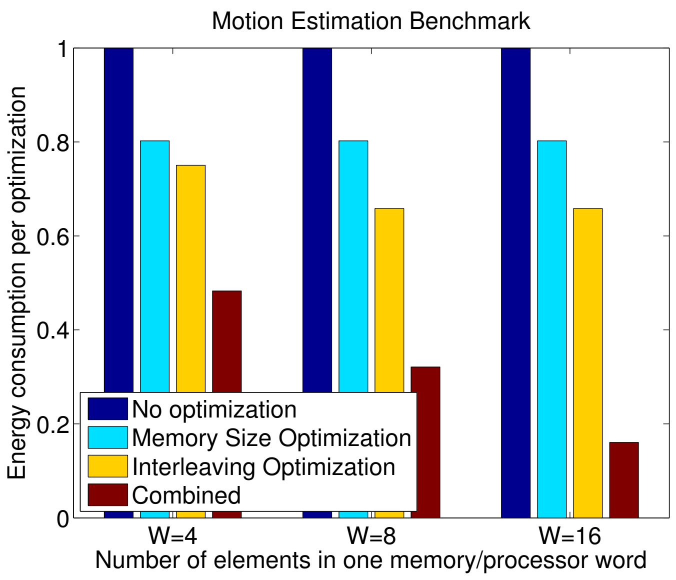


Figure D.9: Motion Estimation Benchmark

D.8 Conclusion

The scope of this work is to presents a methodology for efficient exploration of data interleaving and data-to-memory mapping options for SIMD (Single Instruction Multiple Data) platform architectures. Detailed energy models are presented for the studied architecture. The methodology focuses on reducing the overall energy consumption by reducing the number of memory accesses and the energy per access. The number of memory accesses is reduced by interleaving the application data to construct compact sets of sequential data. The energy per memory access is reduced by employing a reconfigurable scratch-pad memory architecture with multiple banks that can operate independently. A systematic way is presented in order to explore the different options that lead to the interleaving and data-to-memory mapping decisions. A wide range of applications is studied that allow us to draw conclusions about different kinds of dynamic behavior and their effect on the energy gains achieved using the methodology. The improvement in the total system energy consumption after efficient interleaving and mapping of data is between 40% and 80% for the studied benchmarks having the type of irregularities in their access scheme that benefit most from the methodology.

Bibliography

[1] Santosh G Abraham and Scott A Mahlke. Automatic and efficient evaluation of memory hierarchies for embedded systems. In *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on*, pages 114–125. IEEE, 1999.

[2] Luca Benini, Alberto Macii, Enrico Macii, and Massimo Poncino. Increasing energy efficiency of embedded systems by application-specific memory hierarchy generation. *IEEE Design & Test of Computers*, (2):74–85, 2000.

[3] Luca Benini, Alberto Macii, and Massimo Poncino. A recursive algorithm for low-power memory partitioning. In *Low Power Electronics and Design, 2000. ISLPED'00. Proceedings of the 2000 International Symposium on*, pages 78–83. IEEE, 2000.

[4] Erik Brockmeyer, Bart Durinck, Henk Corporaal, and Francky Catthoor. Layer assignment techniques for low energy in multi-layered memory organizations. In *Designing Embedded Processors*, pages 157–190. Springer, 2007.

[5] RTL Cadence. Compiler users manual, 2014.

[6] Francky Catthoor, Sven Wuytack, G.E. de Greef, Florin Banica, Lode Nachtergaele, and Arnout Vandecappelle. *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design.* Springer, 1998.

[7] Fei Chen and Edwin Hsing-Mean Sha. Loop scheduling and partitions for hiding memory latencies. In *Proceedings of the 12th international symposium on System synthesis*, page 64. IEEE Computer Society, 1999.

[8] Eric Cheung, Harry Hsieh, and Felice Balarin. Memory subsystem simulation in software tlm/t models. In *Design Automation Conference, 2009. ASP-DAC 2009. Asia and South Pacific*, pages 811–816. IEEE, 2009.

[9] Iason Filippopoulos, Francky Catthoor, and Per Gunnar Kjeldsberg. Exploration of energy efficient memory organisations for dynamic multimedia applications using system scenarios. *Design Automation for Embedded Systems*, pages 1–24, 2013.

[10] Philip Garcia, Katherine Compton, Michael Schulte, Emily Blem, and Wenyin Fu. An overview of reconfigurable hardware in embedded systems. *EURASIP J. Embedded Syst.*, 2006(1):13–13, January 2006.

[11] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *Solid-State Circuits, IEEE Journal of*, 31(9):1277 – 1284, sep 1996.

[12] Peter Grun, Nikil Dutt, and Alex Nicolau. Mist: An algorithm for memory miss traffic management. In *Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design*, pages 431–438. IEEE Press, 2000.

[13] Yibo Guo, Qingfeng Zhuge, Jingtong Hu, Juan Yi, Meikang Qiu, and Edwin HM Sha. Data placement and duplication for embedded multicore systems with scratch pad memory. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 32(6):809–817, 2013.

[14] J. Hulzink, M. Konijnenburg, M. Ashouei, A. Breeschoten, T. Berset, J. Huisken, J. Stuyt, H. de Groot, F. Barat, J. David, et al. An ultra low energy biomedical signal processing system operating at near-threshold. *Biomedical Circuits and Systems, IEEE Transactions on*, 5(6):546–554, 2011.

[15] Bruce L Jacob, Peter M Chen, Seth R Silverman, and Trevor N Mudge. An analytical model for designing memory hierarchies. *Computers, IEEE Transactions on*, 45(10):1180–1194, 1996.

[16] Axel Jantsch, Peeter Ellervee, Ahmed Hemani, Johnny Öberg, and Hannu Tenhunen. Hardware/software partitioning and minimizing

memory interface traffic. In *Proceedings of the conference on European design automation*, pages 226–231. IEEE Computer Society Press, 1994.

[17] Wang Kai and Xu Zhiwei. Synopsys prime power manual release u-2003.06-qa, 2003.

[18] Mahmut Kandemir, Ugur Sezer, and Victor Delaluz. Improving memory energy using access pattern classification. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 201–206. IEEE Press, 2001.

[19] A Kritikakou, F Catthoor, V Kelefouras, and C Goutis. A scalable and near-optimal representation for storage size management. *ACM transaction architecture and code optimization*, 11(1):1–25, 2014.

[20] Angeliki Stavros Kritikakou. *Development of methodologies for memory management and design space exploration of SW/HW computer architectures for designing embedded systems.* PhD thesis, Department of Electrical and Computer Engineering School of Engineering, University of Patras, 2013.

[21] Chidamber Kulkarni, C Ghez, Miguel Miranda, Francky Catthoor, and Hugo De Man. Cache conscious data layout organization for conflict miss reduction in embedded multimedia applications. *Computers, IEEE Transactions on*, 54(1):76–81, 2005.

[22] Jongeun Lee, Kiyoung Choi, and Nikil D Dutt. Compilation approach for coarse-grained reconfigurable architectures. 2003.

[23] Yanbing Li and Wayne H Wolf. Hardware/software co-synthesis with memory hierarchies. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 18(10):1405–1417, 1999.

[24] Zhe Ma, Pol Marchal, Daniele Paolo Scarpazza, Peng Yang, Chun Wong, José Ignacio Gómez, Stefaan Himpe, Chantal Ykman-Couvreur, and Francky Catthoor. *Systematic methodology for real-time cost-effective mapping of dynamic concurrent task-based systems on heterogenous platforms.* Springer Science & Business Media, 2007.

[25] A. Macii, L. Benini, and M. Poncino. *Memory Design Techniques for Low-Energy Embedded Systems.* Kluwer Academic Publishers, 2002.

[26] Afzal Malik, Bill Moyer, and Dan Cermak. A low power unified cache architecture providing power and performance flexibility. In *Low Power Electronics and Design, 2000. ISLPED'00. Proceedings of the 2000 International Symposium on*, pages 241–243. IEEE, 2000.

[27] Naraig Manjikian and Tarek Abdelrahman. Array data layout for the reduction of cache conflicts. In *Proceedings of the 8th International Conference on Parallel and Distributed Computing Systems*, pages 1–8. Citeseer, 1995.

[28] P Meinerzhagen, C Roth, and A Burg. Towards generic low-power area-efficient standard cell based memory architectures. In *Circuits and Systems (MWSCAS), 2010 53rd IEEE International Midwest Symposium on*, pages 129–132. IEEE, 2010.

[29] Pascal Meinerzhagen, SM Yasser Sherazi, Andreas Burg, and Joachim Neves Rodrigues. Benchmarking of standard-cell based memories in the sub-vt domain in 65-nm cmos technology. *IEEE Transactions on Emerging and Selected Topics in Circuits and Systems*, 1(2), 2011.

[30] Sparsh Mittal. A survey of architectural techniques for improving cache power efficiency. *Sustainable Computing: Informatics and Systems*, 4(1):33–43, 2014.

[31] Yoichi Oshima, Bing J Sheu, and Steve H Jen. High-speed memory architectures for multimedia applications. *Circuits and Devices Magazine, IEEE*, 13(1):8–13, 1997.

[32] Preeti Ranjan Panda, Francky Catthoor, Nikil D Dutt, Koen Danckaert, Erik Brockmeyer, Chidamber Kulkarni, A Vandercappelle, and Per Gunnar Kjeldsberg. Data and memory optimization techniques for embedded systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 6(2):149–206, 2001.

[33] Preeti Ranjan Panda, Nikil D Dutt, and Alexandru Nicolau. *Memory issues in embedded systems-on-chip: optimizations and exploration.* Springer Science & Business Media, 1999.

[34] Preeti Ranjan Panda, Nikil D Dutt, Alexandru Nicolau, Francky Catthoor, Arnout Vandecappelle, Erik Brockmeyer, Chidamber Kulkarni, and Eddy De Greef. Data memory organization and optimizations in application-specific systems. *IEEE Design & Test of Computers*, (3):56–57, 2001.

[35] NL Passes, Edwin Hsing-Mean Sha, and Liang-Fang Chao. Multi-dimensional interleaving for time-and-memory design optimization. In *Computer Design: VLSI in Computers and Processors, 1995. ICCD'95. Proceedings., 1995 IEEE International Conference on*, pages 440–445. IEEE, 1995.

[36] Herman Schmit and Donald E Thomas. Synthesis of application-specific memory designs. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 5(1):101–111, 1997.

[37] Namita Sharma, TV Aa, Prashant Agrawal, Praveen Raghavan, Preeti Ranjan Panda, and Francky Catthoor. Data memory optimization in lte downlink. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 2610–2614. IEEE, 2013.

[38] Tajana Šimunić, Luca Benini, and Giovanni De Micheli. Cycle-accurate simulation of energy consumption in embedded systems. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 867–872. ACM, 1999.