Design-Time System-on-Chip Memory Optimization

Von der Fakultät Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Manuel Strobel

aus Bad Saulgau

Hauptberichter:	Prof. Dr.-Ing. Martin Radetzki
Mitberichter:	Prof. Dr.-Ing. habil. Christian Haubelt

Tag der mündlichen Prüfung: 26.05.2020

Institut für Technische Informatik der Universität Stuttgart

2020

Acknowledgements

Partially, ideas and concepts in this thesis have been developed while I was working for the project CONFIRM (project label 16ES0569).

I am very thankful to Prof. Martin Radetzki for his extremely valuable feedback whenever needed and the opportunity to become a part of his research group. Many thanks also to Prof. Christian Haubelt for evaluating my thesis.

A big thank you to my colleagues here at the university, especially to Marcus Eggenberger, for helpful discussions and an always open ear.

Thanks again to Michael Fouad, Julius Hiller, and all other students, who supported me with input from bachelor/master thesis or project work.

I am grateful to my wife Elisabeth, my parents, and my entire family for being part of my life and for their unconditional support.

Abstract

Trends as miniaturization and more data-intensive applications cause System-on-Chip (SoC) design to become increasingly complex and far from manageable without automated design methods. A central aspect in the corresponding field of Electronic Design Automation (EDA) is optimization, where especially the memory subsystem is of growing interest as the above trend allows to integrate more and more memory on-chip.

Optimization potential of Static Random-Access Memory (SRAM), the most prominent storage technology for on-chip use, is twofold. On the one hand, the memory size is highly decisive. This is due to the fact that dynamic energy from read and write operations turns out to be consumed in the memory periphery to a large degree. As larger SRAM blocks logically require more switching logic in the periphery, using small SRAM resources for frequently used program code and data turns out to be highly energy-efficient. Steady reduction of the feature size in chip fabrication, on the other hand, leads to considerably increasing leakage currents and thus higher static power consumption. Saving potential in this regard is promised by the targeted activation of memory low-power modes. Spin-Transfer Torque Random-Access Memory (STT-RAM), an emerging memory technology, promises the same benefits as SRAM in terms of access performance at lower on-chip area footprint and without being volatile. Optimization potential in terms of energy consumption in this storage technology is particularly found in the costly write operation that allows for an energy/latency trade-off.

This thesis contributes to the field of System-on-Chip design in general and to on-chip memory optimization in particular as follows. At large, a complete workflow for the application-specific optimization of memory subsystems at system design-time is proposed. This involves the automated and transparent connection of software simulation and memory access profiling, optimization of the memory subsystem, and finally the implementation of obtained results, ideally on the software-level. While minor contributions to Instruction Set Simulation (ISS) and code generation round off this workflow, main focus is on the optimization methods for SRAM- and STT-RAM-based memory subsystems that are at the core of this workflow.

Inspired by embedded system synthesis theory, every proposed on-chip memory optimization method is categorized and formally defined as combination of memory allocation, application binding, and memory operation mode scheduling. Each mathematical problem formulation is further implemented by means of mixed-integer linear or quadratic programming, or using a heuristic. Following this uniform structure, this work introduces four different optimization concepts. This includes, at first, a method for dynamic energy minimization in SRAM-based memory subsystems through combined allocation and binding. Next, and with focus on static power consumption in SRAM, a combined solution for allocation, binding, and memory operation mode scheduling. Third, previous aspects are re-considered in the context of multi-core designs, i.e., targeting Multi-Processor System-on-Chip (MPSoC) design. The last optimization method deals with STT-RAM memories and presents a way for the combined determination of memory allocation and application binding while further exploiting the above mentioned trade-off.

Thorough experimental evaluation proves the general functionality and scalability of all optimization methods. Beyond that, high application-specific saving potential can be reported. Concerning dynamic energy consumption, an optimized split memory configuration yields savings of partly over 90 % when compared to a baseline configuration with only one, typically large memory. In terms of static energy, percentage savings of over 60 % can be achieved in selected cases through the utilization of memory low-power modes. Additional impact through different write modes in STT-RAM instead turns out to be dominated by dynamic energy consumption, i.e., similar to SRAM memories, high reductions are most and foremost possible through a split memory setup.

All in all, integrated into the complete flow of simulation, optimization, and code generation, the proposed memory optimization methods show promising results. Due to the sole availability of simulation-based memory figures, future investigation of presented concepts with memory figures from industrial environments for single- and multi-core SoC design is the next logical step.

Kurzfassung

Zunehmende Miniaturisierung und datenintensive Anwendungen bewirken mit Blick auf den System-on-Chip (SoC) Entwurf eine erhöhte Komplexität, die ohne automatisierte Entwurfsmethoden nicht zu bewältigen ist. Ein zentraler Aspekt im Bereich der Entwurfsautomatisierung stellt hierbei die Optimierung dar. Besonders das Speichersubsystem ist in diesem Zusammenhang von großem Interesse, da die genannten Entwicklungen zu einer stetigen Zunahme von Speicherblöcken auf dem Chip führen.

Für SRAM-Speicher, die derzeit meist genutzte On-Chip Speichertechnologie, sind besonders zwei Optimierungsaspekte von Interesse. Zum einen der dynamische Energiebedarf, der sich durch Lese- und Schreibzugriffe ergibt. Hier zeigt sich, dass besonders die Speichergröße aufgrund der Zunahme an Logik in der Speicherperipherie hinsichtlich des Energiebedarfs von Bedeutung ist. Daraus folgt, dass die Nutzung eines kleinen Speichers deutlich effizienter ist als die eines großen Blocks. Zum anderen ergibt sich durch immer kleinere Strukturen bei der Chipherstellung eine deutliche Zunahme der statischen Leistungsaufnahme durch Leckströme. Dieser Entwicklung kann durch den Einsatz von Low-Power Modi begegnet werden kann. Magnetoresistive RAM-Speicher wie z.B. STT-RAM bezeichnen eine neue nichtflüchtige Speichertechnologie, die ähnliche Performanz wie SRAM bei deutlich geringerem Platzbedarf ermöglicht. Optimierungspotential ergibt sich hier vor allem durch einen Trade-Off zwischen Energiebedarf und Dauer eines Schreibzugriffs.

Unter diesen Gesichtspunkten trägt die vorliegende Arbeit zur Weiterentwicklung der Methodik im System-on-Chip Entwurf bei. Konkret wird der Fokus hierbei auf die Optimierung des On-Chip Speichersubsystems gelegt. Der vorgestellte Workflow ermöglicht eine anwendungsspezifische Optimierung zur Entwurfszeit und umfasst folgende Schritte: Softwaresimulation inklusive dem Anlegen von Speicherzugriffsprofilen, tatsächliche Optimierung des Speichersubsystems sowie die abschließende Umsetzung der Optimierungsergebnisse, idealerweise auf Softwareebene. Während einzelne kleinere Beiträge der Arbeit in den Bereichen Instruktionssatzsimulation und Code-Generierung diesen Workflow abrunden, so besteht der Hauptbeitrag in den vorgestellten Optimierungsmethoden für SRAM und STT-RAM Speicher.

In Anlehnung an die Theorie aus dem Bereich Synthese Eingebetteter Systeme werden die einzelnen Optimierungsverfahren kategorisiert und formal als Kombination von Speicherallokation, Applikationsbindung und hinsichtlich der Ablaufplanung von Speicher-Betriebsmodi definiert. Die mathematischen Beschreibungen der Einzelprobleme werden je entweder als gemischt-ganzzahliges lineares bzw. quadratisches Programm oder mit Hilfe einer Heuristik implementiert. Insgesamt werden vier Optimierungsansätze vorgestellt. Die erste Methode optimiert den dynamischen Energiebedarf von SRAM-basierten Speichersystemen durch gezielte Allokation von Speicherressourcen und Abbildung der Anwendung auf diese Speicher durch die Bindung. Der zweite Ansatz stellt die statische Leistungsaufnahme in den Vordergrund und liefert eine optimierte Lösung für Allokation, Bindung sowie Ablaufplanung der Speicher-Betriebsmodi. Die Übertragung sowie Anpassung dieser Optimierungskonzepte auf Multi-Core Systeme wird nachfolgend separat betrachtet. Zuletzt wird ein Konzept für die optimierte Nutzung von STT-RAM vorgestellt, welches neben der Bestimmung von Allokation und Bindung eine Möglichkeit bietet, den zuvor beschriebenen Trade-Off auszunutzen.

Die experimentelle Auswertung belegt zum einen die Funktionalität und Skalierbarkeit der Optimierungsverfahren und zeigt zum anderen deutliches Einsparungspotential beim Energiebedarf des Speichersubsystems auf. Hinsichtlich des dynamischen Energiebedarfs ergeben sich im Vergleich zu einem Basissystem mit nur einem, typischerweise großen Speicherblock, deutliche Einsparungen von teilweise über 90 % durch die optimierte heterogene Speicherarchitektur. Für den statischen Energiebedarf ergeben sich durch gezielten Einsatz von Low-Power Modi der Speicher prozentuale Einsparungen von über 60 % in ausgewählten Fällen. Einsparungen durch verschiedene Modi beim Schreiben eines STT-RAM Speichers werden in den Experimenten hingegen durch dynamische Anteile dominiert. So ist hier, ähnlich wie bei SRAM Speichern, die größte Einsparung durch die Unterteilung des Speichersubsystems in mehrere, kleine Blöcke zu erzielen.

Integriert in den Ablauf von Simulation, Optimierung und Code-Generierung, liefern die vorgestellten Optimierungsverfahren insgesamt vielversprechende Ergebnisse. Aufgrund der ausschließlichen Verfügbarkeit von simulationsbasierten Speicherwerten lässt sich die weitere Untersuchung vorgestellter Verfahren, mit Speicherwerten aus dem industriellen Umfeld beim Entwurf von Single- und Multi-Core SoCs, als nächster Schritt identifizieren.

Contents

1 Introduction . . . 1
1.1 Motivation . . . 2
1.2 Summary of Contributions . . . 3
1.3 Document Structure . . . 4

2 Background . . . 5
2.1 Embedded System Synthesis . . . 5
2.1.1 Basic Modeling . . . 7
2.1.2 Allocation . . . 10
2.1.3 Binding . . . 11
2.1.4 Scheduling . . . 12
2.1.5 Implementation . . . 13
2.2 Instruction Set Simulation and Profiling . . . 14
2.2.1 Basics . . . 14
2.2.2 Interpretive Simulation . . . 15
2.2.3 Host-Compiled Simulation . . . 16
2.2.4 Conclusion . . . 18
2.3 Memory Technologies . . . 18
2.3.1 Static Random-Access Memory (SRAM) . . . 18
2.3.2 Spin-Transfer Torque RAM (STT-RAM) . . . 26

3 Related Work . . . 31
3.1 Hardware-Based Concepts . . . 32
3.2 Software-Based Concepts . . . 37
3.3 Memory Optimization for Multi-Core Systems . . . 43
3.4 STT-RAM Optimization . . . 46
3.5 Summary . . . 47

4 Memory Optimization . . . 49
4.1 Optimization Workflow . . . 49
4.2 Dynamic Energy Optimization (SRAM) . . . 50
4.2.1 Context . . . 51
4.2.2 Optimization Model . . . 52
4.2.3 Combined Allocation and Binding MIQP . . . 55
4.2.4 MIQP Variations . . . 57

4.3 Static Power Optimization (SRAM) 59
4.3.1 Context 60
4.3.2 Optimization Model 61
4.3.3 Implementation Considerations 66
4.3.4 Allocation and Binding Heuristics 69
4.3.5 Operation Mode Scheduling MIQP 72
4.4 Memory Optimization in Multi-Core Systems (SRAM) 75
4.4.1 Context 76
4.4.2 Optimization Model 77
4.4.3 Binding MILP 78
4.4.4 Operation Mode Scheduling 80
4.5 Dynamic Energy Optimization (STT-RAM) 81
4.5.1 Context 82
4.5.2 Optimization Model 82
4.5.3 Combined Allocation and Binding MIQP 84

5 Code Modification and Generation 87
5.1 Basic Concept 88
5.1.1 Binary Parsing 90
5.1.2 Code Insertion 94
5.1.3 Assembly Code Emission 94
5.2 Automated Code and Data Placement 95
5.3 Operation Mode Schedule Integration 96
5.3.1 Memory Activation-on-Access 98
5.3.2 Memory Configuration Handling via Stack 100
5.4 Adjustments for Multi-Core Systems 101
5.5 Write Mode Handling in STT-RAM 102

6 Evaluation 103
6.1 Experimental Setup and Tools 103
6.2 Combined SRAM Allocation and Binding Results 105
6.2.1 Dynamic Energy Minimization 106
6.2.2 MIQP Solving Performance and Scalability 108
6.2.3 Interconnect Impact 110
6.2.4 Area Consumption Minimization 111
6.2.5 Multi-Objective Optimization and Pareto Analysis . . 113
6.2.6 Results for Multitasking Systems 115
6.3 SRAM Operation Mode Scheduling Results 117
6.3.1 Break-Even Point Analysis 117

6.3.2 Static Power Minimization 119
6.3.3 Solving Performance and Scalability 124
6.3.4 Code Modification and Generation Overhead 125
6.3.5 Evaluation of a Sensor Hub System 127
6.3.6 Corridor Constraints 127
6.4 Multi-Core Memory Optimization Results 129
6.4.1 Dynamic Energy and Static Power Minimization 129
6.4.2 Comparison of Scratchpads and Caches 132
6.5 STT-RAM Allocation and Binding Results 134
6.5.1 Dynamic Energy Minimization 134
6.5.2 MIQP Performance and Scalability 137

7 Conclusions 139

Appendix 143
A Combined SRAM Allocation and Binding MIQP 143
B SRAM Operation Mode Scheduling MIQP 146
C Multi-Core Application Binding MILP 149
D Combined STT-RAM Allocation and Binding MIQP 151

Abbreviations 153

References 155

Chapter 1
Introduction

In a universal form, embedded systems can be defined as „computers [that are] lodged in other devices where the presence of the computers is not immediately obvious“ [45]. A further refinement of this definition classifies embedded devices as computer systems of limited complexity that are not able to run external, third-party software. This clearly separates them from desktop computers and servers. Still, the variety of embedded systems is huge, for example reflected by the price range of utilized microprocessors in this domain that varies between 0.01 $ and 100 $ [45]. Driven by the corresponding wide range of target applications, the embedded field represents the fastest growing portion of the computer market. The application in mobile devices, Internet of Things (IoT), natural language processing, cloud technologies, multimedia, robotics, autonomous driving, health service, or renewable and smart energy systems for example make embedded systems and micro electronics in general to become a key factor to progress [34].

Despite this great variety, embedded computing problems are commonly solved in one of the following three ways. Hennessy et al. [45] specify:

1. A hardware/software solution consisting of custom hardware in combination with one or multiple embedded processors plus corresponding software (often realized on a single chip as System-on-Chip (SoC) or Multi-Processor System-on-Chip (MPSoC)).
2. Off-the-shelf hardware in combination with a custom software solution.
3. A Digital Signal Processor (DSP) with customized software.

Regardless, which of those concepts is chosen, there are several critical issues that apply to embedded system design in common. Most and foremost, minimization of memory and power consumption can be named in this regard. Other factors are robustness, system security, and energy efficiency. High market pressure further makes it difficult to develop new and innovative embedded products with increasing complexity in short periods and at a reasonable price. In mobile devices, consumers expect more functionality and better performance at constant or even reduced energy budgets. Batteries, however, do not improve at the same pace and the usual procedure of lowering the system supply voltage is hampered and already close to reaching a predicted strict limit of 0.6 V [54]. Challenges that go beyond scaling according to Moore's law are increasingly relevant. In its 2015 *More Moore* report, the ITRS consortium discusses the Power Performance Area Cost (PPAC) value as representative figure for future technology progress. Instant data generation and related applications in the fields big data, cloud, IoT, or edge computing dictate the needs, named to be >30 % performance and >50 % power improvement at >50 % area reduction and 35-40 % less die cost between subsequent technology nodes, corresponding to a period of 2-3 years.

To cope with these challenges, new methods for automated embedded system design and optimization with respect to performance, power, and area are highly required as central pillar for innovation and continuous development.

1.1 Motivation

For several reasons, especially the embedded memory subsystem is a highly promising target for optimization in SoC design. Continuous integration and technology scaling facilitate on-chip solutions with Static Random-Access Memory (SRAM) as dominant storage technology. Key advantages of SRAM are robustness, high speed, low power consumption, and compatibility with standard CMOS process when compared to other mature memory technologies, e.g., Dynamic Random-Access Memory (DRAM). Driven by increasingly data-intensive applications, meanwhile 50 % to 90 % of the total transistor count is attributed to on-chip SRAM [110]. Logically, memory accounts for the largest part of total system power consumption with a share of about 60 % [84]. Concerning the energy consumption in SoC devices, shares of 50 % up to 75 % [88] are stated for the memory subsystem in literature. Beyond the dynamic energy consumption that results from memory access, SRAM volatility requires the cells to remain constantly powered on in order to hold the data. The resulting static power consumption, mainly due to leakage, increasingly dominates the total dissipation. This fact is further accelerated by constant cell size reduction and projected to exceed 50 % of the overall circuit power consumption in CMOS technologies [88].

Evolving non-volatile memories tackle the leakage problem of SRAM. Especially the Spin-Transfer Torque Random-Access Memory (STT-RAM) technology can be named as promising candidate in this regard. It satisfies CMOS compatibility and scalability demands and furthermore, meets speed and endurance requirements of increasingly data-centric and instant-on applications [146]. However, the STT-RAM technology still matures and especially the mitigation of the costly write operation in terms of energy and latency is subject to active research [115].

Owing to mostly custom software solutions, embedded systems usually perform known and further, often restricted tasks. This fact enables targeted and application-specific optimization without compromise. Focusing on-chip SRAM memory, a break-down analysis shows that 90 % of the energy is consumed by components as pre-charge unit, sense amplifiers, or address transition detection [29]. In short, it is not the memory array but its periphery that accounts for the largest dynamic energy consumption part. Savings are consequently facilitated by splitting SRAM memory into possibly small sub-blocks or banks while respecting access statistics and keeping an eye on the area overhead. Also the efficient utilization of a memory hierarchy with software-programmable memories, so-called scratchpads, as replacement for common caches promises savings in terms of energy, power, and performance. The static power issue can be tackled by low-power modes. In STT-RAM it is the write performance versus energy trade-off that encourages for closer investigation. Altogether, this work is motivated by the objective of finding the best possible embedded system design in general and, the automated optimization of the on-chip memory subsystem in particular.

1.2 Summary of Contributions

Targeting SoC and MPSoC embedded devices, this thesis presents a complete workflow for the application-specific on-chip memory subsystem optimization from different viewpoints. Similar properties to system synthesis theory are formulated and used as to develop a unique formal notation scheme for allocation α, binding β, and schedule γ in the context of memory optimization. On this basis, the different methods can be described, including:

- A memory optimization method for hardware/software co-design of energy-efficient SoCs with area constraints [126].
- A combined solution for the efficient handling of (1) memory instance allocation, (2) application to memory binding, and (3) scheduling of memory low-power modes [113] [129] [132].
- The adaption and extension of the previous concept to MPSoC embedded multi-core platforms [127] [131].

- A solution for optimized memory allocation and application binding when STT-RAM is included as on-chip memory.

Additionally, two minor contributions are made in the form of tools that complement the workflow in direction of full automation. This includes:

- A tool for fast yet accurate Instruction Set Simulation (ISS) and memory access profiling [113] [128].
- A compiler extension for the implementation of obtained optimization results into embedded software at hand by means of targeted code generation and insertions [130].

1.3 Document Structure

This dissertation is organized as follows. Chapter 2 conveys the basics for the following chapters. It covers fundamentals of embedded system synthesis and properly introduces its main aspects, allocation, binding, and scheduling. Besides that, SRAM and STT-RAM memory characteristics are presented while aspects with optimization potential are highlighted. Also, a short primer on instruction set simulation and profiling is given due to its relevance for the generation of important input data. State-of-the-art in the field of embedded system memory optimization is presented and discussed in Chapter 3. The individual optimization concepts and thus the main contributions of this work are detailed in Chapter 4. This includes the mathematical description of each optimization problem as well as implementation details where applicable. All optimization methods are either realized using a heuristic or implemented as Mixed-Integer Linear Program (MILP) respectively Mixed-Integer Quadratic Program (MIQP). Chapter 5 describes the code generation tool as implemented on the basis of LLVM [77], which is used for the automated implementation of optimization results. Thorough evaluation as presented in Chapter 6 proves the functionality of the optimization workflow and all presented optimization methods. Potential savings with respect to energy, execution time, and area are presented and discussed along with general trends. Besides that, comparison with other optimization approaches and memory architectures is included as well as the performance evaluation of presented tools and optimization methods. Chapter 7 concludes the thesis.

Chapter 2
Background

This chapter provides the necessary background information as relevant in the further course of this thesis. In Section 2.1, memory subsystem optimization is described and classified as part of synthesis in embedded system design. Next, the formal model that represents the basis for all following memory optimization concepts is discussed. A short introduction to instruction set simulation as required tool for the generation of memory access statistics is given in Section 2.2 and followed by a description of relevant memory technologies and their characteristics in Section 2.3. These two sections teach the basics for the two main inputs to optimization and conclude the chapter at the same time.

2.1 Embedded System Synthesis

Synthesis in general is defined as: „The combination of components or elements to form a connected whole“ [107]. Applied to embedded system design, synthesis describes the fully automated transformation of a given system specification to a corresponding implementation by means of hardware and software. A specification, i.e., a behavioral model or description in combination with specified requirements serves as basis for design decisions and gradual refinement thereof in the synthesis step as illustrated in Figure 2.1. The finally synthesized structure implements the desired quality features and ideally matches with the previously specified requirements.

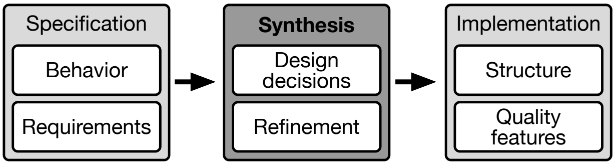


Figure 2.1 Synthesis in embedded system design (based on [44], Figure 1.12)

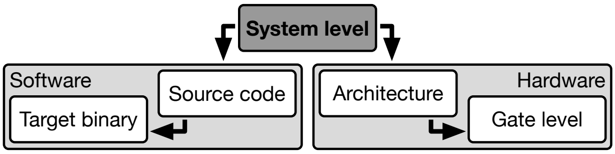


Figure 2.2 Embedded system design flow abstraction levels (based on [138], Figure 1.7)

In order to cope with the high complexity in present-day embedded designs, the basic concept of synthesis as *automated transformation process* is applied to different abstraction levels as illustrated in Figure 2.2. When following a top-down design flow, a step-wise refinement from high to low abstraction levels takes places. System synthesis starts on the highest abstraction level, dealing for example with basic questions of hardware/software co-design, i.e., which part of the system to implement in hardware and which part by means of software. This involves high-level analysis and optimization steps, for example with respect to expenses, area, or power consumption. Also the consideration of constraints can be included. For the hardware part, high-level synthesis on the architectural level involves the derivation of integrated circuits from graph-based control/data path models. Gate level synthesis further brings the resulting description, for example in form of Verilog or VHDL code, down to the gate respectively transistor level. If the target platform is already given by some off-the-shelf product, the hardware synthesis part is dropped. Synthesis steps for the part of the embedded software, however, apply in any case. Starting from an architectural software perspective, for example using Unified Modeling Language (UML), target-independent source code is partly generated automatically from this representation. Additional, manually programmed code further complements the generated code base. Compiler tools finally carry out the transformation from source code to target-specific representation, i.e., to assembly code and finally target binary on the last level.

Despite the different abstraction levels and logically highly varying forms of specification and resulting implementation, Teich and Haubelt [138] identify three common sub-problems in synthesis named *allocation*, *binding*, and *scheduling*. On system level for example, allocation includes the identification and selection of different resource types for a given embedded computing problem. Binding involves the mapping of individual tasks to the set of allocated resources. Scheduling defines the chronological order, i.e., when to execute which task. Other abstraction levels include similar steps, what allows for a general categorization into allocation, binding, and scheduling problems.

Memory optimization as discussed in this thesis can be considered as part of system level synthesis. It is conducted at system design-time and moreover has some interesting similarities with synthesis in general. In fact, the central objectives in memory optimization can also be classified as allocation, binding, or scheduling problems. Now, in order to address differences and similarities but also to give a formal foundation for the remainder of this thesis, the following sections provide a formal model for these synthesis sub-problems on the basis of [138] and adapt it to the context of memory optimization.

2.1.1 Basic Modeling

In fundamental embedded system synthesis as defined by Teich and Haubelt [138], the problem statements for allocation of resources, binding of tasks to resources, and scheduling of tasks are based on two types of graphs. On the one hand, a *problem graph* that models the individual tasks of the embedded computing problem. On the other hand, a *resource graph* that contains the set of available resources and their relation to the nodes of the problem graph. The combination of these graphs is referred to as *specification*. Unfortunately, this classic synthesis model can not be applied out-of-the-box to the memory subsystem and its optimization. In short, tasks in their original sense are not relevant for a consideration of the memory subsystem and a problem graph is thus not required at all. Instead, individual data blocks need to be distinguished, including control and data dependencies. Also, number and type of relevant resources is limited to memory components. Processing elements or other computational components are not of interest. Still, parts of the modeling scheme in [138] apply, which leads to the following adapted model.

The embedded software application is modeled by clearly separable units, referred to as *application profiles* (cf. Definition 2.1). A profile describes a single function, in terms of program code, or a static variable for the part of application data. Dynamic memory sections as heap and stack are considered as monolithic blocks and therefore modeled as a single data profile each.

Definition 2.1 (Application Profile)

The set of application profiles P describes the embedded application. Every profile $p \in P$ represents a unique address range and is characterized by a set of parameters, including:

Parameter	Description	Domain	Unit
t	Profile type (**c**ode or **d**ata)	$\{c, d\}$	
s^p	The profile's memory requirements	$\mathbb{Z}_{>0}$	byte
n^r	Number of read accesses	$\mathbb{Z}_{\geq 0}$	
p^r	Normalized read probability of this profile	[0, 1]	
n^w	Number of write accesses	$\mathbb{Z}_{\geq 0}$	
p^w	Normalized write probability of this profile	[0, 1]	
d	The profile's normalized duty cycle	[0, 1]	

Please note that read/write figures and duty cycle are application-specific and strongly connected to the application period T. That is the execution time of one iteration in case of a periodic application or simply the investigated time frame, for example of one representative execution run. The duty cycle (d) is provided in normalized form and describes a profile's individual share of the period T, i.e., the time share, this particular profile is active. Read and write numbers (n^r and n^w) count the accesses to the address range of the profile. Read and write probability (p^r and p^w) instead yield the normalized memory access quantities with respect to the period T. The example in Figure 2.3 illustrates these profile characteristics. Each bar in diagram (a) represents reading (light gray) or writing activity (dark gray). Grouping of individual memory accesses leads to diagram (b), which explains the relation between memory access probabilities, duty cycle, and application period.

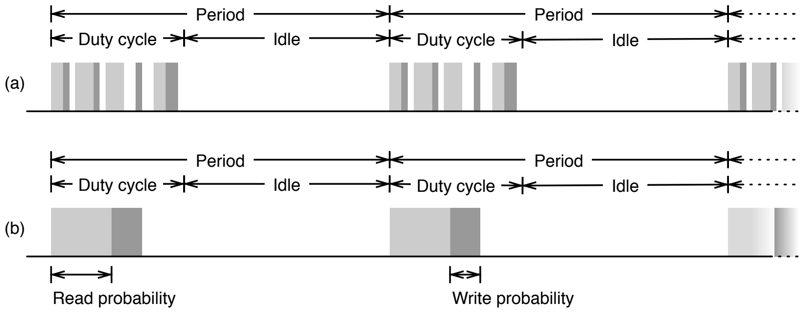


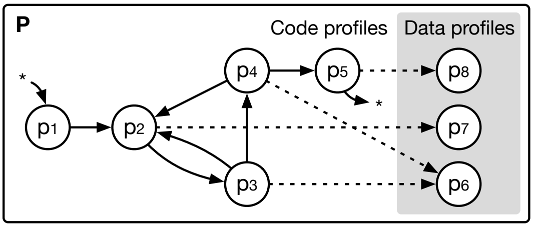
Figure 2.3 Exemplary application profile with period, duty cycle, and r/w probabilities

Next, the relationship between the individual application profiles in P can be modeled as *dependency graph* according to the following definition:

Definition 2.2 (Dependency Graph)

A dependency graph $G_D(P, E_D)$ is a directed graph that consists of node set P and edge set E_D. Each node $p \in P$ represents an application profile. Each edge $e = (p, q) \in E_D$ models the control or data dependency between two profiles. The weight function $w : E_D \rightarrow \mathbb{Z}_{\geq 0}$ assigns each edge in E_D the number of transitions if $(t_p = c) \wedge (t_q = c)$ or the number of data accesses if $(t_p = c) \wedge (t_q = d)$.

An exemplary dependency graph of a periodic application is illustrated in Figure 2.4, showing control flow transitions between code profiles as solid lines and data dependencies, resulting from data accesses between related code and data profiles as dashed lines. The fact that control flow *and* data dependencies are modeled in this type of graph enables an investigation of the connectivity degree between individual memory address ranges as modeled at the granularity of profiles.



Example

$(p_1,p_2) = 1$	$(p_4,p_5) = 1$
$(p_2,p_3) = 10$	$(p_2,p_7) = 10$
$(p_3,p_2) = 5$	$(p_3,p_6) = 1$
$(p_3,p_4) = 5$	$(p_4,p_6) = 7$
$(p_4,p_2) = 4$	$(p_5,p_8) = 2$

Figure 2.4 Exemplary dependency graph

While application profiles and dependency graph describe the memory content, the on-chip memory subsystem is modeled by *memory resources* according to the following definition:

Definition 2.3 (Memory Resource)

The set M describes the memory resources that are available for the memory subsystem. Every element $m \in M$ is characterized by:

Parameter	Description	Domain	Unit
s^m	Memory size	$\mathbb{Z}_{>0}$	byte
A^m	Memory on-chip area footprint	$\mathbb{R}_{>0}$	mm^2
E^r	Dynamic energy consumption per read	$\mathbb{R}_{>0}$	J
t^r	The latency of a read operation	$\mathbb{R}_{>0}$	s
E^w	Dynamic energy consumption per write	$\mathbb{R}_{>0}$	J
t^w	The latency of a write operation	$\mathbb{R}_{>0}$	s
P^s	The memory's static power consumption	$\mathbb{R}_{>0}$	W

Every memory that provides one or multiple low-power modes is further characterized by a set of operation modes O, which describes static power consumption as well as activation and deactivation behavior through the following parameters:

Parameter	Description	Domain	Unit
P^s	Static power consumption of this operation mode	$\mathbb{R}_{>0}$	W
E^a	Energy penalty on mode activation	$\mathbb{R}_{\geq 0}$	J
t^a	Activation time of this mode	$\mathbb{R}_{\geq 0}$	s
E^d	Energy penalty on mode deactivation	$\mathbb{R}_{\geq 0}$	J
t^d	Deactivation time of this mode	$\mathbb{R}_{\geq 0}$	s

A *specification* in this memory optimization model is altogether described by the combination of application profile set P and dependency graph G_D, in conjunction with the set of available memory resources M. Building on the above definitions, the individual optimization problems allocation, binding, and scheduling can now be formulated.

2.1.2 Allocation

Allocation in the context of memory optimization describes the selection of memory instances from the specified set of memory resources M. In embedded system projects, where the hardware part is modifiable and subject to design

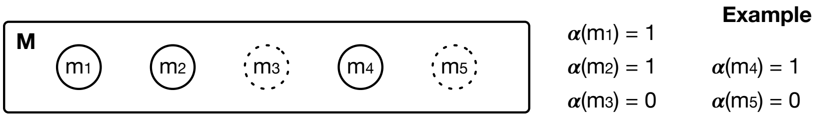


Figure 2.5 Memory allocation example

(cf. Chapter 1), this step defines the architecture of the memory subsystem. If predetermined hardware is used instead, the memory allocation is given.

Definition 2.4 (Allocation)

The allocation of a memory subsystem is a binary function $\alpha : M \rightarrow \{0, 1\}$. It indicates for every memory resource $m \in M$ whether it is part of the memory subsystem or not.

After allocation, as for example illustrated in Figure 2.5, the design space can possibly be pruned from all unallocated memories. The resulting set of allocated memories is denoted $M^{\alpha} \subset M$.

2.1.3 Binding

Binding in memory optimization stands for the question, which application profile shall be mapped to which memory resource. This step is based on the result of allocation. Nevertheless, both problems are strongly related and a joint solving step through combination of allocation and binding is often indicated and useful. If allocation is available as input to binding, M^{α} can be used. Otherwise, a more general definition of β on the basis of all memory resources M is possible as follows:

Definition 2.5 (Binding)

Binding is defined as function $\beta : P \rightarrow M$ that assigns every application profile to one memory resource $m \in M$.

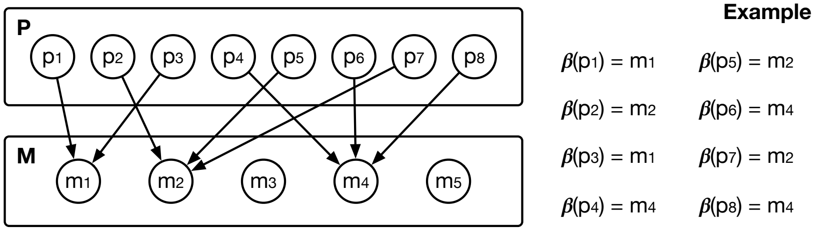


Figure 2.6 Exemplary application profile to memory resource binding

In a directed bipartite graph with vertices $P \cup M$, β can be described as activation function for a subset of edges as illustrated in Figure 2.6. It should be noted that binding typically involves various feasibility checks. The most obvious example is a memory size criterion that ensures a memory block is large enough in order to accommodate all profiles that are mapped to this resource. When solving the described binding problem, it can also be important to consider transition information as encoded in the dependency graph G_D. Especially when multiple low-power operation modes are supported, locality aspects as resulting from the binding impact the scheduling. This step is described next.

2.1.4 Scheduling

Scheduling for the memory subsystem differs from the typically associated meaning of this term, i.e., a chronological organization of tasks. In the context of this work instead, a schedule describes a relation between application profiles (cf. Definition 2.1) and memory operation modes (cf. Definition 2.3). That means, every profile in the set P is assigned an operational state of the memory subsystem as described by means of a *configuration vector*:

Definition 2.6 (Configuration Vector)

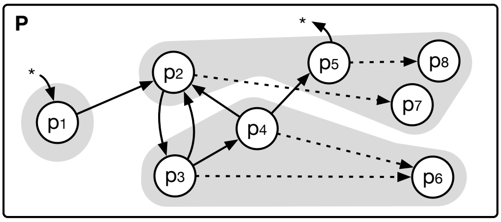
A memory configuration vector $c \in C$ is composed of one operation mode per memory resource $m \in M^\alpha$. The set of all possible configuration vectors is denoted $C \equiv O^{|M^\alpha|}$.

On this basis, the temporal causality between control flow of the application (cf. Definition 2.2) and activation/deactivation of memory operation modes becomes visible as a transition from one application profile to another possibly triggers a memory configuration change. A schedule for the memory subsystem consequently describes the relation between application profiles and configuration vectors according to the following definition:

Definition 2.7 (Schedule)

A schedule is defined as function $\gamma : P \to C$ that assigns every profile $p \in P$ to a memory configuration vector from the set C.

In order to give an example, let every memory type $m \in M^{\alpha}$ be equipped with support for two operation modes, i.e., $O = \{ACT, LP\}$. While read and write access is possible in active mode (ACT), memory access in the low-power mode (LP) is not possible. Applied to the above memory subsystem and previously discussed exemplary solutions for allocation (cf. Figure 2.5) and binding (cf. Figure 2.6), one possible configuration schedule is depicted in Figure 2.7.



Example

$\gamma(p_1) = (ACT, LP, LP)$

$\gamma(p_2, p_5, p_7, p_8) = (LP, ACT, ACT)$

$\gamma(p_3, p_4, p_6) = (ACT, LP, ACT)$

Figure 2.7 A possible memory configuration schedule

2.1.5 Implementation

Similar to general embedded system synthesis, there are certain dependencies between allocation, binding, and scheduling that dictate a specific order of execution. Allocation and binding are tightly connected, i.e., it highly depends on the optimization objective whether to carry out the sub-problems subsequently with allocation first, or both at once. Memory operation mode scheduling instead depends on the outcome of allocation and binding and is therefore executed last. Still, feedback loops with repetitive step-wise refinement of allocation, binding, and scheduling are possible.

Altogether, it is the triple of (α, β, γ) that constitutes an *implementation*, i.e., an optimized application-specific System-on-Chip memory subsystem. Large design spaces in realistic use cases, however, make memory subsystem design a non-trivial task that is tackled in the later course of this thesis on the basis of the above definitions.

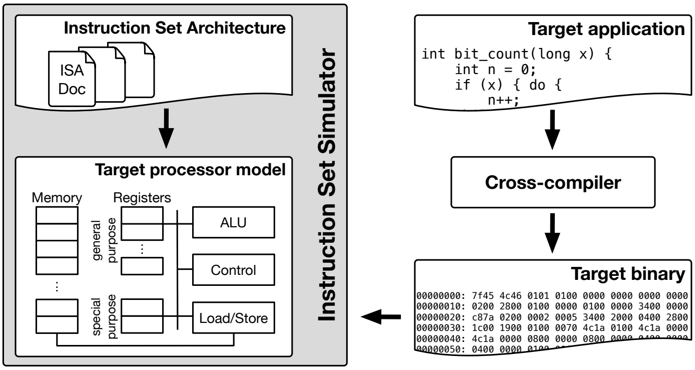
2.2 Instruction Set Simulation and Profiling

Instruction Set Simulation (ISS) describes the simulation of a target system based on a model of the embedded processor respectively its instruction set architecture. This task is executed on a simulation host platform, which is commonly different from the target platform that is subject to simulation.

Due to several reasons, ISS plays an important role in embedded system design. It enables, first of all, a decoupling of hardware and software development and consequently increases productivity in system design. Further, it is an important tool for system analysis and thus supports critical early design decisions. Beyond that, accurate profiling of bus access behavior or memory utilization is possible on the basis of ISS. Especially the latter is of interest in this work, as relevant for the generation of application profiles and dependencies according to Definitions 2.1 and 2.2. With focus on memory access profiling, the following section gives an overview on instruction set simulation in general and further introduces the two most known simulation concepts along with advances in the respective fields.

2.2.1 Basics

Central building block of any instruction set simulator is a behavioral model of the target processor that, at minimum, represents the Instruction Set Architecture (ISA) of this platform. Figure 2.8 depicts this general structure and illustrates the typical input data flow, including embedded software, cross-



```
int bit_count(long x) {
    int n = 0;
    if (x) { do {
        n++;
```

```
00000000: 7f45 4c46 0101 0100 0000 0000 0000 0000
00000010: 0200 2800 0100 0000 0100 0000 3400 0000
00000020: c87a 0200 0002 0005 3400 2000 0400 2800
00000030: 1c00 1900 0100 0070 4c1a 0100 4c1a 0000
00000040: 4c1a 0000 0800 0000 0800
00000050: 0400 0000
```

Figure 2.8 General ISS structure and input data flow (based on [128], Figure 1 and Figure 2)

compilation, and finally, the target platform binary that serves as main input to simulation.

When used for memory access profiling, especially the connection between processor core and memory model is of interest. That means, tracing on this interface during simulation allows to collect access statistics for different address ranges and memory blocks. Together with information from the application binary, application profiles and memory accesses can further be related to each other. While many other analysis possibilities are enabled by ISS, it is this type of application-specific memory access information that is relevant for optimizations as discussed in the later course of this work.

The two most common simulation concepts for this task are *interpretive* ISS and *host-compiled* simulation. Both are introduced below.

2.2.2 Interpretive Simulation

Interpretive simulation is the classic way of instruction set simulation. It comprises an implementation of a complete processor model and pipeline, the individual pipeline stages are simulated iteratively within a main simulation loop. During simulation, the instructions of the target binary are interpreted one after the other as they pass the pipeline of the modeled target architecture (cf. Figure 2.9). This accurate imitation of fetch, decode, execute, and write back stages for example allows for fine-grained memory access traces that

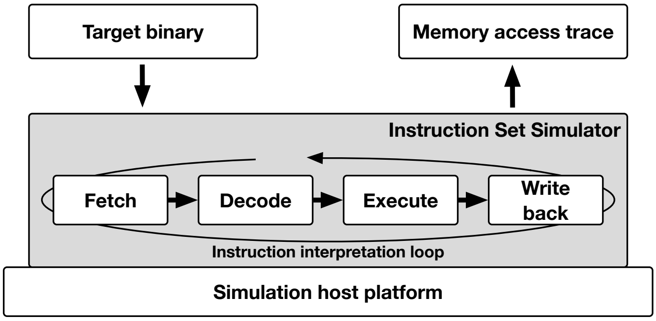


Figure 2.9 Abstract workflow of interpretive ISS

can directly be collected on the interface between load/store unit and memory model (cf. Figure 2.8).

It is important to note that this approach is linked to elaborate modeling and typically sequential execution, which causes a trade-off between accuracy of simulation results and performance. In other words, interpretive simulators are able to provide simulation results with high degree of detail but they are also characterized as slow in terms of execution time. An exemplary concept that addresses this performance problem is the interpretive ISS method with caching mechanism for already decoded instructions as proposed by Cmelik et al. [27]. Another point that affects this trade-off is the degree of detail of the processor model. *Cycle-accurate* simulators provide the most accurate simulation results but come at the cost of high modeling effort and poor performance. If simulation results on a per-cycle resolution are not mandatory, so-called *instruction-accurate* simulation can be applied. According to this concept, individual instructions are used as smallest unit, which allows to reduce the modeling effort. This way, performance is improved in exchange for a less accurate resolution of obtained simulation results.

2.2.3 Host-Compiled Simulation

Host-compiled simulation as originating from Mills et al. [89] mainly aims at improving simulation performance. Instead of costly and step-wise decoding and interpretation of instructions, the target application is brought to an intermediate representation in a first step; then enriched or annotated with

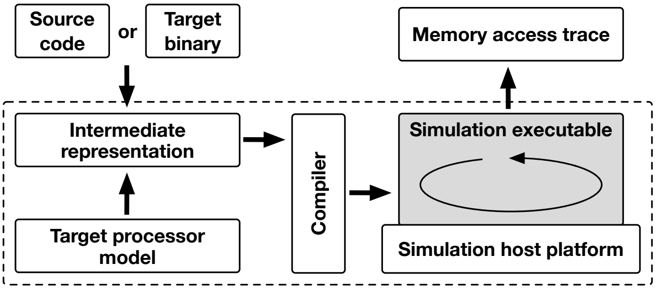


Figure 2.10 Abstract workflow of host-compiled simulation

behavioral descriptions of the target architecture; and finally compiled for the simulation host platform. During execution of the resulting executable on the simulation host machine, the annotated behavioral information mimics the target system and thus enables the generation of target-specific analysis figures.

An abstract depiction of the workflow in compiled simulation is given in Figure 2.10. The two types of input data, i.e., target binary and application software, represent two possible approaches to distinguish. On the one hand, there are methods that start from a target binary, which is brought to a higher abstraction level using binary analysis or decompilation. Afterwards, behavioral models of individual instructions are applied on this representation level as to imitate the target system. On the other hand, there are software-based annotation schemes that start with high-level software and either annotate the source code [80] [123] or work on a target-independent intermediate code representation [18] [63].

Even though host-compiled simulation already provides good performance, some research has been conducted on further improvements in this direction through more efficient utilization of host resources [108] [150] or by means of abstraction [111]. Also retargetability has been researched and solutions using virtualization [150] or abstract modeling [19] have been proposed. Regarding memory access profiling, however, abstraction reduces the possible result accuracy. Further, execution of the embedded application on a simulation host platform with different addressing scheme and address space as compared to the target architecture needs to be considered. This fact again complicates the situation and makes compiled simulation only partially suitable for memory access tracing as for example shown in [26].

2.2.4 Conclusion

In conclusion, interpretive simulation can be seen as most accurate but slow solution, host-compiled simulation instead provides high performance but lacks the modeled level of detail that is required for a determination of precise memory access statistics. Hence, host-compiled simulation is rather a suitable tool for early system design stages where fast estimation of system run-time or power consumption figures is important. These opposing pros and cons of the above simulation concepts finally lead to hybrid solutions for the task of memory access profiling. That is, computationally intensive tasks of interpretive simulation as instruction decoding for example are pre-computed and combined with an accurate but reduced simulation model at run-time. This way, a reasonable degree of memory access trace accuracy and simulation speed can be achieved. One such hybrid simulation concept [128] with its included memory profiler [113] is also a minor contribution of this thesis.

2.3 Memory Technologies

Memory resources (cf. Definition 2.3) are a central part of the specification in the above basic model. The information that characterizes the memory depends on the selected or, in case of some pre-defined hardware platform, implemented memory technology. As described in the introduction, SRAM as state-of-the-art and STT-RAM as emerging memory technology are of main interest in this thesis due to on-chip utilization and CMOS-compatibility. Below, both technologies are introduced from a general perspective but also from the viewpoint of their optimization potential.

2.3.1 Static Random-Access Memory (SRAM)

According to the authors of [110], „embedded SRAM is definitely the workhorse for on-chip data storage owing to its robust operation, high speed, and low power consumption relative to other options“. Also the possibility of direct integration with other logic as based on the CMOS production process makes it, at least at present, the technology of first choice in embedded memory subsystems.

With respect to memory optimization, it is very much the application but also the position of some memory in the system hierarchy that specifies the performance needs and available power budgets. Before introducing different energy respectively power saving approaches and memory hierarchy concepts, the following paragraph shortly describes the most important SRAM basics.

Basics

The base component in SRAM is a storage cell that consists of six transistors, interconnected as depicted in Figure 2.11. The tasks of this so-called 6T cell include, on the one hand, to store (hold) one bit of data and, on the other hand, to enable access to its content by means of read or write. For hold, the wordline is set to $WL = 0$; bitlines do not care ($BLs = X$). In case of a write access, $WL = 1$; the bitlines are driven with the new data value. Reading also includes a wordline select ($WL = 1$); the bitlines are first pre-charged (to V_{DD}) and left floating afterwards. Memory arrays up to a size of 256 KB are constructed from the two-dimensional combination of this basic cell. Storage arrays larger than that are set up from multiple memory array building blocks [110]. Other logic that is required for driving the data into the cells (write) or onto a data bus (read) includes decoders, drivers, and control logic.

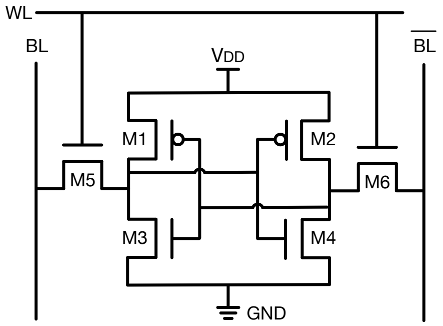


Figure 2.11 Basic 6T SRAM memory cell (based on [110], Slide 7.8)

Dynamic Energy Consumption

Dynamic energy consumption in memories essentially results from reading and writing data. Besides the plain memory cell as described above, some more circuitry is necessary in SRAM for this task. This includes logic for memory control, addressing, but also amplification and driving of access lines. Figure 2.12 depicts a macroscopic view on a SRAM block, illustrating the involved blocks and circuitry. On access, the correct row, column, and block need to be selected based on the applied address. Other circuits drive the data into the cells for write or onto the data bus for read. The combination of all these circuits, including decoders, drivers, and address logic, is referred to as memory periphery.

According to SRAM memory analysis figures in literature, it is especially the peripheral logic and not the memory cell array that accounts for the largest dynamic energy consumption share. In this regard, some references name an energy ratio of about 90 % that can be attributed to the memory periphery [29]. Facing this, especially the well-thought partitioning of the memory subsystem has turned out to be an efficient way to tackle this problem on the architectural level and without sophisticated tuning of electronic or even physical parameters. More detailed, frequently used content is possibly kept in comparatively small memory instances as to reduce the dynamic energy dissipation. Rarely used content instead is mapped to larger memory blocks.

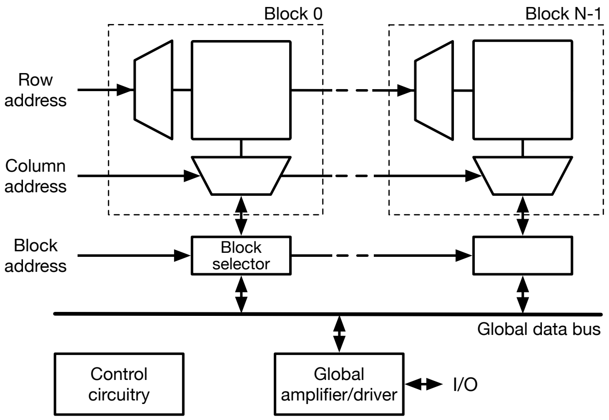


Figure 2.12 SRAM memory block structure (based on [110], Slide 7.5)

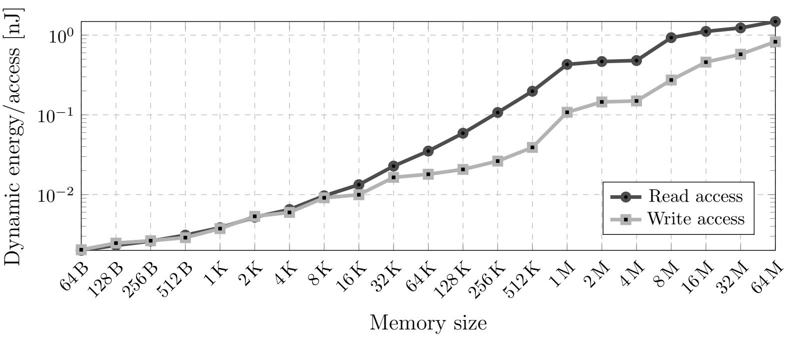


Figure 2.13 Dynamic energy consumption trend for 32-bit single-banked SRAM memories of varying size in 45 nm node (generated with CACTI [95])

Figure 2.13 illustrates this aspect. The given log-log plot depicts the dynamic energy consumption trend in SRAM with increasing memory size on the example of the 45 nm technology node. In the considered range from 64 B to 64 MiB, a steady energy consumption increase can be observed when doubling the memory size between one measuring point and its successor. For both, read and write access, however, the steps are highly variable and range from as few as a 3 % increase between 2 MiB and 4 MiB to more than a doubled dynamic energy consumption when stepping from 512 KiB to a 1 MiB memory instance. This fact further complicates the above defined memory optimization question of finding a suitable memory allocation and application binding (cf. Sections 2.1.2 and 2.1.3).

Static Power Consumption and Low-Power Modes

Besides dynamic power consumption, which accumulates in form of energy portions on memory access activity as just described, there are also static aspects that need to be considered. Still, until recently, most and foremost dynamic parts mattered in terms of power consumption. Regarding optimization of dynamic power, it is particularly effective to reduce the supply voltage levels [65]. Ideally, this approach perfectly aligns with steady technology shrinking, where, according to Moore's law, voltage lowering is allowed or even required. With power consumption being proportional to the square of supply voltage, the resulting reductions from miniaturization thus were enough to

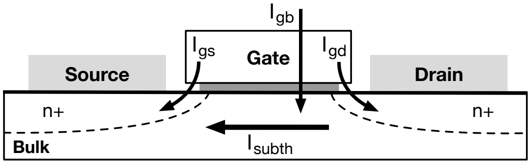


Figure 2.14 Leakage currents in a CMOS transistor (based on [21], Figure 1)

maintain adequate power consumption for a long time. As feature sizes are shrinking below 100 nm, however, leakage currents and resulting static power consumption increasingly gain in importance [67].

In CMOS, leakage describes currents that flow, even though transistors are not switching. According to the authors of [97], this effect is becoming serious as static power is possibly exceeding dynamic power consumption for technology nodes below 90 nm. Figure 2.14 illustrates the two main types of leakage, sub-threshold (I_{subth}) and gate leakage ($I_{gate} = I_{gs} + I_{gb} + I_{gd}$). The former is described as „weak inversion current across the device“; the latter as „tunneling current that flows through the gate oxide insulation“ [67].

In literature, various low-power methodologies for SoC design have been presented. While not all concepts apply to memory, a selection of established and mature low-power solutions for SRAM memory subsystems is presented below, ranging from gate- or cell-level solutions to architectural approaches.

A sleep mode with short wake-up periods, further referred to as *light sleep (LS)*, can be realized through biasing techniques. More precisely, the application of a source biasing scheme results in a raised virtual ground potential that is applied to the complete cell array of a memory bank or instance. Zhang et al. [148] describe this method on the basis of a single SRAM cell as illustrated in Figure 2.15. Two additional transistors (M7, M8) are inserted between the 6T cell and ground (GND) to form a so-called clamping unit. During active operation, M7 is turned on and the source line of the memory cell (V_{SL}) is basically acting as real ground, i.e., conventional operation is possible. If light sleep is enabled though, M7 is turned off and the source line is raised to a virtual ground level. Consequently, sub-threshold and gate leakage are reduced. M8 is turned on in this situation, what clamps V_{SL} to a fixed potential and thus avoids floating and in addition ensures data retention. To sum it up, source biasing describes a method with short reaction times that allows for significant static power reductions of up to 50 % [85]. Memory content is maintained and output lines are kept stable during

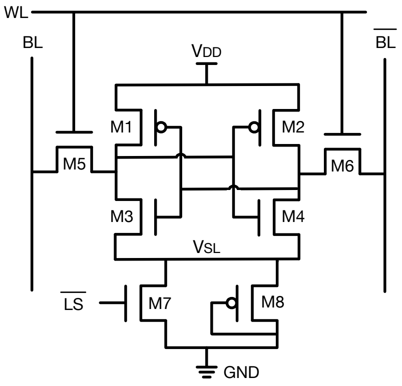


Figure 2.15 Source biasing using a clamping unit (M7,M8) (based on [148], Figure 1)

activation. A complementary low-power mode is adding additional switching logic between V_{DD} and memory cell or array. That means, instead of raising the ground level, supply voltage is lowered to a retention voltage level, e.g., presented in [65] or [141]. Depending on the position of the added sleep mode switch, i.e., source biasing or V_{DD} retention mode, Mohammad et al. [91] talk of head or foot switch.

Another low-power methodology that has established in connection with SRAM memories is power gating [20]. As the memory periphery mainly consists of CMOS logic, it matters not only for dynamic power consumption but also considerably attributes to the static power consumption of the memory subsystem. Separated power gating of this part is therefore a worthwhile option. The memory array is kept active and powered on in this *deep sleep (DS)* mode (cf. Figure 2.16). Data is consequently retained; the memory output lines are reset. Compared to the normal active operation mode, static power can be reduced by up to 70 % [90] with this low-power mode. Wake-up periods instead are higher as compared to the above light sleep modes on the basis of biasing techniques.

As indicated in Figure 2.16, it is also possible to *shut down (SD)* the memory array in addition to the memory periphery. However, stored data is lost in this configuration and possible extra effort needs to be considered on re-activation due to possible restoration of memory content from persistent storage. Following the argumentation of [78], this overhead due to a refill on re-activation from such a non-retention power-down mode is considered to be not practicable for embedded devices. There might be use cases, for example

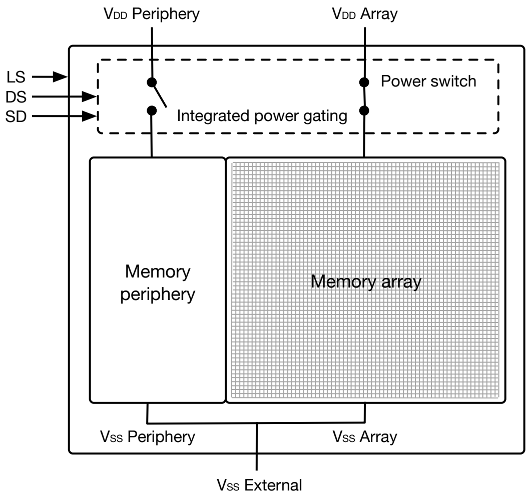


Figure 2.16 SRAM memory with integrated power gating (based on [85], Figure 2 and 4)

when dealing with temporary data that does not need to be restored. For the further course of this work, however, shut down is of no further relevance.

Memory Hierarchies

Especially in data-intensive applications, memory access is considered a central performance bottleneck. Extended access delays not only affect the system performance but also increase the power consumption. Hence, efficient memory design matters and, the most common approach to address this bottleneck is the utilization of a memory hierarchy. Different levels of memory are accordingly used between CPU and main memory. In general, small but fast memories are kept close to the CPU. Down the memory hierarchy in turn, memory instances become larger and slower. In terms of energy consumption, energy efficiency decreases with increasing memory size as previously discussed. Figure 2.17 illustrates four different memory hierarchy schemes that are commonly found in embedded and SoC devices with SRAM memory.

Low-end devices mostly implement a flat memory organization (a). Main reason is a higher importance of low power consumption as opposed to good performance. Main memory in this case is typically represented by SRAM in

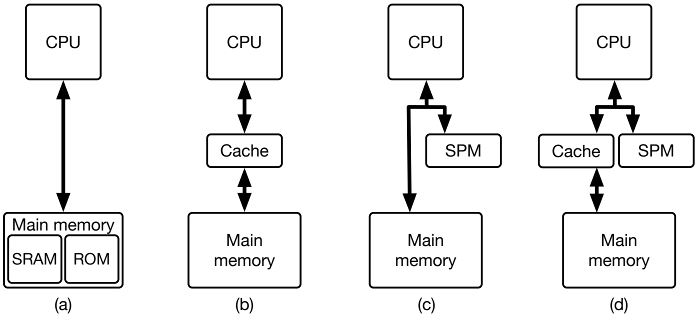
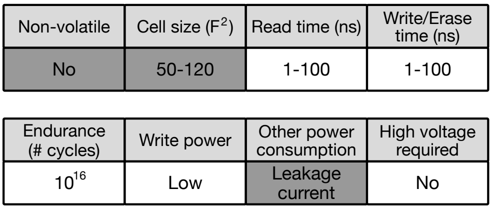


Figure 2.17 Common memory organization alternatives (based on [137], Figure 1)

combination with a non-volatile Read-Only Memory (ROM). With increasing performance demands, different memory hierarchy concepts come into play. Hardware-controlled caches (b) exploit temporal and spatial locality effects of the application and mirror content of the main memory. Positive effects of this concept can, according to Bathen and Dutt [12], for example be reported for database applications. Software-controlled caches (c), referred to as Scratchpad Memory (SPM) instead are said to work well in applications with predictable memory access behavior, e.g., media applications [12]. In scratchpads, the core-local memory content has to be specified by the user, i.e., system designer or programmer respectively. The content of the SPM is, depending on the application, either static and thus determined once at system design-time, or, changed dynamically during operation. The last memory organization (d), a combination of both previous concepts, is increasingly implemented by high-end embedded devices. It should be noted that in all presented alternatives (b,c,d), both, caches and scratchpads are typically subdivided into separate instruction and data parts respectively.

Summary

Finally, the architectural characteristics and technological performance of SRAM are once again summarized in Figure 2.18 from different viewpoints. Dark gray boxes highlight critical points, including the large cell size that results from the footprint of six required transistors per base cell. DRAM in comparison allows cell sizes of 6 to 10 F^2, where F describes the technology node, e.g., in nanometers. Consequently, on-chip area consumption is an im-



Non-volatile	Cell size (F^2)	Read time (ns)	Write/Erase time (ns)
No	50-120	1-100	1-100

Endurance (# cycles)	Write power	Other power consumption	High voltage required
10^{16}	Low	Leakage current	No

Figure 2.18 SRAM performance chart (based on [146], Table 2)

portant subject to optimization. Also volatility brings certain disadvantages as SRAM is sensitive to data loss when the voltage drops below the retention voltage level. In addition, extra non-volatile memory needs to be added to the memory hierarchy for permanent storage and extra overhead is incurred, for example at startup from loading relevant content into the fast operational SRAM memory. Beyond that, leakage increasingly becomes a serious problem as technology nodes are shrinking. Nevertheless, possible solutions for dynamic energy and static power consumption optimization exist.

All in all, SRAM is the central on-chip memory technology in embedded and SoC design. Due to increasing area and power consumption shares, efficient optimization of those aspects is of high interest. With focus on energy and power consumption, especially memory partitioning, for the dynamic part, and application of low-power modes, for the static part, are highly promising concepts in this regard. Still, area footprint and memory access performance are strongly connected parameters that can not be ignored.

2.3.2 Spin-Transfer Torque RAM (STT-RAM)

STT-RAM, more precisely Magnetoresistive Random-Access Memory (MRAM) with spin-transfer torque switching, is an evolving Non-Volatile Memory (NVM) technology. Its benefits are numerous and include low power consumption, especially in terms of leakage, high density, and unlimited endurance [120]. This combination makes it a highly interesting storage technology, which is, according to the authors of [66], the most promising candidate for commercialization among evolving memory technologies at present. Compared to already mature memory types, STT-RAM provides the density of DRAM, the speed of SRAM, as well as the non-volatility property of flash memory. In this section and beyond, STT-RAM is essentially considered as replacement for SRAM in the context of SoC design. Below, a short introduction to the underlying technology is followed by details on some optimization aspect that is based on characteristics of the STT-RAM write operation.

Basics

The basic storage element in STT-RAM is called Magnetic Tunnel Junction (MTJ). This sub-100 nm magnetic element typically consists of three layers as depicted in Figure 2.19. That is, two ferromagnetic layers, separated by a thin insulating oxide layer (of magnesium oxide (MgO) for example). The bottom layer with stable magnetic orientation is named reference layer. On the opposite side, a so-called free layer with variable magnetic orientation is used to represent the information that is stored in the cell. Both layers having the same orientation is referred to as parallel state (P) and representing logic state 0. Reverse orientation and thus logic state 1 is named anti-parallel magnetization (AP). Access to this type of storage cell is implemented using an extra NMOS transistor as illustrated on the right hand side of Figure 2.19. When reading the current state of the cell, a small current is flowing through the bit cell in order to sense its resistance state. Writing instead requires a much higher current to flow for the duration of a given write pulse. This write current, referred to as I^w, has to be large enough as to change the magnetic orientation of the MTJ. The direction of the current allows to control the final magnetic orientation of the cell as illustrated in Figure 2.19.

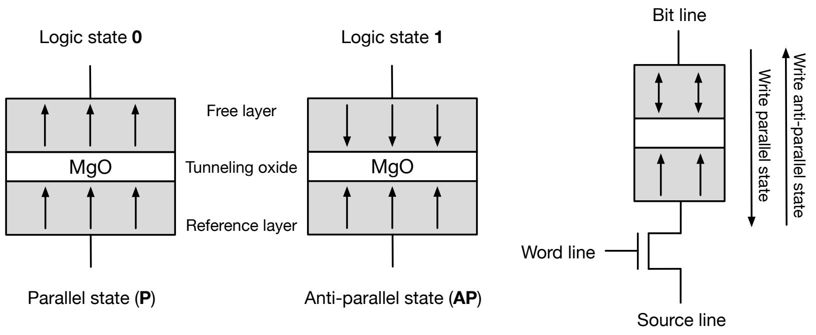
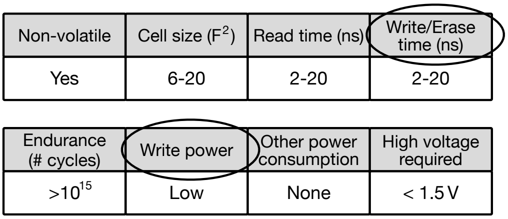


Figure 2.19 STT-RAM bit cell structure (based on [4], Figure 1 and [115], Figure 1)



Non-volatile	Cell size (F^2)	Read time (ns)	Write/Erase time (ns)	Endurance (# cycles)	Write power	Other power consumption	High voltage required
Yes	6-20	2-20	2-20	$>10^{15}$	Low	None	< 1.5 V

Figure 2.20 STT-RAM performance chart (based on [146], Table 2)

A summary on the technological performance of STT-RAM as described by Wolf et al. [146] is given in the following Figure 2.20. In direct comparison to the above SRAM characteristics (cf. Figure 2.18), the essential shortcomings of SRAM with respect to volatility, cell size, and leakage are resolved in STT-RAM. Yet, there is another aspect that needs further attention. Especially the write operation matters in this technology and intensive research on its optimization, mainly on low physical and gate levels is and has been conducted. First concepts and memory cell designs with reasonable write power and write time have already been presented, e.g., by Kishi et al. [68]. Still, there is a direct trade-off between these two values, which is an interesting aspect for high-level optimization as discussed in this thesis and thus worth a closer investigation.

Write Energy/Latency Trade-off

The write operation is, according to Sayed et al. [115] the main bottleneck in STT-RAM concerning performance, energy, and reliability. In this regard, the following two parameters play an important role:

- The thermal stability factor Δ describes the energy barrier that must be overcome in order to flip the magnetic orientation of the bit-cell. This physical parameter is directly related to the retention characteristics of the memory, i.e., the time, the memory cell is able to preserve stored data. In short, lower retention time means lower energy barrier.
- The Write Error Rate (WER) characterizes the memory in describing the write operation's quality. It is due to the stochastic behavior of the MTJ when changing its magnetic orientation. This involves that writing in STT-RAM is not symmetric, i.e., switching of the free magnetic layer to anti-parallel state (AP) takes significantly longer than switching to the

parallel state (P). At system level, however, write time must be set to a fixed value that guarantees an acceptable error rate as to be able to speak of a reliable memory.

In [94], the authors present a model that relates the above parameters to write current and write latency. Using this, Sayed et al. [115] show that with fixed values for Δ and WER, an increasing write current (and thus write energy consumption) leads to a write latency reduction. It can be concluded that once the low-level parameters of the memory are specified, the necessary energy to write the cell depends on the trade-off between writing speed and required writing current level. In other words, writing with high current is fast, reducing the current level prolongs the operation's delay instead. This relation finally allows for the definition of different working points with different performance to energy consumption ratio.

Summary

In the presence of restricted energy budgets, for example, this aspect is highly interesting for system-level memory optimization because static or even dynamic assignment of different working points might help to meet specified constraints. Sayed et al. [115] propose to use additional circuitry for online write current adjustments. However, also voltage scaling techniques allow for equal adjustments and are further considered to be more common in low-power SoC design. Figure 2.21 illustrates this trade-off in showing how

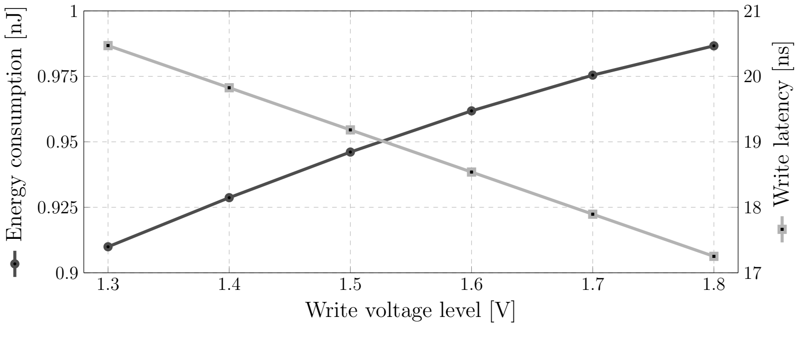


Figure 2.21 Write energy/latency trade-off for a STT-RAM memory of 4 MiB in 45 nm node (generated with STT-CACTI [4] [5])

different operation voltage levels lead to increasing write energy consumption. Simultaneously, write latency goes down when keeping the thermal stability Δ and the error rate constant.

Altogether, STT-RAM is an evolving memory technology with hardly any disadvantages. While already close to commercialization, most improvements in recent literature discuss gate-level or even physical aspects with no use for system-level optimization. Yet, especially the trade-off between memory write latency and energy consumption is very interesting for optimization also on higher abstraction levels.

Chapter 3
Related Work

In the field of embedded system and SoC design, the memory subsystem and its optimization have been and still are subject to intensive research. For that reason and due to the high variety of system types and requirements in the embedded domain, there is a significant amount of related literature.

With focus on SRAM memory, this chapter gives an overview of existing work and contributions to the state-of-the-art in this research domain. Considered related work covers solutions and optimization techniques for the efficient handling of previously discussed memory organizations according to Figure 2.17. This includes flat memory structures as well as hierarchical memory architectures with caches and or scratchpad memories. Presented work deals, amongst others, with design space exploration or the optimization of energy consumption, on-chip area, or performance. Also combined considerations of some of these aspects can be found in literature.

Inspired by the definition of sub-problems in memory optimization as allocation, binding, and scheduling tasks according to Section 2.1, the taxonomy in Figure 3.1 allows for a rough classification of this research field. First, and starting at the bottom left in this diagram, state-of-the-art in hardware-based concepts is introduced in Section 3.1. This includes work that affects the hardware design through memory allocation, partitioning, or required additional circuitry for the implementation of a specific binding for example. Methods

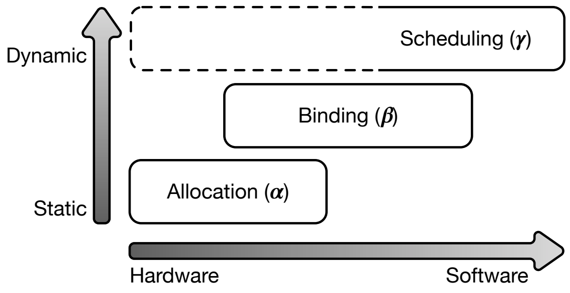


Figure 3.1 Related work classification for memory optimization in embedded systems

that are part of hardware synthesis or based on hardware description languages also belong to this group. In general, respective solutions are typically of static type, i.e., applied and implemented at system design-time. Next, section 3.2 discusses software solutions, i.e., approaches that work without modifications of the embedded hardware. Such concepts are dealing with binding or scheduling problems and often implemented as compiler extension. While solutions that affect the hardware level in any form are rather of static nature and mostly applied at system design-time, utilization of software allows more dynamic approaches and solutions with run-time adaptions. The remaining sections of this chapter on related work discuss dedicated optimization methods for multi-core systems (Section 3.3) and memory subsystems with STT-RAM (Section 3.4). Finally, Section 3.5 concludes this literature overview and summarizes the most relevant aspects in relation to the contributions of this thesis.

3.1 Hardware-Based Concepts

All work as discussed in this section is either evaluating hardware aspects, affecting the memory architecture through allocation steps, or implementing a determined binding or memory operation mode schedule in terms of dedicated circuitry, e.g., as address decoder.

Design Space Exploration

Design Space Exploration (DSE) is typically applied in early design stages. Figure 3.2 schematically illustrates the basic concept and main intention, which is to facilitate design decisions through combined consideration of different system characteristics and constraints. In the context of memory architectures, the following contributions to the state-of-the-art can be named.

Panda et al. [102] propose a design space exploration strategy for the performance evaluation based on application analysis. This includes the consideration of arrays and loops; potential memory components are on-chip scratchpads or data caches as well as (off-chip) main memory. The authors of [71] investigate multi-banked memory architectures for mobile and portable embedded devices. With focus on energy consumption, they present a mathematical energy/cost model for the analysis of relevant parameters and the system design space. This includes, most and foremost, memory bank parti-

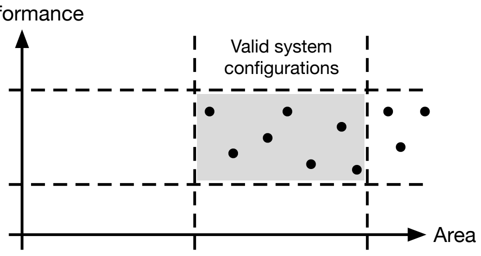


Figure 3.2 Exemplary system design space with constraints

tioning and the consideration of waiting states. Periods where the memory is holding garbage as well as power state transitions with resulting penalties are respected in this context. Another solution for memory DSE is proposed in [47]. The authors aim at finding the best memory hierarchy on the basis of application-specific profiling information. This step is intended as part of a hardware design flow and considers address ranges, number, size, and type of on-chip caches. Down the memory hierarchy, SRAM, DRAM, and non-volatile memories, e.g., EPROM are supported.

Altogether, these approaches aim at the rather general evaluation and comparison of different system setups, for example with respect to system performance and on-chip area as depicted in Figure 3.2. Methods as proposed in this thesis put the focus on a single metric and its optimization instead. This allows, in contrast to the above references, to determine an ideally optimal system configuration, for example in terms of energy consumption.

Methods for Digital Signal Processors

Digital signal processors are widely adopted for highly data-intensive applications and in most cases realized based on the Harvard architecture with separated memory and bus for program code and data. This makes separate optimization feasible, while especially the data memory part is of interest. The following collection of related work mainly addresses this fact through special optimization of data flow and memory architectures in Digital Signal Processor (DSP) designs. Work in this thesis, however, is targeting systems that implement the von Neumann architecture. Direct comparison is therefore limited as program code and data are not separated in these type of systems but share the same memory and pathways.

Zhuge et al. [151] present a graph-based solution to exploit the design space with multiple memories. Using a DFG representation, the authors identify memory access patterns, which allow for the definition of a partitioning problem that results in an optimized memory subsystem architecture. On top, data parallelism between individual variables is considered. Another solution that works on a Data Flow Graph (DFG) representation and thus suited for application at higher abstraction levels is given in [92]. A heuristic for application-specific memory partitioning based on profiling data under a constraint that limits the number of banks is presented in [84]. The authors of [144] take instruction-level parallelism in DSP platforms into account. That means, on top of a graph model and iterative solution to memory partitioning, instruction scheduling is added to the equation. While respecting performance constraints, energy savings are set as optimization goal by maximizing idle intervals and thus the low-power mode time of individual memory blocks. In [109], an ILP-based solution for the performance versus energy consumption problem in memories is presented. It is again based on a data flow graph and includes different optimization aspects with data-to-memory assignment, DSP instruction scheduling, as well as the handling of different memory operation mode settings. A hybrid memory system consisting of DRAM and Phase-Change Memory (PRAM) is considered in [76]. The application for the digital signal processor is required in form of a graph. The presented ILP-based optimization enables the evaluation of trade-offs, caused by write operations to the PRAM memory. These are beneficial in terms of power consumption but in turn, have a negative impact on the system performance. Balasa et al. [9] [10] focus on signal processing in real-time and data-intensive multimedia applications. The proposed Electronic Design Automation (EDA) solution deals with high-level design of hierarchical memory architectures using a formal model and further bases on a behavioral specification of the application. Again working with a behavioral specification, follow-up work of these authors adds scratchpads and low-power memory states to the consideration [8].

Scratchpad Partitioning

The work of Benini et al. [13] [14] is one of the first to consider on-chip SRAM memory partitioning. The proposed optimal recursive partitioning algorithm works with execution profiles and is applied for the determination

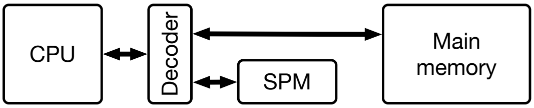


Figure 3.3 System architecture with scratchpad and address decoder (based on [3], Figure 1)

of an application-specific optimal scratchpad partitioning into banks [14]. The maximum number of banks is a constraint; the optimization objective is energy consumption reduction. The hardware-based implementation of the obtained optimization result is realized using an dedicated address decoder as illustrated in Figure 3.3. This circuitry implements the resolution of memory accesses, determination of address ranges, and redirection to the correct memory bank [13]. Later work by Angiolini et al. [3] deals with scratchpad mapping, which is in fact a binding problem. The proposed solution is based on memory access traces; dynamic programming is used to realize the optimization model for the identification of frequently used memory content that shall be mapped to the SPM. This way, a compromise in terms of energy consumption, die area, and system performance can be identified. The proposed integration of obtained results into a system design at hand is again following the scheme in Figure 3.3. In [122], combined partitioning of bus and memory architecture is discussed. The presented solution is again application-specific and based on a genetic algorithm. It considers memory aspects but also includes additional wiring for the interconnect when memory is partitioned in multiple segments. Other work that is motivated by image processing applications deals with automatic memory partitioning as part of a behavioral synthesis flow [28]. The presented optimized arrangement of data arrays in combination with memory partitioning allows for savings in terms of throughput and energy consumption.

In contrast to these solutions for allocation and binding, presented methods in this thesis not only focus on partitioning but allow for the optimal identification of α and β in such and even more heterogeneous memory subsystems. Proposed software-based concepts for the realization of obtained optimization results further allow to avoid overhead from dedicated address decoders.

Hardware-Based Low-Power Mode Scheduling

The following publications deal with memory low-power modes. Obtained schedules are, in any case, realized on the hardware level.

For hybrid memory subsystems with caches and scratchpad, Wen et al. [145] discuss the impact of low-power modes for cache lines. Depending on the idle time of a cache line, a power mode for either short or long intervals is chosen. Lin et al. [75] propose a hardware extension that aims at reducing the energy consumption through an address-aware memory state predictor, which is based on lookup tables and hardware counters that keep track of the memory access pattern. Using this, inter-access times are predicted and used for a controlled activation and deactivation of low-power modes in DRAM memories. This way, overhead from resynchronization in DRAM can be reduced and presented experiments prove the suitability of this method for multimedia applications. In [147], video coding applications are considered as use case. The proposed memory optimization method comprises the evaluation and cost-function-based determination of a multi-banked memory architecture in combination with an application-specific power mode management. Multiple memory sleep modes are available in the search space and mapped to idle memory blocks while considering wake-up overheads. Another, statistical method for video coding applications is presented in [114]. This management of scratchpad memory power states is based on application-specific information as available at system design-time. Adjustment of memory sleep states for individual memory blocks is further possible based on online predictions. The required SPM access management unit is implemented in hardware. Loghi et al. [78] present a leakage-aware memory partitioning method, which includes the consideration of memory sleep states that are implemented using voltage scaling. Temporal locality effects are extracted from input memory traces; address decoders are used to realize the optimized scratchpad partitioning. The proposed optimization method is based on exhaustive search and also considers introduced penalties from additional wiring and the SPM decoder overhead. Solutions are proven to be optimal with respect to the provided memory traces. Follow-up work of Steinfeld et al. [124] also deals with banked SRAM memory subsystems that support low-leakage sleep modes. Their focus is on event-driven applications that possibly come with long idle periods. The proposed optimization method is based on a greatly simplified energy model. Memory blocks are all of equal size. The impact of sleep modes is evaluated on the basis of two hardware-based power management concepts. First, a greedy solution that puts memories directly in a low-power mode when becoming inactive. Second, a fixed time-out policy.

In contrast to Loghi et al. [78] and Steinfeld et al. [124], this thesis proposes a method that not only considers operation mode scheduling but allows for the combined determination of allocation, binding, and scheduling. Further demarcation is given by the consideration of peak power, which is not part of any of the above concepts. Further, it should be noted that with more and more projects in the embedded field being software-dominated, the hardware platform is often pre-defined or simply chosen off-the-shelf. The solutions in this thesis are, if possible and thus except for results from allocation steps, all software-based. Therefore and in contrast to above presented work, no extra hardware as for example decoders is needed. This point directly transfers to the next section, where software-based related work is discussed in more detail.

3.2 Software-Based Concepts

With respect to the above given general classification in Figure 3.1, optimization concepts as presented in this section cover binding and scheduling problems; proposed methods range from static to dynamic. In general, static software-based solutions are based on design-time optimization steps that are typically applied at compile-time, e.g., through a compiler extension. Dynamic solutions instead are for example realized as run-time library that completely works online. The following overview starts with binding concepts while moving from dynamic to static solutions. Subsequently, scheduling methods for the handling of SRAM low-power modes are discussed.

Dynamic Binding and Run-Time Memory Management

The following collection of memory management methods is divided in two parts. First, fully dynamic solutions with online evaluation and application of measures are discussed. Second, partly dynamic solutions are presented. That is, rules for memory modifications are statically pre-computed at design-time and then applied at run-time.

Luz et al. [81] propose an online data migrator for multi-banked memory subsystems. The run-time library that implements this migration scheme depends on temporal access information that is collected during execution. The sampled data set is stored in decision tables and used to adjust the trade-off between power consumption and resynchronization cost in DRAM memories. Possible activities include memory-to-memory copies or the activation of low-power modes. Main application fields are data-intensive and array-dominated applications as common in image or video processing. The authors of [62] and [118] present a dynamic SPM management solution for stack data only. It is exclusively applied at run-time and implemented as so-called scratchpad manager that is added to the software at hand. Whether energy and performance savings are possible with this method highly depends on the application as code overhead from the SPM manager needs to be considered on top. The solution of [93] also works completely without compiler assistance, profiling information, or hardware support. Instead, the efficiency of the proposed run-time management for SPMs depends on the programmer's knowledge of the application. Annotations on source code level are used accordingly to select the most appropriate scratchpad content. Chen et al. [25] present a solution for dynamic binding of code and data to the system's scratchpad. Starting point in this approach is a code repositioning step that aims at improving the memory access locality of the executed program. The proposed online management is implemented as interrupt service routine that takes care of data swapping to and from the scratchpad.

Partly dynamic run-time management is for example investigated by Kandemir et al. [56] [60]. They present an on-chip scratchpad memory management scheme on the basis of loop and data transformations. The main focus is on embedded image and video processing applications, where handling of array-based data structures is dominating. Their SPM scheme is pre-computed but applied at run-time, i.e., data transfers are carried out online. How to maximize the reuse factor, i.e., how to minimize the data transfers between main memory and scratchpad is discussed in [58]. Targeting multimedia applications, the work in [1] deals with the optimization of irregular memory accesses and indirectly-accessed array data structures. The proposed method is based on spatial and temporal locality access patterns as collected at compile-time. Using this, a cost model is used to determine the trade-off between array size and resulting cost for moving a block into the SPM. Benefits from data reuse are also considered in this model. Inserted code finally implements this partly dynamic scratchpad management scheme into the application. A fully automated SPM code management on the basis of profiling information and binary analysis is presented in [32]. The proposed solution is tailored for systems with instruction scratchpad. A Mixed-Integer Quadratic Program (MIQP) formulation is used at design-time to select static scratchpad content and dynamic parts that are loaded and unloaded to and from the SPM at run-time. Hu et al. [48] discuss the possibilities with hybrid scratchpads that consist of SRAM, on the one hand, and an ultra-low leakage non-volatile memory, on the other hand. The presented dynamic data mapping algorithm is based on input from simulation and respects the differing characteristics of both memory technologies for access time, dynamic energy, and leakage power reductions.

In sum, all of the above methods propose a solution for the dynamic application or modification of application to memory binding. In comparison, all methods in this thesis that involve a binding step can be classified as static, which is why there is no basis for comparison here.

Optimization in Real-Time Scenarios

Further related work for partly dynamic scratchpad binding with additional consideration of real-time constraints is discussed next.

The work in [133] considers real-time systems with strict timing limits.Worst-Case Response Time (WCRT), multi-tasking, and scratchpads that are shared among different tasks are defined as outer conditions. This, however, requires the application to be modeled as Message Sequence Chart (MSC), which allows predictable analysis. Scratchpad loading and especially reloading at run-time is scheduled at pre-defined execution points. In [139], the authors present a scratchpad handling scheme that satisfies real-time requirements. Through compiler-inserted code at specific points, predictability is maintained. The presented binding scheme comes with low overhead, yet provides a possibility for run-time adjustments. Comparable to previously discussed solutions, most frequently used global and stack data is mapped to the scratchpad. Also coupling with a Direct Memory Access (DMA) unit for better performance is discussed in this work. The authors of [136] propose a scratchpad management scheme for priority-based preemptive multi-tasking systems. Their solution targets applications with Real-Time Operating System (RTOS) and comprises different ILP formulations for different memory access patterns, including temporal, spatial, and mixed locality. The scratchpad is subdivided among the individual tasks; the SPM content can be swapped using the operating system and with help of a DMA controller depending on the pre-computed ILP results. Another SPM management solution that includes DMA in order to reduce the performance overhead from online data migration is proposed in [37]. An evaluation of data scratchpads as alternative for data caches in real-time applications with predictability requirements is given in [61]. In this work, the authors propose a heuristic for data to scratchpad binding that aims at performance optimization in the context of preemptive real-time systems.

While the methods in thesis also provide highly predictable optimization results, typical real-time constraints in terms of execution time are out of the scope of this work. Instead, restrictions in terms of on-chip area and peak power consumption are considered.

Static Binding Methods

Static methods for optimized application mapping as presented below can be distinguished in methods with focus on program code only, others that perform data placement only, and lastly solutions that consider the combined binding of both, program code and data.

The authors of [117] focus on code only and present a function block to memory mapping heuristic for problem settings, where the application is available as control flow graph. Also here, execution trace information of the application is required as input.

Considering data-only mapping, early work in the field of compiler-assisted memory optimization was conducted by Panda et al. [101]. This includes the distribution of scalar and array variables into scratchpad and main memory using a sorting-based heuristic. According to this approach, variables that cause the most cache conflicts are statically mapped to the scratchpad. Handling of recursive data access in the context of scratchpads is discussed in [30], where based on a fixed stack frame size, data to scratchpad mapping is executed at compile-time. In [7], heterogeneous memory architectures are considered, which includes scratchpads, internal and external DRAM, and ROM. The proposed automated compiler extension optimizes the distribution of global data as well as heap and stack using an ILP. The obtained solution is optimal, relative to the profiling data that is used as input. Integration of the computed data placement into the embedded software binary is statically implemented using the linker. In [42], a polynomial-time data placement algorithm for scratchpads is presented. In minimizing the memory access cost, this dynamic programming solution mainly targets system performance. A static solution for data to scratchpad mapping with compiler support is presented in [11]. This work evaluates scratchpads as alternatives to traditional caches in compute-centric applications and puts special attention to trade-offs in terms of area and performance.

While the above methods either consider code only [117] or exclusively deal with data [7] [30] [42] [101], the authors of [125] present another compiler-integrated solution that allows for an automated selection of program and data parts and their placement in the SPM of the system. The approach is based on application analysis; the optimization step is realized as ILP. The scratchpad content is statically assigned at design-time, hence, no dynamic content loading takes place in this method. In fact, most frequently used content is identified on the basic block level and selected for a placement in the system's scratchpad. Verma et al. [140] further extend this concept through partitioning of arrays. A code and data to scratchpad mapping solution that allows uncertainties about the actual memory sizes is presented in [99]. The method is based on pre-processing and analysis at compile-time. Most frequently used elements including code, global variables, and stack data are considered for a placement in the scratchpad. Implementation of the actual data placement step into the system's bootloader increases the portability of this solution. The work in [142] evaluates the differences of dynamic (overlay) and static (non-overlay) scratchpad mapping methods, showing a better performance of non-overlay SPM utilization, at least for the widely used set of MiBench embedded benchmark applications [43]. In [149], hybrid memory systems with cache and SPM are considered. Memory access profiling information is used as input for an ILP solution that aims at cache miss reduction and optimized scratchpad mapping for either performance or energy improvements. Modified linker scripts are utilized in a recompilation step to implement the optimization result on the software level.

While different solutions for the determination of a static binding are also discussed in this thesis, none of them considers binding as stand-alone problem. That means, in comparison to the above solutions, either allocation of memory instances or allocation plus scheduling of memory operation modes is considered on top and even more important, in combination with the binding problem. This is highly important as α, β, and γ are tightly connected.

Dynamic Power-Down Concepts

Beyond binding, several software-based solutions have been proposed for the determination of an optimized scheduling. This handling of memory operation modes is partly pre-computed but always dynamically applied at run-time. The set of SRAM operation modes as previously introduced in Section 2.3 is once again summarized in Figure 3.4, along with corresponding characteristics in terms of optimization potential or impact respectively.

Regarding power-down concepts, the authors of [82] put their focus on applications that frequently allocate and deallocate data on the heap. Es-

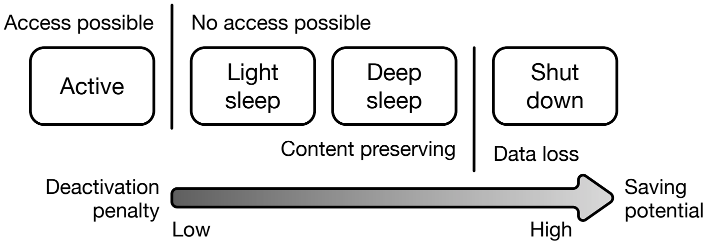


Figure 3.4 Overview of SRAM operation modes and their characteristics

sentially, they propose an alternative implementation of `malloc` and `free` function that can be automatically inserted by the compiler. This allows for an energy-efficient dynamic data migration scheme that aims at occupying a small number of memory banks. The remaining banks are completely shut down for energy savings. Chen et al. [41] present an integer linear program that allows for a trade-off evaluation between loading content into a SPM bank or turning it off while keeping the corresponding data in the main memory instead. The authors of [88] turn the idea back and propose a trace-based optimization method that aims at maximizing the power-down time of the main memory while temporarily operating from the scratchpad only. The focus is on code and constant data; flash is used as part of the memory subsystem as to avoid data loss; the optimization model is formulated as ILP.

It is important to note that shut down in SRAM is not content preserving, i.e., leading to data loss (cf. Figure 3.4). Due to the obviously high overhead from re-loading complete memory blocks from non-volatile off-chip memory, this thesis only considers content preserving low-power modes.

Software-Based Low-Power Mode Scheduling

Instead of completely shutting down individual memory blocks (cf. [41] [82] [88]), the following publications deal with low-power modes and corresponding activation schemes.

A compiler-directed solution for variables to memory bank assignment and determination of memory bank operation modes is presented in [83]. The heuristic solution works with DRAM memory and takes four different memory operation modes into consideration. This includes active, standby, nap, and power-down. In order to apply this approach, however, the problem has to be modeled as graph. A compiler-based leakage reduction scheme for instruction scratchpads is presented in [52]. Idle scratchpad memory is accordingly put into a low-power mode using a special instruction. Same holds for the reactivation step, which has to be executed in time as to keep the performance impact low. The work of Kandemir et al. [57] deals with the reduction of static power from leakage currents in SPMs. The proposed compiler-based solution assumes the memory to be divided into banks. In order to reduce the static power consumption from leakage, scratchpad content is dynamically changed at run-time as to maximize the idle times of individual memory banks. That way, the impact of low-power mode activations can be maximized. When and what data to map to memory and individual banks is determined statically at compile-time using a linear algebra framework. As this framework works on an intermediate code representation, low-level and library code is not supported.

The method for low-power mode handling as proposed in this thesis is, compared to [57], less dynamic and therefore more deterministic, considering memory allocation on top and thus a larger design space, supporting peak power constraints, and further able to support low-level and library code.

3.3 Memory Optimization for Multi-Core Systems

With multi-core designs being more and more common in the embedded field, memory optimization experiences a renaissance. Previous work that was originally designed for single-core systems is ported, newly combined, and evaluated on recent multi-core platforms. Furthermore, new methods and ideas are presented as follows.

Graph-Based Optimization

The work of [104] investigates co-synthesis of memory and communication architecture in MPSoCs on the basis of graph-based modeling. The proposed solution is application-specific and aims at minimizing the number of busses while respecting area and performance constraints. A second optimization goal is the reduction of memory area; experiments are presented for applications from the networking domain. The handling of stream-processing applications on multi-core architectures with SPM is discussed in [24]. This work is again based on graph-based modeling and expects an architecture graph for the platform and a Synchronous Data Flow (SDF) graph representation for the part of the application. Optimization objective is maximizing the streaming throughput by means of re-timing and with respect to memory constraints. The ILP and alternative heuristic solution in [40] deal with simultaneous task-to-core and data-to-scratchpad assignment. Scratchpads are multi-port, meaning each core is able to access its local as well as remote scratchpads of other cores. Required input is a Directed Acyclic Graph (DAG) model and a data access table that reflects the memory access behavior of individual tasks in the system. Main optimization goal in this work is performance, visible in the obtained schedule length.

Multi-core memory optimization in this thesis does, in contrast to these publications, not depend from any specific graph-based model of memory architecture or application and is therefore more generally applicable.

Dynamic Concepts

Alvarez et al. [2] propose a dynamic method that completely works at run-time. Main target are multi-core systems with hybrid memory subsystem, i.e., caches and scratchpads. The proposed dynamic solution consists of a run-time system that collects required information on-the-fly and using that, manages the scratchpad transparently to the user. Compared to a system with caches only, this setup is experimentally proven to be faster at less power consumption. Another run-time SPM mapping concept with focus on performance improvements is the work in [135]. The proposed workload-aware scratchpad pool manager depends on partly pre-computed input from a heuristic algorithm as based on profiled data. A SPM supervisor that aims at improving the handling of local and remote SPMs in a multi-core system is proposed in [23]. The method is inspired by traditional caches and involves the online recording of the memory access behavior, based on which data is moved at the granularity of complete pages. The work of Paul et al. [105] is based on hints of the programmer, on the one hand, and the source code of a multi-threaded application, on the other hand. Using this information, the run-time system is instructed in what to load to the core-local scratchpad memories. Mainly targeted application field of this paper is audio and video processing. In [103], a dynamic scratchpad mapping scheme assisted by a Memory Management Unit (MMU) is proposed. With focus on dynamic stack management and implemented using the exception handlers of the MMU, this method can be realized on top of other optimization steps.

Similar to the above presented dynamic concepts for single-core embedded systems, these concepts are mainly applied at run-time and thus not comparable to the mostly static optimization concepts as presented in this work.

Static Concepts

In [59], Kandemir et al. extend their compiler-based scratchpad management strategy [58] to multi-processor systems. Again, array-dominated embedded applications are the main use case. Central concept for energy and latency improvements is data reuse, e.g., when multiple processors work on the same array or data set. The authors of [86] present research for multi-core SoCs with shared memory architecture and shift the optimization objectives towards minimizing access cost, memory area, and access time. The proposed model for an optimal but application-specific shared memory block allocation and shared data binding is realized as Integer Linear Program (ILP). The results of this step are finally provided as architecture-level description for further use in an existing MPSoC design flow. In [100], a twofold MPSoC memory optimization method is presented. This includes a compiler-based application analysis for the detection of inter-core relations in a first step. Afterwards, an ILP formulation is used for the determination of on-chip memory sizes, sharing policy among the cores, and data binding. A straight-forward scratchpad mapping heuristic that statically selects the most frequently used elements is presented and evaluated for MPSoC architectures in [51].

In comparison to these static binding methods, this thesis contributes with an optimal solution for the binding of application to MPSoC memory subsystem. Beyond that, scheduling in memory operation modes is considered, an aspect that is not found in any previous work for multi-core platforms.

Hybrid Memory Subsystems

Multi-core devices with hybrid memory architecture are considered in [49]. This includes fast access SRAM, global DRAM, and ultra-low leakage NVM with high density. The proposed heuristic algorithm for data to memory binding exploits the pros and cons of the different memory technologies for memory access time and energy consumption reductions. The proposed solution is application-specific and solves in polynomial time. Wang et al. [143] evaluate the possibilities of hybrid scratchpad memories, consisting of SRAM and

NVM. The presented algorithms for scratchpad binding allow either a fully energy-optimized solution, or, a balanced one that works under constraints and has an eye on energy consumption and system performance. This is handled through the controlled utilization of write operations to the NVM.

The consideration of memory optimization in multi-core systems in this thesis is limited to on-chip SRAM. With STT-RAM, however, one prominent emerging non-volatile memory is separately considered as follows next.

3.4 STT-RAM Optimization

Besides research on the costly write operation in this technology, which enables certain optimization potential (cf. Section 2.3.2), some other concepts have been subject to investigation as summarized below.

Low-Level Optimization Concepts

The fact that STT-RAM is still subject to ongoing development can be seen from related research. In fact, many publications on this topic are dealing with low-level optimization of the memory cells on gate or even physical level. This includes Smullen et al. [120], who propose to adjust certain parameters of the memory cell in order to relax non-volatility in exchange for better dynamic energy consumption and memory access performance. The work in [33] proposes an error rate reduction method on the circuit level. The authors of [55] contribute with the identification of critical parameters in STT-RAM development regarding the design goals: higher density, lower power consumption, and shorter latency. Bishnoi et al. [16] instead propose a circuit-level technique that avoids unnecessary write operations.

System-Level Optimization Concepts

Existing solutions on the system level that consider STT-RAM as given memory block with given characteristics includes [134]. The presented garbage collection scheme for object-oriented languages is based on varying the retention time. Key idea is to categorize allocated objects based on their lifetime and assign them to different memory regions with different retention time. A compiler-based memory mapping scheme for STT-RAM scratchpads that also uses varying retention times in different memory regions is presented in [112]. Lifetime aspects are used as indicator, further supplemented by data locality information and iteration bounds of investigated applications. In [70], a dynamic retention time scheme is evaluated for caches. With multiple STT-RAM instances of different retention time characteristics each, a dynamic refresh scheme can be realized while saving energy and gaining performance. Sayed et al. [115] present a hardware-based technique that makes use of additional circuitry for the run-time adjustment of write currents. This way, the STT-RAM can be operated in different energy-performance trade-off states, which allows a well-suited and analysis-based reaction to different workload scenarios.

In extension to the STT-RAM write operation trade-off as exploited in [115], this thesis contributes to memory optimization in STT-RAM by the additional consideration of allocation and binding.

3.5 Summary

Benini et al. [15] state: „Contemporary system design focuses on the trade-off between performance and energy consumption in processing and storage units, as well as in their interconnections.“ With main focus on storage units and partly interconnects, this sentence precisely summarizes the motivation for most of the above presented related work. This also holds for the contributions of this thesis, which go most and foremost into the direction of energy consumption reduction. With respect to the above presented state-of-the-art, these individual contributions are now finally classified and differentiated from other work as follows.

This thesis provides a solution for four different problem settings that all have certain aspects in common. With respect to Figure 3.1, all methods are static and of application-specific type as based on profiling information. Thus, all steps are completely applied at design- or compilation-time respectively. This setup is perfectly suited for devices and scenarios of deterministic type, with for example repeating tasks, high reliability, or strict constraints in terms of energy consumption. As a consequence, there is no need for a comparison with highly dynamic run-time-only methods or non-application-specific solutions. Beyond that, no specific, for example graph-based input format (cf. [104] [117]) is required. Instead, the application software represents the most important input. Impact on the hardware design, however, partly exists. This includes memory partitioning, certain required memory low-power mode features, or control registers. For that reason, a categorization of this work in partly hardware-based and mostly software-based is the most appropriate one. In sum, especially hardware/software co-design projects are the ideal environment for the application of presented methods.

The first method [126] provides a combined solution for optimal memory partitioning and application code and data to memory binding on the basis of profiling data. While presented hardware-based methods (cf. Section 3.1) lack the binding part, software-based solutions (cf. Section 3.2) typically assume a pre-defined memory architecture or scratchpad.

The two-stage optimization method in [129] [132] extends the above idea. Instead of putting the optimization focus on dynamic energy consumption from memory accesses only, this contribution additionally considers static energy as mainly caused from leakage currents in SRAM. Compared to state-of-the-art, no hardware-based activation scheme is required as in the work of Loghi et al. [78] or Steinfeld et al. [124]. Software-based techniques instead leave the optimization potential from memory partitioning out (e.g. [52] or [57]). Another distinguishing point is the possibility to define peak power constraints, which is supported by this method but not found in related work at all.

In the multi-core context [131], the above observations hold too. Beyond that, a trend towards software-only solutions can be observed. More computational power from multiple cores further encourages for run-time methods that, however, are typically not suited for highly constrained and restricted devices as investigated in this thesis.

The proposed STT-RAM optimization method in this thesis can be, if any, put into relation with [115]. Besides the common motivation of exploiting the write operation for different energy-performance trade-offs, however, the differences prevail. Instead of a heuristic write-rate analysis, this thesis presents a mathematical optimization model, implemented as quadratic program. Further, Sayed at al. [115] use additional circuitry for write current modifications as opposed to voltage scaling techniques in this work.

Beyond that, it is the way of implementing obtained optimization results on the software level that distinguishes this work from others. Instead of special hardware solutions (e.g. [13] [78]) or modified linker scripts (e.g. [149]), a modified compiler backend is used in this work [130]. This automated solution is transparent to the user and further enables direct integration into existing hardware/software design flows.

Chapter 4
Memory Optimization

The following chapter is on the SoC memory optimization methods that represent the main contribution of this thesis. All below sections are structured the same way and describe one approach each: First, a general optimization context and problem statement is given. In accordance with Chapter 2, all investigated optimization problems are further characterized as allocation, binding, or scheduling problem. This is followed by a mathematical formulation of the corresponding memory and optimization model. At last, details of either integer linear program or heuristic implementation are provided.

The first Section 4.1 covers a general optimization workflow that applies to all following sections in this chapter. Section 4.2 then starts with the discussion of an optimization method, which aims at dynamic energy consumption reduction of the memory subsystem while respecting area constraints. The following Section 4.3 is attributed to work that shifts the focus towards static power consumption reduction as enabled by the utilization of memory low-power modes. While the solutions in Sections 4.2 and 4.3 target single-core systems, Section 4.4 considers multi-core platforms and corresponding memory optimization concepts. In general, the focus is put on SRAM memory, except for Section 4.5, which describes an optimization method for STT-RAM.

4.1 Optimization Workflow

An important commonality of all presented optimization concepts is the design-time workflow that can be generalized as depicted in Figure 4.1. The listed inputs include memory characteristics, i.e., general information about the memory architecture of the target platform, which is reflected by the set of memory resources M (cf. Section 2.1). Besides that, some of the below presented optimization models allow the definition of user-defined constraints, for example an upper bound on the number of memory resources. According to the illustrated flow of information, optimization further depends on results of instruction set simulation. Corresponding information includes system statistics, as for example the application's execution period, or the results from

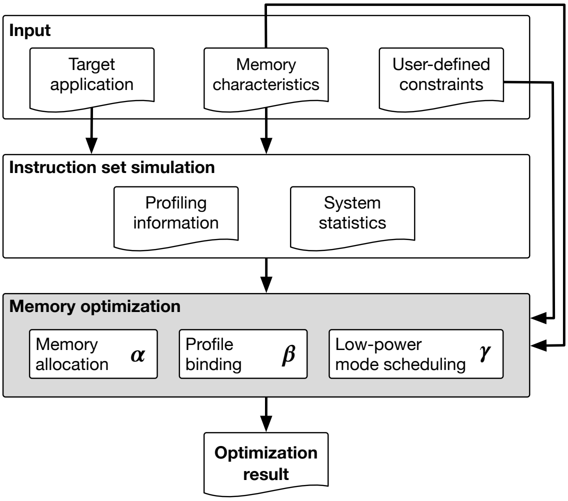


Figure 4.1 Design-time memory optimization workflow

memory access profiling that are relevant for the specification of application profile set P and the dependency graph G_D. As a consequence, determined optimization results are application-specific, i.e., highly tailored to the given target application and investigated use case.

4.2 Dynamic Energy Optimization (SRAM)

Following the explanations on SRAM memories in Section 2.3, dynamic energy consumption in this storage technology highly depends on the memory size and the associated extent of the memory periphery. One can make use of this aspect to reduce the dynamic energy consumption by splitting memory into multiple instances such that frequently accessed segments of the address space reside in separate small memory instances. However, splitting memory into multiple units requires an interconnect, e.g., a bus or a custom fabric, that forwards read and write requests to the individual memory instances. Obviously, this interconnect consumes on-chip area and energy itself and may offset the benefits of the split memories. Furthermore, using multiple small memories instead of a single large one implies an increase of area requirements and thus can become prohibitive. To achieve the highest possible energy consumption savings, it is also necessary to reorganize the logical address space such that the most frequently accessed segments are grouped together and can be mapped to the same physical memory instance. For example, if the most frequently accessed memory addresses are uniformly distributed over the application's address space, frequent and infrequent addresses will inevitably be mapped to the same memory instances voiding any benefit of a split memory architecture. Finding an optimal memory configuration in this context is therefore a non-trivial task. The below explanation of a solution to this problem is based on the contribution in [126].

4.2.1 Context

The focus in this optimization method is on single-core SoC platforms with support for multiple on-chip memories. An abstract system architecture of this target platform is depicted in Figure 4.2. With respect to Section 2.3, this corresponds to a flat memory organization. If applicable, some of the split memories can also be considered as scratchpad memories. Memory low-power modes are not considered. Other components that for example manage the communication with the environment of the system are abstracted as input/output interface but of no further relevance for the optimization.

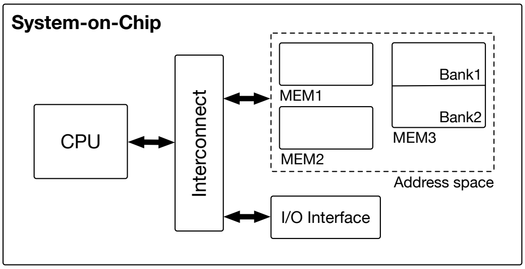


Figure 4.2 Abstract target system architecture

Targeting this system architecture, an energy-efficient on-chip memory subsystem shall be determined at design-time according to the following problem formulation:

Definition 4.1 (Problem Statement)

Find an allocation α of memory instances and a binding β that assigns each application profile $p \in P$ to exactly one memory resource $m \in M$ such that α and β yield the lowest energy consumption of all possible allocations and mappings while satisfying area and user-defined constraints.

4.2.2 Optimization Model

In accordance with the basic optimization model in Section 2.1.1 and the general workflow in Figure 4.1, all relevant input parameters for the following optimization model are shortly summarized in Table 4.1. This comprises activity and memory access statistics of application profiles, memory energy and power characteristics, and lastly, some global system-related figures.

	Parameter	Description	Unit
profile	s^p	Size of the profile	byte
	p^r	Application profile read probability	
	p^w	Application profile write probability	
	d	Profile duty cycle	
memory	s^m	Memory storage capacity	byte
	A^m	Memory on-chip area footprint	mm^2
	E^r	Dynamic energy consumption per read access	J
	t^r	Read operation access time	s
	E^w	Dynamic energy consumption per write access	J
	t^w	Write operation access time	s
	P^s	Static memory power consumption	W
system	P^f	Power consumption of the interconnect	W
	A^f	On-chip area footprint of the interconnect	mm^2
	V	Memory supply voltage	V
	f	System and memory operation frequency	Hz
	T	The application period	s

Table 4.1 Relevant input parameters for dynamic energy optimization

With these parameters, a step-by-step definition of all factors that impact the energy consumption of the memory subsystem is given first. From the perspective of an application profile and for a given memory instance, the required energy per profile due to read and write E^p is formulated as follows:

$$E^p = E^{read} + E^{write} \tag{4.1}$$

Considering one application period, the two dynamic energy consumption components E^{read} and E^{write} are defined as follows:

$$E^{read} = d \cdot p^r \cdot I^r(f) \cdot V \cdot T \tag{4.2}$$

$$E^{write} = d \cdot p^w \cdot I^w(f) \cdot V \cdot T \tag{4.3}$$

Read current I^r and write current I^w are either defined by the technology library or derived from read/write energy consumption and corresponding access times (cf. Table 4.1). Explained on the example of I^r and with respect to the system operation frequency f, the following relation holds:

$$I^r(f) = \frac{E^r \cdot f}{V \cdot \lceil t^r \cdot f \rceil} \tag{4.4}$$

The formulation $\lceil t^r \cdot f \rceil$ yields a factor $\in \mathbb{Z}_{>0}$ that represents the number of clock cycles per memory access. $I^r(f)$ accordingly describes the frequency-dependent read current per clock cycle. Adjusted accordingly, the same relation holds for I^w.

Next, the total dynamic energy consumption of an application, denoted E^{dyn} can be formulated as sum of overall read and write energy:

$$E^{dyn} = \sum_{i=1}^{|P|} E_i^{read} + E_i^{write} \tag{4.5}$$

By substituting E^{read} and E^{write} with Equation 4.2 and Equation 4.3, we obtain:

$$E^{dyn} = T \cdot V \cdot \sum_{i=1}^{|P|} d_i \cdot (p_i^r \cdot I^r(f) + p_i^w \cdot I^w(f)) \tag{4.6}$$

Static energy E^{stat} does not depend on memory access and is consumed independently of any application profile. Instead, it is related to the static current I^s according to Equation 4.7.

$$E^{stat} = T \cdot V \cdot I^{s} \tag{4.7}$$

The average power consumption of one application period is defined according to Equation 4.8:

$$P^{avg} = \frac{E^{dyn} + E^{stat}}{T} \tag{4.8}$$

Substituting E^{dyn} (cf. Equation 4.6) and E^{stat} (cf. Equation 4.7) gives Equation 4.9:

$$P^{avg} = \frac{T \cdot \left((V \cdot I^{s}) + \left(V \cdot \sum_{i=1}^{|P|} d_i \cdot \left(p_i^{r} \cdot I^{r}(f) + p_i^{w} \cdot I^{w}(f)\right)\right)\right)}{T} \tag{4.9}$$

From the optimization perspective, there is no difference whether to minimize the total energy consumption per application period or simply average power. For that reason, this step is useful as it increases readability through elimination of application run-time T. Further, $(V \cdot I^{s})$ can be substituted as it directly corresponds to the static power consumption P^{s} as given in Table 4.1.

$$P^{avg} = P^{s} + \left(V \cdot \sum_{i=1}^{|P|} d_i \cdot \left(p_i^{r} \cdot I^{r}(f) + p_i^{w} \cdot I^{w}(f)\right)\right) \tag{4.10}$$

As discussed above, a complete model of the whole memory architecture should not only consider the individual memory instances but also the interconnect fabric. As the fabric grows in complexity with the number of connected memories, its power consumption and area requirements grow. To capture this effect, both power and area requirements of the interconnect fabric are described as piecewise linear function of the number of connected memories. Let $P^{f} : \mathbb{Z}_{\geq 0} \rightarrow \mathbb{R}$ and $A^{f} : \mathbb{Z}_{\geq 0} \rightarrow \mathbb{R}$ denote the functions for the interconnect fabric's power consumption and area requirements, respectively.

On this basis, the next section discusses an integer linear program that equally respects memory subsystem and interconnect fabric in terms of energy respectively average power and on-chip area consumption.

4.2.3 Combined Allocation and Binding MIQP

The below Mixed-Integer Quadratic Program (MIQP) formulation enables the simultaneous determination of memory allocation and application binding. Please note that the utilization of integer and floating point values and at least one quadratic constraint causes an ILP to become a MIQP. The general feature of integer linear programming, to provide an optimal solution, also holds in this case, however, optimality is considered to be relative to the investigated application or rather its simulation and the resulting input parameters.

Memory Allocation

One part of this optimization method is finding an allocation α. In the model, the allocation variable is modeled as binary vector $\alpha \in \{0, 1\}^{|M|}$ with the set of available memory resources M. It holds:

$$\forall j \in [1, |M|] : \alpha_j = \begin{cases} 1, & \text{if } M_j \text{ is allocated} \\ 0, & \text{otherwise} \end{cases} \tag{4.11}$$

Using the allocation α and a user-defined constraint $mems_{max}$ allows to limit the maximum number of memory instances:

$$\sum_{j=1}^{|M|} \alpha_j \leq mems_{max} \tag{4.12}$$

With $area_{max}$, a similar user-defined constraint can be specified for the total area available for all memory instances and the interconnect. The area requirement per memory resource is denoted A^m; the piecewise linear function A^f describes the area consumption of the interconnect (cf. Table 4.1):

$$\sum_{j=1}^{|M|} \alpha_j \cdot A_j^m + A^f \left(\sum_{j=1}^{|M|} \alpha_j \right) \leq area_{max} \tag{4.13}$$

Application Binding

For each application, the binding variable is represented as binary matrix $\beta \in \{0, 1\}^{|P| \times |M|}$, with the elements $\beta_{i,j}$ indicating whether application profile

i is mapped to memory resource j:

$$\forall i \in [1, |P|] : \forall j \in [1, |M|] : \beta_{i,j} = \begin{cases} 1, & \text{if } P_i \text{ is mapped to } M_j \\ 0, & \text{otherwise} \end{cases} \tag{4.14}$$

To ensure a correct solution, each application profile shall be bound to exactly one memory resource:

$$\forall i \in [1, |P|] : \sum_{j=1}^{|M|} \beta_{i,j} = 1 \tag{4.15}$$

Furthermore, every allocated memory instance shall be large enough in order to accommodate all application profiles that are mapped to it. This memory requirements constraint is defined as follows:

$$\forall j \in [1, |M|] : \sum_{i=1}^{|P|} \beta_{i,j} \cdot s_i^p \leq \alpha_i \cdot s_j^m \tag{4.16}$$

Optimization Goal

According to Section 4.2.2, the average power consumption does not depend on the application run-time. Here, it should be noted that application run-time T is assumed to be independent of the memory selection and thus to be constant. This is justified by the assumption that splitting of memory into multiple instances maintains the timing characteristics. Also the utilized interconnect is expected to barely affect the critical path of a memory access. A timing penalty with effect on the system frequency f is therefore not considered. Based on Equation 4.10, the average power consumption per memory resource can be derived as follows:

$$\forall j \in [1, |M|] : P_j^{avg} = \alpha_j \left(P_j^s + \sum_{i=1}^{|P|} \beta_{i,j} \cdot d_i \cdot \left(p_i^r \cdot I_j^r(f) + p_i^w \cdot I_j^w(f) \right) \cdot V \right) \tag{4.17}$$

Furthermore, combining the quadratic constraint in Equation 4.17 and the interconnect fabric's power consumption P^f allows to postulate the overall average power consumption P^{avg} to be minimized by the MIQP solver by choosing suitable variable assignments for allocation α and binding β.

$$P^{avg} = P^f(\sum_{j=1}^{|M|} \alpha_j) + \sum_{j=1}^{|M|} P_j^{avg} \tag{4.18}$$

The optimization objective is consequently: $minimize(P^{avg})$. The complete formulation of this optimization method in AMPL syntax is provided in Appendix A.

4.2.4 MIQP Variations

On the basis of the above optimization model and corresponding MIQP formulation, several variations are possible as described in the following. Please note that resulting AMPL code from these variations is not separately listed in the appendix as the differences to the above described combined allocation and binding formulation are limited.

Area Consumption Minimization

A big advantage of ILP formulations in general comes with the easy exchangeability of optimization objective and constraints. On the example of the above optimization, few modifications allow to change the objective function from minimization of average power to the minimization of area consumption. In detail, this requires two steps. Equation 4.18 becomes a constraint, which allows to restrict the maximum power consumption $power_{max}$:

$$P^f(\sum_{j=1}^{|M|} \alpha_j) + \sum_{j=1}^{|M|} P_j^{avg} \leq power_{max} \tag{4.19}$$

At the same time, the former area constraint in Equation 4.13 is transformed and becomes the new objective function $minimize(A^{total})$, with A^{total} describing the total area consumption of memory resources and interconnect:

$$A^{total} = \sum_{j=1}^{|M|} \alpha_j \cdot A_j^m + A^f \left(\sum_{j=1}^{|M|} \alpha_j \right) \tag{4.20}$$

The original average power minimization with respect to an area constraint consequently becomes an area consumption minimization problem with possibility to restrict average power. Using both models side by side further enables multi-objective optimization and pareto analysis of the design space. For this, both minimization problems are first executed without a constraint in terms of area or average power respectively. The resulting two configurations mark the energy- and area-optimal limits of the design space. Subsequent optimization runs with iteratively modified constraints within those limits facilitates the identification of a suitable trade-off configuration, e.g., for system designs with restrictions in both dimensions.

Optimization for Multitasking Systems

In the above model, the complete system run-time is attributed to one application. That means, all memory accesses as profiled during simulation are combined into a single set of application profiles. For SoC devices that execute a static schedule of individual and clearly separable algorithms though, this model only fits conditionally. Considering the application to consist of several tasks instead allows a more detailed analysis. The set A denotes a complete multitasking application. Every contained element describes one task, which is assumed to be characterized by its own set of application profiles. The individual share of total execution time per task is further described by the vector $\theta \in [0, 1]^{|A|}$ of weighting factors with:

$$\sum_{a=1}^{|A|} \theta_a = 1 \tag{4.21}$$

Transferred to the above MIQP, all considerations that affect the binding β need to be adjusted in order to reflect this weighted consideration of multiple tasks. That is to say, $\beta_{i,j}$ becomes $\beta_{a,i,j}$ and Equation 4.14 accordingly:

$$\forall a \in [1, |A|] : \forall i \in [1, |P_a|] : \forall j \in [1, |M|] : \beta_{a,i,j} = \begin{cases} 1, & \text{if } P_{a,i} \text{ is mapped to } M_j \\ 0, & \text{otherwise} \end{cases} \tag{4.22}$$

Applying the same modification to Equation 4.15 yields:

$$\forall a \in [1, |A|] : \forall i \in [1, |P_a|] : \sum_{j=1}^{|M|} \beta_{a,i,j} = 1 \tag{4.23}$$

As all profiles of all tasks need to fit into the provided memory space, Equation 4.16 becomes:

$$\forall j \in [1, |M|] : \sum_{a=1}^{|A|} \sum_{i=1}^{|P_a|} \beta_{a,i,j} \cdot s^p_{a,i} \leq \alpha_i \cdot s^m_j \tag{4.24}$$

Applied to the optimization goal, the vector of run-time shares θ needs to be considered as described below:

$$P^{avg} = P^f(\sum_{j=1}^{|M|} \alpha_j) + \sum_{a=1}^{|A|} \sum_{j=1}^{|M|} \theta_a \cdot P^{avg}_{a,j} \tag{4.25}$$

4.3 Static Power Optimization (SRAM)

While the optimization model in Section 4.2 is designed to minimize the dynamic energy consumption of a memory subsystem, this section describes a concept that additionally reduces static power. Optimization potential in this regard is delivered by SRAM low-power operation modes as described in Section 2.3. Main input for the determination of a guided power mode activation schedule γ is the application dependency graph G_D (cf. Definition 2.2). It contains memory access frequencies but also encodes locality of reference patterns for and between code and data profiles. A good implementation in terms of static power reduction is considered to maximize the low-power mode time of individual memory blocks, on the one hand, and further, is expected to minimize the number of mode changes, on the other hand. From the viewpoint of memory allocation α, this implies a memory subsystem with multiple memory blocks, each with support for individual configurability of the operation mode. Further, concerning the binding β of the embedded software, application profiles that are connected by any principle of locality should possibly be mapped to the same memory instance. This is important for the realization of an efficient schedule γ. In other words, highly related code and data profiles shall ideally be in the same memory block and either assigned the same or at least a similar memory configuration vector $c \in C$. The number of operation mode activations and deactivations with the implied penalty in terms of time and energy consumption might otherwise cancel out the savings that result from low-power mode operation time. The explanation of the below solution to this problem is based on the contribution in [132]; a concept for the integration of memory operation mode changes into the system implementation is presented in the subsequent Chapter 5.

4.3.1 Context

This optimization method again targets single-core SoC devices with multiple on-chip memories according to the abstract system architecture as depicted in Figure 4.3. Yet, in comparison to the system architecture for dynamic energy optimization in Figure 4.2, two aspects extend the system model for the consideration of static power optimization. First and motivated by the existence of any type of co-processor, it may be necessary to bound the power consumption level to a certain peak value. This holds especially for mixed-signal designs with supplementary units that are highly sensitive to power peaks. The resulting valid operation range is referred to as power corridor in the following. The second aspect is additional support for memory low-power modes. At any point in time, every memory resource in the system is accordingly operated in one operation mode of the set O as defined for every memory (cf. Definition 2.3) or, in a transition between two of these states (see also [87]). Figure 4.4 illustrates possible SRAM power states and transitions for the operation modes as described in Section 2.3. This includes *active* (ACT), *light sleep* (LS), and *deep sleep* (DS).

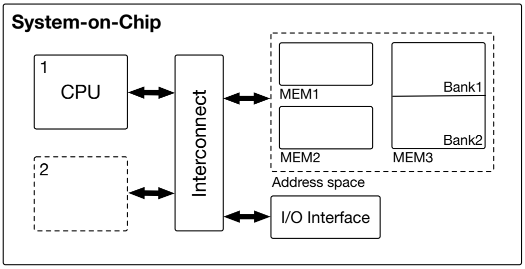


Figure 4.3 Extended abstract target system architecture

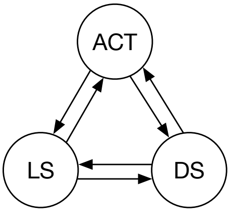


Figure 4.4 Low-power mode transition graph

For this SoC architecture and with respect to the given low-power operation mode alternatives, a dynamic energy- and static power-efficient on-chip memory subsystem shall be determined at design-time according to the following problem statement:

Definition 4.2 (Problem Statement)

Find an allocation α of memory instances and determine a binding β of all application code and data profiles onto the set of allocated memories. Identify an operation mode schedule γ that specifies, which allocated memory shall be operated in which power mode at any point in time within the period T. At the same time, satisfy user-defined corridor constraints for peak power and maximum period.

4.3.2 Optimization Model

Following the basic model in Section 2.1.1, an operation mode schedule γ is defined on the basis of the set of memory configurations C (cf. Definitions 2.6 and 2.7). Accordingly, each configuration $c \in C$ is represented by a vector of length $|M^\alpha|$. Iterating over this vector with index j, every element $c_j \in O$ defines the operation mode per allocated memory $m_j \in M^\alpha$. Using this definition of a memory configuration vector, the state of the memory subsystem can be clearly defined at different points in time within the period T. A power mode schedule γ in turn can be specified as sequence of memory configuration vectors. However, it is simply impracticable to realize such a schedule at the granularity of single time units or cycles, i.e., by specifying the memory configuration for any point in time. This is due to the fact that already the set of possible memory configurations C, which is defined as n-ary cartesian power of O, i.e., $C \equiv O^n$ with $n = |M^\alpha|$, becomes considerably large, even for small numbers of allocated memories. When further combining C with all time units $t \in [0, T)$, the design space for γ is no longer manageable. Beyond that, an implementation of such a schedule in hard- or software seems simply not practicable, let alone meaningful. However, when decreasing the granularity from individual cycles to application profiles instead, the situation changes. That is, each profile $p \in P$ is assigned one $c \in C$ and always executed in this and only this memory power mode configuration.

This direct link between profiles and memory subsystem configurations allows the size of C, and thus of the design space, to be reduced. The reason is, each code profile requires at least the memory block that contains its instructions to be in *active* operation mode (ACT) and thus to be accessible. Configurations that do not satisfy the condition of at least one active memory can consequently be removed from the design space in advance, which leads to the set $C^* \subset C$ with:

$$\forall c \in C^* : \exists i \in [1, |c|] : c_i = ACT \tag{4.26}$$

When looking at data profiles in this context of profile-based configuration assignment, it can be observed that every data profile is connected to one or multiple code profiles in form of data dependencies. Such relations, as encoded in the dependency graph G_D, imply that from the perspective of a function (code profile), all variables and data structures (data profiles) that are under use in this function can not be operated in a low-power mode with restricted accessibility. Illustrated on the example in Figure 4.5, assume that variable X and Y are accessed from within function A. Consequently, MEM1

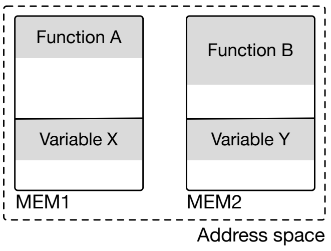


Figure 4.5 Memory configuration example

but also MEM2 need to be operated in *active* mode in order to not impede the access of variable Y.

In short, an individual assignment of memory configurations to variables and data structures (data profiles) provides no additional benefit. Instead, it rather allows for another design space reduction that results from the sole consideration of code profiles in the scheduling process. The set of application profiles is therefore divided into the subset of code profiles P^c and the subset of data profiles P^d according to Equation 4.27:

$$P = P^c \cup P^d \tag{4.27}$$

For the duty cycle of all code profiles $p \in P^c$ it applies:

$$\sum_{i=1}^{|P^c|} d_i = 1 \tag{4.28}$$

For all data profiles $p \in P^d$ instead, the duty cycle stems from the accumulated activity of all code profiles that access data block p as given in Equation 4.29:

$$\forall h \in [1, |P^d|] : d_h = \sum_{i=1}^{|P^c|} \begin{cases} d_i, & \text{if } P_i^c \text{ depends on } P_h^d \\ 0, & \text{otherwise} \end{cases} \tag{4.29}$$

A complete list of all parameters that are relevant for the following optimization model is given in Table 4.2. Please note that in contrast to the average power approach as used in the previously discussed dynamic energy optimization model (cf. Section 4.2), this model works with energy consumption per access (dynamic) or cycle (static) instead. Therefore, read and write quantities are used instead of probabilities. Not listed in the table but also relevant is the dependency graph $G_D(P, E_D)$, including its weighted edge function w (cf. Definition 2.2).

The total amount of energy that is consumed per investigated period T is denoted E^T. According to Equation 4.30, this central optimization variable, which is later subject to minimization, is subdivided into three components.

$$E^T = E^{dyn} + E^{stat} + E^{mode} \tag{4.30}$$

The dynamic part E^{dyn} originates from memory accesses and depends on memory characteristics, on the one hand, and profile access statistics, on the

	Parameter	Description	Unit
profile	s^p	Size of the profile	byte
	n^r	Number of reads to this profile in period T	
	n^w	Number of writes to this profile in period T	
	d	Profile duty cycle	
memory	s^m	Memory storage capacity	byte
	E^r	Dynamic energy consumption per read access	J
	t^r	Read operation access time	s
	E^w	Dynamic energy consumption per write access	J
	t^w	Write operation access time	s
opmode	P^s	Static power consumption of operation mode	W
	E^a	Energy penalty on mode activation	J
	t^a	Operation mode activation time	s
	E^d	Energy penalty on mode deactivation	J
	t^d	Operation mode deactivation time	s
system	f	System and memory operation frequency	Hz
	T	The application period	s

Table 4.2 Relevant input parameters for static power optimization

other hand. This includes the read and write quantities n^r and n^w according to Table 4.2 but also the optimization variables allocation α (cf. Definition 2.4) and binding β (cf. Definition 2.5). In Equation 4.31, β_i describes the memory instance from the set of allocated memories M^α that is assigned to profile P_i.

$$E^{dyn} = \sum_{i=1}^{|P|} n_i^r \cdot E^r(\beta_i) + n_i^w \cdot E^w(\beta_i) \tag{4.31}$$

E^{stat} results from static power consumption in dependence on the operation mode. In SRAM memories, this part is mainly due to leakage currents. It depends, on the one hand, on the normalized duty cycle per code profile d_i. In Equation 4.32, the product of T and d_i accordingly yields the active time per code profile P_i^c. On the other hand, the static power consumption per memory and operation mode (P^s) as defined by the configuration vector schedule γ is relevant for the description of the static energy consumption. Please note that according to Definition 2.7, γ assigns a profile to a vector of memory operation modes from the set C^*.

$$E^{stat} = T \cdot \sum_{i=1}^{|P^c|} d_i \cdot \sum_{j=1}^{|M^\alpha|} P_j^s(\gamma_{i,j}) \tag{4.32}$$

The third and last component is denoted E^{mode} and attributed to the energy penalty from operation mode changes. This information is encoded in the set of weighted edges in the dependency graph. Depending on the static connection of application profiles and operation modes as given by γ, each edge between two code profiles possibly leads to a new memory subsystem configuration $c \in C^*$. In other words, the function w denotes the number of transitions between two functions in the control flow, which in turn describes the number of mode changes that happen during the period T. If any mode change occurs for one or multiple memory instances in the set of allocated memories M^α, it is considered in terms of activation and deactivation energy penalty. The complete formulation of E^{mode} is given in Equation 4.33.

$$E^{mode} = \sum_{h=1}^{|P^c|} \sum_{i=1}^{|P^c|} w_{h,i} \cdot \sum_{j=1}^{|M^\alpha|} \begin{cases} E_j^d(\gamma_{h,j}) + E_j^a(\gamma_{i,j}) & \text{if } \gamma_{h,j} \neq \gamma_{i,j} \\ 0 & \text{otherwise} \end{cases} \tag{4.33}$$

A power corridor, denoted $power_{max}$, can for example be given by a highly sensitive system component, e.g., an analog co-processor. It defines the power level that shall not be exceeded at any point in time within the period T. Accordingly, it bounds the maximum power consumption of the memory subsystem (P^{max}), which consists of two parts for each memory instance in M^α. That is, on the one hand, the maximum value of either memory access energy in dependence on the system frequency f, or, the power mode transition energy over the corresponding switching time. This part is denoted $P^{dyn/mode}$ and defined for every configuration vector $c \in C^*$ according to Equation 4.34.

$$\forall c \in C^* : P^{dyn/mode}(c) = max_{j=1}^{|M^\alpha|} \left\{ \underbrace{\overbrace{\frac{E_j^r \cdot f}{\lceil t_j^r \cdot f \rceil}, \frac{E_j^w \cdot f}{\lceil t_j^w \cdot f \rceil}}^{\text{dyn}}}_{\text{if } c_j = ACT,\ 0 \text{ otherwise}}, \overbrace{\frac{E_j^a(c_j)}{t_j^a(c_j)}, \frac{E_j^d(c_j)}{t_j^d(c_j)}}^{\text{mode}} \right\} \tag{4.34}$$

On the other hand, the total static power consumption of all memories $m \in M^\alpha$ needs to be considered in dependence on the scheduled memory configuration as described in Equation 4.35.

$$\forall c \in C^* : P^{stat}(c) = \sum_{j=1}^{|M^\alpha|} P_j^s(c_j) \tag{4.35}$$

Combined and evaluated for the set of occurring mode changes, the peak power consumption P^{max} can finally be defined as follows:

$$P^{max} = max_{i=1}^{|P^c|} \left\{ P^{dyn/mode}(\gamma_i) + P^{stat}(\gamma_i) : d_i > 0 \right\} \tag{4.36}$$

The peak power corridor constraint is consequently denoted by:

$$P^{max} \leq power_{max} \tag{4.37}$$

A similar consideration for the system period leads to the definition of an upper bound $time_{max}$, further referred to as time corridor (cf. Equation 4.38). As defined above, T denotes the period of the embedded application.

$$T \leq time_{max} \tag{4.38}$$

Since every power mode change possibly entails a delay (cf. Table 4.2), there is a trade-off between energy and time in this regard that needs to be considered. That is to say, frequent power mode switching might be beneficial in terms of energy consumption, however, disadvantageous due to the increasing time penalty. Nevertheless, whenever $time_{max} > T$, the resulting slack time can be used for optimizations, as long as the condition in Equation 4.39 is respected. It states that the sum of all time penalties due to mode switching has to be less than or equal to the slack time.

$$time_{max} - T \geq \sum_{h=1}^{|P^c|} \sum_{i=1}^{|P^c|} w_{h,i} \cdot \sum_{j=1}^{|M^\alpha|} \begin{cases} t_j^d(\gamma_{h,j}) + t_j^a(\gamma_{i,j}) & \text{if } \gamma_{h,j} \neq \gamma_{i,j} \\ 0 & \text{otherwise} \end{cases} \tag{4.39}$$

4.3.3 Implementation Considerations

With regard to the above model and for an optimization goal that, with the help of Equation 4.30, is described as $minimize(E^T)$, the following starting points for an overall reduction of E^T are conceivable:

1. Directly reduce the number of memory accesses, i.e., n^r and n^w in Equation 4.31 respectively. This can be either achieved through direct modification of the application code or through the utilization of a hierarchical memory design, e.g., in using scratchpads.
2. Make use of a heterogeneous memory subsystem and put frequently accessed elements into small memories; rarely used address ranges to larger instances instead. This way, E^r and E^w in Equation 4.31 are accounting for possible savings.
3. Maximize the time, allocated memories are operated in a low-power mode through effects from duty cycle d and γ (cf. Equation 4.32).
4. Minimize the number of mode changes between different power modes. That is to say, keep the number of control flow edges that involve a power mode configuration change, and thus the impact of energy penalties for activation E^a and deactivation E^d in Equation 4.33, as small as possible.

Alternatives (1) and (2) in this enumeration put the focus on E^{dyn} solely. Aspect (2) can be tackled with the above presented dynamic energy optimization method (cf. Section 4.2). The focus in this section, however, is on static aspects and the following optimization concept combines saving potential from (2), (3), and (4) into one, novel optimization method as outlined below.

Due to the fact that the simultaneous consideration of α, β, and γ spans an extremely large design space, a single-step realization could not be realized, wherefore the implementation is split in two subsequent parts. The first part is the allocation of up to $mems_{max}$ memories from the set of available memories M and binding of all application profiles to one of the allocated memory instances $m \in M^{\alpha}$. The second, subsequent part deals with the determination of an energy-optimal configuration $c \in C^*$ for each code profile $p \in P^c$. This way, the design space size issue can be handled and a likewise consideration of E^{dyn}, E^{stat}, and E^{mode} in terms of energy minimization becomes possible.

Using the dependency graph G_D in combination with further characteristics from Table 4.2, the designed two-stage workflow can be described as follows:

1. **Identification of α and β by means of clustering**
   Find clusters of related application profiles in the dependency graph, while minimizing the inter-cluster dependency, i.e., the weight of all edges that connect profiles in different clusters. Each identified cluster finally contains a set of application profiles. Taking each cluster as a memory instance automatically defines the allocation α as well as the binding function β.
2. **Identification of γ based on the result of stage (1) by an ILP**
   Determine an optimal assignment between application profiles and memory

configuration vectors on the basis of the dependency graph and concrete power and energy figures for the memories in M^{α}. This step yields an optimal solution for γ, relative to the quality of the results from software simulation and optimization stage (1). Beyond that, stage (2) allows the definition of corridor constraints.

This implementation concept enables energy savings in multiple ways:

- With each cluster that is identified in stage (1) of the optimization flow, memory subsystem heterogeneity increases along with its positive effects on the dynamic energy consumption E^{dyn}.
- Another co-product from clustering is the grouping of highly related profiles. That means, strongly linked application code is likely to be bound to the same memory instance. This increase of locality favors the expansion of inactivity periods and thus the possibility for longer low-power mode phases of individual memory instances. Hence, a further decreases of the static energy part E^{stat} becomes possible.
- Minimizing the edge weight between clusters in stage (1) directly affects the number of operation mode changes and thus E^{mode}.
- On top of that, stage (2) ensures an optimal operation mode schedule (for a given allocation and binding) and takes its optimization potential from E^{stat} and E^{mode}.

As summarizing illustration, the complete workflow of this two-stage implementation is depicted in Figure 4.6. Other than presented solutions in literature, it is able to consider saving potential from dynamic and static energy consumption. In using a heuristic clustering method in stage (1), followed by a Mixed-Integer Quadratic Program (MIQP) formulation in stage (2), large design spaces are reduced in size and thus can be handled.

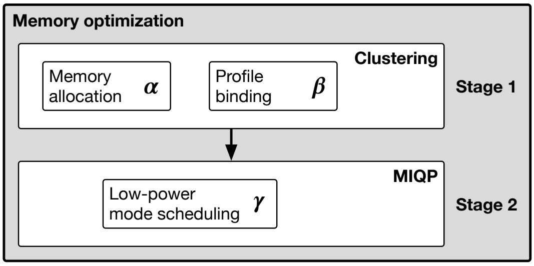


Figure 4.6 Two-stage workflow for static energy minimization

Following the flow in Figure 4.6, different implementation alternatives for stage (1) are outlined in Section 4.3.4 at first. The MIQP realization of stage (2) is then described in Section 4.3.5.

4.3.4 Allocation and Binding Heuristics

Every clustering algorithm and thus heuristic solution that is described in the following section is applied for the purpose of allocation and binding. Similar to the combined allocation and binding MIQP in Section 4.2.3, several general constraints hold here as well.

Concerning the allocation step, a user-defined upper bound for the number of allocated memories $mems_{max}$ shall be respected (cf. Equation 4.12). The storage capacity of every allocated memory $m \in M^\alpha$ shall further be large enough in order to accommodate all profiles that are mapped to it:

$$\forall j \in [1, |M^\alpha|] : s_j^m \geq \sum_{i=1}^{|P|} s_i^p : \beta_i = M_j \tag{4.40}$$

With respect to application binding, each profile shall be mapped to one and only one memory instance:

$$\forall i \in [1, |P|] : \exists! j \in [1, |M^\alpha|] : \beta_i = M_j \tag{4.41}$$

Graph Partitioning

One possibility for the clustering of a given graph into a set of independent groups with minimum inter-dependency is graph partitioning (compare for example [69]). The following approach is realized as integer linear program.

Set/Parameter	Description
V	Set of nodes in targeted graph G
N	Number of clusters to be determined
DM	Dependency matrix of size $\|V\| \times \|V\|$ according to edge weights

Table 4.3 Graph partitioning ILP dataset and parameters

It operates on the dataset V, which describes the nodes of the targeted graph $G(V, E)$. This dataset and other required parameters for the below formulation are summarized in Table 4.3.

Besides these parameters, two variables named ρ and ϕ are defined in the ILP. ρ is specified in Equation 4.42 and realized as binary matrix over the set of nodes V, on the one hand, and the range of clusters $1..N$, on the other hand.

$$\forall i \in [1, |V|] : \forall n \in [1, N] : \rho_{i,n} = \begin{cases} 1, & \text{if node } V_i \text{ is mapped to cluster } n \\ 0, & \text{otherwise} \end{cases} \tag{4.42}$$

The variable ϕ represents a vector of size N. Every entry in this vector describes the accumulated dependency cost between a particular node and all other nodes that are mapped to a different cluster. That means:

$$\forall 1 \le n \le N : \forall 1 \le m \le N : n \neq m \implies \phi_n \ge \sum_{h=1}^{|V|} \sum_{i=1}^{|V|} \rho_{h,n} \cdot \rho_{i,m} \cdot DM_{h,i} \tag{4.43}$$

Other constraints define that each node is assigned to one and only one cluster (cf. Equation 4.44) and that empty clusters are not allowed (cf. Equation 4.45).

$$\forall i \in [1, |V|] : \exists! n \in N : \rho_{i,n} = 1 \tag{4.44}$$

$$\forall n \in [1, N] : \sum_{i=1}^{|V|} \rho_{i,n} \ge 1 \tag{4.45}$$

The final optimization goal is denoted by:

$$minimize\left(\sum_{n=1}^{N} \phi_n\right) \tag{4.46}$$

To be exact, Equation 4.43 causes the integer linear program to become an integer quadratic program. This is due to the multiplication of two variables, here ρ. Nevertheless, the formulation remains manageable with common solvers, most and foremost because ρ is of binary type and the corresponding formula is no equality constraint ($\ge$).

In the context of this work, the partitioning approach is applied to the dependency graph $G_D(P, E_D)$. Accordingly, the set of nodes V is represented by the set of application profiles P; the value of N corresponds to $mems_{max}$; DM is equal to a matrix representation of the weight function w. With varying

N, the design space for memory allocation and application binding can be explored through multiple runs. Each identified cluster represents one memory and thus yields α and β as requested from a stage (1) implementation.

Min-Cut Clustering

An alternative to graph partitioning and thus a different approach to allocation and binding in workflow stage (1) is graph clustering. One such algorithm is min-cut graph clustering as proposed by Flake et al. [35]. The algorithm works on a graph $G(V, E)$ and is described by the following steps:

1. Add an additional node t to the graph G
2. Connect t with every other node in G by an edge with weight a
3. Compute the min-cut tree of the resulting graph
4. Remove t from the graph again
5. The remaining connected elements represent the final clustering

For more details about this heuristic, the reader is kindly referred to, either, the original work [35], or, to the author of [39] (Section 4.4), who gives a thorough introduction to the algorithm. It is important to note that in contrast to graph partitioning, a clustering algorithm in general provides no ability to specify the number N of clusters to be identified. As the result of the min-cut algorithm varies with the edge weight a of the artificial node t, a binary search-like approach is applied to a series of experiments, starting with an initial weight:

$$a_0 = \frac{max_{i=1}^{|E|}\{weight_i\}}{2} \tag{4.47}$$

Subsequently, two solutions with $a_{i+1,1} = a_i - a_i/2$ and $a_{i+1,2} = a_i + a_i/2$ are determined repeatedly. In every iteration, the better of the two solutions is marked as next a_i, as long as the inter-cluster dependency cost according to Equation 4.43 decreases.

In the problematic case, where $mems_{max}$ is less than the number of identified clusters and thus memories, graph partitioning can be used as fallback algorithm. Alternatively, the min-cut binary search quality function can be configured to iterate until $N \leq mems_{max}$ holds, instead of minimizing ϕ according to Equation 4.46.

Modularity Clustering

A third, alternative algorithm for stage (1) clustering is based on the measure modularity [98], a „[...] measure that closely agrees with intuition on a wide range of real-world graphs [...]“ [39] (p. 26). For detailed information about the algorithm as well as a corresponding ILP formulation of this clustering method, the reader is kindly referred to [39] (Sections 1.2 and 2.2).

Please note that modularity is again a clustering algorithm that does not allow to specify the number of searched clusters N. As a consequence, in case of $N > mems_{max}$, min-cut or graph partitioning can be used as fallback solution in order to determine a valid clustering and thus a solution for allocation α and binding β.

4.3.5 Operation Mode Scheduling MIQP

Once, allocation and binding are determined, the second stage of the optimization workflow in Figure 4.6 can be executed. It is realized as Mixed-Integer Quadratic Program (MIQP), meaning, only a part of the parameters and variables in the linear program formulation are integer types. Further, at least one constraint contains a quadratic element, i.e., from a multiplication of two variables. In any case, this method provides an optimal and thus energy-minimal solution for the given set of input data. Based on the result from allocation and binding, but also with respect to the given input parameters as listed in Table 4.2, all datasets and parameters that are used in the following optimization formulation are summarized in Tables 4.4 and 4.5. The constraint parameters $power_{max}$ and $time_{max}$ denote user-defined upper bounds for peak power (power corridor) and time corridor respectively.

Please note that the activity matrix AM directly encodes a determined binding β but reduced to the set of code profiles. Nevertheless, any dependency between code and data profiles as previously illustrated on the example in Figure 4.5 is included in this parameter.

Set	**Description**
P^c	Set of all code profiles
M^α	Set of allocated memory instances
C^*	Set of all possible memory operation mode configuration vectors

Table 4.4 Operation mode scheduling MIQP datasets

Parameter	**Size**	**Description**
d	$\lvert P^c \rvert$	Duty cycle per code profile
DM	$\lvert P^c \rvert \times \lvert P^c \rvert$	Dependency matrix as based on $G_D(P, E_D)$
AM	$\lvert P^c \rvert \times \lvert M^\alpha \rvert$	Required (accessible) memories per code profile (binary)
E^c	$\lvert C^* \rvert$	Passive energy consumption per configuration and cycle
P^p	$\lvert C^* \rvert$	Peak power consumption per configuration $c \in C^*$
CM	$\lvert C^* \rvert \times \lvert M^\alpha \rvert$	Memories in ACT mode per configuration (binary)
PM^E	$\lvert C^* \rvert \times \lvert C^* \rvert$	Penalty matrix for energy per configuration change
PM^T	$\lvert C^* \rvert \times \lvert C^* \rvert$	Extra time on configuration change

Table 4.5 Operation mode scheduling MIQP parameters

The equation for the computation of the individual peak power consumption per configuration P^p is depicted in Equation 4.48. This formulation is considered to be pessimistic as there is no guarantee that the value that results from the max statement is attributed to an actually occurring event for the considered combination of memory instance and configuration vector. If for example the peak power level of the write operation dominates for a configuration that is assigned to read-only profiles exclusively, the value for P^p is higher than actually needed in this case and therefore pessimistic.

$$\forall c \in C^*\colon P^p(c) = \underbrace{max_{j=1}^{|M^\alpha|} \left\{ \underbrace{\frac{E^r_j \cdot f}{\lceil t^r_j \cdot f \rceil}, \frac{E^w_j \cdot f}{\lceil t^w_j \cdot f \rceil}}_{\text{if } c_j = ACT\text{, 0 otherwise}}, \frac{E^a_j(c_j)}{t^a_j(c_j)}, \frac{E^d_j(c_j)}{t^d_j(c_j)} \right\}}_{dyn/mode} + \underbrace{\sum_{j=1}^{|M^\alpha|} P^s_j(c_j)}_{stat} \tag{4.48}$$

Table 4.6 lists the set of required variables. Main goal of workflow stage (2) is the determination of an operation mode schedule γ. In the following MIQP, this information is encoded into a binary matrix, denoted Γ.

Variable	**Size**	**Description**
Γ	$\lvert P^c \rvert \times \lvert C^* \rvert$	Binary matrix for the profile to configuration mapping
E^{stat}	$\lvert P^c \rvert$	Static energy consumption per code profile
E^{mode}	$\lvert P^c \rvert$	Energy consumption per profile due to power mode changes
T^{mode}	$\lvert P^c \rvert$	Time penalty per profile due to power mode changes

Table 4.6 Operation mode scheduling MIQP variables

On this basis, the optimization goal can be stated as follows:

$$minimize\left(\sum_{i=1}^{|P^c|} E_i^{stat} + E_i^{mode}\right) \tag{4.49}$$

The following two constraints ensure the feasibility of a potential solution. According to Equation 4.50, each code profile shall only be assigned to one and only one configuration.

$$\forall i \in [1, |P^c|] : \sum_{k=1}^{|C^*|} \Gamma_{i,k} = 1 \tag{4.50}$$

The formulation in Equation 4.51 ensures that each profile is only assigned to a configuration, in which required memories according to the activity matrix AM are in active (ACT) operation mode, as indicated by the configuration matrix CM (cf. Table 4.5).

$$\forall i \in [1, |P^c|] : \forall j \in [1, |M^\alpha|] : AM_{i,j} = 1 \implies \sum_{k=1}^{|C^*|} \Gamma_{i,k} \cdot CM_{k,j} = 1 \tag{4.51}$$

Equations 4.52 and 4.53 specify constraints for the energy consumption variables E^{stat} and E^{mode} on the basis of passive energy consumption E^c, dependency matrix DM, and energy penalty PM^E.

$$\forall i \in [1, |P^c|] : E_i^{stat} = d_i \cdot T \cdot f \cdot \sum_{k=1}^{|C^*|} \Gamma_{i,k} \cdot E_k^c \tag{4.52}$$

$$\forall h \in [1, |P^c|] : \forall k \in [1, |C^*|] : E_h^{mode} \geq \sum_{i=1}^{|P^c|} DM_{h,i} \cdot \sum_{l=1}^{|C^*|} \Gamma_{h,k} \cdot \Gamma_{i,l} \cdot PM_{k,l}^E \tag{4.53}$$

Accordingly, Equation 4.54 details the corresponding constraint for T^{mode} using the time penalty PM^T.

$$\forall h \in [1, |P^c|] : \forall k \in [1, |C^*|] : T_h^{mode} \geq \sum_{i=1}^{|P^c|} DM_{h,i} \cdot \sum_{l=1}^{|C^*|} \Gamma_{h,k} \cdot \Gamma_{i,l} \cdot PM_{k,l}^T \tag{4.54}$$

Please note that the quadratic constraints in Equations 4.53 and 4.54 are defined as non-equality constraints. This is necessary to get solvability. Finally, period and peak power corridor constraint are given in Equations 4.55 and 4.56 respectively on the basis of Equations 4.36 to 4.39.

$$T + \sum_{i=1}^{|P^c|} T_i^{mode} \leq time_{max} \tag{4.55}$$

$$\forall i \in [1, |P^c|] : \sum_{k=1}^{|C^*|} \Gamma_{i,k} \cdot P_k^p \leq power_{max} \tag{4.56}$$

The complete MIQP implementation of this optimization method in AMPL syntax is given in Appendix B. To the extent that first-order logic has been used in above constraint formulations, they have been transformed into pure MIQP notation using standard methods [50].

4.4 Memory Optimization in Multi-Core Systems (SRAM)

Above discussed optimization concepts focus on single-core SoC devices. With increasingly complex and consequently more compute- and data-intensive applications, however, the demand for multi-core devices, also in the embedded sector, has been steadily increasing in recent years. Unfortunately, multi-core paradigms that are valid in ubiquitous computing are typically not applicable in embedded system development. Limitations in terms of energy and power consumption but also system predictability or given real-time constraints as specially given in the low-power domain, simply require different design approaches. As a consequence, automated concepts for constraint-aware Multi-Processor System-on-Chip (MPSoC) design are increasingly important and in demand. In this context, again the memory subsystem is an interesting target for optimization with respect to performance, energy consumption, or power characteristics as closer investigated below.

Concerning the memory architecture, it can be observed that many multi-core platforms come without traditional caches, i.e., without a cache coherence protocol implemented in hardware. Instead, software-programmable caches alias scratchpad memories are used. As it is the choice of the system designer what to map to these core-local memories, automated concepts are unavoidable in order to evaluate power-performance trade-offs. Besides that, solutions for single-core devices, as discussed in Chapter 3 or given above in Sections 4.2 and 4.3, are not applicable out-of-the-box. Most and foremost, this is due to the fact that, other than for single-core, the memory subsystem is shared among multiple processing units in a multi-core system. Parallel access and possible congestion because of limited memory ports is the consequence. In the worst case, this can heavily impact the system performance with negative effects on the overall energy consumption.

The following memory subsystem optimization concept is based on the contribution in [131]. It takes characteristics of multi-core systems into account, presents a solution for application to memory binding β, and discusses the handling and scheduling γ of memory low-power modes.

4.4.1 Context

The targeted system architecture in this section is, on the one hand, inspired by commercially available Multi-Processor System-on-Chip (MPSoC) devices for industrial use cases [53], and, on the other hand, in line with platforms as considered in other academic work for embedded multi-core and scratchpad-based architectures, e.g., in [22] or [40]. According to Figure 4.7, it consists of a set of processing units, each equipped with a core-local scratchpad memory. On the global level, a shared memory instance is available. The building blocks of the system are further considered to belong to different frequency domains

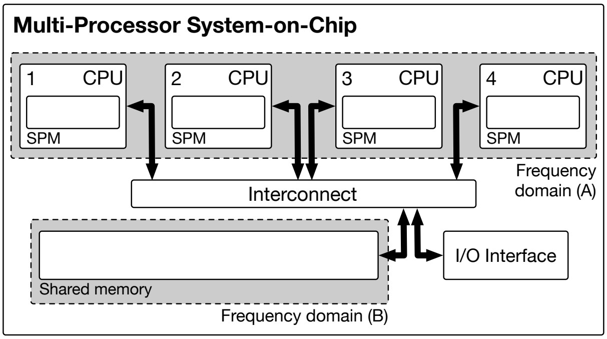


Figure 4.7 Abstract multi-core target system architecture

as exemplified in the diagram. All memories of the system are accessible by all processing units. That is, all cores have access to their local SPM and the global shared memory, but also to the remote scratchpads of other cores. The number of simultaneous accesses to one memory block is limited by the number of access ports. Access delays and energy consumption logically vary with increasing distance and in dependence of assigned frequency domains. In any case, however, core-local memory access is faster than remote scratchpad access, which again is faster than global memory access. Every memory is further expected to support the previously discussed set of operation and low-power modes as illustrated in Figure 4.4.

When working with a pre-defined hardware platform as in this case, the allocation α of the memory subsystem is predetermined and consequently given as input to the optimization flow. The corresponding problem statement for the above stated setup can be described as follows:

Definition 4.3 (Problem Statement)

Assuming a given MPSoC architecture and memory allocation α, find an application to memory binding β such that the overall dynamic energy consumption from memory accesses is minimized while user-defined constraints are met. Further, determine a memory operation mode schedule γ that specifies when to enable/disable which memory operation mode such that the static power consumption of the memory subsystem is additionally optimized.

4.4.2 Optimization Model

The memory optimization model for multi-core is directly based on the solution for static power consumption optimization as presented in Section 4.3. This fact implies two things as follows. On the one hand and in relation to Figure 4.6, the optimization flow is again divided into a first stage for the determination of β, followed by a second, subsequent step for the identification of γ. On the other hand, the list of relevant input parameters as given in Table 4.2 also holds here. Additional modeling instead is necessary for the consideration of multiple processing units in a multi-core system. The set of processing units is therefore added to the plan and denoted U in the following.

Altogether, previously introduced models for dynamic and static energy consumption of the memory subsystem also hold in this section. Differences instead result from the consideration of parallelism, which is mainly visible in terms of multiple, possibly simultaneous memory accessing units $u \in U$. The pre-defined system architecture further changes the task of optimization stage (1) to the determination of β only. This step directly leads to the following paragraph, where a solution for this mapping problem is presented as ILP.

4.4.3 Binding MILP

The first optimization stage aims at finding an optimal binding of application profiles to available memories in the given MPSoC architecture. This includes both, core-local and global memories. The corresponding formulation is based on the datasets in Table 4.7 and parameters as listed in Table 4.8. Due to partly non-integer types it is defined as Mixed-Integer Linear Program (MILP).

Set	**Description**
U	Set of processing units
P	Set of application profiles
M^α	Set of allocated memory instances according to system architecture

Table 4.7 Binding MILP datasets

Parameter	**Size**	**Description**
s^p	$\|P\|$	Profile size (required memory space)
n^r	$\|P\| \times \|U\|$	Number of read accesses per profile and core
n^w	$\|P\| \times \|U\|$	Number of write accesses per profile and core
d	$\|P\|$	Duty cycle of every profile
DM	$\|P\| \times \|P\|$	Dependency matrix of size $\|P\| \times \|P\|$
s^m	$\|M^\alpha\|$	Memory storage capacity
E^r	$\|U\| \times \|M^\alpha\|$	Dynamic read energy per access
t^r	$\|U\| \times \|M^\alpha\|$	Memory read access delay per core
E^w	$\|U\| \times \|M^\alpha\|$	Dynamic write energy per access
t^r	$\|U\| \times \|M^\alpha\|$	Memory read access delay per core
P^s	$\|M^\alpha\|$	Static memory power consumption

Table 4.8 Binding MILP parameters

Variable	**Size**	**Description**
β	$\lvert P\rvert \times \lvert M^{\alpha}\rvert$	Binary matrix for the profile to memory mapping
E^{sum}	$\lvert U\rvert$	Dynamic memory subsystem energy consumption per core
T^{sum}	$\lvert U\rvert$	Active time per processing unit

Table 4.9 Binding MILP variables

The set of variables in this optimization model, including the application binding β, are listed in Table 4.9.

Using these definitions, the following optimization constraints can be defined. Please note that the central optimization variable β is of binary type according to Equation 4.57.

$$\forall i \in [1, |P|] : \forall j \in [1, |M^{\alpha}|] : \beta_{i,j} = \begin{cases} 1, \text{if } P_i \text{ bound to } M_j^{\alpha} \\ 0, \text{otherwise} \end{cases} \tag{4.57}$$

The constraint Equation 4.58 specifies that the footprint of all profiles that are mapped to a memory instance shall be less than or equal to its storage capacity.

$$\forall j \in [1, |M^{\alpha}|] : \sum_{i=1}^{|P|} \beta_{i,j} \cdot s_i^p <= s_j^m \tag{4.58}$$

Further, each function alias code profile $p \in P^c$ with $t(p) = c$ shall be assigned to at least one memory component (cf. Equation 4.59).

$$\forall i \in [1, |P^c|] : \sum_{j=1}^{|M^{\alpha}|} \beta_{i,j} >= 1 \tag{4.59}$$

To ensure data coherency, each data profile (r/w data) $p \in P^d$ with $t(p) = d$ shall be mapped to one and only one memory instance (cf. Equation 4.60).

$$\forall i \in [1, |P^d|] : \sum_{j=1}^{|M^{\alpha}|} \beta_{i,j} = 1 \tag{4.60}$$

The following two constraints define the rules for the computation of consumed energy (cf. Equation 4.61) and time (cf. Equation 4.62) per processing unit as resulting from interaction with the memory subsystem.

$$\forall k \in [1, |U|] : E_k^{sum} >= \sum_{i=1}^{|P|} \sum_{j=1}^{|M^\alpha|} \beta_{i,j} \cdot \Big((n_{i,k}^r \cdot E_{k,j}^r) + (n_{i,k}^w \cdot E_{k,j}^w) \Big) \quad (4.61)$$

$$\forall k \in [1, |U|] : T_k^{sum} >= \sum_{i=1}^{|P|} \sum_{j=1}^{|M^\alpha|} \beta_{i,j} \cdot \Big((n_{i,k}^r \cdot t_{k,j}^r) + (n_{i,k}^w \cdot t_{k,j}^w) \Big) \quad (4.62)$$

The final optimization goal is formulated according to Equation 4.63.

$$minimize \left(\sum_{k=1}^{|U|} E_k^{sum} \right) \quad (4.63)$$

Alternatively, this optimization model can also be used to optimize the memory access performance by replacing E_{sum} with T_{sum} in Equation 4.63. In any case, implemented and solved as integer linear program, this optimization model provides an application-specific optimal solution for β. The complete formulation of this optimization in AMPL syntax is given in Appendix C.

4.4.4 Operation Mode Scheduling

The second optimization stage for MPSoC deals, similar to the single-core case as described in Section 4.3.5, with the scheduling of memory operation modes. The main difference in a multi-core system, the existence of multiple processing units $u \in U$, however, is basically irrelevant for the determination of γ. This is due to the fact that this optimization step deals with the set of code profiles P^c, on the one hand, and the set of allocated memory instances M^α, on the other hand. Combined with the above introduced MPSoC system architecture (cf. Figure 4.7), where all cores of the system are able to access all memories, there is no impact on the optimization model from having multiple processing units. The set of allocated memory instances is predetermined and the complete address range is visible to all processing units. The set P^c simply contains all code profiles of the investigated multi-core application. Hence, the assignment of configuration vectors to code profiles can be regarded as independent from set and number of processing units. Altogether, power mode scheduling and corresponding MIQP formulation for single-core SoC devices (cf. Section 4.3.5) can be applied unchanged to MPSoC platforms as well.

4.5 Dynamic Energy Optimization (STT-RAM)

The last section in this chapter is attributed to an optimization concept for memory subsystems that consist of STT-RAM blocks. The proposed method is built on optimization potential as provided by the following two characteristics that can be observed for this memory technology. First, the STT-RAM write operation energy/latency trade-off according to Section 2.3.2 is an important aspect to consider. Second, and similar to SRAM memories, the impact of required memory access logic leads to increasing dynamic energy consumption with increasing memory size. A corresponding optimization method that exploits this fact in SRAM memory subsystems is described in Section 4.2. The determination of an energy-optimal set of STT-RAM memories, in any case, involves aspects that again can be classified as allocation and binding problem. That is to say, an allocation of memory blocks with different size matters in terms of read and write energy consumption, especially when combined with application segment binding and the consideration of varying memory access frequencies. The assignment of different operation voltage levels to individual memory blocks is another aspect that affects operation frequency of the memory subsystem and thus the performance of the overall system. That means, a higher system operation frequency can be traded in for a less energy-efficient design, depending on the specified system constraints. On the example of the write energy/latency trade-off diagram as depicted in Figure 2.21 for a 4 MiB STT-RAM memory in 45 nm node, the maximum operation frequency ranges from 48 MHz to 57 MHz depending on the operation mode. Analysis of this write operation impact for other memory sizes further reveals an even wider range of design possibilities that extend from frequencies around 40 MHz for megabyte memories to over 100 MHz for smaller memories with a storage

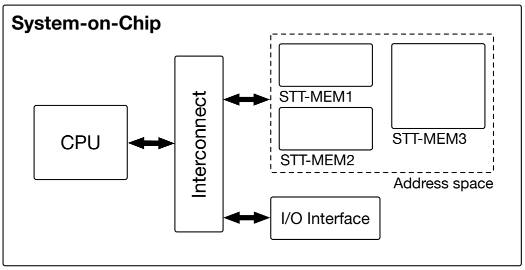


Figure 4.8 Abstract target system architecture

capacity of some kilobyte. Logically, energy consumption varies at the same time, which explains, why the determination of a good or even optimal solution in this design space is not trivial. The below optimization method deals with both, the impact of memory size but also the effects of different operation voltage levels in STT-RAM memories and describes a possible solution for the simultaneous consideration of those aspects.

4.5.1 Context

Again focusing on single-core SoC platforms, this optimization methods evaluates the design space that is spanned by memory subsystems with multiple STT-RAM memory blocks. The considered system architecture in this section is depicted in Figure 4.8. Similar to the targeted architecture in Section 4.2, memory organization is flat. Distinct memory operation modes with different write characteristics are characterized in terms of an operation voltage level and statically determined. That is to say, every memory block is assigned one and only one operation mode, i.e., a run-time adjustment of voltage levels is not part of the following optimization model.

Targeting this system architecture, an on-chip STT-RAM memory subsystem shall be determined at design-time according to the following problem statement:

Definition 4.4 (Problem Statement)

Find an allocation α of memory instances with a specific operation voltage level each and further, determine a binding β that assigns every application profile $p \in P$ to exactly one memory resource $m \in M$ such that α and β yield the lowest energy consumption of all possible allocations and mappings while meeting operation frequency and user-defined constraints.

4.5.2 Optimization Model

A complete list of all parameters that are relevant for the following optimization model is given in Table 4.10. Despite common information about

	Parameter	Description	Unit
profile	s^p	Size of the application profile	byte
	n^r	Number of reads to this profile in period T	
	n^w	Number of writes to this profile in period T	
memory	s^m	Memory storage capacity	byte
	E^r	Dynamic energy consumption per read access	J
	P^s	Static memory power consumption	W
opmode	V	Memory supply voltage in this mode	V
	E^w	Dynamic energy per write and mode	J
	t^w	Write operation latency per mode	s
	f^{max}	Corresponding maximum operation frequency	Hz
system	N	The application period in number of cycles	

Table 4.10 Relevant input parameters for dynamic energy optimization in STT-RAM

memory access statistics and application profiles as also used for previously discussed SRAM optimization models, special attention is to be paid to the operation mode row. In contrast to the low-power operation modes in SRAM, an operation mode in STT-RAM describes an operation voltage level. Every mode is characterized by varying write energy consumption and write operation latency per access. The maximum operation frequency parameter per memory resource and mode is directly related to the latency t^w and computed according to Equation 4.64.

$$f^w = \left\lfloor \frac{1}{t^w} \right\rfloor \tag{4.64}$$

The system period T consequently depends on the determined memory allocation and operation mode selection as the memory instance with the lowest value for f^w bounds the memory subsystem operation frequency. For that reason, the application period is given as N in number of cycles, whereas the actual period in a unit of time is only computable after optimization.

As mentioned above, operation modes are statically assigned to memory instances, i.e., every memory is always operated at one and only one voltage level V, which is either implemented in hardware or adjusted by software at system startup. This way, dedicated handling of operation modes can be avoided in the optimization model. Instead, the question of which mode to apply to which memory can be captured as part of the allocation problem by replicating every memory $m \in M$ by the number of supported operation modes. This obviously enlarges the size of M and thus the number of design choices, however, keeps the optimization problem practicable as shown below.

The total energy consumption of the memory subsystem in period T is modeled by a static and a dynamic part according to Equation 4.65.

$$E^T = E^{dyn} + E^{stat} \tag{4.65}$$

The static part E^{stat} is defined in Equation 4.67. It depends on the set of allocated memories M^α. Please note again that every element $m \in M^\alpha$ is not only characterized by its size, read energy, and static power consumption but also by an operation mode or voltage level respectively with its connected write energy/latency and operation frequency. The STT-RAM instance in the set of allocated memories with the lowest maximum operation frequency is considered as limiting factor in this regard. The period T is accordingly computed as given in Equation 4.66.

$$T = N \cdot min_{j=1}^{|M^\alpha|} \left\{ \frac{1}{f_j^w} \right\} \tag{4.66}$$

$$E^{stat} = T \cdot \sum_{j=1}^{|M^\alpha|} P_j^s \tag{4.67}$$

The dynamic part E^{dyn} is given in Equation 4.68. It depends on the binding β, which links the set of allocated memories M^α with all application profiles. As mentioned above, the dynamic energy consumption per write depends on memory instance and voltage operation mode.

$$E^{dyn} = \sum_{i=1}^{|P|} n_i^r \cdot E^r(\beta_i) + n_i^w \cdot E^w(\beta_i) \tag{4.68}$$

4.5.3 Combined Allocation and Binding MIQP

The following MIQP is based on two datasets. That is, on the one hand, the set of application profiles P. On the other hand, the set of all combinations

Parameter	Size	Description
s^p	$\|P\|$	Profile size (required memory space)
n^r	$\|P\|$	Number of read accesses per profile
n^w	$\|P\|$	Number of write accesses per profile
s^m	$\|M\|$	Memory storage capacity
E^r	$\|M\|$	Dynamic read energy per access
E^w	$\|M\|$	Dynamic write energy per access (in this mode)
P^s	$\|M\|$	Static memory power consumption
f^{max}	$\|M\|$	Maximum supported memory operation frequency

Table 4.11 STT-RAM allocation and binding MIQP parameters

between memory types and corresponding write operation modes, captured by M. That is, every element $m \in M$ describes a memory resource that is operated at a specific operation voltage level.

The collection of all required parameters that characterize the elements in these datasets are listed in Table 4.11. Besides those parameters, the total number of cycles per period N according to Table 4.10 is needed as input. User-defined constraints include $mems_{max}$, denoting the maximum allowed number of allocated memories. The constraint $freq_{min}$ describes the minimum acceptable memory subsystem operation frequency. The variables in the following MIQP formulation are listed in Table 4.12.

For the set of constraints, there are several similarities with the combined allocation and binding MIQP for SRAM memories as discussed in Section 4.2.3:

- Allocation α and binding β are of binary type. That means, multiple instances of the same memory type need to be provided as part of the input data set for allocation.
- $mems_{max}$ represents a user-defined upper bound for the maximum number of allowed memory instances (cf. Equation 4.12).

Variable	Size	Description
α	$\|M\|$	Allocated memory instances (binary vector)
β	$\|P\| \times \|M\|$	Binary matrix for the profile to memory mapping
E^{sum}	$\|M\|$	Total memory subsystem energy consumption in period T

Table 4.12 STT-RAM allocation and binding MIQP variables

- For the binding β, every profile shall be bound to one and only one memory instance (cf. Equation 4.15).
- Every allocated memory instance shall be large enough in order to accommodate all profiles that are mapped it (cf. Equation 4.16).

Beyond that and with respect to the user-defined frequency constraint $freq_{min}$, Equation 4.69 shall hold.

$$\forall i \in [1, |P|] : \sum_{j=1}^{|M|} \beta_{i,j} \cdot f_j^{max} \geq freq_{min} \tag{4.69}$$

The individual energy consumption per memory resource as based on allocation and binding is computed according to Equation 4.70.

$$\forall j \in [1, |M|] : E_j^{sum} \geq \alpha_j \cdot \left(\frac{P_j^s}{f_j^{max}} \cdot N \right) + \sum_{i=1}^{|P|} \beta_{i,j} \cdot \left((n_i^r \cdot E_j^r) + (n_i^w \cdot E_j^w) \right) \tag{4.70}$$

The optimization goal defines the minimization of the total energy consumption as specified in Equation 4.71.

$$minimize \left(\sum_{j=1}^{|M|} E_j^{sum} \right) \tag{4.71}$$

The complete formulation of this optimization method in AMPL syntax is given in Appendix D.

Chapter 5
Code Modification and Generation

After successful optimization of the memory subsystem, e.g., by using one of the above presented methods (cf. Chapter 4), one step remains for the workflow to be complete. That is, the integration of an obtained *implementation* (α, β, γ) (cf. Section 2.1.5) into the System-on-Chip (SoC) design at hand.

Recalling the discussion of related work in Chapter 3, hardware- and software-based solutions can be distinguished for this purpose. As to provide a way that enables transparent integration, general applicability, and support for all presented memory optimization methods, the proposed solution in this chapter is based on code modification and generation steps. This software-centric approach is the preferred way because only the allocation of memory instances α actually affects the hardware design directly. Binding of application to memory β instead can be perfectly realized on the software level only. Same holds for the handling of memory operation modes γ, as long as the memory architecture provides an interface for the software-based configuration of operation modes for example. This way, access to the hardware design is not necessarily needed; modifications and expensive application-specific hardware extensions can be avoided at all. Using code generation further allows some of the obtained optimization concepts even to be applied to commercially available off-the-shelf embedded devices, which is definitely the way to go with increasing complexity of hardware in general and a trend towards multi-core devices in particular.

Starting from target application and memory optimization result, Figure 5.1 depicts the workflow of the proposed post-optimization code modification and generation tool. It connects directly to the optimization flow according to Figure 4.1 and finally generates an application binary that contains all modifications as specified by the optimization step. The presented solution is designed as extension to the compiler backend of LLVM, a modular and open-source compiler framework [72] [73]. This makes it, according to the terminology of Angiolini et al. [3] a compiler-assisted technique.

Next, the basic concept of the backend tool is introduced, followed by the discussion of different use cases, each of which describes the application of the tool to one of the memory optimization methods as presented in Chapter 4.

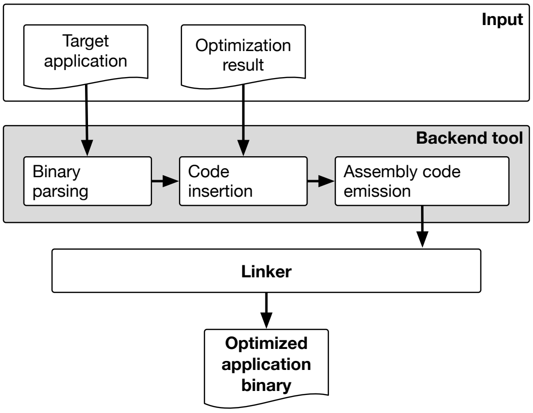


Figure 5.1 Code modification and generation workflow

5.1 Basic Concept

Code generation or insertion in general can be applied at different stages of a cross-compilation flow. Depending on this decision, a different code representation is targeted, which goes along with certain pros and cons in each case. Figure 5.2 depicts a typical cross-compilation flow.

The first option to do code modifications is to work directly on the target application source code. One benefit of this approach is human readability of the source code, typically written in C. Further, applied changes on this level are platform-independent as only afterwards the target-specific cross-compilation tool chain is selected and applied. While passing the compilation pipeline, the source code runs through various representations. First, most compilers transform the input to an Intermediate Representation (IR). On this level, readability is still given but already decreased; platform-independency is mostly maintained, e.g., in case of the LLVM IR (cf. [79], p. 105 ff.). This changes after the application of compiler optimizations and the emission of the assembly code. The assembler stage then transforms the assembler code to a machine-readable object file format. With additional linker information and possibly further pre-compiled code in form of libraries, the generated object file is finally linked to the application binary.

At first glance, it seems reasonable to apply any form of code generation or modification respectively to the earliest possible stage and code representation

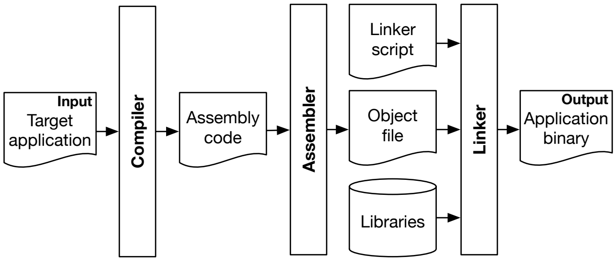


Figure 5.2 Typical cross-compilation tool flow

form. However, there are some pitfalls that need to be considered. Due to the fact that embedded devices are often highly constrained, processing hardware is often kept minimal, i.e., hardware support for a large set of operations is not given and therefore emulated by means of software. For that reason, embedded software is highly dependent on library code, e.g., for otherwise unsupported arithmetic instructions. This library code can, depending on the application, make up for a large share of the overall system execution time and consequently matters in terms of system and memory optimization respectively. However, as illustrated in Figure 5.2, libraries are commonly not available in source code format and added to the compilation flow as input to the linker. This raises the question of how to enable code generation that covers both, application code as defined by the user, but also library code that is part of the tool chain or provided by some other supplier.

In order to address this problem, the following solution is likewise able to deal with user and library code. Due to the fact that libraries are typically not available in a general-purpose language but specific to the target instruction set, the implementation of the proposed tool is automatically bound to a target-specific code representation of limited human-readable form for the application of modifications. Technically, the functionality for the implementation of memory optimization results on the software level is added as specifically designed extension to the LLVM backend, more detailed to the so-called low-level compiler tool *llc*. Following the proposed workflow as depicted in Figure 5.1, the application binary, which initially served as input for system simulation and optimization, is parsed based on symbol table and object dump in a first step. This includes reversing the machine code back to a target-specific LLVM-internal code representation. This representation consists of several hierarchical classes including `MachineFunction`, `MachineBasicBlock`, and

`MachineInstr` [77]. After this step, code modifications and insertions can be applied, depending on the input from the previously executed optimization stage. Finally, existing functionality of *llc* is used to emit an assembly file that contains user and library code as well as all applied modifications and inserted code snippets. Next, each step of this three-stage code generation concept is explained in detail. Please note that except for the last stage, the assembly emission, all steps and related functionality is completely based on own implementations that is added to *llc* in form of passes.

5.1.1 Binary Parsing

With respect to the compilation flow as depicted in Figure 5.2, the LLVM backend constitutes the last part of the compiler block that finally emits the application assembly code. Its pipeline structure is illustrated in Figure 5.3.

Expected input to the backend is a LLVM intermediate representation file as usually provided by the frontend of the compiler. In the original flow, this file is parsed and afterwards passed along the depicted pipeline. The different phases of the backend are highlighted in gray. Between these stages, so-called passes are executed on the current representation of the compilation data. Every pass stage contains pre-defined optimizations but also allows for the integration of own functionality into the backend flow. That is to say, by implementing a set of backend passes that follow the modular design of LLVM, the intended functionality for target-specific binary parsing or code

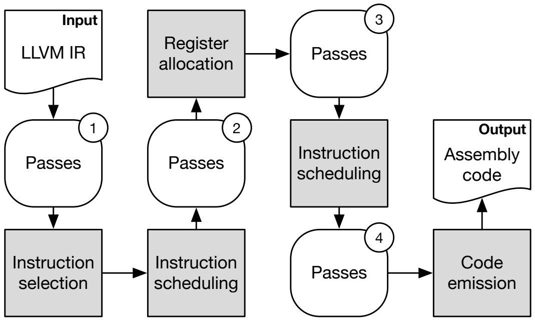


Figure 5.3 LLVM backend pipeline (based on [79], p.134)

modification and insertion can be hooked into the compilation backend in a straightforward way.

Following the above reasoning, the *llc* tool is re-executed, however, not working with input as usually provided by the LLVM frontend but using an existing application binary as input instead. For this, *llc* is invoked with a dummy IR file. Besides that, the added code generation pass is activated, taking the file path of the original application binary as parameter. In fact, this pass is divided into several sub-passes that are hooked into the pipeline at different positions. All following steps are specifically designed extensions and next explained in more detail, sorted by their order of appearance.

CodeGenInitPass

This step is added to pass stage (1) (cf. Figure 5.3). It is responsible for reading the given application binary ELF file into an internal class-based structure of the LLVM backend. This task is subdivided in five steps or functions respectively as shown in Listing 5.1.

Listing 5.1 Code generation initialization pass

```
// parse the symbol table of the given ELF file
parseSymTable(elfFile);
// parse the complete ELF file (as -D dump)
parseElfFile(elfFile);
// find and identify all constant elements
identifyConstData();
// check data blocks for pointers
identifyPtrBlocks();
// final handling and assignment to const block
finalizeConstData();
```

Parsing the symbol table is based on the output of an object dump tool. In case of the ARMv6-M system architecture [6], which is used as the running example in the following, this is calling `arm-none-eabi-objdump -t binary-file`. From this header section data, all relevant symbols for functions, global data, heap, and stack, are collected in a first step by terms of name, address range, and type.

Afterwards, a complete disassembly is generated, for example by calling: `arm-none-eabi-objdump -D binary-file`. This dump is parsed according to the hierarchical structure of functions that contain basic blocks, which in turn contain individual instructions. This step completes the raw structure of the application as previously extracted from the symbol table. Further, every kind of direct branch instruction is filed by means of source and destination. Global static data blocks are also parsed in this step.

The third stage in this initialization pass explicitly searches for constant data as located within code sections. For the exemplarily targeted ARMv6-M architecture, this includes constants, static loop bounds, blocks thereof, branch symbols, branch offsets, or jump tables.

Some of these constants and data blocks further represent a possible pointer block or might contain pointers that need to be identified. For this, potential pointer targets, i.e., functions, basic blocks and static data are matched accordingly, based on their previously determined address. In this connection, it is important to note that the reconstruction of various, especially indirect addressing modes is only possible through detailed analysis of the target-specific LLVM backend implementation. In other words, detailed investigation of code conversion steps for different pointer-based expressions facilitates their correct interpretation during the binary parsing step. As a consequence, only binaries that originate from the LLVM tool chain can be safely processed in this reconstruction step. Experiments with binaries as generated by a cross-compiler from the GNU Compiler Collection (GCC) confirm this point. Particularly expressions with pointer-based addressing and offsets lead to different assembly code in GCC as compared to the output of LLVM. For example in case of pointer-based function calls, LLVM makes use of jump tables. Even though functionally equivalent, GCC uses hard-coded offsets instead, which can not be properly identified by the above presented binary parsing.

The last step in binary parsing deals with a final inspection of constant data blocks and orders them into the hierarchical data structure. Pointers are re-checked and assigned to their corresponding LLVM backend data structure.

CodeGenFctPass

This pass is also hooked into pass stage (1) (cf. Figure 5.3) and directly follows the previously discussed initialization pass. It is implemented as so-called `ModulePass` and therefore executed on a compilation module, the LLVM data structure that encapsulates everything related to one backend run. Once parsed, the content of the application binary needs to be properly inserted into this compilation module as to allow its further processing along the workflow of the backend pipeline. As mentioned before, the invocation of *llc* includes a dummy intermediate representation file. Besides some target specific information, this file contains nothing but a single function stub. For all identified functions of the parsed embedded application, the function stub is duplicated, renamed accordingly, and finally, after handling of all functions, the dummy function is removed from the function list. At this point in the pipeline, however, no content is added to any created `MachineFunction` in form of basic blocks or even instructions yet. This is different for global data blocks and function pointer tables. Corresponding blocks are instantiated, inserted into the `Module` and directly filled with raw content in case of data blocks or function labels in case of a function pointer tables.

With this basic but, for the part of functions, empty data set, the backend pipeline is passed until pass stage (4). Please note that all main phases and passes in stage (2) and (3) as illustrated in Figure 5.3 work on an empty set. This, however, is OK as the complete code as parsed from the application binary has passed this pipeline before and therefore reflects all treatments and modifications of these steps already.

CodeGenBlockPass

At the end of pass stage (4), the insertion the remaining content, consisting of basic blocks, instructions, and constant data is started. In form of a `MachineFunctionPass`, this step iterates over all previously created function stubs. Doing this, the set of `MachineBasicBlock` instances is added to the corresponding function in correct order. This also includes bidirectional linking of blocks as predecessor and successor respectively but also in case of a connection via direct branch. Please note that the identification of basic blocks and their relation to each other is based on a complete list of branch and jump targets and already determined in the binary parsing step.

CodeGenInstrPass

In this last step of the binary parsing phase, the remaining content is added to the `Module` and its sub-components. Targeting the ARMv6-M instruction set, this includes the following data:

- Constant data (blocks)
- Symbols (labels)

- Symbol offsets
- Inline jump tables
- Absolute jump tables

Finally, the still missing instructions are inserted one by one as `MachineInstr` instances. This includes instantiation, appending the instruction to the corresponding `MachineBasicBlock`, parameterization with the right set of registers and condition codes, and if applicable, linking to one of the above listed data instances.

This point marks a state at which the complete application is loaded into the *llc* data structures, just as in an ordinary cross-compilation run (cf. Figure 5.2). The main difference, however, is the fact that this time also library contents are included. On this basis, the code generation flow can be further executed as follows.

At this point, thank you to Julius Hiller, who has contributed to the implementation of this binary parsing step [46].

5.1.2 Code Insertion

In order to modify the application code, additional passes can now be added to pass stage (4). For this, either a `MachineFunctionPass` or a `MachineInstrPass` can be used. The former exemplarily allows the insertion of code snippets or even complete `MachineBasicBlocks` into a function. The latter enables the fine-grained modification of single instructions. This can for instance be used to filter out certain branches in order to apply a redirect. Also class attributes can be used for the implementation of modifications. This allows for example to alter the label of a `MachineFunction`.

5.1.3 Assembly Code Emission

As last step, the modified code is lowered once again to the target assembly level and finally emitted. Due to the neat integration of this tool with the LLVM backend flow, the assembly code generation can completely and without further modifications be reused. Nevertheless, if needed, some late modifications can also be added for consideration in this stage, e.g., the adjustment of labels.

The assembly code as provided by *llc* can then be further processed to object code by an assembler. Subsequent linking yields a binary that eventually reflects all applied changes as based on the memory optimization results.

5.2 Automated Code and Data Placement

The first use case in this section discusses the implementation of a result from the dynamic energy optimization method in Section 4.2 into an embedded software application at hand. This includes the implementation of a determined application-specific memory allocation α and the associated application profile to memory instance binding β. As the allocation part affects the memory architecture, this memory optimization method is only applicable in a hardware/software co-design environment. The allocation α accordingly triggers hardware modifications, for the binding β instead, the proposed backend tool can be used as follows.

While the actual placement of code and data to address ranges is performed by the linker, most assembly dialects provide some directive that enables the guided adjustment thereof. In case of the running example ARM platform, this directive is named `.section`. Prepending individual function or data blocks in the assembly code with a corresponding line enables the desired fine-grained placement of address ranges to memory instances.

From the implementation perspective, this feature is realized as `ModulePass` and appended to pass stage (1), directly following the *CodeGenFctPass* (cf. Section 5.1.1). In first place, the memory optimization result is parsed, which reflects the binding and thus the placement of address ranges to memory sections. Afterwards, the LLVM `setSectionPrefix()` function is used for every `Function` instance; the LLVM `addAttribute()` interface for every `GlobalVariable` and read-only data instance as to make a note for the section prefix. This prefix attribute is respected by the backend later on and finally printed in the assembly emission stage. On the example of the ARMv6-M architecture and its libc helper function for a signed (integer) division `__divsi3`, binding of this code profile to memory section `.text2` results in the emitted assembly code as depicted in Listing 5.2. Subsequent processing of this target-specific assembly code in combination with a matching linker script ensures that all functions with prefix `.text2` are finally placed in the second memory instance. Same holds for data blocks that use the prefix `.data` in combination with the number of the assigned memory block. This way, the obtained application binary finally implements the correct code and data placement.

Listing 5.2 Assembly code after section placement pass

```
    .section   .text2 ,"ax",%progbits
    .weak  __divsi3
    .p2align   1
    .type  __divsi3 ,%function
    .code  16
    .thumb_func
__divsi3:
    .fnstart
@ %bb.0:
    ...
```

5.3 Operation Mode Schedule Integration

The second use case as described in this section discusses the transparent integration of a memory operation mode schedule γ or the handling of memory operation modes by means of software respectively. For the part of allocation and binding that are also provided by the corresponding two-stage memory optimization method in Section 4.3, the previously discussed concept in Section 5.2 can be applied.

In order to implement a memory operation mode switching schedule in software, the memory subsystem necessarily has to provide a suitable interface. One possibility for that is using a so-called configuration register, a

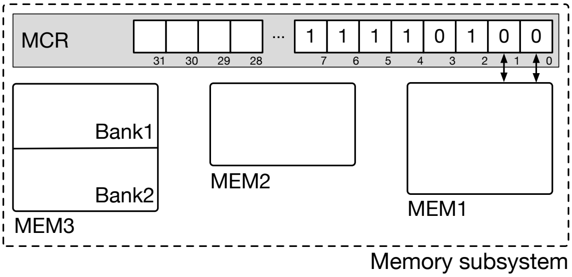


Figure 5.4 Exemplary memory subsystem with configuration register

commonly adopted concept in SoC devices for the configuration of hardware components at system startup, for example the setup of timers, prescalers, or clock dividers. Applied to the memory subsystem and the set of operation modes $O = \{ACT, LS, DS\}$ (cf. Figure 4.4), two bits per memory are sufficient for a binary encoding of this information. In the exemplary ARM platform, a 32-bit architecture, one register allows accordingly to manage the configuration of up to 16 memory blocks, which is considered a sufficiently large number. The corresponding memory architecture with Memory Configuration Register (MCR) is illustrated in Figure 5.4. With $ACT = 00$, $LS = 01$, and $DS = 11$, the depicted example represents the memory configuration vector $c \in C^* = (ACT, LS, DS, DS)$ for the memory blocks MEM1, MEM2, MEM31, and MEM32.

Now, to realize an operation mode schedule γ as provided at the granularity of functions, the prologue of every function is prepended by a short code snippet, which handles the operation mode switching. Instead of directly modifying the individual functions for that purpose, a duplicate stub is inserted for every function during the execution of the *CodeGenFctPass* (cf. Section 5.1.1). This duplicate is appended with a short label, for example `_impl`. An additional `MachineFunctionPass` that is added to pass stage (4) is then used to insert the operation mode switching code snippets into the duplicate functions. To ensure that this code is executed as additional prologue in front of the original function code, the assembly code emission stage of LLVM is slightly modified as to exchange duplicate (e.g. `__divsi3_impl`) and original function labels (e.g. `__divsi3`) during assembly code printing (cf. Figure 5.5). That way, every function call (e.g. to `__divsi3`) lands on the label of the inserted code snippet.

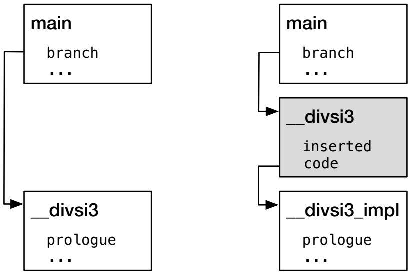


Figure 5.5 Function call example - plain (left), modified (right)

A huge benefit of this solution originates from the fact that while passing the stages of *llc*, direct recursion can be detected and properly handled by skipping the inserted code snippet, i.e., in directly jumping to label `ntbl_bitcnt_impl` as illustrated by the short excerpt for this case in Listing 5.3. Same holds for jumps between functions that share the same configuration vector. This allows to reduce the introduced code overhead and its negative effect on system execution time to the minimum.

Listing 5.3 Efficient handling of direct recursion

```
ntbl_bitcnt_impl:
   .fnstart
@ %bb.0:
   push   {r4, r6, r7, lr}
   ...
   beq  .LBB23_2
@ %bb.1:
   bl   ntbl_bitcnt_impl
   ...
```

Concerning the actual inserted code that is added to the duplicate functions for operation mode switching, it is important to note that SRAM memories are not accessible when operated in a low-power mode (e.g. light sleep (LS) or deep sleep (DS)). As a consequence and in terms of a required re-activation from one of those low-power modes, it is problematic if corresponding code is located in the same, possibly still sleeping memory block. With this restriction in mind, two conceivable solutions are presented next. The first concept in Section 5.3.1 requires partial hardware support, the second proposed solution in Section 5.3.2 instead is purely software-based.

5.3.1 Memory Activation-on-Access

Activation-on-access describes a hardware feature that automatically triggers a change to the active operation mode (ACT), whenever a read or write takes place during a low-power mode period. This can for example be realized through extra circuitry and based on the detection of an edge on a read- or write-enable signal. Accordingly, every access to a still sleeping memory block is served without exception due to this automatic activation scheme. The default configuration scheme, however, remains software-based via the

Memory Configuration Register (MCR). Exemplary operation mode switching code for this purpose and on the example of the ARMv6-M architecture is depicted for the function `__divsi3` in Listing 5.4.

Listing 5.4 Inserted operation mode switching code for the activation-on-access case

```
__divsi3:
  .fnstart
@ %bb.0:
  push   {r5, r6}
  ldr    r5, .LCPI14_0    @ Load address of MCR
  ldr    r6, .LCPI14_1    @ Load memory configuration vector
  str    r6, [r5]
  pop    {r5, r6}
  b      __divsi3_impl    @ To original function
```

First, the configuration vector as specific to that function is loaded from label `.LCPI14_1` and stored into the memory-mapped configuration register (address stored at label `.LCPI14_0`). Afterwards a direct branch to the original function (now labeled `__divsi3_impl`) is executed. This way, a determined memory operation mode schedule γ can be integrated on the software level.

The activation-on-access feature becomes relevant as both, activation code (duplicate function) as well as original function are located in the same memory block. This is problematic whenever a change of the memory configuration register is indicated, e.g., due to a function call or return, but the memory block of the targeted code profile is still operated in a low-power mode. In short, any transition to a sleeping memory makes the above described hardware feature necessary as fallback plan. It ensures the availability of the required memory block in time, however, causes a temporal deviation of configuration register and actual memory configuration. In case of a function call, the operation mode switching code in Listing 5.4 is executed directly afterwards, which causes the prompt correction of the MCR. On the return path, e.g., via the link register, the inserted function prologue and thus the adjustment of the configuration register is not re-executed, which will cause a longer deviation of configuration register and actual memory configuration. Still, the effect from this is expected to be low as highly related code profiles, i.e., functions with frequent mutual branches, are most likely grouped together and assigned the same or at least a similar memory configuration vector (cf. Section 4.3).

5.3.2 Memory Configuration Handling via Stack

If an automated activation-on-access feature is not available or supported by the memory subsystem, the responsibility of keeping the MCR correct and all required memory blocks in active operation mode at any time must be handled in software.

The proposed solution is realized via the stack, similar to register context saving and restoring and based on two pillars. First, an always-on memory instance that contains all duplicate functions is added to the memory subsystem. This way, all operation mode switching code is constantly accessible, however, at the cost of an extra, albeit small memory. The inserted code snippet per function in this case is illustrated in Listing 5.5, again on the example of `__divsi3`. In contrast to the above solution (cf. Listing 5.4), only memory configuration vector and the target address of the original function need to be loaded, similar to the access of a lookup table.

Listing 5.5 Inserted operation mode switching code for the stack-based case

```
__divsi3:
  .fnstart
@ %bb.0:
  push  {r5, r6, lr}
  ldr   r5, .LCPI14_0     @ Load memory configuration vector
  ldr   r6, .LCPI14_1     @ Load address of original function
  bl    lpmJump           @ To jump handler
```

The second pillar is an extra introduced jump handler function. That means, besides the already discussed duplicate functions, one additional function is inserted to the compilation module via the backend tool. As illustrated in Listing 5.4, any branch or jump instruction that leads to the address range of another function is redirected via this jump handler. Here, the correct handling of the MCR is implemented in software by setting and restoring its correct value. An exemplary implementation of this function, again for the ARMv6-M architecture, is given in Listing 5.6, where the address of the MCR is stored at label `.LCPI115_0`. Content of registers `r5` and `r6` depends on the previously executed code snippet, which ensures the correct parameterization of `lpmJump`. Please note that on function return the jump handler is passed again, what allows to restore the MCR via the stack to its earlier content. Hence, the state of memory subsystem is always in line with the operation mode schedule γ.

Listing 5.6 Branch and jump handler function

```
lpmJump:
   .fnstart
@ %bb.0:
   push  {r0, r1}        @ Store content of working registers
   ldr r0, .LCPI115_0    @ Load address of MCR
   ldr r1, [r0]          @ Load current MCR vector
   str r5, [r0]          @ Set (new) vector of target function
   mov r5, r1            @ Move old MCR value
   pop {r0, r1}          @ Restore working registers
   push  {r5}            @ Push old MCR value to the stack
   blx r6                @ Execute jump/branch ...
   pop {r5}              @ Get old MCR value from stack
   ldr r6, .LCPI115_0    @ Load address of MCR
   str r5, [r6]          @ Restore content of MCR
   pop {r5, r6, pc}      @ Return
```

5.4 Adjustments for Multi-Core Systems

With respect to the memory optimization concept for multi-core according to Section 4.4, allocation and binding can be implemented the same way as described for the single-core case (cf. Section 5.2). For a software-controlled memory operation mode control mechanism instead, multiple processing elements make a difference that needs to be considered. One possible solution that extends the Memory Configuration Register (MCR) concept in Section 5.3 is illustrated in Figure 5.6. Each processing unit is accordingly equipped with a core-local MCR. This register contains the individual configuration of the

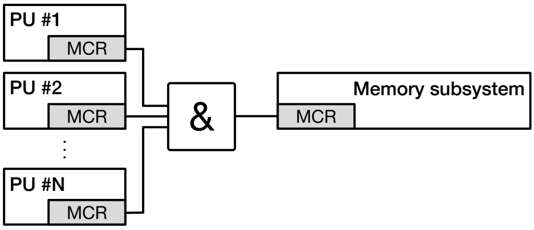


Figure 5.6 Memory subsystem configuration register concept for multi-core

memory subsystem as required by every core. Since the memory subsystem is shared among multiple processing units, a simple bit-wise AND is used as to determine the global memory subsystem configuration. Active is considered dominant in this regard and again encoded by $ACT = 00$ (cf. Figure 5.4). Light and deep sleep are accordingly defined by $LS = 01$, and $DS = 11$.

5.5 Write Mode Handling in STT-RAM

The different write operation modes in STT-RAM as provided by the optimization step (cf. Section 4.5) are completely static and therefore not adjusted at run-time. For that reason, a fixed implementation of the voltage level per memory block can either be realized on the hardware level if applicable. Or, corresponding configuration code can be added to the system startup code if a matching interface, e.g., a MCR is available for that purpose. Code modification or generation steps are not required in this case.

Chapter 6
Evaluation

This chapter discusses the systematic experimental evaluation of all memory optimization concepts as presented in Chapter 4 of this thesis. It is structured as follows. First, Section 6.1 lists general information about utilized platforms and tools. Next, Sections 6.2 and 6.3 discuss experimental results for the presented dynamic energy and static power optimization methods and their variations. Memory optimization results for multi-core systems are provided in Section 6.4. Optimization results for STT-RAM are discussed in Section 6.5. Which optimization method from Chapter 4 applies in each case is stated in the introduction of the individual sections.

6.1 Experimental Setup and Tools

Platforms

The simulation and optimization host machine runs Gentoo Linux, is equipped with a x86_64 Intel Xeon CPU E5-2660 v2 (10 cores @ 2.20 GHz), and has 128 GB of RAM as well as 256 GB of SSD swap space.

Main target evaluation platform is the ARMv6-M architecture [6] as commonly implemented by ARM Cortex-M0, ARM Cortex-M0+, and ARM Cortex-M1 processors.

Benchmark Applications

Test software for all experiments is taken from representative and well-established embedded benchmark collections. This includes the EEMBC benchmark suite [31] and the MiBench benchmark suite [43]. The evaluated use cases cover applications from the most relevant embedded computing fields, including automotive and industrial control, telecommunication, network, and security. The individual examples are characterized by varying memory access behavior as to enable a meaningful evaluation.

Memory Simulation

Unfortunately, intellectual property protection makes it difficult to get industrial-grade memory figures. Black box modeling of memory properties using non-linear regression [116] is one alternative to cope with this lack of available numbers. The tool CACTI, originally published by HP, however, has over the years become a defacto-standard for memory simulation in scientific work. It is based on technology models according to the ITRS roadmap, publicly available, and further supports simulation of SRAM [95] and STT-RAM [4] [5] memories. All memory characteristics that are used in the experimental evaluation as presented in this chapter are, for that reason, generated from memory simulation with either version of CACTI.

Instruction Set Simulators

Memory access traces for selected benchmarks on the targeted ARM architecture are generated by means of Instruction Set Simulation (ISS). Utilized simulators include the SystemC-based simulation framework SoCLib [106] [121] as well as a specially for this purpose developed in-house simulator [128] that is written in C++. A specially developed memory power state machine that works on the basis of CACTI figures is implemented as library [113] and used with both simulation environments. Using this setup, highly accurate memory performance, energy, and power figures can be determined through simulation before and after optimization. This enables meaningful comparative experimental figures that also capture overheads from extra code, e.g., as inserted for the implementation of a memory operation mode schedule γ (cf. Section 5.3).

Software and Tools

Any sort of linear or quadratic program is implemented in AMPL syntax [36], solved by AMPL frontend version 20111121, and using the Gurobi Optimizer in version 8.1.0.

C++ code for heuristics or simulators is compiled using GNU g++ (GCC) version 5.4.1 for execution on the host platform. Automation of data parsing, generation, and evaluation is realized as Python or Bash script.

Target application code is cross-compiled using the LLVM framework in version 7.0.0. The code generation tool according to Chapter 5 is also integrated into the LLVM framework version 7.0.0. Assembler, linker, and object dump tools are from GNU Binutils version 2.27.51.

6.2 Combined SRAM Allocation and Binding Results

This section evaluates savings in terms of energy and on-chip area consumption as well as the solving performance of the dynamic energy optimization method and its variations as presented in Section 4.2. The applied memory design space for the following experiments consists of 54 different SRAM memory resource types as generated with CACTI in 45 nm technology node and using the low standby-power cell type (LSTP). The individual memories differ in terms of storage capacity and banking. All considered memories are of size 2^x with $x \in [8, 25]$. This corresponds to a range from 256 B to 32 MiB. For every memory size in this set, a memory resource with either one, two, or four memory banks is available. Power and area figures for the interconnect are based on power simulations for a parameterized, multiplexer-based VHDL model that was synthesized using the NanGate 45 nm Open Cell Library [119]. The investigated range for $mems_{max}$, that constrains the maximum number of allowed memories is $[1, 8]$. Please note that the interconnect prolongs the critical path for memory access, however, all setups in the considered range for up to 8 allowed memory instances are still able to run at clock frequencies of up to 800 MHz. For that reason and assuming a system operation frequency of $f = 100$ MHz, the timing impact of the interconnect is considered uncritical. The system operation voltage is assumed to be 1.0 V.

The remainder of this section is structured as follows. For the MIQP according to Section 4.2.3 and Appendix A respectively, Section 6.2.1 first evaluates achieved energy savings and the associated impact on the on-chip area consumption. Next, Section 6.2.2 discusses the solving performance of this optimization for the above listed parameters, experimental setup of platforms, and benchmark applications. The impact of the interconnect fabric is discussed in Section 6.2.3. Subsequently, the MIQP variations according to Section 4.2.4 are evaluated. Section 6.2.4 presents results from area consumption minimization experiments. The combination of energy and area minimization MIQPs further allows for multi-objective optimization and pareto analysis as discussed in Section 6.2.5. Results for a multitasking setup in Section 6.2.6 conclude this section on experiments for dynamic energy optimization.

6.2.1 Dynamic Energy Minimization

The obtained results from dynamic energy minimization experiments for 16 benchmark applications are depicted in Figure 6.1 in terms of normalized average power consumption. The given figures include dynamic energy and static power consumption of all memory resources plus the overhead from the interconnect. The baseline configuration, which is marked by 1 (allowed memory instance) and thus $mems_{max} = 1$, describes the optimal memory configuration that uses a single memory block only. The following cases with 2-8 allowed memory instances vary the value for the $mems_{max}$ constraint and are compared to this baseline.

It should be noted that a separate consideration of program code and data profiles is possible with this optimization method. However, as the example ARM platform follows the von Neumann architecture with a shared memory subsystem, a separate optimization is not applicable here. In other words, below experiments consider all profiles together for the determination of a combined memory subsystem for program code and data.

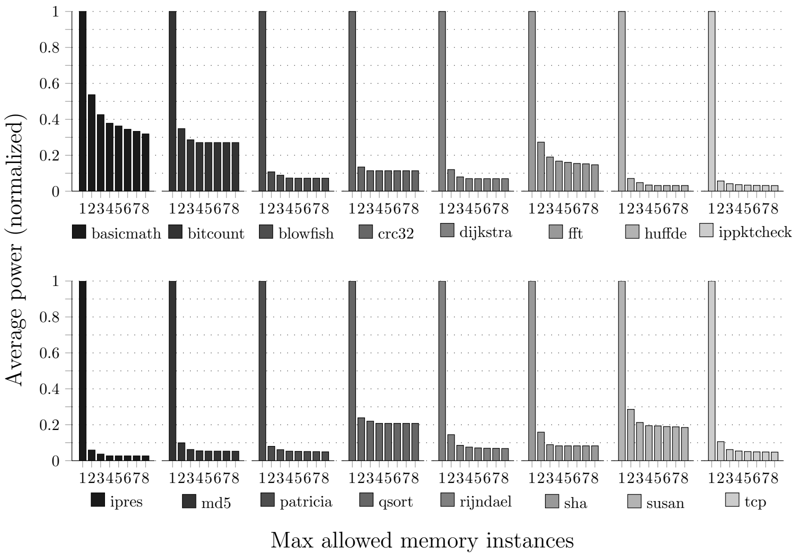


Figure 6.1 Normalized average power consumption with varying $mems_{max}$ constraint

Mems	Num	Size	Banks	P [mW]	Accesses [%]	Profiles [#]
1	1	4 M	1	48.1044	100	135
5	-	-	-	1.2895	-	-
	1	256 B	1	0.2370	90.88	4
	1	2 K	2	0.0170	3.38	14
	1	8 K	2	0.0134	1.42	32
	1	1 M	4	0.0933	0.51	1
	1	2 M	4	0.8482	3.81	88
			Interconnect	0.0806		

Table 6.1 Power consumption details for ipres (IP reassembly) benchmark

When looking at the plots in Figure 6.1, it is striking that already the utilization of two memories leads to considerable savings of over 82 % on average. Allowing a third memory yields 86 % savings and with $mems_{max} = 8$ the average improvement is over 88 %. This shows, that for most applications a consideration of up to three memory resources is sufficient in order to exploit the optimization potential of this method. Only highly compute-intensive applications, e.g., basicmath, fft, or susan, represent an exception.

In order to better understand, where these average power savings of close or partly over 90 % are coming from, Table 6.1 gives a detailed listing of memory blocks, application profiles, and their relationship in terms of power consumption on the example of the IP reassembly benchmark.

Compared to the first row, which describes the baseline case with a single memory of 4 MiB, the given dynamic energy-optimized configuration uses 5 memory resources instead. When looking at the share of memory accesses, it can be observed that only 4 out of 135 total application code and data profiles make up for over 90 % of all memory accesses. A binding of those address ranges to an extremely small block of 256 B consequently enables the observed average power reduction of 97 % in this example.

Obviously, the uncompromising optimization of dynamic energy has some impact on the on-chip area consumption. Surprisingly, energy minimization does not exclude a reduction of on-chip area consumption as can be seen from the experiments that show an average area reduction of over 12 % for $mems_{max} >= 3$ (cf. Figure 6.2). Nevertheless, some benchmarks also show a, though little, degradation in this regard. Again, the compute-intensive applications fft and susan can be named here.

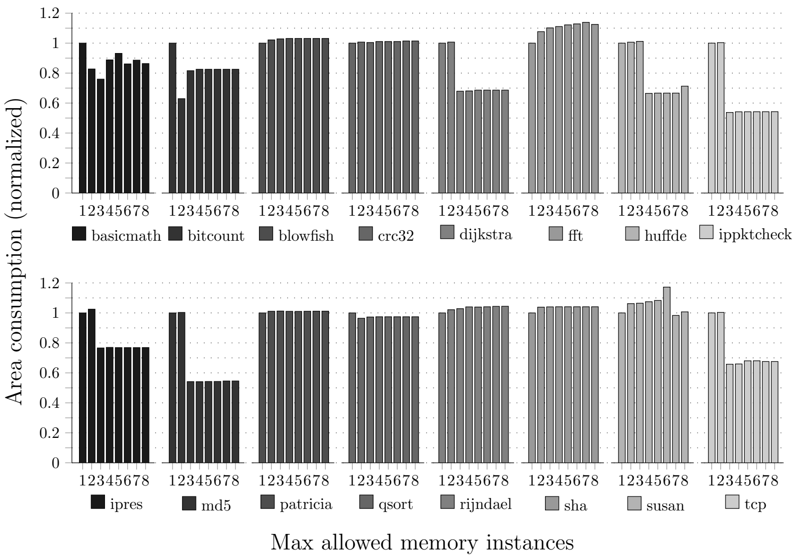


Figure 6.2 Normalized on-chip area consumption after energy minimization

6.2.2 MIQP Solving Performance and Scalability

It is important to note that the presented experimental results not only require the value of $mems_{max}$ to be varied but also make the replication of the above introduced 54 memory resources necessary. This is due to the fact that the allocation α is binary function (cf. Definition 2.4) and every memory type can possibly be allocated $mems_{max}$ times. That means, the number of available resources considerably increases with every increment of $mems_{max}$, i.e., with $mems_{max} = 1$ there are 54 available memories, then 108, 162, and so on. This has a quite severe impact on the size of the design space, which in turn negatively affects the solving performance as illustrated in Table 6.2.

As the design space of the investigated optimization problem is basically spanned by the set of application profiles, on the one hand, and the set of memory resources, on the other hand, it is not surprising that every duplication of the provided memory resources leads to increasing solving times. Some cases with > 100 application profiles and for $mems_{max} = 8$, i.e., 432 provided memory resources, even exceed the defined solving time limit of two hours.

		basicmath	bitcount	blowfish	crc32	dijkstra	fft	huffde	ippktcheck
Lines of code		1662	1096	1569	758	992	1391	2135	2928
Profiles [#]		115	70	65	57	69	99	165	128
Solving time [s] $mems_{max}$	1	0.11	0.09	0.08	0.07	0.07	0.12	0.15	0.13
	2	1.64	1.4	0.83	0.39	2.74	0.72	3.48	2.45
	3	3.6	4.31	2.28	0.55	2.49	2.01	29.63	5.59
	4	10.03	7.12	1.57	0.76	2.6	4.1	22.37	17.28
	5	65.65	14.61	3.64	2.24	4.53	13.36	10.24	15.69
	6	2538.51	24.29	5.63	1.04	8.35	4.93	29.19	13.47
	7	1979.8	76.24	5.35	4.06	11.91	1991.82	38.93	187.86
	8	▨ 7200	134.78	7.56	6.03	39.68	▨ 7200	▨ 7200	▨ 7200

		ipres	md5	patricia	qsort	rijndael	sha	susan	tcp
Lines of code		3788	2656	1085	782	1757	975	2520	5506
Profiles [#]		135	106	65	55	78	58	126	216
Solving time [s] $mems_{max}$	1	0.13	0.1	0.07	0.07	0.09	0.07	0.15	0.18
	2	2.74	1.25	0.6	0.62	1.39	0.87	2.66	3.28
	3	15.85	4.9	1.12	2.18	2.16	1.36	2.03	13.26
	4	7.84	6.74	0.98	2.19	4.3	0.71	8.15	44.04
	5	49.29	4.4	1.74	3.45	1.69	1.21	108.18	39.29
	6	57.39	51.07	2.69	5.64	4.4	3.27	324.11	63.2
	7	357.95	77.13	138.84	7.79	5.47	4.7	4108.49	333.92
	8	▨ 7200	▨ 7200	▨ 7200	9.94	5.11	5.61	▨ 7200	▨ 7200

▨ Stopped after a time limit of 2 h (feasible solution only)

Table 6.2 Overview on benchmark extent and MIQP solving time for varying $mems_{max}$

In these gray-marked cases, a feasible (not guaranteed optimal) solution is returned. As these results turn out to be equal to or even better than the previously determined optimal solution with $mems_{max} = 7$, there is no need to withdraw the resulting memory configurations, which are therefore also part of the depicted plots in Figure 6.1 and 6.2.

The following details show further that it is quite unlikely that, for example in case of $mems_{max} = 8$, an optimal allocation consists of 8 memory resources of exactly the same type. As shown on the example of the IP reassembly benchmark in Table 6.1, it is rather the combination of small memory blocks for frequently used profiles and large memory resources for hardly used code and data blocks, which provides the observed high saving potential. Another conducted set of experiments, with $mems_{max} = 8$ and varying replication factor in the range of $[1, 8]$ and therefore $|M| \in \{54, 108, 162, \ldots, 432\}$ provided memory resources, confirms this assumption. Even without replicating any resource, i.e., for the scenario with up to 8 allowed memories and a replication factor of 1 and thus only 54 unique memory resources, the average deviation (over all 16 benchmarks) from the optimization result with all resources replicated 8 times is below 0.5 %. With every increment of the replication factor, the deviation further vanishes to zero, the latest at factor 4. Altogether, it can be stated that using this switch, the solving performance of the optimization model can be tuned in exchange for an extremely small deviation from the optimal and most energy-efficient solution. Still, the other way round, the overall optimal solution can be determined at a greater amount of time.

6.2.3 Interconnect Impact

With increasing number of instances in the memory subsystem, the impact of the interconnect is expected to grow. Table 6.3 lists both, area and power consumption figures for memory blocks and the required bus as taken from

	basicmath	**bitcount**	**blowfish**	**crc32**	**dijkstra**	**fft**	**huffde**	**ippktcheck**
$\vert M^{\alpha}\vert$	8	4	5	4	5	8	8	8
Memory area [%]	98.84	98.87	99.93	99.96	99.89	99.77	99.95	99.96
Bus area [%]	1.16	1.13	0.07	0.04	0.11	0.23	0.05	0.04
Memory power [%]	84.36	82.30	87.34	94.26	86.97	84.29	81.87	88.93
Bus power [%]	15.64	17.70	12.66	5.74	13.03	15.71	18.13	11.07

	ipres	**md5**	**patricia**	**qsort**	**rijndael**	**sha**	**susan**	**tcp**
$\lvert M^\alpha \rvert$	8	8	8	4	8	4	8	8
Memory area [%]	99.99	99.96	99.96	99.93	99.93	99.97	99.53	99.97
Bus area [%]	0.01	0.04	0.04	0.07	0.07	0.03	0.47	0.03
Memory power [%]	93.15	93.43	88.41	95.11	85.11	91.82	81.38	92.84
Bus power [%]	6.85	6.57	11.59	4.89	14.89	8.18	18.62	7.16

Table 6.3 Number of allocated memories $\lvert M^\alpha \rvert$ and impact of the interconnect in terms of area and power consumption for $mems_{max} = 8$

an experimental series with $mems_{max} = 8$. Figures for the interconnect are based on simulation of a multiplexer-based VHDL model as introduced above.

The first interesting finding in Table 6.3 is that a constraint of $mems_{max} = 8$ not necessarily leads to an allocation of $\lvert M^\alpha \rvert = 8$ memories. In terms of power consumption it can be observed that with 12 % on average, the impact of the interconnect definitely matters. A closer investigation of these figures shows further that, contrary to expectation, there is no relation between number of allocated memories and impact of the interconnect. In contrast to that, the on-chip area consumption of the interconnect, with an average share of 0.22 %, turns out to be negligible. This is due to the severe on-chip footprint of memory blocks that clearly dominates the impact of the interconnect. This holds especially for benchmarks with high memory consumption. The ipres benchmark for example requires over 2 MiB of storage capacity. In comparison to memory blocks of that size, the footprint of the bus simply does not matter.

6.2.4 Area Consumption Minimization

According to Section 4.2.4, few modifications allow the optimization model to minimize the on-chip area consumption instead of average power. Figure 6.3 depicts the results from this experiment, again in relation to the baseline configuration, which is given by the case with one allowed memory instance, i.e., $mems_{max} = 1$. The plotted results can, on the one hand, be divided into benchmark applications that do not benefit from this optimization step

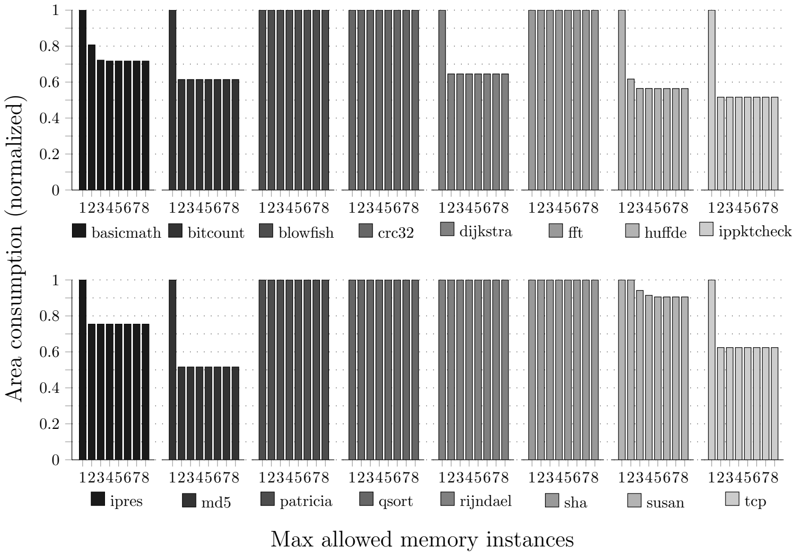


Figure 6.3 Normalized on-chip area consumption with varying $mems_{max}$ constraint

in terms of on-chip area consumption at all. Some examples, on the other hand, benefit from the consideration of two memory blocks instead of one. Additional savings for $mems_{max} > 2$ can only be reported in three cases, basicmath, huffde, and susan.

The corresponding average power figures are given in Figure 6.4. All applications that do not benefit from the area optimization step obviously do not show any change in terms of power consumption as well. For all other cases, connected average power savings highly vary and range from a few per cent in case of ipres to an improvement of over 90 % in case of ippktcheck or md5. A deterioration in contrast to the baseline does not occur in any case.

It can be concluded that similar to on-chip area consumption in the above discussed energy minimization case (cf. Figure 6.2), also no clear trend can be observed for average power when minimizing area. However, this does not mean that a simultaneous consideration of on-chip area and energy consumption is not of interest. A possible approach for the optimization with respect to multiple objective functions is therefore presented in the following section.

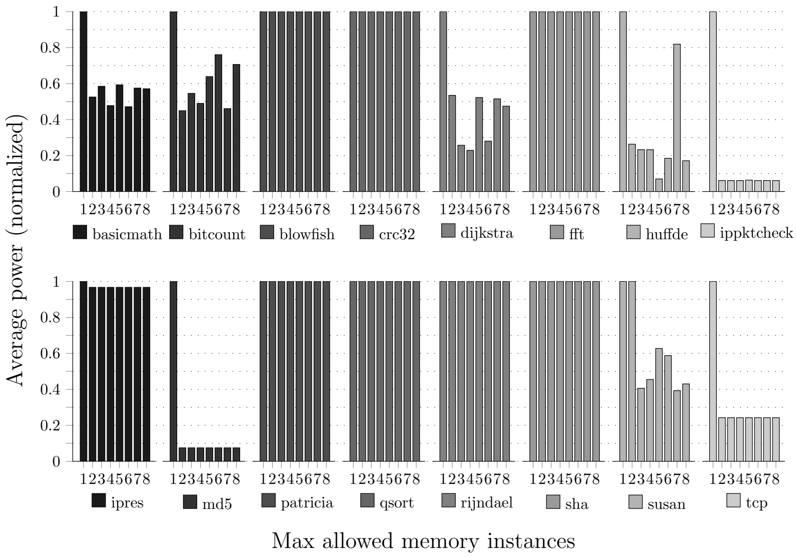


Figure 6.4 Normalized average power consumption after area minimization

6.2.5 Multi-Objective Optimization and Pareto Analysis

When facing system designs with both, energy and area restrictions, the sole minimization of one metric is not effective. The reason is, instead of an energy-optimal solution with probably bad area consumption and vice versa, rather a trade-off solution with acceptable performance on both metrics is wanted. This, however, requires the co-consideration of two objective functions. Following the above presented evaluation steps, dynamic energy minimization with area constraint as well as on-chip area minimization with energy constraint are possible with this optimization model. Using both variations side by side enables the desired design space exploration as follows.

In a first step, energy- and area-optimal configuration are determined without any constraint for on-chip area ($area_{max}$) or average power ($power_{max}$). Those two solutions span the design space of interest and bound one dimension each. In a set of subsequent experiments, either the value of $area_{max}$ in dynamic energy minimization or the value of $power_{max}$ in on-chip area minimization is iteratively altered within the previously determined design space limits. Doing this, the number of resulting trade-off configurations can be influenced by varying the constraint interval. Also a combination of results from both, energy and area optimization experiments according to this iterative procedure is effective in order to increase the coverage of possible solutions in the design space. All pareto-optimal, i.e., dominant configurations are finally

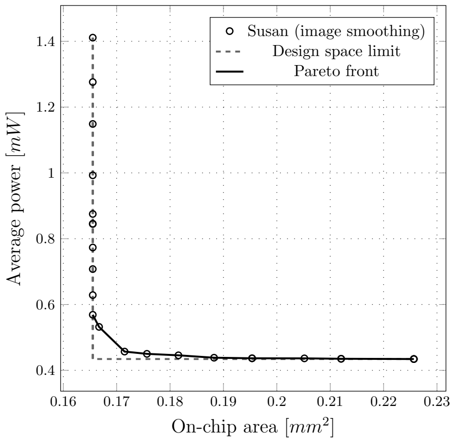


Figure 6.5 Multi-objective optimization results and pareto curve for Susan benchmark

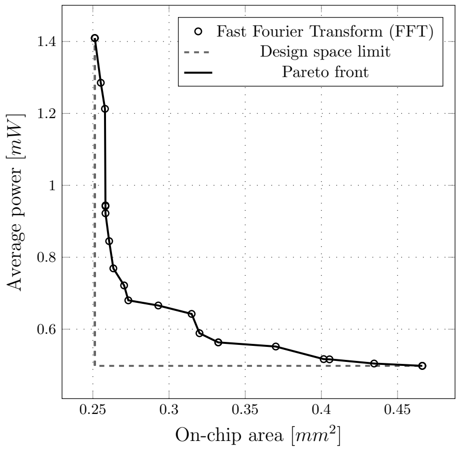


Figure 6.6 Multi-objective optimization results and pareto curve for fft benchmark

connected to a curve that is referred to as pareto front. The best trade-off configuration can then be chosen from this pareto front in dependence on the limitations as given by the system design.

Figure 6.5 illustrates design space, resulting configurations, and pareto curve for the susan benchmark application. Notable in this example is the fact that all obtained configurations do not or only slightly deviate from the energy- and area-optimal design space limits. The best trade-off configurations are therefore easily identifiable in this example.

This is different for the fft benchmark example as depicted in Figure 6.6. Here, the individual configurations differ more from each other, which leads to a pareto curve with a rounder shape. In any case, both examples clearly show that multi-objective optimization enables more informed design decisions than the isolated optimization of one objective function only.

6.2.6 Results for Multitasking Systems

Another variation of the optimization model enables the handling of multitasking scenarios that follow a pre-defined static schedule (cf. Section 4.2.4). The following experiments evaluate possible energy savings and resulting on-chip area consumption on the example of four use cases as listed in Table 6.4.

All use cases can be seen as periodic stream or data processing examples. A given static schedule specifies the number of task activations per period as used for this evaluation series. θ_a accordingly specifies the resulting share of total execution time per task. In caseA, the exemplary system comprises a decryption step (blowfish), some processing of the packet content (bitcount), and finally the encryption of the determined result for example (rijndael). IP communication with Huffman coding and MD5 checksum is represented

	Task set	**Task duty cycle**
caseA	A = {blowfish, bitcount, rijndael}	$\theta_a = \{\frac{1}{5}, \frac{3}{5}, \frac{1}{5}\}$
caseB	A = {ipres, ippktcheck, md5, huffde}	$\theta_a = \{\frac{1}{7}, \frac{4}{7}, \frac{1}{7}, \frac{1}{7}\}$
caseC	A = {patricia, qsort, susan, tcp}	$\theta_a = \{\frac{1}{6}, \frac{1}{3}, \frac{1}{6}, \frac{1}{3}\}$
caseD	A = {basicmath, dijkstra, fft, crc32, sha}	$\theta_a = \{\frac{1}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5}\}$

Table 6.4 Overview on the set of experimental multitasking use cases

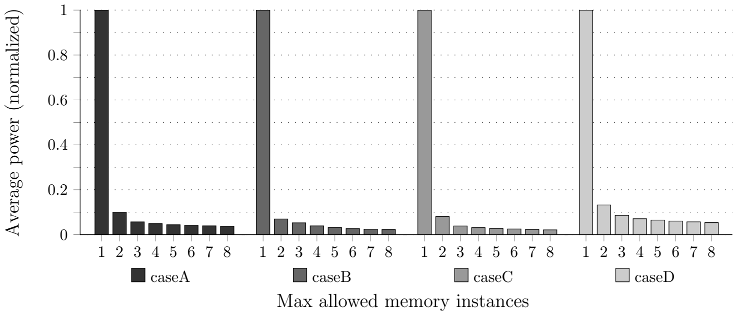


Figure 6.7 Average power consumption for multitasking with varying $mems_{max}$

by caseB. The example in caseC combines kernels from automotive and industrial applications (qsort and susan) with others from the networking domain (patricia and tcp). The last caseD intends to mimic a multitasking setup that combines compute-intensive mathematical operations.

The resulting figures from dynamic energy minimization for all investigated use cases are illustrated in terms of normalized average power in Figure 6.7. Compared to the above presented results for individual benchmarks (cf. Figure 6.1), the effect from utilizing a second memory block is even more visible in this plot. In total and with up to 8 allowed memory instances, average savings of over 96 % can be reported. Main explanation for this improvement is the higher memory footprint of the above listed multitasking examples as

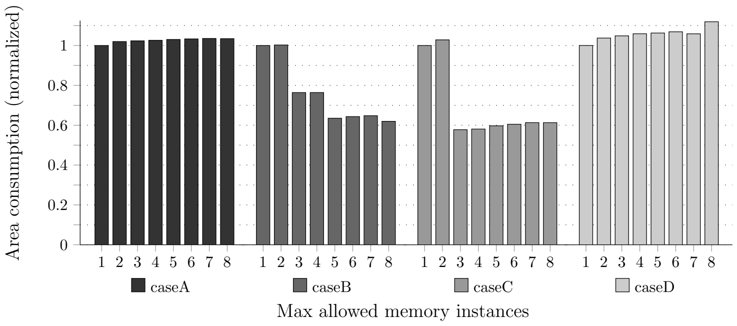


Figure 6.8 On-chip area consumption for multitasking after energy optimization

compared to individual benchmarks alone. This fact favours dynamic energy savings as in most cases only few application profiles make up for the major part of memory accesses (cf. Table 6.1).

In terms of on-chip area consumption as illustrated in Figure 6.8, it once again depends on the use case whether the optimization results in an improvement or a slight degradation.

6.3 SRAM Operation Mode Scheduling Results

The following experiments evaluate the static power optimization method as presented in Section 4.3. The design space again comprises CACTI-based memories for an assumed system operation frequency of 100 MHz. More detailed, 21 single-banked memories are considered with low-standby-power cell type and a storage capacity of 2^x and $x \in [6, 26]$, i.e., covering the range from 64 B to 64 MiB.

Below, an investigation of the general saving potential as provided by memory low-power modes is provided in Section 6.3.1 at first. Next, details on experimental results for all variations of the two-stage static power optimization concept are presented in Section 6.3.2. After that, Section 6.3.3 puts the focus on solving performance and scalability of this optimization method. Possible overhead as resulting from code insertions and modifications is investigated in Section 6.3.4. The application of this optimization method to a sensor hub system is presented as use case in Section 6.3.5. Last, the definition of corridor constraints is analyzed in more detail in Section 6.3.6.

6.3.1 Break-Even Point Analysis

The following general evaluation is based on a metric that is referred to as *break-even point*. It describes the minimum number of cycles that a memory instance at least has to be operated in a specific low-power mode such that the activation of this mode pays off. In other words, the break-even point specifies the minimum number of low-power mode active cycles that are needed in order to equalize the energy penalty due to activation and deactivation. The comparison covers the technology nodes 65 nm, 45 nm, and 32 nm and uses the values for memory energy and power consumption according to Table 6.5.

Variable	**Description**
E^r	Total dynamic read energy per access
E^w	Total dynamic write energy per access
$P^s(ACT)$	Total static power of a bank
$P^s(LS)$	According to the author of [90] set to: $0.58 \cdot P^s(ACT)$
$P^s(DS)$	According to the author of [90] set to: $0.3 \cdot P^s(ACT)$
$E^{a/d}(ACT \rightarrow LS)$	Set configuration register, extra cycle
$E^{a/d}(ACT \rightarrow DS)$	Set configuration register, extra cycle
$E^{a/d}(LS \rightarrow DS)$	Set configuration register, extra cycle
$E^{a/d}(LS \rightarrow ACT)$	Set register, two extra cycles, reverse biasing [90]
$E^{a/d}(DS \rightarrow ACT)$	Set register, three extra cycles, reverse biasing and power gating
$E^{a/d}(DS \rightarrow LS)$	Set register, three extra cycles, reverse periphery power gating

Table 6.5 Experimental memory energy and power figures

The values for E^r, E^w, and $P^s(ACT)$ are provided by CACTI. Same holds for the energy penalty on deep sleep mode deactivation $E^d(DS)$, which denotes the energy consumption for the deactivation of memory periphery power gating. This includes the energy figures for the activation of the data array plus the value for pre-charging. Other penalties result from extra energy consumption due to setting of a memory-mapped configuration register and resulting additional execution cycles. The set of instructions as needed for the implementation of a completely software-based operation mode switching (cf. Chapter 5) is excluded from this evaluation step, which aims to show a general trend. Please note that all parameters to optimization method and tool workflow as used in this and following evaluation steps aim to model a realistic setting but can easily be replaced if other memory characteristics are available, as for example given in an industrial environment.

The conducted experiments evaluate the break-even point for light (LS) and deep sleep (DS) low-power mode. In both cases, activation and deactivation is expected from and to active operation mode (ACT). The result of this investigation as depicted in the log-log plot in Figure 6.9 allows for three general observations:

1. The break-even point considerably decreases with increasing memory size.
2. The slope of the log-log plot is not affected by the technology node.
3. The number of measured break-even cycles decreases by $\frac{1}{3}$ to $\frac{1}{2}$ with every technology step.

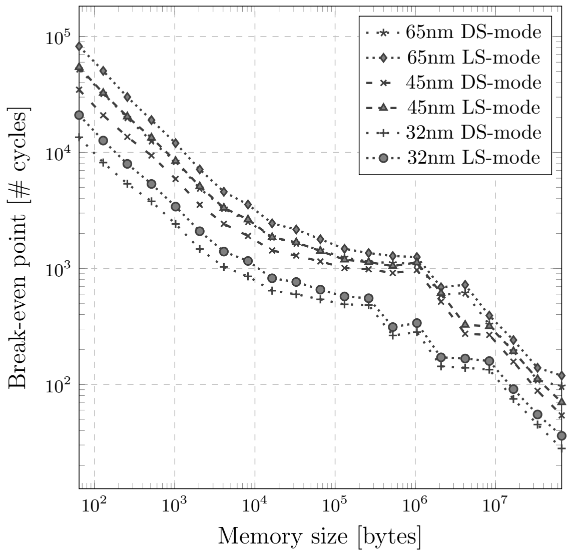


Figure 6.9 Break-even point analysis (log-log plot)

This leads to the conclusion that the impact of low-power modes considerably increases with advancing miniaturization and that higher saving potential can be expected from applications with large data footprints and a demand for appropriate memories.

6.3.2 Static Power Minimization

The following section presents the experimental results for the two-stage static power optimization method according to Section 4.3.2. The conducted evaluation on the example of 11 benchmark applications covers all combinations of first and second optimization stage. That is, either graph partitioning, min-cut clustering, or modularity clustering are used for the determination of memory allocation α and application binding β. Similar to experiments in Section 6.2, the allocation constraint for the maximum number of allowed memories is set to $mems_{max} = 8$. Subsequently, the MIQP formulation according to Appendix B is applied for the scheduling of memory operation modes γ. In a last step, the code generation tool (cf. Chapter 5) is used for the implementation of α, β, and γ on the software level. Operation mode handling in this experimental setup is realized using the stack-based solution according to Section 5.3.2. Due to the adjustments of memory operation modes at run-time and the accordingly inserted code for the implementation of γ, all experimental figures as presented and discussed in this section result from Instruction Set Simulation (ISS) with activated memory power state machine. This enables detailed comparative figures for energy and power consumption of the memory subsystem before and after optimization, while respecting different memory states and possible overhead from inserted code.

Recall that optimization stage 1 and resulting allocation and binding aim at a reduction of the dynamic energy consumption. At the same time, locality in terms of memory accesses is respected in this step as to create a basis for optimization step 2, where static power is reduced on top. This is achieved by minimizing static energy due to leakage as well as the energy overhead that results from switching between active operation and memory low-power modes. Figure 6.10 depicts the experimental results in terms of energy consumption of the memory subsystem after both optimization steps for a memory subsystem in 32 nm technology node. The baseline configuration that is used for comparison consists of a single memory block only. That means, for every benchmark application, the smallest but large enough memory resource from the above mentioned set of 21 memories is chosen for this basic setup.

The investigation of Figure 6.10 allows for several observations. In general, obtained energy savings from this optimization method highly vary from one

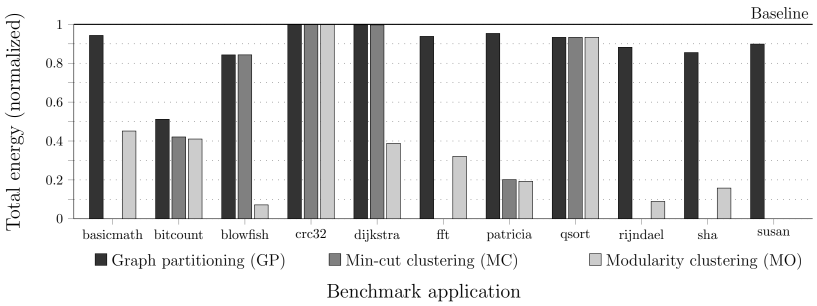


Figure 6.10 Normalized total energy consumption of the memory subsystem after two-stage static power optimization compared to the baseline (technology node: 32 nm)

application to the other. The optimization flow with graph partitioning applied in stage 1 yields a solution for all 11 benchmarks, however, only reaches an average reduction of 11 %. Min-cut clustering turns out to be extremely memory-intensive for applications with close to or above 100 application profiles, i.e., basicmath, fft, or susan (cf. Table 6.2). The demand in these cases even exceeds the memory limit of the host platform, which leads to a termination of the heuristic without result (cf. Figure 6.10 and Table 6.6). For the applications rijndael and sha, the utilized memory limit of $mems_{max} = 8$ is violated by the min-cut solution, thus there is no result there as well. The

Total memory energy consumption [mJ]	**basicmath**	**bitcount**	**blowfish**	**crc32**	**dijkstra**	**fft**	**patricia**	**qsort**	**rijndael**	**sha**	**susan**
65nm											
Base	22.65	1.48	16.58	1.60	23.56	25.55	25.83	1.29	43.65	10.19	18.05
GP	21.33	0.86	13.93	1.60	23.51	23.93	24.77	1.20	38.61	8.70	16.18
MC	▓ x	0.72	13.93	1.60	23.49	▓ x	5.03	1.20	░ x	░ x	▓ x
MO	10.81	0.71	0.95	1.60	5.97	8.49	4.80	1.20	3.11	1.50	░ x
45nm											
Base	12.45	0.80	9.65	0.93	13.71	14.61	17.10	0.72	25.37	5.92	10.12
GP	11.73	0.44	8.10	0.93	13.68	13.69	16.34	0.68	22.46	5.06	9.08
MC	▓ x	0.36	8.10	0.93	13.67	▓ x	3.26	0.68	░ x	░ x	▓ x
MO	5.77	0.36	0.52	0.93	3.37	4.72	3.10	0.68	1.71	0.86	░ x
32nm											
Base	6.84	0.43	3.54	0.34	5.02	8.48	6.33	0.43	9.34	2.18	5.66
GP	6.45	0.22	2.98	0.34	5.01	7.95	6.04	0.40	8.24	1.87	5.08
MC	▓ x	0.18	2.98	0.34	5.00	▓ x	1.28	0.40	░ x	░ x	▓ x
MO	3.09	0.18	0.25	0.34	1.95	2.72	1.22	0.40	0.83	0.34	░ x

Base	– Baseline configuration without low-power modes and one memory block only
GP	– Result with graph partitioning used in optimization stage 1
MC	– Result with min-cut clustering used optimization stage 1
MO	– Result with modularity clustering used in optimization stage 1
░	Solution exceeds maximum number of allowed memory instances $mems_{max} = 8$
▓	Exceeded memory limit of host platform or time limit of 2 h

Table 6.6 Total energy consumption after two-stage static power optimization for different stage 1 clustering methods and compared to non-optimized baseline

remaining, successful experiments that use min-cut clustering as optimization step 1, result in an average energy reduction of 26 %. By far the best results in terms of energy savings, however, are possible when allocation and binding are computed with modularity clustering. In the depicted set of experiments, this setup yields average savings of 60 %. Only the solution for the benchmark susan exceeds the $mems_{max}$ constraint. Therefore, it is not forwarded to the second optimization stage, i.e., excluded from further consideration.

Table 6.6 lists the absolute energy consumption numbers for all conducted experiments. The given figures show, on the one hand, that the total energy consumption almost halves with every technology step towards a smaller structure size, i.e., when moving from 65 nm over 45 nm to 32 nm. On the other hand, irrespective of the chosen technology node, the experiments show the same pattern concerning examples that exceed host memory limit or $mems_{max}$ constraint. Most notable, however, is the observation of percentage savings that appear to be equal in all cases and thus independent of the technology node. Figure 6.11 visualizes this aspect by illustrating the normalized total energy consumption for different technology nodes after optimization. Here, absolutely no trend is visible, which is particularly striking as the above discussed break-even point analysis (cf. Section 6.3.1) suggests higher savings for static energy with increasing miniaturization.

To further investigate this finding, Table 6.7 provides a breakdown analysis of dynamic and static energy consumption before and after optimization in 32 nm. The table shows that already in the baseline configuration with only one memory block, the static energy consumption E^{stat} amounts to a share

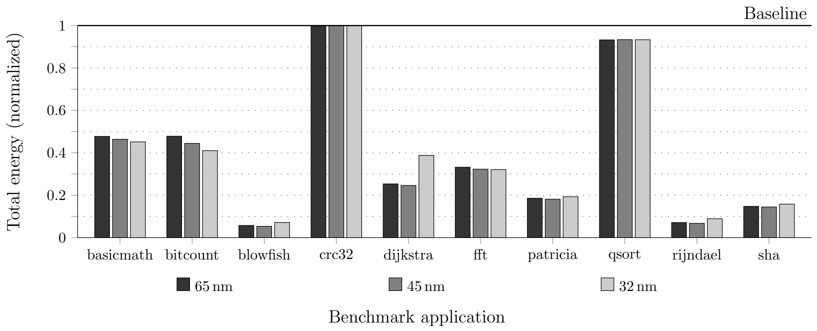


Figure 6.11 Normalized total energy consumption of the memory subsystem after two-stage static power optimization in different technology nodes (stage 1: modularity)

		basicmath	**bitcount**	**blowfish**	**crc32**	**dijkstra**	**fft**	**patricia**	**qsort**	**rijndael**	**sha**
Base [mJ]	E^{dyn}	6.793	0.430	3.482	0.334	4.940	8.405	6.243	0.423	9.187	2.148
	E^{stat}	0.043	0.003	0.056	0.005	0.079	0.074	0.092	0.004	0.151	0.036
Opt. [mJ]	E^{dyn}	3.048	0.176	0.232	0.334	1.897	2.662	1.178	0.392	0.774	0.329
	E^{stat}	0.034	0.002	0.020	0.005	0.050	0.047	0.042	0.006	0.057	0.015
Sav. [%]	E^{dyn}	55.13	59.10	93.34	0.02	61.60	68.32	81.13	7.21	91.57	84.67
	E^{stat}	13.49	39.47	62.87	-1.25	36.81	19.94	52.93	-48.67	61.52	57.14
	Total	54.87	58.98	92.85	0.00	61.21	67.90	80.72	6.68	91.09	84.22

Base	–	Baseline configuration without low-power modes and one memory block only
Opt.	–	Optimized case after modularity clustering (stage 1) and MIQP (stage 2)
Sav.	–	Savings after optimization compared to the baseline

Table 6.7 Energy consumption breakdown before and after optimization (stage 1: modularity, technology node: 32 nm)

of only 1.24 % on average for the set of investigated benchmark applications. The remainder with over 98 % on average instead is attributed to dynamic energy consumption E^{dyn} as resulting from read and write activity. This is strange as for example Kim et al. [67] write of low-power design challenges through leakage with feature sizes below 100 nm and even cite projections of the static power consumption share through leakage in caches of 70 % in 70 nm technology. A closer investigation of related work in the field of static power optimization further reveals certain workarounds. Kandemir et al. [57] for example work with dynamic energy figures from CACTI but skip its static power values and assume leakage energy instead to be equal to the dynamic energy consumption of a memory access. Similar assumptions are used in [64] and [74]. Later work by Loghi et al. [78] even completely avoids memory simulation-based values as for example provided by CACTI. This work uses different imaginary dynamic-static energy ratios instead for the purpose of evaluation. In conclusion, the simulation-based values for static power as provided by CACTI [95] are not realistic. For that reason, further evaluation in terms of dynamic and especially static energy savings is presented with focus on percentage and not in terms of absolute reduction.

A closer investigation of Table 6.7 reveals that E^{dyn}, which is affected by optimization stage 1 only, can be reduced by 60 % on average through allocation and binding as determined by the modularity clustering heuristic. Minimization of E^{stat} is covered by optimization stage 2 and amounts to a reduction of 29 % on average for this set of experiments. A notable deterioration for E^{stat} in case of the qsort benchmark in contrast to considerable savings for blowfish, however, suggest an even more application-specific character of stage 2 as compared to the dynamic energy optimization part in stage 1. Please note that these figures already contain energy penalties that result from memory operation mode switches as well as the additionally inserted instructions that implement this functionality in software. A closer evaluation of this aspect is given in the subsequent Section 6.3.4.

All in all it can be stated that the proposed static power optimization method including code insertions works. Achieved savings turn out to be highly application-specific but dynamic energy savings of partly over 90 % and relative static energy reductions of more than 60 % in selected cases are a promising result. Unfortunately, unrealistically low static power figures as provided by CACTI avoid a more detailed analysis of absolute savings as well as the impact of the technology node.

6.3.3 Solving Performance and Scalability

The measured solving times of all two-stage static power optimization flows are summarized in Table 6.8. Graph partitioning, which is highly flexible as the number of wanted partitions or clusters is configurable, performs extremely fast, however, provides only poor dynamic energy savings as discussed above. Min-cut clustering suffers from extremely high memory consumption and solving times as compared to the other stage 1 heuristic solutions, which makes it a quite unreliable choice. Modularity clustering, which provides by far the best results in terms of dynamic energy consumption reduction yields an excellent performance with an average solving time below 2 s. This makes it a good heuristic alternative for dynamic energy minimization.

Operation mode scheduling as performed by the MIQP in optimization stage 2 performs fast and within acceptable time for memory configurations of less than 8 memories. Beyond that limit, solvability worsens drastically. That is, with $|M^{\alpha}| = 8$, an optimal solution is only found for benchmark examples with only few code profiles. In the case of basicmath and $|P^c| > 50$,

		basicmath	**bitcount**	**blowfish**	**crc32**	**dijkstra**	**fft**	**patricia**	**qsort**	**rijndael**	**sha**	**susan**
	$\|P^c\|$	55	12	8	4	11	35	19	7	9	11	40
Solving time [s]	GP	0.25	0.27	0.47	0.04	2.39	0.26	5.28	0.07	0.54	0.15	0.34
	$\|M^\alpha\|$	4	5	7	6	8	4	8	6	7	7	4
	MIQP	0.06	0.05	1.62	0.07	16.30	0.05	39.40	0.21	0.15	3.18	0.05
	MC	▓ x	22.03	0.03	0.01	848.47	▓ x	606.15	0.01	23.66	0.09	▓ x
	$\|M^\alpha\|$	-	9	8	8	8	-	8	7	12	9	-
	MIQP	-	13.67	19.39	0.87	81.29	-	192.8	1.66	▓ x	▓ x	-
	MO	13.16	0.16	0.05	0.02	0.40	3.04	0.32	0.03	0.24	0.06	4.22
	$\|M^\alpha\|$	8	7	6	4	5	7	6	4	5	5	10
	MIQP	░ 7200	4.28	0.59	0.03	0.06	267.25	1.12	0.03	0.1	0.06	▓ x

$\|P^c\|$	– Number of active code profiles as considered in optimization stage 2
GP	– Graph partitioning (optimization stage 1)
MC	– Min-cut clustering (optimization stage 1)
MO	– Modularity clustering (optimization stage 1)
$\|M^\alpha\|$	– Number of allocated memories after optimization stage 1
MIQP	– Operation mode MIQP (optimization stage 2)
░	Stopped after a time limit of 2 h (feasible solution only)
▓	Stopped after a time limit of 2 h (no solution)

Table 6.8 Static power optimization solving times (average over all experiments)

only a feasible solution is possible within the set time limit of 2 h. For cases with $|M^\alpha| > 8$, a successful solution within a reasonable time becomes quite unlikely. This is due to the fact that the size of the solution space considerably grows with every additional memory resource (cf. Section 4.3.2).

6.3.4 Code Modification and Generation Overhead

Besides static energy savings from the second optimization stage, the question remains, what overhead is caused by the inserted code snippets in terms of additional execution time.

In Section 6.3.2 and Table 6.7, the deterioration of almost 50 % for the static energy consumption in case of the qsort application already attracted attention. The experimental figures for the application execution time after optimization and code modification as depicted in Figure 6.12 now yield the explanation. With an over 40 % higher application execution time as compared to the baseline system, the impact of inserted code seems unproportionately high in this case. Same holds for the benchmark application bitcount. This impression is reinforced as the simulation of these optimized binaries yields that just 48 penalty cycles in case of bitcount and only 5 extra cycles for qsort are actually used for memory operation mode changes. In comparison to several million total cycles, this penalty is not visible at all and definitely no explanation for the reported increase in execution time and static energy consumption. A closer investigation of application sources and optimized assembly code finally delivers the answer. Both benchmarks make heavy use of pointer-based function calls. As these jumps are executed indirectly via a register, the code generation tool is not able to detect them automatically, which leads to a redirect via the software-based operation mode handler `lpmJump` (cf. Section 5.3.2) in each case, even though no operation mode change is indicated. This effect sums up to the observed increase of close to or even above 40 %. Manual correction, however, is possible in this situation, what allows to reduce the overhead to the above mentioned penalty cycles. All other investigated examples instead show an average increase in execution time of 1.82 %, which is considered acceptable.

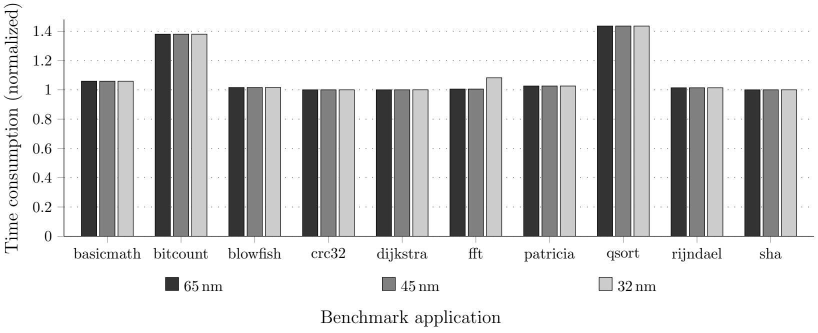


Figure 6.12 Normalized application execution time after optimization and code generation step for different technology nodes (stage 1: modularity)

6.3.5 Evaluation of a Sensor Hub System

While the above examples on the basis of benchmarks are well-suited for a basic evaluation, they are still only able to mimic a real system. In order to fill that gap, further experiments with a virtual system prototype of a single-core sensor hub as inspired by the Bosch Sensortec BMF055 device [17] are presented below. This common ARM-based demonstration platform resulted from a cooperation with the University of Rostock and the University of Tübingen in the research project CONFIRM [38].

For this platform, in combination with a firmware that implements the detection of a tap gesture, the results from static power optimization as presented in Table 6.9 can be given.

	Unit	**Baseline**	**Optimized**	**Savings**
Allocated memories $\lvert M^{\alpha} \rvert$		1	5	
Dynamic energy E^{dyn}	nJ	188097.38	61134.83	67.50 %
Static energy E^{stat}	nJ	340.22	205.39	39.63 %
Peak power P^{p}	mW	0.967	0.536	44.57 %
Execution cycles		19489757	19495554	–0.03 %

Table 6.9 Static power optimization results from experiments with a gesture detection firmware on a virtual system prototype of a sensor hub (technology node: 45 nm)

Altogether, also in this sensor hub use case, the proposed static power optimization method works well and above findings from experiments with benchmarks are confirmed. Besides considerable relative savings for dynamic and static energy at an ignorable increase of execution time, also peak power is positively affected in this example. However, please note that again CACTI-based memory figures are used for evaluation. That means, absolute static energy values as well as peak power consumption and related saving ratio are expected to be different in a real fabricated system.

6.3.6 Corridor Constraints

Finally, this section discusses the corridor constraint feature of the operation mode scheduling MIQP. This part of the optimization model allows for the

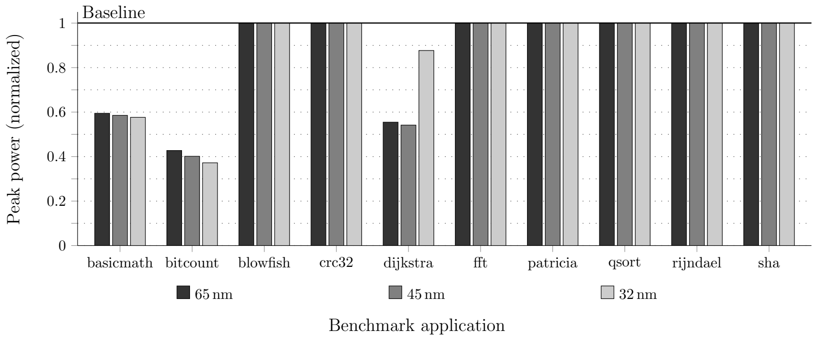


Figure 6.13 Normalized peak power after two-stage static power optimization for different technology nodes (stage 1: modularity)

definition of acceptable time and peak power ranges and by that, to trade a reduction of the maximum power consumption level for an increase in execution time. Unfortunately, the evaluation of this constraint pair is again restricted by the imperfect memory figures. In fact and due to the above discussed small static power share in CACTI-simulated SRAM memories (cf. Table 6.7), peak power is dominated by read and write accesses and resulting dynamic energy consumption. That means, the largest memory resource with highest dynamic energy consumption on access always dominates the peak power metric. In consequence, these memory figures leave no margin for the variation of peak power in the second optimization stage.

Nevertheless, the experimental peak power results as depicted in Figure 6.13 allow for one finding that is related to the above mentioned impact of large memory instances in terms of dynamic energy consumption. While most benchmarks show no effect in Figure 6.13, a considerable peak power reduction is given for the examples basicmath, bitcount, and dijkstra. Closer investigation of these cases turns out that the optimized memory allocation contains exclusively smaller memory blocks as compared to the baseline. Same holds for the above discussed sensor use case (cf. Section 6.3.5). In consequence, a peak power reduction of the memory subsystem can be achieved by a strictly split memory configuration, i.e., by means of allocation.

6.4 Multi-Core Memory Optimization Results

The experiments in this section evaluate the presented memory optimization method for Multi-Processor System-on-Chip (MPSoC) platforms according to Section 4.4. This includes the binding of application profiles to memory and the scheduling of operation modes. The used experimental platform (cf. Figure 4.7) is equipped with four processor cores and inspired by the Infineon Aurix architecture [53]. Memory allocation is pre-defined by the chosen platform and exemplarily set to scratchpad memories of 64 KiB and a global shared memory of 8 MiB. Frequency domain (A), which includes processor cores and scratchpads, is operated at 250 MHz, frequency domain (B) and thus the shared memory runs at 50 MHz. All memory resources are modeled as two-port instances in 32 nm technology node. Code generation for the implementation of a determined memory operation mode schedule γ follows the activation-on-access scheme according to Section 5.3.1 and further uses the adaptation for multi-core systems as discussed in Section 5.4.

Section 6.4.1 at first presents a general evaluation of the two-stage optimization method for multi-core as introduced in Section 4.4.2 . This includes both, savings in terms of dynamic energy from application binding β as well as static power reduction through an optimized operation mode scheduling γ. Section 6.4.2 separately evaluates and compares cache- and scratchpad-based memory subsystems in multi-core SoC designs.

6.4.1 Dynamic Energy and Static Power Minimization

Dynamic power management through clock or power gating is widely used in low-power SoC design (cf. for example [96]). Especially in multi-core environments, where inter-core dependencies and varying system load rarely cause all cores of the system to be fully occupied, dynamic activation of sleep modes is a widely adopted technique to keep the overall power consumption of these MPSoC devices manageable and within a tolerable range. For deterministic multi-core setups with periodic schedule, where individual tasks are computed by different processing units (PU) and inter-core communication only takes place in between subsequent cycles, two different operation schemes can be distinguished. This stems from the fact that always the most time consuming task determines the maximum iteration frequency or minimum period respectively. Figure 6.14 schematically illustrates the corresponding operation

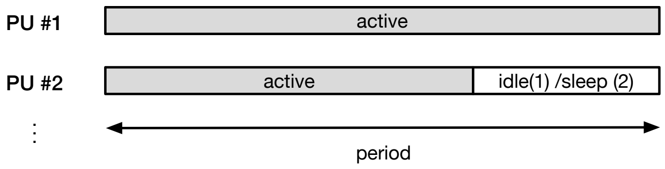


Figure 6.14 Run-to-idle (1) and run-to-sleep (2) multi-core system operation scheme

modes, run-to-idle and run-to-sleep. In run-to-idle, on the one hand, inactive processor time is spent in an active idle state, what entails ongoing instruction fetches. Run-to-sleep, on the other hand, schedules a sleep mode activation for all processing units that finish early. Thus, no further memory accesses of these PUs take place in this time frame.

Considering this MPSoC system setup, the following evaluation is based on ten experiments according to Table 6.10. All use cases are built from individual benchmarks of the MiBench suite. For this set of examples, Figure 6.15 depicts the results with processing units operated in run-to-idle configuration.

The baseline in this experimental series is referred to as *β Heuristic* and describes a reference implementation of the MPSoC scratchpad mapping algorithm as proposed in [51]. *β MILP* marks the energy consumption after the first memory optimization stage for multi-core, i.e., with an optimal application-specific binding (cf. Section 4.4.3). The figures as represented by the last bar

	PU #1	**PU #2**	**PU #3**	**PU #4**
case1	basicmath	qsort	sha	susan_corners
case2	bitcount	crc32	rijndael_enc	susan_smooth
case3	blowfish_enc	dijkstra	rijndael_dec	susan_edges
case4	blowfish_dec	qsort	sha	susan_corners
case5	bitcount	crc32	qsort	sha
case6	blowfish_enc	blowfish_dec	rijndael_enc	rijndael_dec
case7	fft	susan_smooth	susan_edges	susan_corners
case8	rijndael_enc	blowfish_dec	dijkstra	bitcount
case9	basicmath	fft	patricia	susan_smooth
case10	patricia	dijkstra	rijndael_dec	fft

Table 6.10 Overview on the set of experimental multi-core optimization use cases

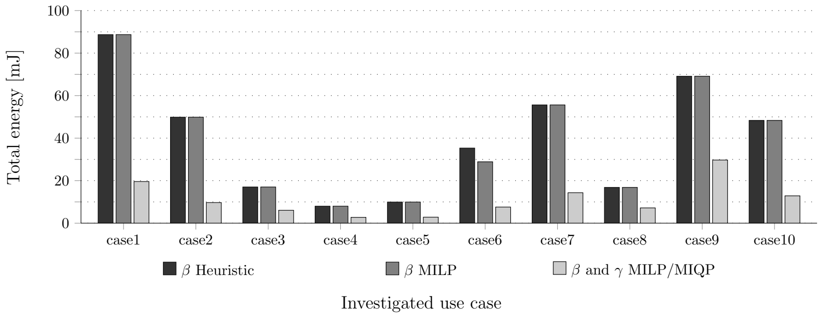


Figure 6.15 Total memory subsystem energy consumption in run-to-idle configuration

(*β and γ MILP/MIQP*) include the second optimization step, i.e., operation mode scheduling on top.

The direct comparison of both binding methods in Figures 6.15 and 6.16 shows that the heuristic solution essentially performs equally good as the linear program. This is striking as *β MILP* is characterized to deliver an optimal solution. *β Heuristic* instead implements a simple algorithm that assigns most frequently used objects in descending order of access frequency to the scratchpads and as long as core-local memory space is available. Consequently, it can be concluded that a good mapping can be reduced to this simple algorithm.

Concerning the static power minimization MIQP in optimization stage 2, there is no difference in terms of findings as compared to the evaluation for single-core systems according to Section 6.3.2. Processor cores are not part of the MIQP formulation and further, from the perspective of the memory subsystem, cause nothing but a different memory access pattern. As still bound to memory figures from CACTI, with unrealistically low static power consumption values, absolute savings are not visible in these experimental figures. Possible relative savings, a breakdown analysis of dynamic and static energy consumption, and resulting conclusions are already given in Table 6.7 respectively Section 6.3.2 and therefore not discussed again. The time overhead through extra code snippets for memory operation mode changes plus further delay from congestion due to the limited number of memory ports amounts to 4 % on average in this setup, which is considered to be acceptable.

Yet, the direct comparison of experimental results for the second optimization stage in Figure 6.15 and Figure 6.16 exposes one interesting finding that is related to dynamic energy from memory accesses. At first glance, it

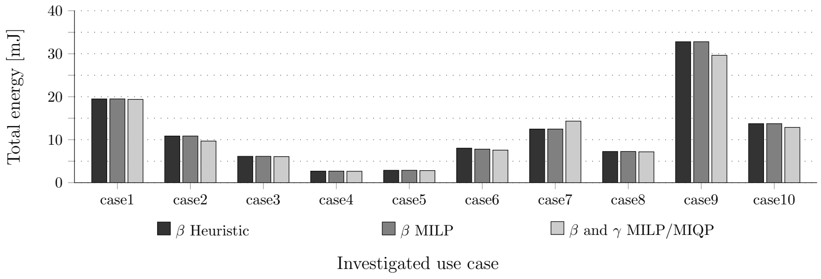


Figure 6.16 Total memory subsystem energy consumption in run-to-sleep configuration

appears that in run-to-idle (cf. Figure 6.15), the application of an operation mode schedule allows for considerable savings of 70 % on average, compared to results from optimized binding only. A direct comparison of given absolute energy values for *β and γ MILP/MIQP* with figures that result from experiments in run-to-sleep configuration, however, reveals basically no difference. In Figure 6.16, further no difference between binding only in optimization stage 1 and the additional application of optimization stage 2 can be observed, which is due to the CACTI-based memory figures. This leads to the conclusion that active idle time in run-to-idle mode, with resulting extra instruction fetches as well as delays through congestion at memory ports, causes the already mentioned 70 % difference on average, which in sum is avoidable overhead. All in all, coupling of processor and memory low-power modes in MPSoC devices turns out to be extremely relevant.

In terms of performance and scalability, all above discussed optimization experiments successfully terminated in less than 100 s. A separate study of this aspect for the above investigated optimization methods is therefore considered to be of no further relevance.

6.4.2 Comparison of Scratchpads and Caches

As many recent MPSoC architectures come with both, traditional caches and scratchpad memories, this section aims at evaluating the differences between these two types of core-local memory. For comparison, a direct mapped cache architecture with 64 KiB cache memory per core and a cache-line width of 32 B

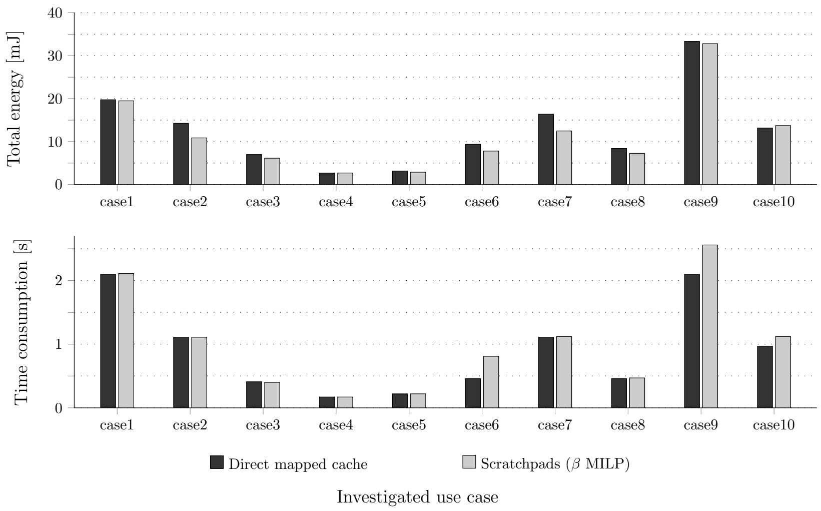


Figure 6.17 Comparison of direct mapped caches and optimized scratchpad-based memory subsystem in run-to-sleep configuration

is used. As introduced above, scratchpads in the used experimental multi-core architecture are of equal size, again 64 KiB per core. The optimized application binding for the scratchpad-based setup is determined by the MILP formulation according to Section 4.4.3 or Appendix C respectively.

As illustrated in Figure 6.17, the performance of both core-local memory concepts is more or less balanced. Still, two observations can be made. First, in terms of energy consumption, scratchpads are slightly ahead. This can be explained by the highly deterministic nature of the investigated use cases, a setup that is commonly known to be beneficial for this type of memory subsystem. Second and in terms of time consumption, however, several use cases show better performance for the cache-based architecture. This can be explained by the ability of traditional hardware-based caches to change content at run-time and in dependence on the memory access pattern. The content of the scratchpads instead is, at least in this set of experiments, static and thus not adjusted during execution. All in all, having both core-local memory types side-by-side seems to be the right way as it allows to combine the benefits of both concepts depending on the application. In addition, unneeded memory blocks can always be switched off.

6.5 STT-RAM Allocation and Binding Results

This section evaluates the memory optimization method for STT-RAM memories according to Section 4.5. The memory design space in the presented set of experiments comprises 19 different memory sizes with storage capacities of 2^x and $x \in [7, 25]$. This corresponds to a range from 128 B to 32 MiB. Per memory size, 6 memory types with different operation voltage are provided. The supported voltage level range is from 1.3 V to 1.8 V in steps of 0.1 V. This amounts to a total number of 114 different STT-RAM memories with access transistor and memory periphery modeled in 32 nm node. Corresponding memory characteristics are based on simulation with STT-CACTI [4].

The following experimental evaluation covers two aspects. Section 6.5.1 starts with an investigation of energy reduction potential and possible impact on memory operation frequency as resulting from this optimization method. Second, the solving performance of the STT-RAM allocation and binding MIQP formulation according to Appendix D is discussed in Section 6.5.2.

6.5.1 Dynamic Energy Minimization

The resulting figures for 16 benchmark applications in terms of total energy consumption of the memory subsystem are depicted in Figure 6.18. Compared to the baseline configuration with one memory block only, the most significant reduction in all cases is already achieved with two allowed memory instances, i.e., $mems_{max} = 2$. Additional savings with a further increment of this constraint are possible but turn out to depend on the application. The measured energy savings for $mems_{max} = 8$ in this experimental series amount to 81 % on average.

In a way, this result reminds of presented figures for the dynamic energy minimization of SRAM memories in Section 6.2.1. That is to say, besides savings in terms of dynamic energy from varying the operation voltage level in STT-RAM, remaining CMOS parts in the memory periphery still cause a quite notable effect from splitting memory blocks into smaller units. A general trend on how much savings result from different write modes in STT-RAM and what share is due to a split memory configuration, however, is difficult to quantify. A conducted additional experimental series with a fixed operation voltage level of 1.5 V for all memory resources, though, allows for the identification of a trend in this regard. These results are not depicted separately as the

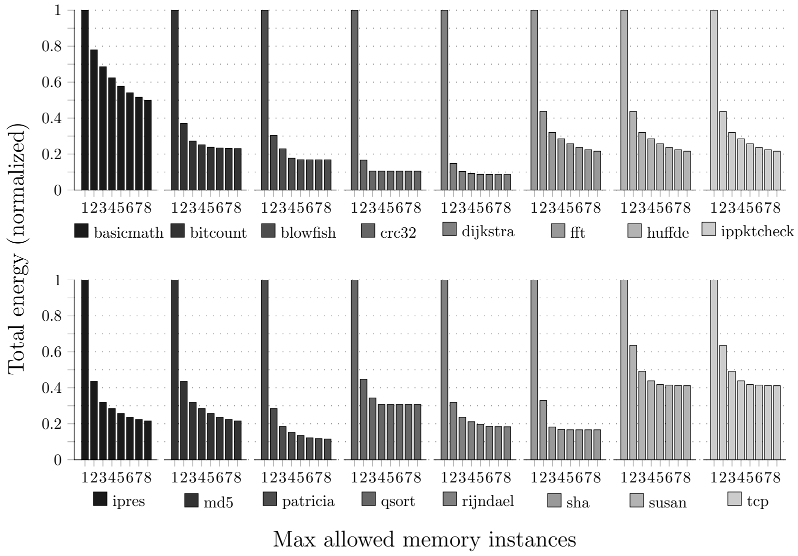


Figure 6.18 Normalized total energy consumption with varying $mems_{max}$ constraint

average energy reduction in this experimental series is again over 80 %. The resulting plot consequently does not visibly differ from Figure 6.18. In total, this evaluation leads to the conclusion that STT-RAM-specific savings from the observed write energy/latency trade-off as adjusted by varying operation voltages have a negligible effect.

The memory access time and thus the maximum supported operation frequency in STT-RAM depends on two things, the memory size and the applied operation voltage level. As both characteristics are modeled as part of the design space and, for that reason, considered a variable in the optimization model, it is further of interest, whether the above discussed energy minimization affects the access performance of the memory subsystem. Figure 6.19 illustrates this aspect in terms of maximum operation frequency of the memory subsystem. In most cases, at least one step towards a higher frequency level can be observed. However, no general trend is emerging as for example in case of the benchmark application tcp, the maximum supported frequency even decreases. This effect of a simultaneous reduction for the part of energy consumption and delay, however, is not related to the STT-RAM operation

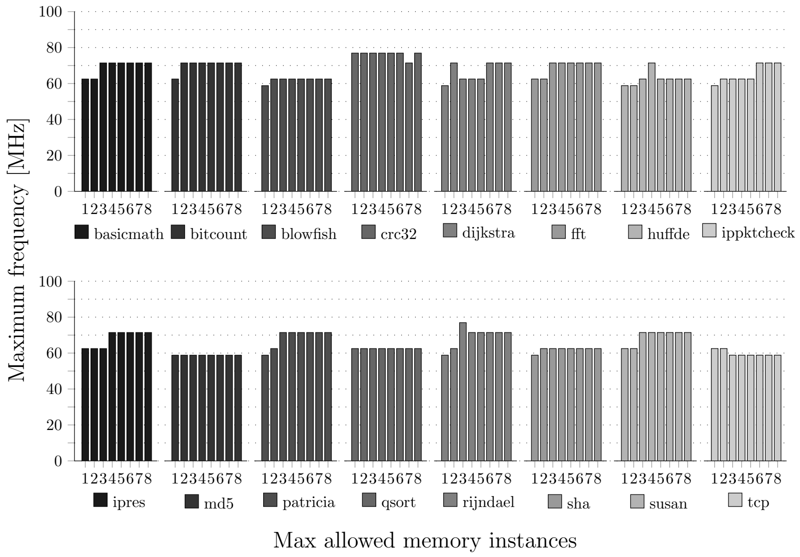


Figure 6.19 Maximum memory operation frequency with varying $mems_{max}$ constraint

voltage but results from varying memory allocation, i.e., the utilization of smaller memory blocks with less access delay.

Compared to the trade-off between dynamic energy and on-chip area consumption as given in case of SRAM memories, the trend of energy and frequency in this optimization model is rather contrary. In other words, decreasing energy consumption in STT-RAM is likely to be accompanied by an operation frequency increase. In consequence, the expected trade-off relationship between these two units is simply overruled by other factors as for example memory size. A separate investigation of frequency minimization as well as pareto analysis is therefore considered to provide no extra information.

Still it should be noted that it remains unclear, whether the utilized memory figures as provided by STT-CACTI are able to accurately model STT-RAM-specific features. The corresponding publication by Arcaro et al. [4] at least is not able to prove the accuracy of this simulation tool as no comparison with actual STT-RAM hardware is documented.

6.5.2 MIQP Performance and Scalability

The performance of the STT-RAM allocation and binding MIQP in terms of solving time is summarized in Table 6.11. Also in this allocation and binding problem, the memory allocation variable α is of binary type in order to enable an unambiguous binding β. Each of the 114 available memory types with varying size and operation voltage level needs therefore to be duplicated $mems_{max}$ times for the respective experiment as to provide enough resources for possible multiple selections of the same memory type. Therefore, any increase of $mems_{max}$ directly causes a growth of the design space.

			basicmath	**bitcount**	**blowfish**	**crc32**	**dijkstra**	**fft**	**huffde**	**ippktcheck**
Solving time [s]	$mems_{max}$	1	0.31	0.15	0.12	0.12	0.15	0.20	0.37	0.23
		2	2.60	10.95	1.15	2.17	5.41	1.67	46.39	9.13
		3	3.34	14.01	3.76	0.82	6.64	4.90	309.20	31.08
		4	3.95	32.14	6.36	2.68	12.23	10.30	263.81	107.91
		5	7.47	16.48	3.78	1.86	9.36	21.03	54.15	274.96
		6	61.36	40.02	2.02	2.31	7.73	54.89	292.70	2355.79
		7	1278.23	61.63	2.53	2.13	5.07	7200	468.83	95.49
		8	2123.83	15.94	3.09	2.46	5.84	7200	2733.04	1853.70

			ipres	**md5**	**patricia**	**qsort**	**rijndael**	**sha**	**susan**	**tcp**
Solving time [s]	$mems_{max}$	1	0.24	0.20	0.13	0.11	0.16	0.12	0.26	0.40
		2	17.31	6.78	2.29	2.07	1.68	2.48	4.14	12.75
		3	125.22	12.48	4.16	4.38	4.86	2.18	10.85	93.38
		4	38.30	31.43	10.83	1.73	6.38	2.42	7.77	60.14
		5	75.42	19.55	17.73	1.61	16.12	1.78	12.33	7200
		6	284.86	34.52	223.04	2.49	7.31	1.57	201.41	94.21
		7	1360.60	69.95	138.92	3.07	115.62	2.50	916.73	242.36
		8	1744.25	18.44	629.50	4.08	19.22	2.84	47.10	1036.22

▨ Stopped after a time limit of 2 h (feasible solution only)

Table 6.11 Overview on STT-RAM MIQP solving time for varying $mems_{max}$

The general trend in Table 6.11 confirms the expected increase in solving time with increasing number of allowed memories. Only three examples exceed the set time limit of 2 h. Still, an acceptable feasible solution is returned in these cases for fft and tcp benchmark applications. Please note that similar to the related SRAM MIQP (cf. Section 6.2.2), it is quite unlikely that for example in case of $mems_{max} = 8$, exactly 8 memories of the same type, i.e., with equivalent size and operation voltage level, mark the optimal solution. Accordingly, also in this case, a decrease of duplicated memory instances to a factor of 4 allows to considerably improve the solving performance of the quadratic program without loss in terms of energy reduction.

Chapter 7
Conclusions

This thesis presents design-time optimization methods that allow for the automated identification of an ideally optimal on-chip memory subsystem in terms of energy and power consumption.

Chapter 2 gives a general introduction to memory optimization in System-on-Chip design and shows that theoretical formulations from embedded system synthesis apply to particular problems in memory optimization. Accordingly, adapted definitions of allocation, binding, and scheduling are presented as to provide a formal basis for the remainder of the thesis. The introduction to on-chip memory technologies helps in clarifying the optimization potential in SRAM and STT-RAM memories, which exists in several ways. Instruction Set Simulation (ISS) and minor contributions to this field are briefly cut as all presented optimization methods are of application-specific type and thus based on input from simulation.

As outlined in Chapter 3, the central distinctive feature of all optimization methods in this work is given by the taken software-centric approach. That is, compared to the state-of-the-art, optimization steps are, if possible, designed in a way that makes an implementation on the software level and without modification of hardware parts possible. This consistent approach is, in conclusion, considered highly relevant as it makes proposed methods applicable to most, if not all system types, no matter if the hardware platform is chosen off-the-shelf or whether it is developed or at least adjusted in-house.

The four memory optimization methods according to Chapter 4 are the main contribution of this thesis. In combination with the corresponding experimental results in Chapter 6, the following conclusions can be drawn.

The first method enables application-specific dynamic energy minimization in SRAM memory subsystems through combined optimization of memory allocation and application binding. Determined results are optimal and show reductions of over 80 % on average when compared to the utilization of a single memory block only. Split SRAM memory subsystem configurations, finally, turn out to be a highly efficient switch in low-power System-on-Chip design. The most significant improvement in this regard is achieved when using two instead of only one memory block. The allocation of further memories, on the other hand, turns out to result in only little extra energy savings.

The second optimization concept shifts the focus towards the minimization of static power from leakage currents in SRAM memories. As simulation-based memory figures from CACTI turn out to model leakage aspects in an insufficient way, conclusions about absolute savings and possible impact from peak power constraints can not be drawn at this point. However, on the example of a use case for a sensor hub system, percentage savings over 60 % for dynamic energy and almost 40 % for static energy consumption prove the functionality and high efficiency of this method. Conducted experiments with benchmark applications provide partially even better results. The lack of availability of industrial-grade memory characteristics leaves further evaluation with hardware-based memory figures as future work.

The evaluation of proposed optimization methods for MPSoC design shows that optimal application-to-scratchpad binding is reducible to an algorithm that assigns most frequently used elements to core-local scratchpads. Concerning caches and scratchpads in the multi-core context, conducted experiments prove benefits for either concept. Combination of both types of core-local memory, as already common practice in recent MPSoC architectures, therefore turns out to be the right way to go.

The last optimization method specifically targets STT-RAM memories and combines the determination of allocation and binding with the selection of an operation mode per memory block. Conducted experiments show that the observed energy/latency trade-off that characterizes the write operation in this storage technology is dominated by dynamic energy as caused by peripheral and access circuitry. Similar to SRAM, future work into the direction of more detailed evaluation with memory figures that are based on real hardware is also indicated here and might change the picture.

Concerning the solving performance of presented optimization methods, large design spaces as even spanned by investigated benchmark applications cause every proposed mixed-integer linear or quadratic problem formulation to reach the limit of solvability, at least when a reasonable time limit is set. In case of the static power optimization problem, the solution flow is moreover already divided in two steps as to keep the problem manageable. In turn, obtained solutions are not guaranteed to be optimal any more. Also, certain workarounds, as for example constraint variations, are partially needed in order to get a result in time. Future work could therefore go into the direction of new solving methods with probably better scalability. This might allow, for example, to solve the static power minimization problem (cf. Section 4.3) in a single step and with optimal result.

The proposed concept for code modification and generation according to Chapter 5 enables the implementation of above mentioned memory optimization results on the software level in an automated way. The prototype implementation of this tool for the ARMv6-M instruction set reveals, however, that most and foremost the involved binary parsing step, which is required for the support of frequently used library code, is not trivial and extremely costly in terms of development time. This point impedes a simple porting of the tool to other platforms with different instruction set architecture and therefore marks another, more practical aspect for future work.

To sum it up, all parts for the application-specific optimization of memory subsystems in System-on-Chip design are covered in this work. The presented optimization methods allow for highly considerable dynamic energy and static power savings and wait for further evaluation in industrial environments.

Appendix

A Combined SRAM Allocation and Binding MIQP

```
# ---------------------------------------------------------------------
#  Application profiles and related parameters
# ---------------------------------------------------------------------
set PROFILE_SET;

# Profile size parameter
param P_MEM_SIZE {PROFILE_SET} >= 0;

# Duty cycle
param P_DUTY_CYC {PROFILE_SET} >= 0, <= 1;

# Read probability
param P_READ_PROB {PROFILE_SET} >= 0, <= 1;

# Write probability
param P_WRITE_PROB {PROFILE_SET} >= 0, <= 1;

# ---------------------------------------------------------------------
#  Memory resources and their characteristics
# ---------------------------------------------------------------------
set MEM_SET;

# Memory size parameter
param M_SIZE {MEM_SET} >= 0;

# Memory on-chip area consumption footprint
param M_AREA {MEM_SET} >= 0;

# Read current per cycle
param M_READ_CURR {MEM_SET} >= 0;

# Write current per cycle
param M_WRITE_CURR {MEM_SET} >= 0;

# Static power consumption
param M_STAT_POW {MEM_SET} >= 0;

# ---------------------------------------------------------------------
#  User-defined limiting constraints
# ---------------------------------------------------------------------
# Supply voltage
param V > 0 default 1;

# Maximum number of memories
param MEMS_MAX integer >= 0 default 8;

# Upper on-chip area consumption limit
param AREA_MAX >= 0 default 0;
```

```
# ---------------------------------------------------------------------
#  Bus parameters (modeled as piecewise linear function)
# ---------------------------------------------------------------------
# Power consumption
param B_POW {1..MEMS_MAX};

# Area footprint
param B_AREA {1..MEMS_MAX};

# ---------------------------------------------------------------------
#  Memory allocation
# ---------------------------------------------------------------------
var MEM_ALLOC {MEM_SET} binary default 0;

var MEM_ALLOC_SUM integer >= 1;
subject to MemAllocSum:
  MEM_ALLOC_SUM = sum {j in MEM_SET} MEM_ALLOC[j];

# Constrain the maximum number of instantiated memories
# If MEMS_MAX = 0, the number of memories is unconstrained
subject to MaxInstances {if MEMS_MAX > 0}:
  sum {j in MEM_SET} MEM_ALLOC[j] <= MEMS_MAX;

# ---------------------------------------------------------------------
#  Application binding
# ---------------------------------------------------------------------
var MEM_MAP {PROFILE_SET, MEM_SET} binary default 0;

# Every application profile must be mapped to exactly one memory
subject to ProfileMapped {i in PROFILE_SET}:
  sum {j in MEM_SET} MEM_MAP[i,j] = 1;

# Memory size feasibility constraint
subject to MemSize {j in MEM_SET}:
  sum {i in PROFILE_SET} MEM_MAP[i,j] * P_MEM_SIZE[i] <=
    M_SIZE[j] * MEM_ALLOC[j];

# ---------------------------------------------------------------------
#  Power consumption model
# ---------------------------------------------------------------------
var MEM_POWER {MEM_SET};
subject to MemPower {j in MEM_SET}:
  MEM_POWER[j] = MEM_ALLOC[j] * M_STAT_POW[j] +
    sum {i in PROFILE_SET} V * MEM_MAP[i,j] * P_DUTY_CYC[i] *
      (P_READ_PROB[i] * M_READ_CURR[j] + P_WRITE_PROB[i] * M_WRITE_CURR[j]);

var BUS_POWER >= 0;
subject to BusPower:
  BUS_POWER = <<{j in 1..MEMS_MAX-1} j;
                {k in 1..MEMS_MAX} B_POW[k]>> MEM_ALLOC_SUM;

# ---------------------------------------------------------------------
#  Area consumption model
# ---------------------------------------------------------------------
var MEM_AREA {MEM_SET};
subject to MemArea {j in MEM_SET}:
  MEM_AREA[j] = MEM_ALLOC[j] * M_AREA[j];
```

```
var BUS_AREA >= 0;
subject to BusArea:
  BUS_AREA = <<{j in 1..MEMS_MAX-1} j;
                {k in 1..MEMS_MAX} B_AREA[k]>> MEM_ALLOC_SUM;

# Constrain the maximum area consumed by memory and bus
# If AREA_MAX = 0, no area limit is set
subject to AreaConstraint {if AREA_MAX > 0}:
  BUS_AREA + sum {j in MEM_SET} MEM_AREA[j] <= AREA_MAX;

# -----------------------------------------------------------------------
#  Optimization goal: Minimize the dynamic energy consumption
# -----------------------------------------------------------------------
minimize AveragePower:
  BUS_POWER + sum {j in MEM_SET} MEM_POWER[j];
```

B SRAM Operation Mode Scheduling MIQP

```
# -----------------------------------------------------------------------
#  Application profiles and related parameters (functions only!)
# -----------------------------------------------------------------------
set FCT_SET;

# Execution time in cycles (here # instructions)
param F_EXEC_TIME {FCT_SET} integer >= 0;

# Dependency matrix between functions
param F_DEPS {FCT_SET, FCT_SET} integer >= 0 default 0;

# -----------------------------------------------------------------------
#  Available memories - as specified by the allocation
# -----------------------------------------------------------------------
set MEM_SET;

# Binary matrix indicating, which function needs which memory to be active
# in order to be executed properly...includes dependencies to data profiles
param F_M_MAP {FCT_SET, MEM_SET} binary default 0;

# -----------------------------------------------------------------------
#  Memory subsystem configuration and related characteristics
# -----------------------------------------------------------------------
set CFG_SET;

# Energy consumption per configuration and cycle
param E_IDLE {CFG_SET} >= 0;

# Peak power (pessimistic) => C_PWR + Max(R/W)
param P_PEAK {CFG_SET} >= 0;

# Binary matrix representing the active memories per configuration
param C_M_ACT {CFG_SET, MEM_SET} binary default 0;

# Energy penalty on configuration change
param E_PENALTY {CFG_SET, CFG_SET} >= 0 default 0;

# Corresponding timing penalty
param T_PENALTY {CFG_SET, CFG_SET} integer >= 0 default 0;

# -----------------------------------------------------------------------
#  Corridor constraint limit values
# -----------------------------------------------------------------------
# Time corridor
param T_MAX integer >= 0;

# Power corridor
param P_MAX >= 0;

# -----------------------------------------------------------------------
#  Variables
# -----------------------------------------------------------------------
# Central variable for mapping between code profiles and configuration
var F_C_MAP {FCT_SET, CFG_SET} binary default 0;
```

```
# Energy consumption per function (idle)
var E_IDLE_SUM {FCT_SET} >= 0;

# Energy penalty due to mode switching
var E_MODE {FCT_SET, CFG_SET} >= 0;

# Overall energy penalty per function
var E_MODE_SUM {FCT_SET} >= 0;

# Timing penalty due to mode switching
var T_MODE {FCT_SET, CFG_SET} integer >= 0;

# Overall timing penalty (needed if T_MAX is set)
var T_MODE_SUM {FCT_SET} integer >= 0;

# ----------------------------------------------------------------------
#  Constraints - feasibility
# ----------------------------------------------------------------------
# Each function shall only be assigned to one and only one configuration
subject to OneCfgPerFct {f in FCT_SET}:
  sum {c in CFG_SET} F_C_MAP[f,c] = 1;

# Assign each function a configuration where required memories are active
subject to ValidCfgBinding {f in FCT_SET, m in MEM_SET : F_M_MAP[f,m]}:
  sum {c in CFG_SET} F_C_MAP[f,c] * C_M_ACT[c,m] = 1;

# ----------------------------------------------------------------------
#  Constraints - related to energy consumption
# ----------------------------------------------------------------------
# Active idle part
subject to EnergyIdle {f in FCT_SET}:
  E_IDLE_SUM[f] =
    F_EXEC_TIME[f] * (sum {c in CFG_SET} (F_C_MAP[f,c] * E_IDLE[c]));

# This vector contains the (energy) penalties for each function
# in configuration c1 to any configuration c2 in CFG_SET
subject to EnergyPenalty {f in FCT_SET, c2 in CFG_SET}:
  E_MODE[f,c2] = sum {c1 in CFG_SET} (F_C_MAP[f,c1] * E_PENALTY[c1,c2]);

# Sum up the energy penalty
subject to EnergyPenaltySum {f1 in FCT_SET}:
  E_MODE_SUM[f1] >= sum {f2 in FCT_SET}
    (F_DEPS[f1,f2] * sum {c in CFG_SET} (E_MODE[f1,c] * F_C_MAP[f2,c]));

# ----------------------------------------------------------------------
#  Constraints - timing and time corridor
# ----------------------------------------------------------------------
# Model the timing penalties due to mode changes. This vector contains
# the (timing) penalties for each f in c1 to any c2 in CFG_SET
subject to TimingPenalty {f in FCT_SET, c2 in CFG_SET}:
  T_MODE[f,c2] = sum {c1 in CFG_SET} (F_C_MAP[f,c1] * T_PENALTY[c1,c2]);

# Sum up the timing penalty - only if T_MAX is set!
subject to TimingPenaltySum {f1 in FCT_SET}:
  T_MODE_SUM[f1] >= sum {f2 in FCT_SET}
    (F_DEPS[f1,f2] * sum {c in CFG_SET} (T_MODE[f1,c] * F_C_MAP[f2,c]));
```

```
# Ensure that period T is within the time corridor
# If T_MAX = 0, no time corridor is set
subject to TimeCorridor {if T_MAX > 0}:
  sum {f in FCT_SET} (F_EXEC_TIME[f] + T_MODE_SUM[f]) <= T_MAX;

# -----------------------------------------------------------------------
#  Constraint - power corridor
# -----------------------------------------------------------------------
# Ensure peak of all active configurations is within corridor
subject to PowerCorridor {f in FCT_SET}:
  sum {c in CFG_SET} F_C_MAP[f,c] * P_PEAK[c] <= P_MAX;

# -----------------------------------------------------------------------
# Optimization goal: Minimize the total energy consumption
# -----------------------------------------------------------------------
minimize TotalEnergy:
  sum {f in FCT_SET} (E_IDLE_SUM[f] + E_MODE_SUM[f]);
```

C Multi-Core Application Binding MILP

```
# ---------------------------------------------------------------------
#  Set of processing units
# ---------------------------------------------------------------------
set PU_SET;

# ---------------------------------------------------------------------
#  Application profiles and related parameters
# ---------------------------------------------------------------------
set PROFILE_SET;

# code profiles
param PR_CODE {PROFILE_SET} binary default 0;

# data profiles
param PR_DATA {PROFILE_SET} binary default 0;

# Memory footprint
param PR_SIZE {PROFILE_SET} integer >= 0;

# ---------------------------------------------------------------------
#  Available memories - as specified by the allocation
# ---------------------------------------------------------------------
set MEM_SET;

# Memory size parameter
param M_SIZE {MEM_SET} >= 0;

# ---------------------------------------------------------------------
#  Relations between processing units and profiles
# ---------------------------------------------------------------------
# Number of read accesses per profile and pu
param PR_PU_READS {PROFILE_SET, PU_SET} integer >= 0;

# Number of write accesses per profile and pu
param PR_PU_WRITES {PROFILE_SET, PU_SET} integer >= 0;

# ---------------------------------------------------------------------
#  Relations between processing units and memories
# ---------------------------------------------------------------------
# Consumed energy in case of a read
param PU_M_READ {PU_SET, MEM_SET} >= 0;

# Same for write access
param PU_M_WRITE {PU_SET, MEM_SET} >= 0;

# Access time pu -> mem
param PU_M_TIME {PU_SET, MEM_SET} >= 0;

# ---------------------------------------------------------------------
#  User-defined limiting constraints
# ---------------------------------------------------------------------
# Maximum period (of one iteration)
param T_MAX integer >= 0;
```

```
# ----------------------------------------------------------------------
#  Variables
# ----------------------------------------------------------------------
# Binding of profiles to memories
var M_PR_MAP {MEM_SET, PROFILE_SET} binary default 0;

# Energy consumption per pu
var E_SUM {PU_SET} >= 0;

# Time consumed per pu (in one iteration)
var T_SUM {PU_SET} >= 0;

# ----------------------------------------------------------------------
#  Constraints - feasibility
# ----------------------------------------------------------------------
# Ensure memory instance storage capacity is large enough
subject to MemorySizeLimit {m in MEM_SET}:
  sum {pr in PROFILE_SET} M_PR_MAP[m, pr] * PR_SIZE[pr] <= M_SIZE[m];

# All code profiles have to be mapped to >>at least one<< memory
subject to BindingCode {pr in PROFILE_SET : PR_CODE[pr] > 0}:
  sum {m in MEM_SET} M_PR_MAP[m, pr] >= 1;

# Each data profile shall be mapped to >>one and only one<< memory
subject to BindingData {pr in PROFILE_SET : PR_DATA[pr] > 0}:
  sum {m in MEM_SET} M_PR_MAP[m, pr] = 1;

# ----------------------------------------------------------------------
#  Constraints - related to energy and time consumption
# ----------------------------------------------------------------------
# Sum up energy consumption for all profiles...
subject to ActiveEnergy {pu in PU_SET}:
  E_SUM[pu] >= sum {pr in PROFILE_SET} sum {m in MEM_SET}
    M_PR_MAP[m, pr] * ((PR_PU_READS[pr, pu] * PU_M_READ[pu,m]) +
      (PR_PU_WRITES[pr, pu] * PU_M_WRITE[pu,m]));

# Sum up consumed time of all profiles...
subject to TotalTime {pu in PU_SET}:
  T_SUM[pu] >= sum {pr in PROFILE_SET} sum {m in MEM_SET} M_PR_MAP[m, pr] *
    (PU_M_TIME[pu,m] * (PR_PU_READS[pr, pu] + PR_PU_WRITES[pr, pu]));

# ----------------------------------------------------------------------
#  Corridor constraints
# ----------------------------------------------------------------------
# Time corridor - the period of each PU has to be less or equal than T_MAX
# If T_MAX = 0, no time corridor is set
subject to TimeCorridor {pu in PU_SET : T_MAX > 0}:
  T_SUM[pu] <= T_MAX;

# ----------------------------------------------------------------------
#  Optimization goal: Minimize the total energy consumption
# ----------------------------------------------------------------------
minimize AccessEnergy:
  sum {pu in PU_SET} E_SUM[pu];
```

D Combined STT-RAM Allocation and Binding MIQP

```
# ---------------------------------------------------------------------
#  Application profiles and related parameters
# ---------------------------------------------------------------------
set PROFILE_SET;

# Profile size parameter
param P_SIZE {PROFILE_SET} integer >= 0;

# Number of read accesses
param P_NUM_READ {PROFILE_SET} integer >= 0;

# Number of write accesses
param P_NUM_WRITE {PROFILE_SET} integer >= 0;

# ---------------------------------------------------------------------
#  Available memories and their characteristics
# ---------------------------------------------------------------------
set MEM_SET;

# Memory size parameter
param M_SIZE {MEM_SET} integer > 0;

# Energy consumption per read access
param M_E_READ {MEM_SET} > 0;

# Energy consumption per write access
param M_E_WRITE {MEM_SET} > 0;

# Static power consumption
param M_P_STAT {MEM_SET} > 0;

# Maximum supported operation frequency
param M_FREQ_MAX {MEM_SET} > 0;

# ---------------------------------------------------------------------
#  System parameters and user-defined limiting constraints
# ---------------------------------------------------------------------
# Period of the application (in cycles)
param N integer > 0;

# Maximum number of allocated memories
param MEMS_MAX integer > 0;

# Minimum operation frequency of the memory subsystem
param FREQ_MIN >= 0;

# ---------------------------------------------------------------------
#  Variables
# ---------------------------------------------------------------------
# Allocation variable
var M_ALLOC {MEM_SET} binary default 0;

# Binding variable
var P_M_BIND {PROFILE_SET, MEM_SET} binary default 0;
```

```
# Energy consumption per memory
var E_SUM {MEM_SET} >= 0;

# ---------------------------------------------------------------------------
#  Constraints
# ---------------------------------------------------------------------------
# Number of allocated memories shall respect the upper bound
subject to MaxMems:
  sum {m in MEM_SET} M_ALLOC[m] <= MEMS_MAX;

# Ensure the individual frequencies are in the constrained range
subject to MaxFreq {p in PROFILE_SET}:
  sum {m in MEM_SET} M_FREQ_MAX[m] * P_M_BIND[p,m] >= FREQ_MIN;

# Allocated memory space shall be large enough for all profiles
subject to ValidBinding {m in MEM_SET}:
  sum {p in PROFILE_SET} P_M_BIND[p,m] * P_SIZE[p] <= M_ALLOC[m] * M_SIZE[m];

# Map every profile to one and only one memory
subject to BindOnlyOne {p in PROFILE_SET}:
  sum {m in MEM_SET} P_M_BIND[p,m] = 1;

# Compute the total energy consumption per memory
subject to EnergySum {m in MEM_SET}:
  E_SUM[m] >= ( M_ALLOC[m] * ((M_P_STAT[m]/M_FREQ_MAX[m]) * N) ) +
          ( sum {p in PROFILE_SET} P_M_BIND[p,m] *
          ((P_NUM_READ[p] * M_E_READ[m]) + (P_NUM_WRITE[p] * M_E_WRITE[m])) );

# ---------------------------------------------------------------------------
#  Optimization goal: Minimize the total energy consumption
# ---------------------------------------------------------------------------
minimize TotalEnergy:
  sum {m in MEM_SET} E_SUM[m];
```

Abbreviations

IoT	Internet of Things
SoC	System-on-Chip
MPSoC	Multi-Processor System-on-Chip
DSP	Digital Signal Processor
PPAC	Power Performance Area Cost
SRAM	Static Random-Access Memory
DRAM	Dynamic Random-Access Memory
CMOS	Complementary Metal-Oxide-Semiconductor
STT-RAM	Spin-Transfer Torque Random-Access Memory
ILP	Integer Linear Program
MILP	Mixed-Integer Linear Program
MIQP	Mixed-Integer Quadratic Program
ISS	Instruction Set Simulation
UML	Unified Modeling Language
ROM	Read-Only Memory
SPM	Scratchpad Memory
MRAM	Magnetoresistive Random-Access Memory
NVM	Non-Volatile Memory
MTJ	Magnetic Tunnel Junction
WER	Write Error Rate
ISA	Instruction Set Architecture
DFG	Data Flow Graph
DSE	Design Space Exploration
EPROM	Erasable Programmable Read-Only Memory
RTOS	Real-Time Operating System
DMA	Direct Memory Access
PRAM	Phase-Change Memory
EDA	Electronic Design Automation
WCRT	Worst-Case Response Time
MSC	Message Sequence Chart
SDF	Synchronous Data Flow
DAG	Directed Acyclic Graph
MMU	Memory Management Unit
IR	Intermediate Representation
MCR	Memory Configuration Register
GCC	GNU Compiler Collection

References

[1] Absar, M.J., Catthoor, F.: Compiler-based approach for exploiting scratch-pad in presence of irregular array access. In: Design, Automation and Test in Europe, pp. 1162–1167 (2005). DOI 10.1109/DATE.2005.97

[2] Alvarez, L., Moretó, M., Casas, M., Castillo, E., Martorell, X., Labarta, J., Ayguadé, E., Valero, M.: Runtime-guided management of scratchpad memories in multicore architectures. In: 2015 International Conference on Parallel Architecture and Compilation (PACT) (2015). DOI 10.1109/PACT.2015.26

[3] Angiolini, F., Benini, L., Caprara, A.: An efficient profile-based algorithm for scratch-pad memory partitioning. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **24**(11), 1660–1676 (2005). DOI 10.1109/TCAD.2005.852299

[4] Arcaro, S., Di Carlo, S., Indaco, M., Pala, D., Prinetto, P., Vatajelu, E.I.: Integration of stt-mram model into cacti simulator. In: 2014 9th International Design and Test Symposium (IDT), pp. 67–72 (2014). DOI 10.1109/IDT.2014.7038589

[5] Arcaro, S., Pala, D.: Integration of stt-rams into cacti simulator. Tech. rep., Politecnico di Torino (2014)

[6] ARM Ltd.: Armv6-m architecture reference manual (2010). URL `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0419c/index.html`. Accessed 12/2019

[7] Avissar, O., Barua, R., Stewart, D.: An optimal memory allocation scheme for scratch-pad-based embedded systems. ACM Transactions on Embedded Computing Systems **1**(1), 6–26 (2002). DOI 10.1145/581888.581891

[8] Balasa, F., Abuaesh, N., Luican, I.I., David Zhu, H.: Scratch-pad memory banking by dynamic programming for embedded data-intensive applications. In: Sixteenth International Symposium on Quality Electronic Design, pp. 490–494 (2015). DOI 10.1109/ISQED.2015.7085474

[9] Balasa, F., Gingu, C.V., Luican, I.I., Zhu, H.: Design space exploration for low-power memory systems in embedded signal processing applications. In: Proc. of Embedded and Real-Time Computing Systems and Applications (RTCSA). IEEE, Taipei (2013)

[10] Balasa, F., Luican, I.I., Abuaesh, N., Gingu, C.V.: Compiler-directed memory hierarchy design for low-energy embedded systems. In: 2013 Eleventh ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2013), pp. 147–156 (2013)

[11] Banakar, R., Steinke, S., Bo-Sik Lee, Balakrishnan, M., Marwedel, P.: Scratchpad memory: a design alternative for cache on-chip memory in embedded systems. In: Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002 (IEEE Cat. No.02TH8627), pp. 73–78 (2002). DOI 10.1145/774789.774805

[12] Bathen, L.A.D., Dutt, N.D.: Software controlled memories for scalable many-core architectures. In: IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (2012). DOI 10.1109/RTCSA.2012.60

[13] Benini, L., Macii, A., Macii, E., Poncino, M.: Increasing energy efficiency of embedded systems by application-specific memory hierarchy generation. IEEE Design Test of Computers **17**(2), 74–85 (2000). DOI 10.1109/54.844336

[14] Benini, L., Macii, A., Poncino, M.: A recursive algorithm for low-power memory partitioning. In: Proc. of the 2000 International Symposium on Low Power Electronics and Design (ISLPED ’00), pp. 78–83 (2000). DOI 10.1145/344166.344518

[15] Benini, L., Macii, A., Poncino, M.: Energy-aware design of embedded memories: A survey of technologies, architectures, and optimization techniques. ACM Trans. Embed. Comput. Syst. **2**(1), 5–32 (2003). DOI 10.1145/605459.605461

[16] Bishnoi, R., Oboril, F., Ebrahimi, M., Tahoori, M.B.: Avoiding unnecessary write operations in stt-mram for low power implementation. In: Fifteenth International Symposium on Quality Electronic Design, pp. 548–553 (2014). DOI 10.1109/ISQED.2014.6783375

[17] Bosch Sensortec GmbH: BMF055 Application Note. Tech. rep. (2016). URL `https://ae-bst.resource.bosch.com/media/_tech/media/datasheets/BST-BMF055-DS000.pdf`

[18] Bouchhima, A., Gerin, P., Petrot, F.: Automatic instrumentation of embedded software for high level hardware/software co-simulation. In: Proc. of Asia and South Pacific Design Automation Conference (2009). DOI 10.1109/ASPDAC.2009.4796537

[19] Braun, G., Nohl, A., Hoffmann, A., Schliebusch, O., Leupers, R., Meyr, H.: A universal technique for fast and flexible instruction-set architecture simulation. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **23**(12) (2004). DOI 10.1109/TCAD.2004.836734

[20] Calimera, A., Benini, L., Macii, A., Macii, E., Poncino, M.: Design of a flexible reactivation cell for safe power-mode transition in power-gated circuits. IEEE Transactions on Circuits and Systems I: Regular Papers **56**(9), 1979–1993 (2009). DOI 10.1109/TCSI.2008.2010151

[21] Calimera, A., Macii, A., Macii, E., Poncino, M.: Design Techniques and Architectures for Low-Leakage SRAMs. IEEE Trans. Circuits Syst. I, Reg. Papers **59**(9), 1992–2007 (2012)

[22] Casini, D., Biondi, A., Nelissen, G., Buttazzo, G.: Memory feasibility analysis of parallel tasks running on scratchpad-based architectures. In: 2018 IEEE Real-Time Systems Symposium (RTSS) (2018). DOI 10.1109/RTSS.2018.00047

[23] Chang, D., Lin, I., Yong, L.: Rohom: Requirement-aware online hybrid on-chip memory management for multicore systems. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **36**(3) (2017). DOI 10.1109/TCAD.2016.2584048

[24] Che, W., Chatha, K.: Compilation of stream programs onto scratchpad memory based embedded multicore processors through retiming. In: 48th ACM/EDAC/IEEE Design Automation Conference (DAC) (2011)

[25] Chen, Z.H., Su, A.W.Y.: A hardware/software framework for instruction and data scratchpad memory allocation. ACM Trans. Archit. Code Optim. **7**(1), 2:1–2:27 (2010). DOI 10.1145/1736065.1736067

[26] Cheung, E., Hsieh, H., Balarin, F.: Memory subsystem simulation in software tlm/t models. In: Proc. of Asia and South Pacific Design Automation Conference (2009). DOI 10.1109/ASPDAC.2009.4796580

[27] Cmelik, B., Keppel, D.: Shade: A fast instruction-set simulator for execution profiling. Proc. of ACM SIGMETRICS Conference (1994). DOI 10.1145/183018.183032
[28] Cong, J., Jiang, W., Liu, B., Zou, Y.: Automatic memory partitioning and scheduling for throughput and power optimization. In: Proceedings of the 2009 International Conference on Computer-Aided Design, ICCAD '09, pp. 697–704. ACM, New York, NY, USA (2009). DOI 10.1145/1687399.1687528
[29] Coumeri, S.L., Thomas, D.E.: Memory modeling for system synthesis. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **8**(3) (2000)
[30] Dominguez, A., Nguyen, N., Barua, R.K.: Recursive function data allocation to scratch-pad memory. In: Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES '07, pp. 65–74. ACM, New York, NY, USA (2007). DOI 10.1145/1289881.1289897
[31] EEMBC: EEMBC Multibench 1.0 Multicore Benchmark Software. URL http://www.eembc.org/benchmark/multi_sl.php. Accessed 12/2019
[32] Egger, B., Kim, S., Jang, C., Lee, J., Min, S.L., Shin, H.: Scratchpad memory management techniques for code in embedded systems without an mmu. IEEE Transactions on Computers **59**(8), 1047–1062 (2010). DOI 10.1109/TC.2009.188
[33] Emre, Y., Yang, C., Sutaria, K., Cao, Y., Chakrabarti, C.: Enhancing the reliability of stt-ram through circuit and system level techniques. In: 2012 IEEE Workshop on Signal Processing Systems, pp. 125–130 (2012). DOI 10.1109/SiPS.2012.11
[34] edacentrum e.V.: edesign 2018-2022 (2017)
[35] Flake, G.W., Tarjan, R.E., Tsioutsiouliklis, K.: Graph clustering and minimum cut trees. Internet Mathematics **1**(4), 385–408 (2004). DOI 10.1080/15427951.2004.10129093
[36] Fourer, R., Gay, D.M., Kernighan, B.: AMPL - A Modeling Language for Mathematical Programming, 2 edn. Duxbury-Thomson (2003)
[37] Francesco, P., Marchal, P., Atienza, D., Benini, L., Catthoor, F., Mendias, J.M.: An integrated hardware/software approach for run-time scratchpad management. In: Proceedings of the 41st Annual Design Automation Conference, DAC '04, pp. 238–243. ACM, New York, NY, USA (2004). DOI 10.1145/996566.996634
[38] edacentrum GmbH: Confirm - automatisierter firmware-entwurf für anwendungsspezifische elektroniksysteme (2019). URL https://www.edacentrum.de/confirm/. Accessed 12/2019
[39] Görke, R.: An algorithmic walk from static to dynamic graph clustering. Ph.D. thesis, Karlsruhe Institute of Technology (KIT) (2010). URL https://publikationen.bibliothek.kit.edu/1000018288. Accessed 12/2019
[40] Gu, S., Zhuge, Q., Yi, J., Hu, J., Sha, E.H.: Optimizing task and data assignment on multi-core systems with multi-port spms. IEEE Transactions on Parallel and Distributed Systems **26**(9) (2015). DOI 10.1109/TPDS.2014.2356194
[41] Guangyu Chen, Feihui Li, Ozturk, O., Guilin Chen, Kandemir, M., Kolcu, I.: Leakage-aware spm management. In: IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI'06), pp. 6 pp.– (2006). DOI 10.1109/ISVLSI.2006.58
[42] Guo, Y., Zhuge, Q., Hu, J., Sha, E.H..: Optimal data placement for memory architectures with scratch-pad memories. In: 2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications, pp. 1045–1050 (2011). DOI 10.1109/TrustCom.2011.143

[43] Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: Mibench: A free, commercially representative embedded benchmark suite. In: Proc. of the 2001 IEEE International Workshop on Workload Characterization. IEEE (2001). DOI 10.1109/WWC.2001.990739

[44] Haubelt, C., Teich, J.: Digitale Hardware/Software-Systeme - Spezifikation und Verifikation. Springer Berlin Heidelberg (2010). DOI 10.1007/978-3-642-05356-6

[45] Hennessy, J.L., Patterson, D.A.: Computer Architecture - A Quantitative Approach, 5 edn. Elsevier Inc. (2012)

[46] Hiller, J.A.: Entwurf und implementierung eines code-generator frameworks (2018)

[47] Hiser, J.D., Davidson, J.W., Whalley, D.B.: Fast, accurate design space exploration of embedded systems memory configurations. In: Proceedings of the 2007 ACM Symposium on Applied Computing, SAC '07, pp. 699–706. ACM, New York, NY, USA (2007). DOI 10.1145/1244002.1244159

[48] Hu, J., Xue, C.J., Zhuge, Q., Tseng, W., Sha, E.H..: Towards energy efficient hybrid on-chip scratch pad memory with non-volatile memory. In: 2011 Design, Automation Test in Europe, pp. 1–6 (2011). DOI 10.1109/DATE.2011.5763127

[49] Hu, J., Zhuge, Q., Xue, C.J., Tseng, W., Sha, E.H.M.: Management and optimization for nonvolatile memory-based hybrid scratchpad memory on multicore embedded processors. ACM Transactions on Embedded Computing Systems **13**(4) (2014). DOI 10.1145/2560019

[50] Hu, T.C., Kahng, A.B.: Linear and Integer Programming Made Easy. Springer International Publishing Switzerland (2016). DOI 10.1007/978-3-319-24001-5

[51] Hu, W.: Scratchpad memory based power efficient optimization for mpsoc. In: 2011 International Conference on Electronics, Communications and Control (ICECC) (2011). DOI 10.1109/ICECC.2011.6067865

[52] Huangfu, Y., Zhang, W.: Compiler-directed leakage energy reduction for instruction scratch-pad memories. In: Fifteenth International Symposium on Quality Electronic Design, pp. 392–399 (2014). DOI 10.1109/ISQED.2014.6783352

[53] Infineon Technologies AG: 32-bit aurix microcontroller based on tricore (2019). URL `https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller/`. Accessed 12/2019

[54] ITRS: More moore (2015). URL `https://www.semiconductors.org/main/2015_international_technology_roadmap_for_semiconductors_itrs/`. Accessed 12/2019

[55] Jin, Y., Shihab, M., Jung, M.: Area, power and latency considerations of stt-mram to substitute for main memory. In: The Memory Forum co-located with the 41st International Symposium on Computer Architecture (ISCA-41) (2014)

[56] Kandemir, M., Choudhary, A.: Compiler-directed scratch pad memory hierarchy design and management. In: Proceedings 2002 Design Automation Conference (IEEE Cat. No.02CH37324), pp. 628–633 (2002). DOI 10.1109/DAC.2002.1012701

[57] Kandemir, M., Irwin, M.J., Chen, G., Kolcu, I.: Compiler-guided leakage optimization for banked scratch-pad memories. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **13**(10), 1136–1146 (2005). DOI 10.1109/TVLSI.2005.859478

[58] Kandemir, M., Kadayif, I., Sezer, U.: Exploiting scratch-pad memory using presburger formulas. In: Proceedings of the 14th International Symposium on Systems Synthesis, ISSS '01, pp. 7–12. ACM, New York, NY, USA (2001). DOI 10.1145/500001.500004

[59] Kandemir, M., Ramanujam, J., Choudhary, A.: Exploiting shared scratch pad memory space in embedded multiprocessor systems. In: Proceedings 2002 Design Automation Conference (IEEE Cat. No.02CH37324), pp. 219–224 (2002). DOI 10.1109/DAC.2002.1012623

[60] Kandemir, M., Ramanujam, J., Irwin, J., Vijaykrishnan, N., Kadayif, I., Parikh, A.: Dynamic management of scratch-pad memory space. In: Proceedings of the 38th Annual Design Automation Conference, DAC '01, pp. 690–695. ACM, New York, NY, USA (2001). DOI 10.1145/378239.379049

[61] Kang, S., Dean, A.G.: Leveraging both data cache and scratchpad memory through synergetic data allocation. In: 2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium, pp. 119–128 (2012). DOI 10.1109/RTAS.2012.22

[62] Kannan, A., Shrivastava, A., Pabalkar, A., Lee, J.e.: A software solution for dynamic stack management on scratch pad memory. In: Proceedings of the 2009 Asia and South Pacific Design Automation Conference, ASP-DAC '09, pp. 612–617. IEEE Press, Piscataway, NJ, USA (2009)

[63] Karuri, K., Huben, C., Leupers, R., Ascheid, G., Meyr, H.: Memory access micro-profiling for asip design. In: Proc. of IEEE International Workshop on Electronic Design, Test and Applications (2006). DOI 10.1109/DELTA.2006.63

[64] Kaxiras, S., Hu, Z., Martonosi, M.: Cache decay: Exploiting generational behavior to reduce cache leakage power. ACM SIGARCH Computer Architecture News **29** (2001). DOI 10.1145/384285.379268

[65] Keating, M., Flynn, D., Aitken, R., Gibbons, A., Shi, K.: Low Power Methodology Manual for System-on-Chip Design. Springer Science and Business Media (2007)

[66] Khvalkovskiy, A.V., Apalkov, D., Watts, S., Chepulskii, R., Beach, R.S., Ong, A., Tang, X., Driskill-Smith, A., Butler, W.H., Visscher, P.B., Lottis, D., Chen, E., Nikitin, V., Krounbi, M.: Basic principles of stt-mram cell operation in memory arrays. Journal of Physics D: Applied Physics **46**(7) (2013). DOI 10.1088/0022-3727/46/7/074001

[67] Kim, N.S., Austin, T., Baauw, D., Mudge, T., Flautner, K., Hu, J.S., Irwin, M.J., Kandemir, M., Narayanan, V.: Leakage current: Moore's law meets static power. Computer **36**(12), 68–75 (2003). DOI 10.1109/MC.2003.1250885

[68] Kishi, T., Yoda, H., Kai, T., Nagase, T., Kitagawa, E., Yoshikawa, M., Nishiyama, K., Daibou, T., Nagamine, M., Amano, M., Takahashi, S., Nakayama, M., Shimomura, N., Aikawa, H., Ikegawa, S., Yuasa, S., Yakushiji, K., Kubota, H., Fukushima, A., Oogane, M., Miyazaki, T., Ando, K.: Lower-current and fast switching of a perpendicular tmr for high speed and high density spin-transfer-torque mram. In: 2008 IEEE International Electron Devices Meeting, pp. 1–4 (2008). DOI 10.1109/IEDM.2008.4796680

[69] Krishnamoorthy, S., Catalyurek, U., Nieplocha, J., Rountev, A., Sadayappan, P.: Hypergraph Partitioning for Automatic Memory Hierarchy Management. In: Proc. of The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC). IEEE, Tampa, FL, USA (2006)

[70] Kuan, K., Adegbija, T.: Lars: Logically adaptable retention time stt-ram cache for embedded systems. In: 2018 Design, Automation Test in Europe Conference Exhibition (DATE), pp. 461–466 (2018). DOI 10.23919/DATE.2018.8342053

[71] Lafond, S., Lilius, J.: Static energy saving through multi-bank memory architecture. In: Proc. of the 2006 International Conference on Embedded Com-

puter Systems: Architectures, Modeling and Simulation, pp. 43–49 (2006). DOI 10.1109/ICSAMOS.2006.300807

[72] Lattner, C.: LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign (2002). URL `http://llvm.org/pubs/2002-12-LattnerMSThesis.html`. Accessed 12/2019

[73] Lattner, C., Adve, V.: Llvm: a compilation framework for lifelong program analysis transformation. In: Proc. of International Symposium on Code Generation and Optimization (2004). DOI 10.1109/CGO.2004.1281665

[74] Li, L., Kadayif, I., Tsai, Y.F., Vijaykrishnan, N., Kandemir, M.T., Irwin, M.J., Sivasubramaniam, A.: Leakage energy management in cache hierarchies. In: Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques, PACT '02, pp. 131–140. IEEE Computer Society, Washington, DC, USA (2002). URL `http://dl.acm.org/citation.cfm?id=645989.674306`

[75] Lin, W.C., Chen, C.H.: An energy-delay efficient power management scheme for embedded system in multimedia applications. In: Proc. of the 2004 IEEE Asia-Pacific Conference on Circuits and Systems, vol. 2, pp. 869–872 (2004). DOI 10.1109/APCCAS.2004.1413017

[76] Liu, T., Zhao, Y., Xue, C.J., Li, M.: Power-Aware Variable Partitioning for DSPs With Hybrid PRAM and DRAM Main Memory. IEEE Trans. Signal Process. **61**(14), 3509–3520 (2013)

[77] LLVM: Llvm source code documentation (2019). URL `http://llvm.org/doxygen/`. Accessed 12/2019

[78] Loghi, M., Golubeva, O., Macii, E., Poncino, M.: Architectural leakage power minimization of scratchpad memories by application-driven subbanking. IEEE Transactions on Computers **59**(7), 891–904 (2010). DOI 10.1109/TC.2010.43

[79] Lopes, B.C., Auler, R.: Getting Started with LLVM Core Libraries. Packt Publishing (2014)

[80] Lu, K., Müller-Gritschneder, D., Schlichtmann, U.: Memory access reconstruction based on memory allocation mechanism for source-level simulation of embedded software. In: Proc. of Asia and South Pacific Design Automation Conference (2013). DOI 10.1109/ASPDAC.2013.6509687

[81] Luz, V.D.L., Kandemir, M., Kolcu, I.: Automatic data migration for reducing energy consumption in multi-bank memory systems. In: Proc. of the 2002 Design Automation Conference, pp. 213–218 (2002). DOI 10.1109/DAC.2002.1012622

[82] Luz, V.D.L., Kandemir, M., Kolcu, I.: Reducing memory energy consumption of embedded applications that process dynamically allocated data. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **25**(9), 1855–1860 (2006). DOI 10.1109/TCAD.2005.859521

[83] Lyuh, C.G., Kim, T.: Memory access scheduling and binding considering energy minimization in multi-bank memory systems. In: Proceedings of the 41st Annual Design Automation Conference, DAC '04, pp. 81–86. ACM, New York, NY, USA (2004). DOI 10.1145/996566.996596

[84] Mai, S., Zhang, C., Zhao, Y., Chao, J., Wang, Z.: An application-specific memory partitioning method for low power. In: Proc. of International Conference on ASIC (ASICON). IEEE, Guilin, China (2007)

[85] Mathur, A., Minwell, L.: Memory power reduction in soc design using powerpro mg (2009). URL `https://www.design-reuse.com/articles/21806/memory-power-reduction-soc-design.html`. Accessed 12/2019
[86] Meftali, S., Gharsalli, F., Rousseau, F., Jerraya, A.A.: An optimal memory allocation for application-specific multiprocessor system-on-chip. In: Proceedings of the 14th International Symposium on Systems Synthesis, ISSS '01, pp. 19–24. ACM, New York, NY, USA (2001). DOI 10.1145/500001.500006
[87] Meng, Y., Sherwood, T., Kastner, R.: Exploring the limits of leakage power reduction in caches. ACM Trans. Archit. Code Optim. **2**(3), 221–246 (2005). DOI 10.1145/1089008.1089009
[88] Menichelli, F., Olivieri, M.: Static minimization of total energy consumption in memory subsystem for scratchpad-based systems-on-chips. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **17**(2), 161–171 (2009). DOI 10.1109/TVLSI.2008.2001940
[89] Mills, C., Ahalt, S.C., Fowler, J.: Compiled instruction set simulation. Software: Practice and Experience **21**(8) (1991). DOI 10.1002/spe.4380210807
[90] Minwell, L.: Advanced power management in embedded memory subsystems (2011). URL `https://www.design-reuse.com/articles/26402/power-management-in-embedded-memory-subsystems.html`. Accessed 12/2019
[91] Mohammad, B.: Embedded Memory Design for Multi-Core and Systems on Chip. Springer Science and Business Media (2014). DOI 10.1007/978-1-4614-8881-1
[92] Mohanty, S.P., Ranganathan, N., Chappidi, S.K.: Ilp models for simultaneous energy and transient power minimization during behavioral synthesis. ACM Transactions on Design Automation of Electronic Systems **11**(1), 186–212 (2006). DOI 10.1145/1124713.1124725
[93] Mück, T.R., Fröhlich, A.A.: A run-time memory management approach for scratch-pad-based embedded systems. In: 2010 IEEE 15th Conference on Emerging Technologies Factory Automation (ETFA 2010), pp. 1–4 (2010). DOI 10.1109/ETFA.2010.5641082
[94] Munira, K., Butler, W.H., Ghosh, A.W.: A quasi-analytical model for energy-delay-reliability tradeoff studies during write operations in a perpendicular stt-ram cell. IEEE Transactions on Electron Devices **59**(8), 2221–2226 (2012)
[95] Muralimanohar, N., Balasubramonian, R., Jouppi, N.P.: Cacti 6.0: A tool to model large caches. Tech. Rep. HPL-2009-85, HP Laboratories (2009). URL `http://www.hpl.hp.com/techreports/2009/HPL-2009-85.pdf`. Accessed 12/2019
[96] Nakada, T., Nakamura, H. (eds.): Normally-Off Computing. Springer Japan (2017). DOI 10.1007/978-4-431-56505-5
[97] Nebel, W., Helms, D.: On leakage currents: sources and reduction for transistors, gates, memories and digital systems. In: Proceeding of the 13th international symposium on Low power electronics and design (ISLPED '08), pp. 349–350 (2008). DOI 10.1145/1393921.1394014
[98] Newman, M.E.J., Girvan, M.: Finding and evaluating community structure in networks. Phys. Rev. E **69**, 026113 (2004). DOI 10.1103/PhysRevE.69.026113
[99] Nguyen, N., Dominguez, A., Barua, R.: Memory allocation for embedded systems with a compile-time-unknown scratch-pad size. In: Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded

Systems, CASES ’05, pp. 115–125. ACM, New York, NY, USA (2005). DOI 10.1145/1086297.1086313

[100] Ozturk, O., Chen, G., Kandemir, M., Karakoy, M.: An integer linear programming based approach to simultaneous memory space partitioning and data allocation for chip multiprocessors. In: IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI’06), pp. 6 pp.– (2006). DOI 10.1109/ISVLSI.2006.22

[101] Panda, P.R., Dutt, N.D., Nicolau, A.: Efficient utilization of scratch-pad memory in embedded processor applications. In: Proc. of European Design and Test Conference. ED TC 97 (1997). DOI 10.1109/EDTC.1997.582323

[102] Panda, P.R., Dutt, N.D., Nicolau, A.: Local memory exploration and optimization in embedded systems. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **18**(1), 3–13 (1999). DOI 10.1109/43.739054

[103] Park, S., Park, H.w., Ha, S.: A novel technique to use scratch-pad memory for stack management. In: Proceedings of the Conference on Design, Automation and Test in Europe, DATE ’07, pp. 1478–1483. EDA Consortium, San Jose, CA, USA (2007)

[104] Pasricha, S., Dutt, N.D.: A Framework for Cosynthesis of Memory and Communication Architectures for MPSoC. IEEE Trans. Comput.-Aided Design Integr. Circuits Syst. (TCAD) **26**(3), 408–420 (2007)

[105] Paul, J., Stechele, W., Kroehnert, M., Asfour, T.: Improving efficiency of embedded multi-core platforms with scratchpad memories. In: ARCS 2014; Workshop Proc. on Architecture of Computing Systems (2014)

[106] Pouillon, N., Becoulet, A., d. Mello, A.V., Pecheux, F., Greiner, A.: A generic instruction set simulator api for timed and untimed simulation and debug of mp2-socs. In: Proc. of IEEE/IFIP International Symposium on Rapid System Prototyping (2009). DOI 10.1109/RSP.2009.11

[107] Press, O.U.: Oxford dictionary (2019). URL `https://www.lexico.com`. Accessed 12/2019

[108] Qin, W., Hu, B.: A technique to exploit memory locality for fast instruction set simulation. In: Proc. of the 6th International Conference on ASIC, vol. 2 (2005). DOI 10.1109/ICASIC.2005.1611459

[109] Qiu, M., Zhang, L., Sha, E.H.M.: Ilp optimal scheduling for multi-module memory. In: Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS ’09, pp. 277–286. ACM, New York, NY, USA (2009). DOI 10.1145/1629435.1629473

[110] Rabaey, J.: Low Power Design Essentials. Springer Science and Business Media (2009). DOI 10.1007/978-0-387-71713-5

[111] Reshadi, M., Mishra, P., Dutt, N.: Instruction set compiled simulation: a technique for fast and flexible instruction set simulation. In: Proc. of Design Automation Conference (2003). DOI 10.1145/775832.776026

[112] Rodríguez, G., Touriño, J., Kandemir, M.T.: Volatile stt-ram scratchpad design and data allocation for low energy. ACM Trans. Archit. Code Optim. **11**(4), 38:1–38:26 (2014). DOI 10.1145/2669556

[113] Rudolf, J., Strobel, M., Benz, J., Haubelt, C., Radetzki, M., Bringmann, O.: Automated sensor firmware development - generation, optimization, and analysis. In: MBMV 2019; 22nd Workshop - Methods and Description Languages for Modelling and Verification of Circuits and Systems, pp. 1–12 (2019)

[114] Sampaio, F., Shafique, M., Zatt, B., Bampi, S., Henkel, J.: dSVM: Energy-efficient distributed Scratchpad Video Memory Architecture for the next-generation High Efficiency Video Coding. In: Proc. of Design, Automation and Test in Europe Conference and Exhibition (DATE). IEEE, Dresden, Germany (2014)

[115] Sayed, N., Bishnoi, R., Oboril, F., Tahoori, M.B.: A cross-layer adaptive approach for performance and power optimization in stt-mram. In: 2018 Design, Automation Test in Europe Conference Exhibition (DATE), pp. 791–796 (2018). DOI 10.23919/DATE.2018.8342114

[116] Schmidt, E., von Cölln, G., Kruse, L., Theeuwen, F., Nebel, W.: Memory power models for multilevel power estimation and optimization. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **10**(2), 106–109 (2002)

[117] Seung, C.J., Shrivastava, A., Bai, K.: Dynamic code mapping for limited local memory systems. Proc. of IEEE Int'l Conference on Application-specific Systems Architectures and Processors (ASAP) (2010)

[118] Shrivastava, A., Kannan, A., Lee, J.: A Software-Only Solution to Use Scratch Pads for Stack Data. IEEE Trans. Comput.-Aided Design Integr. Circuits Syst. (TCAD) **28**, 1719–1727 (2009)

[119] Silvaco Inc: PDK 45nm Open Cell Library (2011). URL `https://www.silvaco.com/products/nangate/FreePDK45_Open_Cell_Library/index.html`. Accessed 12/2019

[120] Smullen, C.W., Mohan, V., Nigam, A., Gurumurthi, S., Stan, M.R.: Relaxing non-volatility for fast and energy-efficient stt-ram caches. In: 2011 IEEE 17th International Symposium on High Performance Computer Architecture, pp. 50–61 (2011). DOI 10.1109/HPCA.2011.5749716

[121] SoCLib Consortium: The soclib project: An integrated system-on-chip modelling and simulation platform. Tech. rep., CNRS (2011). URL `http://www.soclib.fr`. Accessed 12/2019

[122] Srinivasan, S., Angiolini, F., Ruggiero, M., Benini, L., Vijaykrishnan, N.: Simultaneous memory and bus partitioning for soc architectures. In: Proc. of the 2005 IEEE International SOC Conference, pp. 125–128. IEEE (2005). DOI 10.1109/SOCC.2005.1554478

[123] Stattelmann, S., Gebhard, G., Cullmann, C., Bringmann, O., Rosenstiel, W.: Hybrid source-level simulation of data caches using abstract cache models. In: Proc. of Design, Automation & Test in Europe Conference (2012). DOI 10.1109/DATE.2012.6176500

[124] Steinfeld, L., Ritt, M., Silveira, F., Carro, L.: Low-Power Processors Require Effective Memory Partitioning, pp. 73–81. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). DOI 10.1007/978-3-642-38853-8_7

[125] Steinke, S., Wehmeyer, L., Lee, B.S., Marwedel, P.: Assigning program and data objects to scratchpad for energy reduction. In: Proc. of the 2002 Design, Automation and Test in Europe Conference and Exhibition, pp. 409–415 (2002). DOI 10.1109/DATE.2002.998306

[126] Strobel, M., Eggenberger, M., Radetzki, M.: Low power memory allocation and mapping for area-constrained systems-on-chips. EURASIP Journal on Embedded Systems **2017**(1) (2016). DOI 10.1186/s13639-016-0039-5

[127] Strobel, M., Führ, G., Radetzki, M., Leupers, R.: Combined mpsoc task mapping and memory optimization for low-power. In: Proc. of the 2019 IEEE 15th Asia Pacific Conference on Circuits and Systems (APCCAS) (2019)

[128] Strobel, M., Radetzki, M.: Hybrid instruction set simulation for fast and accurate memory access profiling. In: Proc. of the 13th Workshop on Intelligent Solutions in Embedded Systems (WISES), pp. 23–28 (2017). DOI 10.1109/WISES.2017.7986927

[129] Strobel, M., Radetzki, M.: Design-time optimization techniques for low-power embedded memory subsystems. In: Proc. of the 1st International Workshop on Embedded Software for Industrial IOT (ESIIT) (2018)

[130] Strobel, M., Radetzki, M.: A backend tool for the integration of memory optimizations into embedded software. In: 2019 Forum for Specification and Design Languages (FDL), pp. 1–7 (2019). DOI 10.1109/FDL.2019.8876895

[131] Strobel, M., Radetzki, M.: Design-time memory subsystem optimization for low-power multi-core embedded systems. In: 2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC), pp. 347–353 (2019). DOI 10.1109/MCSoC.2019.00056

[132] Strobel, M., Radetzki, M.: Power-mode-aware memory subsystem optimization for low-power system-on-chip design. ACM Trans. Embed. Comput. Syst. **18**(5), 43:1–43:25 (2019). DOI 10.1145/3356583

[133] Suhendra, V., Roychoudhury, A., Mitra, T.: Scratchpad allocation for concurrent embedded software. In: Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS ’08, pp. 37–42. ACM, New York, NY, USA (2008). DOI 10.1145/1450135.1450145

[134] Swaminathan, K., Pisolkar, R., Cong Xu, Narayanan, V.: When to forget: A system-level perspective on stt-rams. In: 17th Asia and South Pacific Design Automation Conference, pp. 311–316 (2012). DOI 10.1109/ASPDAC.2012.6164965

[135] Tajik, H., Donyanavard, B., Dutt, N., Jahn, J., Henkel, J.: Spmpool: Runtime spm management for memory-intensive applications in embedded many-cores. ACM Trans. Embed. Comput. Syst. **16**(1), 25:1–25:27 (2016). DOI 10.1145/2968447

[136] Takase, H., Tomiyama, H., Takada, H.: Partitioning and allocation of scratch-pad memory for priority-based preemptive multi-task systems. In: 2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010), pp. 1124–1129 (2010). DOI 10.1109/DATE.2010.5456977

[137] Takase, H., Tomiyama, H., Zeng, G., Takada, H.: Energy Efficiency of Scratch-Pad Memory at 65 nm and Below: An Empirical Study. In: Proc. of Embedded Software and Systems (ICESS). IEEE, Sichuan (2008)

[138] Teich, J., Haubelt, C.: Digitale Hardware/Software-Systeme - Synthese und Optimierung. Springer Berlin Heidelberg (2007)

[139] Udayakumaran, S., Barua, R.: Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In: Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES ’03, pp. 276–286. ACM, New York, NY, USA (2003). DOI 10.1145/951710.951747

[140] Verma, M., Steinke, S., Marwedel, P.: Data partitioning for maximal scratchpad usage. In: Proceedings of the 2003 Asia and South Pacific Design Automation Conference, ASP-DAC ’03, pp. 77–83. ACM, New York, NY, USA (2003). DOI 10.1145/1119772.1119788

[141] Verma, N.: Analysis towards minimization of total sram energy over active and idle operating modes. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **19**(9), 1695–1703 (2011). DOI 10.1109/TVLSI.2010.2055906

[142] Volpato, D.P., Mendonca, A.K.I., Güntzel, J.L.A., Santos, L.C.V.d.: Cache-tuning-aware scratchpad allocation from binaries. In: Proceedings of the 24th Symposium on Integrated Circuits and Systems Design, SBCCI '11, pp. 221–226. ACM, New York, NY, USA (2011). DOI 10.1145/2020876.2020926

[143] Wang, Y., Li, K., Zhang, J., Li, K.: Energy optimization for data allocation with hybrid sram+nvm spm. IEEE Transactions on Circuits and Systems I: Regular Papers **65**(1), 307–318 (2018). DOI 10.1109/TCSI.2017.2720678

[144] Wang, Z., Hu, X.S.: Energy-aware variable partitioning and instruction scheduling for multibank memory architectures. ACM Trans. Des. Autom. Electron. Syst. **10**(2), 369–388 (2005). DOI 10.1145/1059876.1059885

[145] Wen, H., Zhang, W.: Reducing cache leakage energy for hybrid spm-cache architectures. In: Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES '14, pp. 21:1–21:9. ACM, New York, NY, USA (2014). DOI 10.1145/2656106.2656124

[146] Wolf, S.A., Lu, J., Stan, M.R., Chen, E., Treger, D.M.: The promise of nanomagnetics and spintronics for future logic and universal memory. Proceedings of the IEEE **98**(12), 2155–2168 (2010). DOI 10.1109/JPROC.2010.2064150

[147] Zatt, B., Shafique, M., Bampi, S., Henkel, J.: A low-power memory architecture with application-aware power management for motion & disparity estimation in Multiview Video Coding. Proc. of IEEE/ACM Int'l Conference on Computer-Aided Design (ICCAD) (2011)

[148] Zhang, L., Wu, C., Mao, L.F., Zheng, J.: Integrated sram compiler with clamping diode to reduce leakage and dynamic power in nano-cmos process. IET Micro Nano Letters **7**(2), 171–173 (2012). DOI 10.1049/mnl.2011.0680

[149] Zhou, Z., Ju, L., Jia, Z., Li, X.: Managing hybrid on-chip scratchpad and cache memories for multi-tasking embedded systems. In: The 20th Asia and South Pacific Design Automation Conference (2015). DOI 10.1109/ASPDAC.2015.7059043

[150] Zhu, J., Gajski, D.D.: A retargetable, ultra-fast instruction set simulator. In: Proc. of Design, Automation & Test in Europe Conference (1999). DOI 10.1109/DATE.1999.761137

[151] Zhuge, Q., Sha, E.H.M., Xiao, B., Chantrapornchai, C.: Efficient variable partitioning and scheduling for DSP processors with multiple memory modules. IEEE Trans. Signal Process. (SP) **52**(4), 1090–1099 (2004)