Manish Verma
Peter Marwedel

Advanced Memory Optimization Techniques for Low-Power Embedded Processors

Springer

Advanced Memory Optimization Techniques for
Low-Power Embedded Processors

Advanced Memory Optimization Techniques for Low-Power Embedded Processors

By

Manish Verma
Altera European Technology Center, High Wycombe, UK

and

Peter Marwedel
University of Dortmund, Germany

Springer

A C.I.P. Catalogue record for this book is available from the Library of Congress.

ISBN-13 978-1-4020-5896-7 (HB)
ISBN-13 978-1-4020-5897-4 (e-book)

Published by Springer,
P.O. Box 17, 3300 AA Dordrecht, The Netherlands.

www.springer.com

Printed on acid-free paper

Dedicated to my father

Manish Verma

Acknowledgments

This work is the accomplishment of the efforts of several people without whom this work would not have been possible. Numerous technical discussions with our colleagues, *viz.* Heiko Falk, Robert Pyka, Jens Wagner and Lars Wehmeyer, at Department of Computer Science XII, University of Dortmund have been a greatly helpfull in bringing the book in its current shape. Special thanks goes to Mrs. Bauer for so effortlessly managing our administrative requests.

Finally, we are deeply indebted to our families for their unflagging support, unconditional love and countless sacrifices.

Dortmund, November 2006

Manish Verma
Peter Marwedel

Contents

1 Introduction

In a relatively short span of time, computers have evolved from huge mainframes to small and elegant desktop computers, and now to low-power, ultra-portable handheld devices. With each passing generation, computers consisting of processors, memories and peripherals became smaller and faster. For example, the first commercial computer UNIVAC I costed $1 million dollars, occupied 943 cubic feet space and could perform 1,905 operations per second [94]. Now, a processor present in an electric shaver easily outperforms the early mainframe computers.

The miniaturization is largely due to the efforts of engineers and scientists that made the expeditious progress in the microelectronic technologies possible. According to Moore's Law [90], the advances in technology allow us to double the number of transistors on a single silicon chip every 18 months. This has lead to an exponential increase in the number of transistors on a chip, from 2,300 in an Intel 4004 to 42 millions in Intel Itanium processor [55]. Moore's Law has withstood for 40 years and is predicted to remain valid for at least another decade [91].

Not only the miniaturization and dramatic performance improvement but also the significant drop in the price of processors, has lead to situation where they are being integrated into products, such as cars, televisions and phones which are not usually associated with computers. This new trend has also been called the *disappearing computer*, where the computer does not actually disappear but it is everywhere [85].

Digital devices containing processors now constitute a major part of our daily lives. A small list of such devices includes microwave ovens, television sets, mobile phones, digital cameras, MP3 players and cars. Whenever a system comprises of information processing digital devices to control or to augment its functionality, such a system is termed *an embedded system*. Therefore, all the above listed devices can be also classified as *embedded systems*. In fact, it should be no surprise to us that the number of operational embedded systems has already surpassed the human population on this planet [1].

Although the number and the diversity of embedded systems is huge, they share a set of common and important characteristics which are enumerated below:

(a) Most of the embedded systems perform a fixed and *dedicated* set of functions. For example, the microprocessor which controls the fuel injection system in a car will perform the same functions for its entire life-time.

(b) Often, embedded systems work as *reactive systems* which are connected to the physical world through sensors and react to the external stimuli.
(c) Embedded systems have to be *dependable*. For example, a car should have high reliability and maintainability features while ensuring that fail-safe measures are present for the safety of the passengers in the case of an emergency.
(d) Embedded systems have to satisfy varied, tight and at times conflicting constraints. For example, a mobile phone, apart from acting as a phone, has to act as a digital camera, a PDA, an MP3 player and also as a game console. In addition, it has to satisfy QoS constraints, has to be light-weight, cost-effective and, most importantly, has to have a long battery life time.

In the following, we describe issues concerning embedded devices belonging to consumer electronics domain, as the techniques proposed in this work are devised primarily for these devices.

1.1 Design of Consumer Oriented Embedded Devices

A significant portion of embedded systems is made up of devices which also belong the domain of consumer electronics. The characteristic feature of these devices is that they come in direct contact with users and therefore, demand a high degree of user satisfaction. Typical examples include mobile phones, DVD players, game consoles, etc. In the past decade, an explosive growth has been observed in the consumer electronics domain and it is predicted to be the major force driving both the technological innovation and the economy [117].

However, the consumer electronic devices exist in a market with cut-throat competition, low profit per piece values and low shelf life. Therefore, they have to satisfy stringent design constraints such as performance, power/energy consumption, predictability, development cost, unit cost, time-to-prototype and time-to-market [121]. The following are considered to be the three most important objectives for consumer oriented devices as they have a direct impact on the experience of the consumer.

(a) performance
(b) power (energy) efficiency
(c) predictability (real time responsiveness)

System designers optimize hardware components including the software running on the devices in order to not only meet but to better the above objectives. The memory subsystem has been identified to be the bottleneck of the system and therefore, it offers the maximum potential for optimization.

1.1.1 Memory Wall Problem

Over the past 30 years, microprocessor speeds grew at a phenomenal rate of 50-100% per year, whereas during the same period, the speed of typical DRAM memories grew at a modest rate of about 7% per year [81]. Nowadays, the extremely fast microprocessors spend a large number of cycles idle waiting for the requested data to arrive from the slow memory. This has lead to the problem, also known as the *memory wall problem*, that the

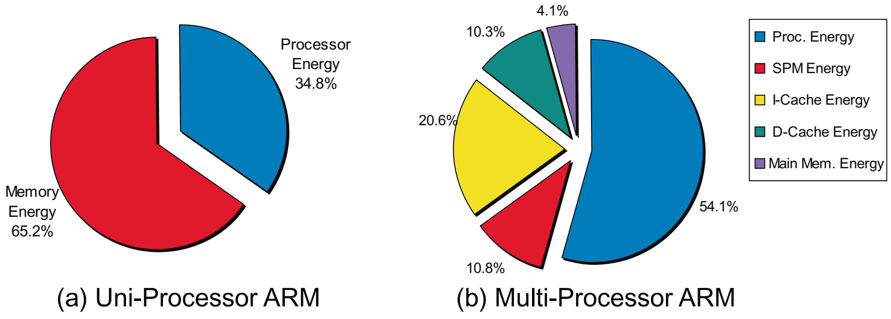


Fig. 1.1. Energy Distribution for (a) Uni-Processor ARM (b) Multi-Processor ARM Based Setups

performance of the entire system is not governed by the speed of the processor but by the speed of the memory [139].

In addition to being the performance bottleneck, the memory subsystem has been demonstrated to be the energy bottleneck: several researchers [64, 140] have demonstrated that the memory subsystem now accounts for 50-70% to the total power budget of the system. We did extensive experiments to validate the above observation for our systems. Figure 1.1 summarizes the results of our experiments for uni-processor ARM [11] and multi-processor ARM [18] based setups.

The values for uni-processor ARM based systems are computed by varying the parameters such as size and latency of the main memory and onchip memories *i.e.* instruction and data caches and scratchpad memories, for all benchmarks presented in this book. For multiprocessor ARM based systems, the number of processors was also varied. In total, more than 150 experiments were conducted to compute the average processor and memory energy consumption values for each of the two systems. Highly accurate energy models, presented in Chapter 3 for both systems, were used to compute the energy consumption values. From the figure, we observe that the memory subsystem consumes 65.2% and 45.9% of the total energy budget for uni-processor ARM and multi-processor ARM systems, respectively. The main memory for the multi-processor ARM based system is an onchip SRAM memory as opposed to offchip SRAM memory for the uni-processor system. Therefore, the memory subsystem accounts for a smaller portion of the total energy budget for the multi-processor system than for the uni-processor system.

It is well understood that there does not exists a silver bullet to solve the memory wall problem. Therefore, in order to diminish the impact of the problem, it has been proposed to create memory hierarchies by placing small and efficient memories close to the processor and to optimize the application code such that the working context of the application is always contained in the memories closest to the processor. In addition, if the silicon estate is not a limiting factor, it has been proposed to replace the high speed processor in the system by a number of simple and relatively lower speed processors.

1.1.2 Memory Hierarchies

Up till very recently, caches have been considered as a synonym for memory hierarchies and in fact, they are still the standard memory to be used in general purpose processors. Their

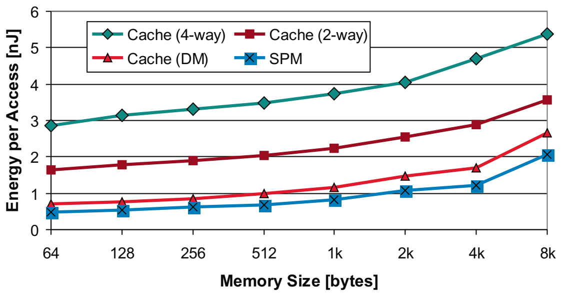


Fig. 1.2. Energy per Access Values for Caches and Scratchpad Memories

main advantage is that they work autonomously and are highly efficient in managing their contents to store the current working context of the application. However, in the embedded systems domain where the applications that can execute on the processor are restricted, the main advantage of the caches turns into a liability. They are known to have high energy consumption [63], low performance and exaggerated worst case execution time (WCET) bounds [86, 135].

On the other end of the spectrum are the recently proposed *scratchpad memories* or *tightly coupled memories*. Unlike a cache, a scratchpad memory consists of just a data memory array and an address decoding logic. The absence of the tag memory and the address comparison logic from the scratchpad memory makes it both area and power efficient [16]. Figure 1.2 presents the energy per access values for scratchpads of varying size and for caches of varying size and associativity. From the figure, it can be observed that the energy per access value for a scratchpad memory is always less than those for caches of the same size. In particular, the energy consumed by a 2k byte scratchpad memory is a mere quarter of that consumed by a 2k byte 4-way set associative cache memory.

However, the scratchpad memories, unlike caches, require explicit support from the software for their utilization. A careful assignment of instructions and data is a prerequisite for an efficient utilization of the scratchpad memory. The good news is that the assignment of instructions and data enables tighter WCET bounds on the system as the contents of the scratchpad memory at runtime are already fixed at compile time. Despite the advantages of scratchpad memories, a consistent compiler toolchain for their exploitation is missing. Therefore, in this work, we present a coherent compilation and simulation framework along with a set of optimizations for the exploitation of scratchpad based memory hierarchies.

1.1.3 Software Optimization

All the embedded devices execute some kind of firmware or software for information processing. The three objectives of performance, power and predictability are directly dependent on the software that is executing on that system. According to Information Technology Roadmap for Semiconductors (ITRS) 2001, embedded software now accounts for 80% of the total development cost of the system [60]. Traditionally, the software for embedded systems was programmed using the assembly language. However, with the software becoming increasingly complex and with tighter time-to-market constraints, the software development is currently done using high-level languages.

Another important trend that has emerged over the last few years, both in the general computing and the embedded systems domain, is that processors are being made increasingly regular. The processors are being stripped of complex hardware components which tried to improve the average case performance by predicting the runtime behavior of applications. Instead, the job of improving the performance of the application is now entrusted to the optimizing compiler. The best known example of the current trend is the CELL processor [53].

The paradigm shift to give an increasing control of hardware to software, has twofold implications: Firstly, a simpler and a regular processor design implies that there is less hardware in its critical path and therefore, higher processor speeds could be achieved at lower power dissipation values. Secondly, the performance enhancing hardware components always have a local view of the application. In contrast, optimizing compilers have a global view of the application and therefore, they can perform global optimizations such that the application executes more efficiently on the regular processor.

From the above discussion, it is clear that the onus lies on optimizing compilers to provide consumers with high performance and energy efficient devices. It has been realized that a regular processor running an optimized application will be far more efficient in all parameters than an irregular processor running an unoptimized application. The following section provides an overview of the contribution of the book towards the improvement of consumer oriented embedded systems.

1.2 Contributions

In this work, we propose approaches to ease the challenges of performance, energy (power) and predictability faced during the design of consumer oriented embedded devices. In addition, the proposed approaches attenuate the effect of the memory wall problem observed on the memory hierarchies of the following three orthogonal processor and system architectures:

(a) Uni-Processor ARM [11]
(b) Multi-Processor ARM System-on-a-Chip [18]
(c) M5 DSP [28]

Two of the three considered architectures, *viz.* Uni-Processor ARM and M5 DSP [33], are already present in numerous consumer electronic devices.

A wide range of memory optimizations, progressively increasing in complexity of analysis and the architecture, are proposed, implemented and evaluated. The proposed optimizations transform the input application such that it efficiently utilizes the memory hierarchy of the system. The goal of the memory optimizations is to minimize the total energy consumption while ensuring a high predictability of the system. All the proposed approaches determine the contents of the scratchpad memory at compile time and therefore, a worst case execution time (WCET) analysis tool [2] can be used to obtain tight WCET bounds for the scratchpad based system. However, we do not explicitly report WCET values in this work. The author of [133] has demonstrated that one of our approaches for a scratchpad based memory hierarchy improved the WCET bounds by a factor of 8 when compared to a cache based memory hierarchy.

An important feature of the presented optimizations which makes them unique from the contemporaries is that they consider both the instruction segments and data variables together for optimization. Therefore, they are able to optimize the total energy consumption of the system. The known approaches to optimize the data do not thoroughly consider the impact of the optimization on the instruction memory hierarchy or on the control flow of the application. In [124], we demonstrated that one such optimization [23] results in worse total energy consumption values compared to the scratchpad overlay based optimization (*cf.* Chapter 6) for the uni-processor ARM based system.

In this work, we briefly demonstrate that the memory optimizations are NP-hard problems and therefore, we propose both optimal and near-optimal approaches. The proposed optimizations are implemented within two compiler backends as well as source level transformations. The benefit of the first approach is that they can use precise information about the application available in the compiler backend to perform accurate optimizations. During the course of research, we realized that access to optimizing compilers for each different processor is becoming a limiting factor. Therefore, we developed memory optimizations as "compiler-in-loop" source level transformations which enabled us to achieve the retargetability of the optimizations at the expense of a small loss of accuracy.

An important contribution of this book is the presentation of a coherent memory hierarchy aware compilation and simulation framework. This is in contrast to some ad-hoc frameworks used otherwise by the research community. Both the simulation and compilation frameworks are configured from a single description of the memory hierarchy and access the same set of accurate energy models for each architecture. Therefore, we are able to efficiently explore the memory hierarchy design space and evaluate the proposed memory optimizations using the framework.

1.3 Outline

The remainder of this book is organized as follows:

- **Chapter 2** presents the background information on power and performance optimizations and gives a general overview of the related work in the domain covered by this dissertation.
- **Chapter 3** describes the memory aware compilation and simulation framework used to evaluate the proposed memory optimizations.
- **Chapter 4** presents a simple non-overlayed scratchpad allocation based memory optimization for a memory hierarchy composed of an L1 scratchpad memory and a background main memory.
- **Chapter 5** presents a complex non-overlayed scratchpad allocation based memory optimization for a memory hierarchy consisting of an L1 scratchpad and cache memories and a background main memory.
- **Chapter 6** presents scratchpad overlay based memory optimization which allows the contents of the scratchpad memory to be updated at runtime with the execution context of the application. The optimization focuses on a memory hierarchy consisting of an L1 scratchpad memory and a background main memory.

- **Chapter 7** presents a combined data partitioning and a loop nest splitting based memory optimization which divides application arrays into smaller partitions to enable an improved scratchpad allocation. In addition, it uses the loop nest splitting approach to optimize the control flow degraded by the data partitioning approach.
- **Chapter 8** presents a set of three memory optimizations to share the scratchpad memory among the processes of a multiprocess application.
- **Chapter 9** concludes the dissertation and presents an outlook on the important future directions.

2

Related Work

Due to the emergence of the handheld devices, power and energy consumption parameters have become one of the most important design constraints. A large body of the research is devoted for reducing the energy consumption of the system by optimizing each of its energy consuming components. In this chapter, we will an introduction to the research on power and energy optimizing techniques. The goal of this chapter is provide a brief overview, rather than an in-depth tutorial. However, many references to the important works are provided for the reader.

The rest of this chapter is organized as follows: In the following section, we describe the relationship between power dissipation and energy consumption. A survey of the approaches used to reduce the energy consumed by the processor and the memory hierarchy is presented in Section 2.2.

2.1 Power and Energy Relationship

In order to design low power and energy-efficient systems, one has to understand the physical phenomenon that lead to power dissipation or energy consumption. In the literature, they are often used as synonyms, though there are underlying distinctions between them which we would like to elucidate in the remainder of this section. Since most digital circuits are currently implemented using CMOS technology, it is reasonable to describe the essential equations governing power and energy consumption for this technology.

2.1.1 Power Dissipation

Electrical power can be defined as the product of the electrical current through times the voltage at the terminals of a power consumer. It is measured in the unit Watt. In the following, we analyze the electric power dissipated by a CMOS inverter (*cf.* Figure 2.1), though the issues discussed are valid for any CMOS circuit. A typical CMOS circuit consists of a pMOS and an nMOS transistor and a small capacitance. The power dissipated by any CMOS circuit can be decomposed into its static and dynamic power components.

$$P_{CMOS} = P_{static} + P_{dynamic} \tag{2.1}$$

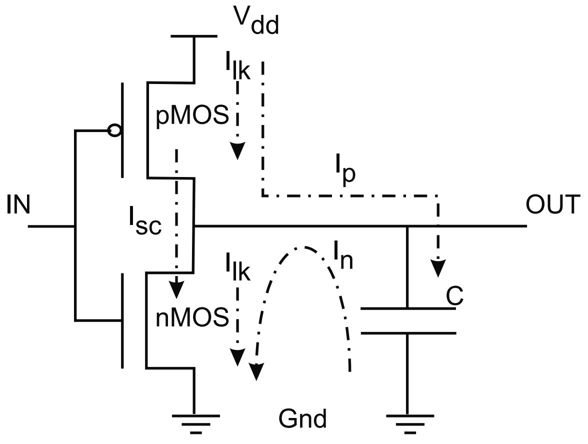


Fig. 2.1. CMOS Inverter

In an ideal CMOS circuit, no static power is dissipated when the circuit is in a steady state, as there is no open path from source (V_{dd}) to ground (Gnd). Since MOS (*i.e.* pMOS and nMOS) transistors are never perfect insulators, there is always a small leakage current I_{lk} (*cf.* Figure 2.1) that flows from V_{dd} to Gnd. The leakage current is inversely related to the feature size and exponentially related to the threshold voltage V_t. For example, the leakage current is approximately 10-20 pA per transistor for 130 nm process with 0.7 V threshold voltage, whereas it exponentially increases to 10-20 nA per transistor when the threshold voltage is reduced to 0.3 V [3].

Overall, the static power P_{static} dissipated due to leakage currents amounts to less than 5% of the total power dissipated at 0.25 µm. It has been observed that the leakage power increases by about a factor of 7.5 for each technological generation and is expected to account for a significant portion of the total power in deep sub-micron technologies [21]. Therefore, the leakage power component grows to 20-25% at 130 nm [3].

The dynamic component $P_{dynamic}$ of the total power is dissipated during the switching between logic levels and is due to charging and discharging of the capacitance and due to a small short circuit current. For example, when the input signal for the CMOS inverter (*cf.* Figure 2.1) switches from one level logic level to the opposite, then there will be a short instance when both the pMOS and nMOS transistors are open. During that time instant a small short circuit current I_{sc} flows from V_{dd} to Gnd. Short circuit power can consume up to 30% of the total power budget if the circuit is active and the transition times of the transistors are substantially long. However, through a careful design to transition edges, the short circuit power component can be kept below 10-15% [102].

The other component of the dynamic power is due to the charge and discharge cycle of the output capacitance C. During a high-to-low transition, energy equal to CV_{dd}^2 is drained from V_{dd} through I_p, a part of which is stored in the capacitance C. During the reverse low-to-high transition, the output capacitance is discharged through I_n. In CMOS circuits, this component accounts for 70-90% of the total power dissipation [102].

From the above discussion, the power dissipated by a CMOS circuit can approximated to be its dynamic power component and is represented as follows:

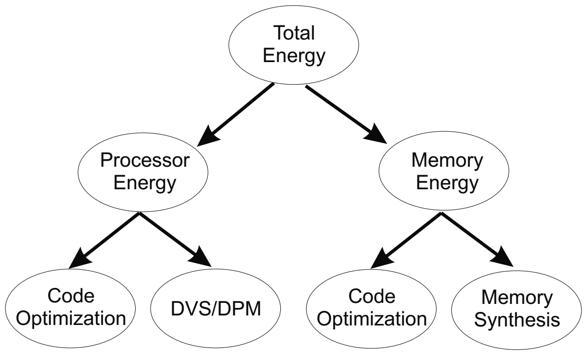


Fig. 2.2. Classification of Energy Optimization Techniques (Excluding Approaches at the Process, Device and Circuit Levels)

$$P_{CMOS} \approx P_{dynamic} \sim \alpha f C V_{dd}^2 \tag{2.2}$$

where, α is the switching activity and f is the clock frequency supplied to the CMOS circuit. Therefore, the power dissipation in a CMOS circuit is proportional to the switching activity α, clock frequency f, capacitive load C and the square of the supply voltage V_{dd}.

2.1.2 Energy Consumption

Every computation requires a specific interval of time T to be completed. Formally, the energy consumed E by a system for the computation is the integral of the power dissipated over that time interval T and is measured in the unit Joule.

$$E = \int_0^T P(t)dt = \int_0^T V * I(t)dt \tag{2.3}$$

The energy consumption decreases if the time T required to perform the computation decreases and/or the power dissipation $P(t)$ decreases. Assuming that the measured current does not show a high degree of variation over the time interval T and considering that the voltage is kept constant during this period, Equation 2.3 can be simplified to the following form:

$$E \approx V * I_{avg} * T \tag{2.4}$$

Equation 2.4 was used to determine the energy model (*cf.* Subsection 3.1.1) for the uni-processor ARM based system. Physical measurements were carried out to measure the average current I_{avg} drawn by the processor and the on-board memory present on the evaluation board. In the following section, we present an introduction to power and energy optimization techniques.

2.2 Survey on Power and Energy Optimization Techniques

Numerous researchers have proposed power and energy consumption models [77, 78, 114, 118] at various levels of granularity to model the power or energy consumption of a processor or a complete system. All these models confirm that the processor and the memory subsystem are major contributors of the total power or the energy budget of the system with the interconnect being the third largest contributor. Therefore, for the sake of simplicity, we have classified the optimization techniques according to the component which is the optimization target. Figure 2.2 presents the classification of the optimization techniques into those which optimize the processor energy and which optimize the memory energy. In the remainder of this section, we will concentrate on different optimization techniques but first we would like to clarify if optimizing for power is also optimizing for energy and vice-versa.

2.2.1 Power vs. Energy

According to its definition (*cf.* Equation 2.2), power in a CMOS circuit is dissipated at a given time instant. In contrast, energy (*cf.* Equation 2.3) is the sum of the power dissipated during a given time period. A compiler optimization reduces energy consumption if it reduces the power dissipation of the system and/or the execution time of the application. However, if an optimization reduces the peak power but significantly increases the execution time of the application, the power optimized application will not have optimized energy consumption. In wake of the above discussion, we deduce that the answer to the question of relationship between power and energy optimizations depends on a third parameter *viz.* the execution time. Therefore, the answer could be either yes or no, depending on the execution time of the optimized application.

There are optimization techniques whose objective is to minimize the power dissipation of a system. For example, approaches [72, 116] perform instruction scheduling to minimize bit-level switching activity on the instruction bus and therefore, minimize its power dissipation. The priority for scheduling an instruction is inversely proportional to its Hamming distance from an already scheduled instruction. Mehta et al. [88] presented a register labeling approach to minimize transitions in register names across consecutive instructions. A different approach [84] smoothens the power dissipation profile of an application through instruction scheduling and reordering to increase the usable energy in a battery. All the above approaches also minimize the energy consumption of the system as the execution time of the application is either reduced or kept constant. In the remainder of this chapter, we will not distinguish between optimizations which minimize the power dissipation or the energy consumption.

2.2.2 Processor Energy Optimization Techniques

We further classify the approaches which optimize the energy consumption of a processor core into the following categories:

(a) Energy efficient code generation and optimization
(b) Dynamic voltage scaling (DVS) and dynamic power management (DPM)

Energy Efficient Code Generation and Optimization:
Most of the traditional compiler optimizations [93], *e.g.* common subexpression elimination, constant folding, loop invariant code motion, loop unrolling, etc. reduce the number of executed instructions (operations) and as a result reduce the energy consumption of the system. Source level transformations such as strength reduction and data type replacement [107]

are known to reduce the processor energy consumption. The strength reduction optimization replaces a costlier operation with a equivalent but cheaper operation. For example, the multiplication of a number by a constant of the type 2^n can be replaced by an n bit left shift operation because a shift operation is known to be cheaper than a multiplication. The data type replacement optimization replaces, for example, a floating point data type with a fixed point data type. Though, care must be taken that the replacement does not affect the accuracy bound, usually represented as Signal-to-Noise Ratio (SNR), of the application.

In most of the optimizing compilers, the code generation step consists of the *code selection*, *instruction scheduling* and *register allocation* step. Approaches [114, 118] use instruction-level energy cost models to perform an energy optimal *code selection*. The ARM processors feature two different bit-width instruction sets, *viz* 16-bit Thumb and 32-bit ARM mode instruction sets. The 16-bit wide instructions result in an energy efficient but slower code, whereas the 32-bit wide instructions result in faster code. Authors in [71] use this property to propose a code selector which can choose between 16-bit and 32-bit instruction sets depending on the performance and energy requirements of the application.

The energy or power optimizing *instruction scheduling* is already described in the previous subsection. Numerous approaches [25, 42, 45, 70, 109] to perform *register allocation* are known. The *register allocation* step is known to reduce the energy consumption of a processor by efficiently utilizing its register file and therefore, reducing the number of accesses to the slow memory. Authors of [42] proposed an Integer Linear Programming (ILP) based approach for optimal register allocation, while the approach [70] performs optimal allocation for loops in the application code. The approach [109] presents a generalized version of the well known graph coloring based register allocation approach [25].

Dynamic Voltage Scaling and Dynamic Power Management:
Due to the emergence of embedded processors with voltage scaling and power management features, a number of approaches have been proposed which utilize these features to minimize the energy consumption. Typically, such an optimization is applied after the code generation step. These optimizations require a global view of all tasks in the system, including their dependences, WCETs, deadlines etc.

From Equation 2.2, we know that the power dissipation of a CMOS circuit decreases quadratically with the decrease in the supply voltage. The maximum clock frequency f_{max} for a CMOS circuit also depends on the supply voltage V_{dd} using the following relation:

$$\frac{1}{f_{max}} \sim \frac{V_{dd}}{(V_{dd} - V_t)^2} \tag{2.5}$$

where V_t is the threshold voltage [102] in a CMOS transistor. The power dissipation decreases faster than the speed of the circuit on reducing the supply voltage. Therefore, we could reduce the energy consumption of the circuit by appropriately scaling the supply voltage. A number of interesting approaches have been proposed which apply voltage scaling to minimize the energy consumption and also ensure that each task just finishes at its deadline.

Authors in [57] proposed a design time approach which statically assigns a maximum of two voltage levels to each task running on a processor with discretely variable voltages. However, an underlying assumption of the approach is that it requires a constant execution time or a WCET bound for each task.

A runtime voltage scaling approach [75] is proposed for tasks with variable execution times. In this approach, each task is divided in regions corresponding to time slots of equal length. At the end of each region's execution, a re-evaluation of the execution state of the task is done. If the elapsed execution time after a certain number of regions is smaller than the allotted time slots, the supply voltage is reduced to slow down the processor. Authors in [105] proposed an approach to insert system calls at those control decision points which affect the execution path. At these points, a re-evaluation of the task execution state is done in order to perform voltage scaling.

The above approaches can be classified as compiler-assisted voltage scaling approaches, as each task is pre-processed off-line by inserting system calls for managing the supply voltage. Another class of approaches [49, 105] which combine traditional task scheduling algorithms, such as Rate Monotonic Scheduling (RMS) and Earliest Deadline First (EDF) with dynamic voltage scheduling are also known.

Dynamic Power Management (DPM) is used to save energy in devices that can be switched on and off under the operating system's control. It has gained a considerable attention over the last few years both from the research community [20, 110] and the industry [56]. The DPM approaches can be classified into predictive schemes [20, 110] and stochastic optimum control schemes [19, 106]. Predictive schemes attempt to predict a device's usage behavior depending on its past usage patterns and accordingly change the power states of the device. Stochastic schemes make probabilistic assumptions on the usage pattern and exploit the nature of the probability distribution to formulate an optimization problem. The optimization problem is then solved to obtain a solution for the DPM approach.

2.2.3 Memory Energy Optimization Techniques

The techniques to optimize the energy consumption of the memory subsystem can also be classified into the following two broad categories:

(a) Code optimization techniques for a given memory hierarchy.
(b) Memory synthesis techniques for a given application.

The first set of approaches optimizes the application code for a given memory hierarchy, whereas, the second set of approaches synthesizes application specific memory hierarchies. Both sets of approaches are designed to minimize the energy consumption of the memory subsystem.

Code Optimization Techniques:
Janet Fabri [38] presented one of the earliest approach on optimizing an application code for a given memory hierarchy. The proposed approach overlays arrays in the application such that the required memory space for their storage can be minimized.

Numerous approaches [24, 101, 119, 138], both in the general computing and the high-performance computing domain, have been proposed to optimize an application according to a given cache based memory hierarchy. The main objective of all the approaches is to improve the locality of instruction fetches and data accesses through code and data layout transformations.

Wolf et al. [138] evaluated the impact of several loop transformations such as data tiling, interchange, reversal and skewing on locality of data accesses. Carr et al. [24] considered two additional transformations, *viz.* scalar replacement and unroll-and-jam, for data cache optimization.

Authors of [101, 119] proposed approaches to reorganize the code layout in order to improve locality of instruction fetches and therefore, improve the performance of the instruction cache. The approach [101] uses a heuristic which groups basic blocks within a function according to their execution counts. In contrast, the approach [119] formulates the code reorganization problem to minimize the number cache misses as an ILP problem which is then solved to obtain an optimal code layout.

Another set of approaches is known to optimize the application code for Flash memories and multi-banked DRAM main memories. Flash memories are use to store the application code because of their non-volatile nature. Authors in [98, 133] proposed approaches to manage the contents of the Flash memory and also utilize its execute-in-place (XIP) features to minimize the overall memory requirements. Authors in [95] proposed an approach to manage data within different banks of the main memory such that the unused memory banks could be moved to the power-down state to minimize the energy consumption. In contrast, authors in [133] use the scratchpad to move the main memory into the power-down state for a maximum time duration.

Numerous approaches [23, 65, 97, 115] which optimize the application code such that it efficiently utilizes scratchpad based memory hierarchies have been proposed. We will not discuss these approaches here, as they are extensively discussed in subsequent chapters on memory optimization techniques.

Application Specific Memory Hierarchy Synthesis:
There exists an another class of approaches which generate memories and/or memory hierarchies which are optimized for a given application. These approaches exploit the fact that most embedded systems typically run a single application throughout their entire life time. Therefore, a custom memory hierarchy could be generated to minimize the energy consumption of these embedded systems.

Vahid et al. [48, 141] have extensively researched the generation of application specific and configurable memories. They observed that typical embedded applications spend a large fraction of their time executing a small number of tight loops. Therefore, they proposed a small memory called a *loop cache* [48] to store the loop bodies of the loops found in applications. In addition, they proposed a novel cache memory called *way-halting cache* [141] for the early detection of cache misses. The tag comparison logic of the proposed memory includes a small fully-associative memory that quickly detects a mismatch in a particular cache way and then halts further tag and data access to that way.

Authors in [27] proposed a software managed cache where a particular way of the cache can be blocked at runtime through control instructions. The cache continues to operate in the same fashion as before, except that the replacement policy is prohibited from replacing any data line from the blocked way. Therefore, the cache can be configured to ensure predictable accesses to time-critical parts of an application.

The generation of application specific memory hierarchies has been researched by [82] and [99]. Approaches in [82] can generate only scratchpad based memory hierarchies, whereas those in [99] can create a memory hierarchy from a set of available memory modules such as caches, scratchpads and stream buffers.

3 Memory Aware Compilation and Simulation Framework

A coherent compilation and simulation framework is required in order to develop memory optimizations and to evaluate their effectiveness for complex memory hierarchies. The three most important properties of such a framework should be the following:

(a) configurability
(b) accuracy
(c) coherency

The framework should have a high degree of configurability to simulate complex multi-level memory hierarchies having a wide range of configurable parameters. In addition, it should have access to accurate energy and timing models for the components of the system under optimization. The accurate models enable us to guarantee the effectiveness of the optimizations for real-life memory hierarchies. The coherence between the compilation and simulation frameworks is required as it facilitates a systematic exploration of the design-space. Unfortunately, much of the research community still utilizes ad-hoc frameworks for the design and analysis of memory optimizations.

In this chapter, we describe the memory aware compilation and simulation framework [131] specifically developed to study memory optimization techniques. Figure 3.1 presents the workflow of the developed framework. The coherence property of the framework emerges from the fact that both the compilation and simulation frameworks are configured (*cf.* Figure 3.1) from a unified description of the memory hierarchy. The configurability of the framework is evident from the fact that it supports optimization of complex memory hierarchies found in three orthogonal processor and system architectures, *viz.* uni-processor ARM [11], multi-processor ARM [18] and M5 DSP [28] based systems. The accuracy of the framework is due to the fact that both compilation and simulation frameworks have access to accurate energy and timing models for the three systems. For the uni-processor ARM [9] based system, the framework features a measurement based energy model [114] with an accuracy of 98%. The framework also includes accurate energy models from ST Microelectronics [111] and UMC [120] for multi-processor ARM and M5 DSP based systems, respectively.

The compilation framework includes an energy optimizing compiler [37] for ARM processors and a genetic algorithm based vectorizing compiler [79] for M5 DSPs. All the memory optimizations proposed in this book are integrated within the backends of these

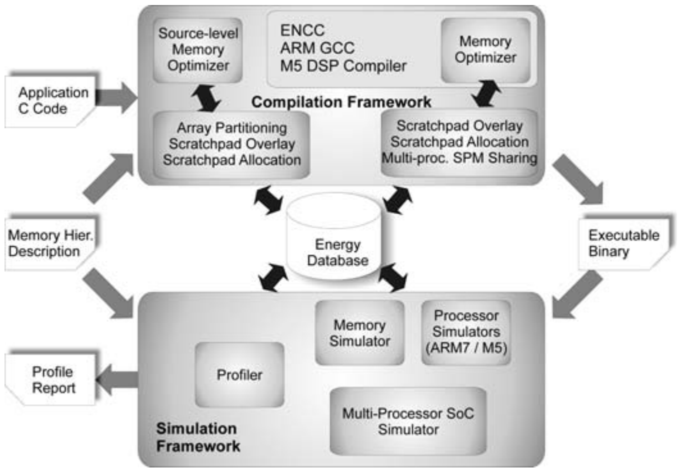


Fig. 3.1. Memory Aware Compilation and Simulation Framework

compilers. Unlike most of the known memory optimizations, the proposed optimization consider both application code segments and data variables for optimization. They transform the application code such that it efficiently utilizes the given memory hierarchy.

The benefit of generating optimizing compilers is that the memory optimizations can utilize precise information about the system and the application available in the compiler backend to perform accurate optimizations. However, the limiting factor is that optimizing compilers are required for every different processor architecture. This prompted us to develop a processor independent "compiler-in-loop" source level memory optimizer. The optimizer collects application specific information from the compiler and then drives the compiler to perform memory optimizations. Currently, the optimizer supports the GCC tool chain [44], though it can be easily made compatible with other compilers. Consequently, the optimizer can optimize memory hierarchies for a wide spectrum of processors supported by the GCC tool chain.

The simulation framework includes processor simulators for ARM and M5 DSP and a highly configurable memory hierarchy simulator [89]. In addition, it includes an energy profiler which uses the energy model and the execution statistics obtained from the simulators to compute the energy consumed by the system during the execution of the application. The simulation framework also includes a multi-processor system simulator [18] which is a SystemC based cycle true simulator of the complete multi-processor system. Currently, it has limited support for multi-level memory hierarchies. Therefore, the integration of the memory hierarchy simulator [89] and the multi-processor simulator is part of our immediate future work.

The workflow of the compilation and simulation framework, common for all the three system architectures, is as follows: The user supplies an application C source code and an XML description of the memory hierarchy to the compilation framework. In addition, the user selects one of the several available memory optimizations to be performed on the application. If a multi-processor ARM based system is under consideration, the chosen memory optimization is applied as a source level transformation and the transformed application is compiled using the GCC tool chain. Otherwise, the memory optimization is applied in the backend of the corresponding compilers.

The compilation framework generates the optimized executable binary of the application which is then passed to the simulation framework for the evaluation of the memory optimization. For uni-processor ARM and M5 DSP based systems, the executable binary is first executed on the processor simulator to generate the instruction trace. The instruction trace is then passed through the memory hierarchy simulator which simulates the memory hierarchy described in the XML file and collects the access statistics for all memories in the hierarchy. The energy profiler collects these statistics from the processor and memory hierarchy simulators and uses the accurate timing and energy models to compute the total execution time and the total energy consumed by the system. On the other hand, the multi-processor simulator simulates the entire system including the processors, memories, buses and other components. In addition, it collects system statistics and reports the total energy consumption of the system.

The remainder of the chapter is organized as follows: The following section describes in-depth the energy model, the compilation and simulation frameworks for the uni-processor ARM based systems. Sections 3.2 and 3.3 provide a similar description of the compilation and simulation frameworks for multi-processor ARM and M5 DSP based systems, respectively.

3.1 Uni-Processor ARM

The experiments for the uni-processor ARM based system are based on an ARM7TDMI evaluation board (AT91EB01) [13]. The ARM7TDMI processor is a simple 32 bit RISC processor which implements the ARM Instruction Set Architecture (ISA) version 4T [11]. It is the most widely used processor core in contemporary low power embedded devices. Therefore, it was chosen as the target processor for evaluating the proposed memory aware energy optimizations.

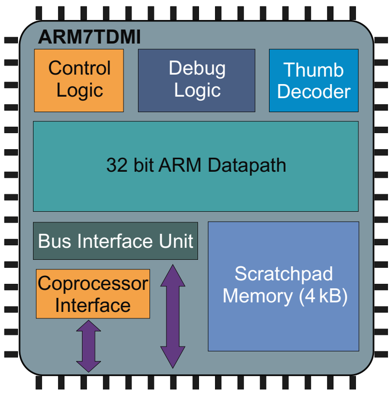


Fig. 3.2. ARM7TDMI Processor

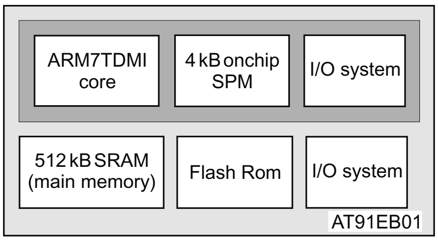


Fig. 3.3. ATMEL Evaluation Board

Figure 3.2 depicts the block diagram of the ARM7TDMI processor core. The data-path of the processor core features a 32 bit ALU, 16 registers, a hardware multiplier and a barrel shifter. The processor has a single unified bus interface for accessing both data and instructions. An important characteristic of the processor core is that it supports two instruction modes, *viz.* ARM and Thumb. The ARM mode allows the 32 bit instructions to exploit the complete functionality of the processor core, whereas Thumb mode instructions are 16 bits wide and can utilize only a reduced functionality of the processor core. For example, Thumb mode instructions can access only the first 8 of 16 registers available in the core. The other important restriction is that predicated instructions enabling conditional execution are not allowed in the Thumb mode.

The processor also includes a hardware decoder unit (*cf.* Thumb Decoder in Figure 3.2) to internally convert 16 bit Thumb instructions to the corresponding 32 bit instructions. The use of Thumb mode instructions is recommended for low power applications, as it results in a high density code which leads to around 30% reduction in the energy dissipated by instruction fetches [71]. The ARM mode instructions are used for performance critical application code segments, as they can utilize the full functionality of the processor core. The availability of predicated instructions in ARM mode reduces the number of pipeline stalls which further improves the performance of the code. Our research compiler (ENCC) generates only Thumb mode instructions because the focus of our research is primarily directed towards energy optimizations.

In addition to the ARM7TMDI processor core, the evaluation board (AT91EB01) has a 512 kB on-board SRAM which acts as the main memory, a Flash ROM for storing the startup code and some external interfaces. Figure 3.3 presents the top-level diagram of the evaluation board. The ARM7TDMI processor core features a 4 kB onchip SRAM memory, commonly known as *scratchpad memory*. Extensive current measurements on the evaluation board were performed to determine an instruction level energy model which is described in the following subsection.

3.1.1 Energy Model

The energy model for the uni-processor ARM based system is based on the energy model from Tiwari et al. [118] and was derived after performing numerous physical measurements on the evaluation board. A detailed description of the energy model can be found in [112].

Tiwari et al. [118] proposed an instruction level energy model of an Intel 486DX2 processor. According to the model, the energy consumption of each instruction consists of two components, namely *base cost* and *inter-instruction cost*. The *base cost* for an instruction refers to the energy consumed by the instruction when it is executed in isolation on the processor. Therefore, it is computed by executing a long sequence of the same instruction and measuring the average energy consumed (or the average current drawn) by the processor core. The *inter-instruction cost* refers to the amount of energy dissipated when the processor switches from one instruction to another. The reason for this energy cost is that on an instruction switch, extra current is drawn because some parts of the processor are switched on while some other parts are switched off. Tiwari et al. also found that for RISC processors, the inter-instruction cost is negligible, *i.e.* around 5% for all instructions.

The energy model [112] used in our setup extends the energy model as it incorporates the energy consumed by the memory subsystem in addition to that consumed by the processor

Instruction	Instruction Memory	Data Memory	Energy (nJ)	Execution Time (CPU Cycles)
MOVE	Main Memory	Main Memory	32.5	2
	Main Memory	Scratchpad	32.5	2
	Scratchpad	Main Memory	5.1	1
	Scratchpad	Scratchpad	5.1	1
LOAD	Main Memory	Main Memory	113.0	7
	Main Memory	Scratchpad	49.5	4
	Scratchpad	Main Memory	76.3	6
	Scratchpad	Scratchpad	15.5	3
STORE	Main Memory	Main Memory	98.1	6
	Main Memory	Scratchpad	44.8	3
	Scratchpad	Main Memory	65.2	5
	Scratchpad	Scratchpad	11.5	2

Table 3.1. Snippet of Instruction Level Energy Model for Uni-Processor ARM System

core. According to the energy model, the energy $E(inst)$ consumed by the system during the execution of an instruction $(inst)$ is represented as follows:

$$E(inst) = E_{cpu\_instr}(inst) + E_{cpu\_data}(inst) + E_{mem\_instr}(inst) + E_{mem\_data}(inst) \tag{3.1}$$

where $E_{cpu\_instr}(inst)$ and $E_{cpu\_data}(inst)$ represent the energy consumed by the processor core during the execution of the instruction $(inst)$. Similarly, the energy values $E_{mem\_instr}(inst)$ and $E_{mem\_data}(inst)$ represent the energy consumed by the instruction and the data memory, respectively.

The ARM7TDMI processor core features a scratchpad memory which could be utilized for storing both data variables and instructions. Therefore, additional experiments were carried by varying the location of variables and instructions in the memory hierarchy. An energy model derived from these additional experiments is shown as follows:

$$E(inst, imem, dmem) = E_{if}(imem) + E_{ex}(inst) + E_{da}(dmem) \tag{3.2}$$

where $E(inst, imem, dmem)$ returns the total energy consumed by the system during the execution of the instruction $(inst)$ fetched from the instruction memory $(imem)$ and possibly accessing data from the data memory $(dmem)$. The validation of the energy model revealed that it possesses a high degree of accuracy, as the average deviation of the values predicted by the model and the measured values was found to be less than 1.7%.

A snippet of the energy model for MOVE, LOAD and STORE instructions is presented in Table 3.1. The table returns the energy consumption of the system due to the execution of an instruction depending upon the instruction and the data memory. It also returns the execution time values for the instructions which are derived from the reference manual [11]. From the table, it can be observed that the energy and execution time values for MOVE instruction are independent of the data memory, as the instruction makes no data access in the memory. A reduction of 50% in the energy consumption values for LOAD and STORE instructions can be observed when the scratchpad memory is used as the data memory. It should also be noted that when both the instruction and data memories are mapped to the scratchpad memory, the system consumes the least energy and time to execute the

Memory	Size (Bytes)	Access Type	Access Width (Bytes)	Energy Per Access (nJ)	Access Time (CPU Cycles)
Main Memory	512k	Read	1	15.5	2
Main Memory	512k	Write	1	15.0	2
Main Memory	512k	Read	2	24.0	2
Main Memory	512k	Write	2	29.9	2
Main Memory	512k	Read	4	49.3	4
Main Memory	512k	Write	4	41.1	4
Scratchpad	4096	Read	x	1.2	1
Scratchpad	4096	Write	x	1.2	1

Table 3.2. Energy per Access and Access Time Values for Memories in Uni-Processor ARM System

instructions. This underscores the importance of the scratchpad memory in minimizing the energy consumption of the system and the execution time of the application.

Table 3.2 summarizes the energy per access and access time values for the main memory and the scratchpad memory. The energy values for the main memory are computed through physical current measurements on the ARM7TDMI evaluation board. The scratchpad is placed on the same chip as the processor core. Hence, the sum of the processor energy and the scratchpad access energy can only be measured. Several test programs which utilized the scratchpad memory were executed and their energy consumption was computed. This energy data along with the linear equation of the energy model (*cf.* Equation 3.2) was used to derive the energy per access values for the scratchpad memory.

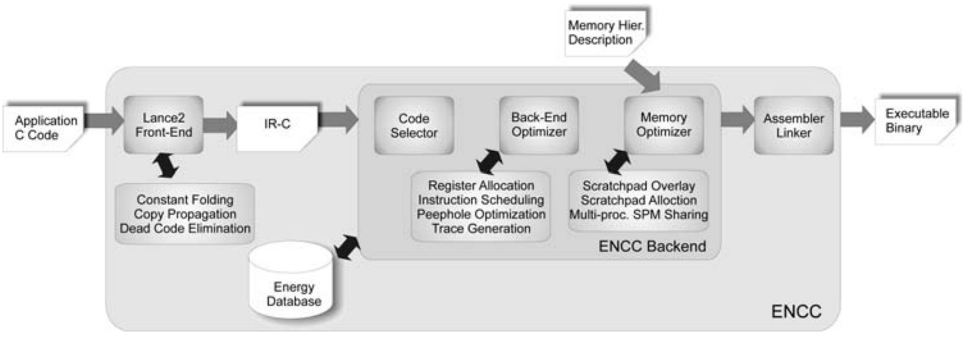


Fig. 3.4. Energy Aware C Compiler (ENCC)

3.1.2 Compilation Framework

The compilation framework for a uni-processor ARM is based on the energy optimizing C compiler ENCC [37]. As shown in the Figure 3.4, ENCC takes application source code written in ANSI C [7] as input and generates an optimized assembly file containing Thumb mode instructions. The assembly file is then assembled and linked using the standard tool chain from ARM, and the executable binary of the application is generated.

In the first step, the source code of the application is scanned and parsed using the LANCE2 [76] front-end which after lexical and syntactical analysis generates a LANCE2 specific intermediate representation also known as IR-C. IR-C is a low-level representation of the input source code where all instructions are represented in *three address code* format. All high-level C constructs such as loops, nested *if*-statements and address arithmetic in the input source code are replaced by primitive IR-C statements. Standard processor independent compiler optimizations such as *constant folding*, *copy propagation*, *loop invariant code motion* and *dead code elimination* [93] are performed on the IR-C.

The optimized IR-C is passed to the ENCC backend where it is represented as a forest of data flow trees. The tree pattern matching based code selector uses the instruction level energy model and converts the data flow trees into a sequence of Thumb mode instructions. The code selector generates an energy optimal cover of the data flow trees as it considers the energy value of an instruction to be its cost during the process of determining a cover. The instruction-level energy model described in the previous subsection is used to obtain energy consumption values or cost for the instructions. After the code selection step, the control flow graph (CFG) which represents basic blocks as nodes and the possible execution flow as edges, is generated.

The control flow graph is then optimized using standard processor dependent optimizations, like *register allocation*[1], *instruction scheduling* and *peephole optimization*. The backend optimizer also includes a well known instruction cache optimization called *trace generation* [101]. The instruction cache optimization provides the foundation for the memory optimizations proposed in the subsequent chapters and therefore, it is described seperately in the following subsection.

In the last step, one of the several memory optimizations is applied and the assembly code is generated which is then assembled and linked to generate the optimized executable binary of the input application. The proposed memory optimizations utilize the energy model and the description of the memory hierarchy to optimize the input application such that on execution it efficiently utilizes the memory hierarchy.

3.1.3 Instruction Cache Optimization

Trace generation [101] is an optimization which is known to have a positive effect on the performance of both the instruction cache and the processor. The goal of the trace generation optimization is to create sequences of basic blocks called *traces* such that the number of branches taken by the processor during the execution of the application is minimized.

Definition 3.1 (Trace). *A trace is a sequence of basic blocks $B_i \cdots B_j$ which satisfy the property that if the execution control flow enters any basic block $B_k : i \le k \le j-1$ belonging to the trace, then there must exist a path from B_k to B_j consisting of only fall-through edges,* i.e. *the execution control flow must be able to reach basic block B_j from basic block B_k without passing through a taken branch instruction.*

A sequence of basic blocks which satisfies the above definition of a trace, has the following properties:

(a) Basic blocks $B_i \cdots B_j$ belonging to a trace are sequentially placed in adjacent memory locations.

[1] Some researchers disagree on register allocation being classified as an optimization.

(b) The last instruction of each trace is always an unconditional jump or a return instruction.

(c) A trace, like a function, is an atomic unit of instructions which can be placed at any location in the memory without modifying the application code.

The third property of traces is of particular importance to us, as it allows the proposed memory optimizations to consider traces as objects of finest granularity for performing memory optimizations. The problem of trace generation is formally defined as follows:

Problem 3.2 (Trace Generation). Given a weighted control flow graph $G(N, E)$, the problem is to partition the graph G such that the sum of weights of all edges within the traces is maximized.

The control flow of the application is transformed to generate traces such that each intra-trace edge is a fall-through edge. In the case that intra-trace edge represents a conditional taken branch, then the conditional expression is negated and the intra-trace edge is transform to a fall-through edge.

The edge weight $w(e_i)$ of an edge $e_i \in E$ represents its execution frequency during the execution of the application. The sum of execution frequencies of taken and non-taken branch instruction is a constant for each run of application with the same input parameters. Therefore, the maximization of the sum of intra-trace edge weights results in the minimization of the sum of inter-trace edge weights which leads to the minimization of execution frequencies of unconditional jumps and taken branches.

The trace generation optimization has twofold benefits. First, it enhances the locality of instruction fetches by placing frequently accessed basic blocks in adjacent memory locations. As a result, it improves the performance of the instruction cache. Second, it improves the performance of the processor's pipeline by minimizing the number of taken branches. In our setup, we restrict the trace generation problem to generate traces whose total size is smaller than the size of the scratchpad memory.

The trace generation problem is known to be an NP-hard optimization problem [119]. Therefore, we propose a greedy algorithm which is similar to the algorithm for the *maximum size bounded spanning tree* problem [30]. For the sake of brevity, we refrain from presenting the algorithm which can alternatively be found in [130]. Trace generation is a fairly common optimization and has been used by a number of researchers to perform memory optimizations [36, 103, 119] which are similar to those proposed in this dissertation.

3.1.4 Simulation and Evaluation Framework

The simulation and evaluation framework consists of a processor simulator, a memory hierarchy simulator and a profiler. In the current setup, the processor simulator is the standard simulator *viz.* ARMulator [12] available from ARM Ltd. ARMulator supports the simulation of only the basic memory hierarchies. Therefore, we decided to implement a custom memory hierarchy simulator (MEMSIM) with the focus on accuracy and configurability.

In the current workflow (*cf.* Figure 3.1), the processor simulator executes the application binary considering a flat memory hierarchy and generates a file containing the trace of executed instructions. The instruction trace is then fed into the memory simulator which simulates the specified memory hierarchy. The profiler accesses the instruction trace, the

Benchmark	Code Size (bytes)	Data Size (bytes)	Description
adpcm	804	4996	Encoder and decoder routines for Adaptive Differential Pulse Code Modulation
edge detection	908	7792	Edge detection in a tomographic image
epic	12132	81884	A Huffman entropy coder based lossy image compression
histogram	704	133156	Global histogram equalization for 128x128 pixel image
mpeg4	1524	58048	mpeg4 decoder kernel
mpeg2	21896	32036	Entire mpeg2 decoder application
multisort	636	2020	A combination of sorting routines
dsp	2784	61272	A combination of various dsp routines (fir, fft, fast-idct, lattice-init, lattice-small)
media	3280	75672	A combination of multi-media routines (adpcm, g721, mpeg4, edge detection)

Table 3.3. Benchmark Programs for Uni-Processor ARM Based Systems

statistics from the memory simulator and the energy database to compute the system statistics, *e.g.* the execution time in CPU cycles and the energy dissipation of the processor and the memory hierarchy. In the following, we briefly describe the memory hierarchy simulator.

Memory Hierarchy Simulator:
In order to efficiently simulate different memory hierarchy configurations, a flexible memory hierarchy simulator (MEMSIM) was developed. While a variety of cache simulators is available, none of them seemed suitable for an in-depth exploration of the design space of a memory hierarchy. In addition to scratchpad memories, the simulation of other memories *e.g.* the loop caches, is required. This kind of flexibility is missing in previously published memory simulation frameworks which tend to focus on one particular component of the memory hierarchy.

The two important advantages of MEMSIM over other known memory simulators, such as Dynero [34], are its cycle true simulation capability and configurability. Currently, MEMSIM supports a number of different memories with different access characteristics, such as caches, loop caches, scratchpads, DRAMs and Flash memories. These memories can be connected in any manner to create a complex multilevel memory hierarchy. MEMSIM takes the XML description of the memory hierarchy and an instruction trace of an application as input. It then simulates the movement of each address of the instruction trace within the memory hierarchy in a cycle true manner.

A graphical user interface is provided so that the user can comfortably select the components that should be simulated in the memory hierarchy. The GUI generates a description of the memory hierarchy in the form of an XML file. Please refer to [133] for a complete description of the memory hierarchy simulator.

Benchmark Suite:
The presentation of the compilation and simulation framework is not complete without the description of the benchmarks that can be compiled and simulated. Our research compiler ENCC has matured into a stable compiler supporting all ANSI-C data types and can compile

and optimize applications from the Mediabench [87], MiBench [51] and UTDSP [73] benchmark suites.

Table 3.3 summarizes the benchmarks that are used to evaluate the memory optimizations. The table also presents the code and data sizes along with a small description of the benchmarks. It can be observed from the table that small and medium size real-life applications are considered for optimization.

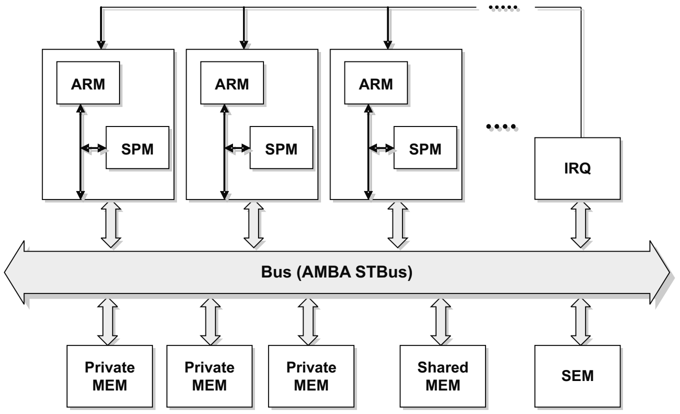


Fig. 3.5. Multi-Processor ARM SoC

3.2 Multi-Processor ARM

The Multi-Processor ARM simulator shown in Figure 3.5 is a SystemC based cycle true multiprocessor simulation framework. It is a full system simulation framework and allows simulation of a configurable number of ARM processors, their local memories, a shared main memory, hardware interrupt and semaphore modules and the bus interconnect. The simulation framework can be configured to simulate an *AMBA AHB bus* [10] or an *ST-Bus* a proprietary bus by STMicroelectronics, as the bus interconnect.

As shown in the figure, each ARM-based processing unit has its own private memory which can be a unified cache or separate caches for data and instructions. A wide range of parameters may be configured, including the size, associativity and the number of wait states. Besides the cache, a scratchpad memory of configurable size can be attached to each processing unit. The simulation framework represents a homogeneous multi-processor system. Therefore, each processor is configured to have the same configuration of its local memory as the other processors.

The multi-processor ARM simulation framework does not support a configurable multilevel memory hierarchy. The memory hierarchy consists of instruction and data caches, scratchpads and the shared main memory. Currently, an effort is being made to integrate MEMSIM into the multi-processor simulator.

3.2.1 Energy Model

The multi-processor ARM simulation framework includes energy models for the processors, the local memories and the interconnect. These energy models compute the energy spent by the corresponding component, depending on its internal state. The energy model for the ARM processor differentiates between *running* or *idle* states of the processor and returns 0.055 nJ and 0.036 nJ as the energy consumption values for the processor states. The above values were obtained from STMicroelectronics for an implementation of ARM7 on an 0.13 μm technology. Though the energy model is not as detailed as the previous measurement based instruction level energy model, it is sufficiently accurate for a simple ARM7 processor.

The framework includes an empirical energy model for the memories created by the memory generator from STMicroelectronics for the same 0.13 μm technology. In addition, the framework includes energy models for the ST-Bus also obtained from STMicroelectronics. However, no energy model is included for the AMBA-Bus. A detailed discussion on the energy models for the multi-processor simulation framework can be found in [78].

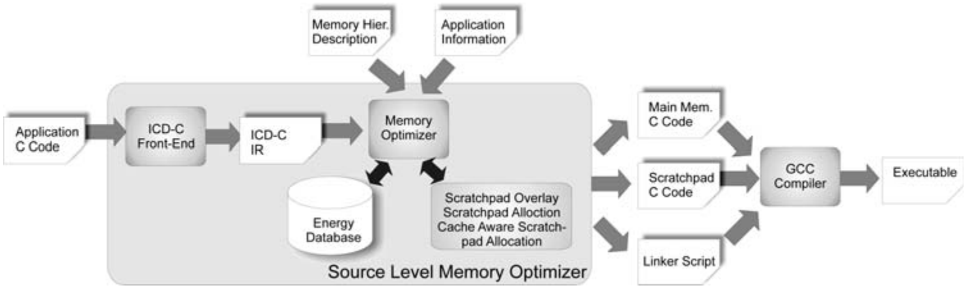


Fig. 3.6. Source Level Memory Optimizer

3.2.2 Compilation Framework

The compilation framework for the multi-processor ARM based systems includes a source level memory optimizer which is based on the ICD-C compilation framework [54] and GCC's cross compiler tool chain for ARM processors. Figure 3.6 demonstrates the workflow of the compilation framework. The application source code is passed through the ICD-C front-end which after lexical and syntactical analysis generates a high-level intermediate representation (ICD-IR) of the input source code. ICD-IR preserves the original high-level constructs, such as loops, *if*-statements and is stored in the format of an *abstract syntax tree* so that the original C source code of the application can be easily reconstructed.

The memory optimizer takes the abstract syntax tree, the memory hierarchy description and an application information file as input. It considers both the data variables and application code fragments for optimization. The information regarding the size of data variables can be computed at the source level but not for code fragments. Therefore, the underlying compiler is used to generate this information for the application and is stored in the application information file.

The memory optimizer accesses the accurate energy model and performs transformations on the abstract syntax trees of the application. On termination, the memory optimizer generates application source files one for each non-cacheable memory in the memory hierarchy. Since the multi-processor ARM simulator does not support complex memory hierarchies, it is sufficient to generate two source files, one for the shared main memory and one for the local scratchpad memory.

The generated source files are then compiled and linked by the underlying GCC tool chain to generate the final executable. In addition, the optimizer generates a linker script which guides the linker to map the contents of the source files to the corresponding memories in order to generate the final executable. The executable is then simulated using the multi-processor ARM simulator, and detailed system statistics, *i.e.* total execution cycles, memory accesses, energy consumption values for processors and memories are collected.

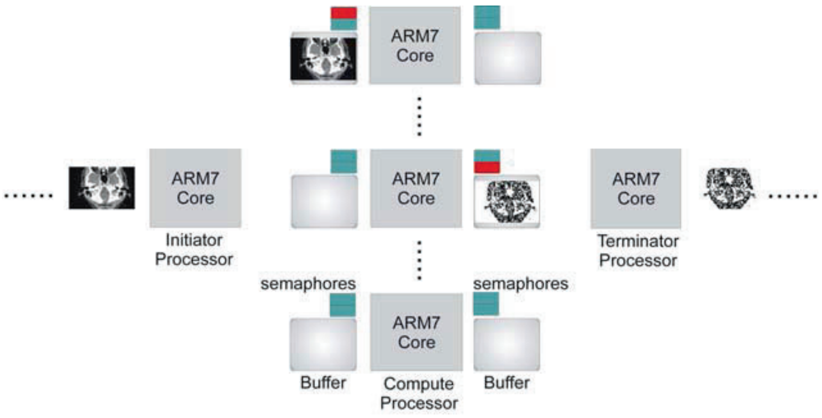


Fig. 3.7. Multi-Process Edge Detection Application

Multi-Process Edge Detection Benchmark:
The memory optimizations for the multi-processor ARM based system are evaluated for the multi-process edge detection benchmark. The original benchmark was obtained from [50] and was parallelized so that it can execute on a multi-processor system.

The multi-processor benchmark consists of an initiator process, a terminator process and a variable number of compute processes to detect the edges in the input tomographic images. The mapping of the processes to the processors is done manually and is depicted in Figure 3.7. As can be seen from the figure, each process is mapped to a different processor. Therefore, a minimum of three processors is required for executing the multi-process application. Each processor is named according to the mapped process.

The multi-process application represents the producer-consumer paradigm. The initiator process reads an input tomographic image from the stream of images and writes it to the input buffer of a free compute process. The compute process then determines the edges on the input image and writes the processed image onto its output buffers. The terminator process then reads the image from the output buffer and then writes it to a backing store. The synchronization between the initiator process and the compute processes is handled by a pair of semaphores. Similarly, another of pair semaphores is used to maintain the synchronization between the compute processes and the terminator process.

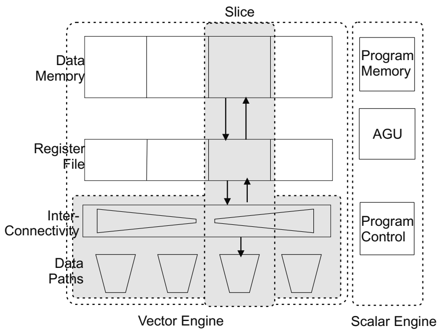


Fig. 3.8. Block Diagram of M5 DSP

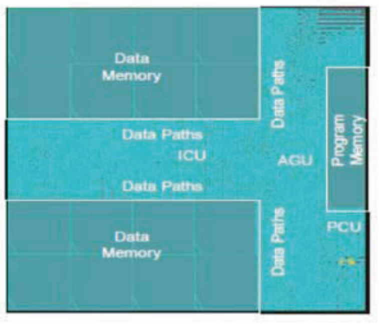


Fig. 3.9. Die Image of M5 DSP

3.3 M5 DSP

The M5 DSP [28] was designed with the objective to create a low power and high throughput digital signal processor. It has a core power consumption of 23 mW and a peak performance of 3.6 GFLOPS/s. The M5 DSP, depicted in Figures 3.8 and 3.9, consists of a fixed control processing part (*scalar engine*) and a scalable signal processing part (*vector engine*). The functionality of the data paths in the *vector engine* can be tailored to suit the application.

The vector engine consists of a variable number of slices where each slice comprises of a register file and a data path. The *interconnectivity unit* (ICU) connects the slices with each other and with the control part of the processor. All the slices are controlled using the *single instruction multiple data* (SIMD) paradigm and are connected to a 64 kB data memory featuring a read and a write port for each slice. The scalar engine consists of a *program control unit* (PCU), *address generation unit* (AGU) and a program memory. The PCU performs operations like jumps, branches and loops. It also features a zero-overhead loop mechanism supporting two-level nested loops. The AGU generates addresses for accessing the data memory.

The processor was synthesized for a standard-cell library by Virtual Silicon$^{\mathrm{TM}}$ for the 130 nm 8-layer-metal UMC process using Synopsys Design Compiler$^{\mathrm{TM}}$. The resulting layout of the M5 DSP is presented in Figure 3.9. The total die size was found to be 9.7 mm^2 with data memory consuming 73% of the total die size.

In our setup, we inserted a small scratchpad memory in between the large data memory and the register file. The scratchpad memory is used to store only data arrays found in the applications. The energy consumption of the entire system could not be computed as the instruction-level energy model for the M5 DSP is currently unavailable. An accurate memory energy model from UMC is used to compute the energy consumption of the data memory subsystem. However, due to copyright reasons, we are forbidden to report exact energy values. Therefore, only normalized energy values for the data memory subsystem of the M5 DSP will be reported in this work.

The compilation framework for the M5 DSP is similar to that for the uni-processor ARM based system. The only significant difference between the two is that the compiler for the M5 DSP uses a phase coupled code generator [80]. The code generation is divided into four subtasks: *code selection* (CS), *instruction scheduling* (IS), *register allocation* (RA) and *address code generation* (ACG). Due to the strong inter-dependencies among these subtasks, the code generator uses a genetic algorithm based phase-coupled approach to generate highly optimized code for the M5 DSP. A genetic algorithm is preferred over an Integer Linear Programming (ILP) based approach because of the non-linearity of the optimization problems for the subtasks. Interested readers are referred to [79] for an in-depth description of the compilation framework.

The proposed memory optimizations are integrated into the backend of the compiler for M5 DSP. The generated code is compiled and linked to create an executable which is then simulated on a cycle accurate processor and memory hierarchy simulator. Statistics about the number and type of accesses to the background data memory and the scratchpad memory are collected. These statistics and the energy model are used to compute the energy dissipated by the data memory subsystem of the M5 DSP. The benchmarks for M5 DSP based systems are obtained from the UTDSP [73] benchmark suite.

4

Non-Overlayed Scratchpad Allocation Approaches for Main / Scratchpad Memory Hierarchy

In this first chapter on approaches to utilize the scratchpad memory, we propose two simple approaches which analyze a given application and select a subset of code segments and global variables for scratchpad allocation. The selected code segments and global variables are allocated onto the scratchpad memory in a non-overlayed manner, *i.e.* they are mapped to disjoint address regions on the scratchpad memory. The goal of the proposed approaches is to minimize the total energy consumption of the system with a memory hierarchy consisting of an L1 scratchpad and a background main memory. The chapter presents an ILP based non-overlayed scratchpad allocation approach and a greedy algorithm based fractional scratchpad allocation approach. The presented approaches are not entirely novel as similar techniques are already known. They are presented in this chapter for the sake of completeness, as the advanced scratchpad allocation approaches presented in the subsequent chapters improve and extended these approaches.

The rest of the chapter is organized as follows: The following section provides an introduction to the non-overlayed scratchpad allocation approaches, which is followed by the presentation of a motivating example. Section 4.3 surveys the wealth of work related to non-overlayed scratchpad allocation approaches. In Section 4.4, preliminaries are described and based on that the scratchpad allocation problems are formally defined. Section 4.5 presents the approaches for non-overlayed scratchpad allocation. Experimental results to evaluate the proposed approaches for uni-processor ARM, multi-processor ARM and M5 DSP based systems are presented in Section 4.6. Finally, Section 4.7 concludes the chapter with a short summary.

4.1 Introduction

In earlier chapters, we discussed that a scratchpad memory is a simple SRAM memory invariably placed onchip along with the processor core. An access to the scratchpad consumes much less energy and CPU cycles than that to the main memory. However, unlike the main memory the size of the scratchpad memory, due to price of the onchip real estate, is limited to be a fraction of the total application size.

The goal of the non-overlayed scratchpad allocation (SA) problem is to map memory objects (code segments and global variables) to the scratchpad memory such that the total

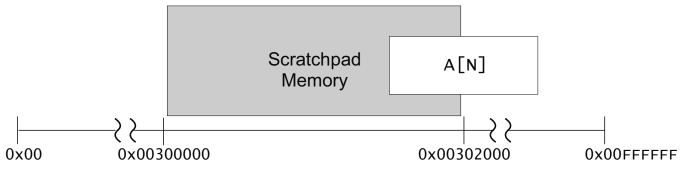


Fig. 4.1. Processor Address Space Containing a Scratchpad Memory

energy consumption of the system executing the application is minimized. The mapping should be done under the constraint that the aggregate size of memory objects mapped to the scratchpad memory should be less than the size of the memory. The proposed approaches use an accurate energy model which, based on the number and the type of accesses originating from a memory object and the target memory, compute the energy consumed by the memory object.

A closer look at the scratchpad allocation (SA) problem reveals that there exists an exact mapping between the problem and the knapsack problem (KP) [43]. According to the knapsack problem, the hitch-hiker has a knapsack of capacity W and has access to various objects $o_k \in O$ each with a size w_k and a perceived profit p_k. Now, the problem of the hitch-hiker is to choose a subset of objects $O_{kp} \subseteq O$ to fill the knapsack ($\sum_{o_k \in O_{kp}} w_k \leq W$) such that the total profit ($\sum_{o_k \in O_{kp}} p_k$) is maximized. Unfortunately, the knapsack problem is known to be an NP-complete problem [43].

In most embedded systems, the scratchpad memory occupies a small region of the processor's address space. Figure 4.1 shows that in the considered uni-processor ARM7 setup, the scratchpad occupies a 4k address region (`[0x00300000, 0x00302000]`) from the processor's address space (`[0x00000000, 0x00FFFFFF]`). Any access to the 4k address region is translated to a scratchpad access, whereas any other address access is mapped to the main memory. We utilize this property to relax the scratchpad allocation problem such that a maximum of one memory object can be fractionally allocated to the scratchpad memory. We term the relaxed problem as the fractional scratchpad allocation (Frac. SA) problem. Figure 4.1 depicts the scenario when an array `A` is partially allocated to the scratchpad memory. It should be noted that this seamless scratchpad and main memory accesses may not be available in all systems.

The Frac. SA problem demonstrates a few interesting properties. First, it is similar to the fractional knapsack problem (FKP) [30], a variant of the KP, which allows the knapsack to be filled with partial objects. Second, a greedy approach [30], which fills the knapsack with objects in the descending order of their valence (profit per unit size p_k/w_k) and breaking only the last object if it does not fit completely, finds the optimal solution for the fractional knapsack problem. This implies that the greedy approach for FKP can be also use to solve the Frac. SA problem, as it allows the factional allocation of a maximum of one memory object.

Third, the total profit obtained by solving the fractional knapsack problem is larger than or equal to the profit of the corresponding knapsack problem as the former is a relaxation of the latter. An unsuspecting reader might imply that the solution to Frac. SA problem achieves larger reduction in energy consumption than that to the SA problem. Unfortunately, this is not always true for the scratchpad allocation problems as the memory objects do not have homogenous valences (p_k/w_k), *i.e.* each element within a memory object is not accessed an equal number of times. For example, consider the following loop nest:

```
for(i=0;i<N;i++) {
  for(j=i;j<N;j++) {
    A[j] = ...; } }
```

The mapping of array A as the last object to the scratchpad (*cf.* Figure 4.1) will not result in an optimal solution, as element A[N-1] is accessed far more times than element A[0]. The Frac. SA problem achieves better solutions than the SA problem if the fractionally allocated memory object has uniform valence for each of its elements or is biased towards the scratchpad allocated portion. The computation of a fine grained valence for each memory object was not considered as it is not trivial and requires a significant computation overhead for profiling. Nevertheless, we will demonstrate that the greedy approach for the Frac. SA problem computes solutions which are very close to optimal solutions for the SA problem. In the following section, we begin by describing a motivating example.

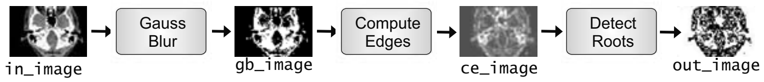


Fig. 4.2. Workflow of Edge Detection Application

Instruction Memory			Data Memory		
Function	Size (bytes)	Execution Count	Array	Size (bytes)	Access Count
ReadImage	32	38,092	in_image	5,120	9,400
GaussBlur	324	785,313	gb_image	5,120	41,320
ComputeEdges	144	1,137,824	ed_image	5,120	9,960
DetectRoots	224	964,147	out_image	5,120	5,648
WriteImage	20	14,411	tmp_image	5,120	50,720
			Gauss	16	13,686
			x_offset	32	36,480
			y_offset	32	36,480
Total Inst	744	2,939,787	Total Data	30,800	203,694

Table 4.1. Execution and Access Counts for Functions and Arrays in Edge Detection Application

4.2 Motivation

We begin by deciding which fragments of the application should be considered as memory objects or as candidates for allocation on the scratchpad memory. Should the set of memory objects consist of only data elements or only instructions, or a combination of both? In order to obtain the answer to the above question, we compiled the Edge Detection application for an ARM7 processor using an energy optimizing research compiler (ENCC) [37].

Memory	Size (bytes)	Access Width (bytes)	Energy per Access (abstract units)
Scratchpad	512	2	2
Scratchpad	512	4	2
Main Memory	64k	2	10
Main Memory	64k	4	20

Table 4.2. Energy per Access Values for Scratchpad and Main Memory

Figure 4.2 presents the workflow of the Edge Detection application which determines edges in a tomographic image. The profile information, gathered by profiling the generated application binary, is presented in Table 4.1. The left side and the right side of the table present the profile information for functions and arrays of the application, respectively. The execution count for a function is the sum of the execution counts of every instruction in the function, whereas the access count for an array is the sum of the access counts of each array element.

We make the following observations upon studying Table 4.1. First, the total instruction size is much smaller than the total data size of the application, while the total execution count for instructions is an order of magnitude larger than the total access count for data arrays. This implies that instructions should belong to the set of memory objects as they have high execution count and consume much less space.

Second, the access count per unit size for array `Gauss` is larger than the execution count per unit size for function WriteImage. Similarly, the access counts per unit size for arrays `x_offset` and `y_offset` are comparable to that of ReadImage and are larger than that of WriteImage. This implies that arrays should also be included in the set of memory objects. Moreover, all the access functions of the image arrays in the application are affine functions. Hence, DTSE techniques [140] can be utilized to generate small slices for the image arrays which can be assigned to the small scratchpads. We did not consider generating array slices as DTSE techniques are orthogonal to our approach. However, they can be implemented as pre-pass optimizations in our setup. Based on the above two observations, we conclude that the set of memory objects should comprise of data elements as well as code segments.

After having decided that both code segments and global variables should be considered as memory objects, we would like quantify the energy reduction that could be achieved by allocating the memory objects onto the scratchpad memory. Let us assume that we have a memory hierarchy consisting of a 512 bytes onchip scratchpad and a 64k bytes main memory. Additionally, assume that we have a simple per access energy model (*cf.* Table 4.2) for the memories. The code generated by the ENCC [37] compiler is Thumb-mode code for the ARM processor, *i.e.* each instruction is 2 bytes or 16 bits wide. Therefore, the energy consumed by the memory hierarchy when the scratchpad memory is not utilized is $2,939,787*10+203,694*20 = 33,471,750$ units. The product of the number of instruction fetches $(2,939,787)$ (*cf.* last row of Table 4.1) and the energy per access (10) to the main memory computes the energy consumed due the instruction fetches. Similarly, the energy consumed due to the data accesses can be computed. The sum of the energy consumed due to instruction fetches and data accesses is the energy consumed by the memory hierarchy.

Now, let us assume that we use the optimal SA approach, described in Subsection 4.5.1, to allocate memory objects to the 512 bytes scratchpad memory. The approach selects both functions and array variables for the allocation on the scratchpad memory. The selected functions are ReadImage, ComputeEdges, DetectRoots and WriteImage, while the selected array variables are `Gauss`, `x_offset` and `y_offset`. Like previously, the energy consumed by the memory hierarchy after the scratchpad allocation is computed to be $14,676,330$ units. It should be noted that by using a scratchpad memory having a size of just 1.6% of the total application size of $30,800 + 744 = 31,544$ bytes, we could reduce the energy consumption of the memory hierarchy to $(14,676,33/33,471,750) * 100 = 43.84\%$ of its original value. For this example, the Frac. SA approach achieves a slightly better allocation. However, for the sake of brevity, we refrain from presenting the allocation due to Frac. SA approach.

4.3 Related Work

Non-overlayed scratchpad allocation approaches [5, 6, 14, 29, 97, 115, 134] have been thoroughly researched by numerous research groups in the past decade. The proposed approaches allocated either data variables or instruction segments or both onto one or many scratchpads.

Panda et al. [97] were the first ones to demonstrate the effectiveness of the scratchpad memory in minimizing the energy consumption of the system. They considered a memory hierarchy consisting of instruction and data caches and a scratchpad as the onchip memories and a background main memory. The proposed approach [97] analyzes the application characteristics, such as life-times, interference and access counts of variables and then allocates array variables onto the scratchpad memory with the objective to minimize their interference in the data cache.

The authors [14] proposed an approach to utilize the scratchpad for storing global variables as well as stack frames. They formulated the allocation problem as an *integer linear programming* (ILP) problem and solved it to obtain an optimal allocation. Sjödin et al. [108] proposed a similar approach to allocate only global variables onto the scratchpad. In contrast, authors [29] utilized the scratchpad memory as a cheap alternative for storing spilled register values. They demonstrated that the spilled register values cause a notable interference in the data cache, and therefore justified that the spilled values should be moved to the non-cacheable scratchpad memory.

The authors in [5] divided aggregate array variables into disjoint partitions based on the footprints of their references. Profit values, based on the number of distinct accesses, are assigned to the partitions, then a knapsack algorithm is used to perform non-overlayed allocation of the array partitions on the scratchpad memory.

Angiolini et al. [6] proposed an approach to allocate only instruction segments onto the scratchpad memory. The approach deviates from the previous approaches in the fact that it modifies a given executable binary of the application for scratchpad allocation. Therefore, the approach can even optimize those legacy applications for which source code is unavailable. On the downside, the approaches which modify application executables are known to be very instruction set architecture (ISA) specific and error-prone as not all the information could be stripped out of an executable.

The authors in [115] demonstrated that significant energy reductions could be achieved by allocating both instruction segments and data variables onto the scratchpad. The authors assigned the profit values to instruction segments and variables, based on an accurate energy model [114] and the execution profile of the application. They formulated the scratchpad allocation problem as a knapsack problem. The optimal solution to the problem achieved the highest energy reductions among the contemporary approaches. An approach [126] to allocate smaller array partitions extended the previous approach.

The authors in [134] presented an approach to allocate both instructions and variables onto partitioned scratchpad memories in a non-overlayed manner. They formulated two ILP versions *viz.* top-down and bottom-up, of the problem which are then solved using a commercial ILP solver to achieve a 22% improvement compared to a single scratchpad approach [115]. In the following section, we present the analysis and the definitions of the non-overlayed scratchpad allocation problems.

4.4 Problem Formulation and Analysis

The proposed scratchpad allocation approaches allocate memory objects to the disjoint address regions on the scratchpad memory with the objective to minimize the total energy consumption of the system executing the given application. The approaches allocate memory objects in a static manner, *i.e.* the address location for each memory object is fixed at compile time which then remains invariant or static during the entire execution of the application.

Before presenting the formal definition of the allocation problem, we first define the components of a given application that are considered as memory objects. This is necessary because the memory objects represent the finest granularity application fragments which are considered for allocation onto the scratchpad memory. Moreover, the allocation approaches are implemented for three different system architectures. In Subsection 4.4.2, we describe the energy model used by the proposed allocation approaches to compute the energy dissipated by a memory object. Finally, we formally define the allocation problems.

4.4.1 Memory Objects

The non-overlayed scratchpad approaches are implemented for a uni-processor ARM, a multi-processor ARM and an M5 DSP based system. As described in Chapter 3, the approaches for uni-processor ARM and M5 DSP based systems are available in the backend of their respective compilers, whereas, for the multi-processor ARM based system, they are developed as source-level transformations. The memory objects for a uni-processor ARM based system consist of the following:

(a) Aggregate global variables (V) including *scalar* and *non-scalar* variables.
(b) Code segments including *traces* (T) and *functions* (F).

The following are the memory objects for the multi-processor ARM based system:

(a) Aggregate global variables (V) including *scalar* and *non-scalar* variables.
(b) Code segments including only *functions* (F).

The memory objects for the M5 DSP based system include only global array variables, because for DSPs the energy consumed by the data memory is much larger than that by the instruction memory. Table 4.3 summarizes the memory objects for each system architecture.

Memory Optimization	System Architecture	Memory Objects	Explanation
Non-Overlayed Scratchpad Allocation for MM / SPM Hierarchy (Chapter 4)	Uni-processor ARM	$MO \subseteq V \cup T \cup F$	global variables, traces and functions
	M5 DSP	$MO \subseteq V$	global data arrays
	Multi-processor ARM	$MO \subseteq V \cup F$	global variables and functions

Table 4.3. Memory Objects for Non-Overlayed Scratchpad Allocation Approach

4.4.2 Energy Model

The energy function $E(mo, mem)$ shown below returns the energy consumed by a memory object when it is located in a memory mem.

$$E(mo, mem) = \begin{cases} E_{var}(mo, mem) & \text{if } mo \in V \\ E_{inst}(mo, mem) & \text{if } mo \in T \cup F \end{cases} \tag{4.1}$$

The above equation represents the observed fact that the energy $E_{inst}(mo, mem)$ dissipated during the execution of an instruction segment is different than the energy dissipated $E_{var}(mo, mem)$ due to an access to a variable. The energy dissipation function for variables is presented in the following:

$$E_{var}(mo, mem) = n_r(mo) * E_{read}(mem) + n_w(mo) * E_{write}(mem) \tag{4.2}$$

where $n_r(mo)$ and $n_w(mo)$ are the total number of read and write accesses to a memory object $mo \in V$. $E_{read}(mem)$ and $E_{write}(mem)$ return the energy consumed by memory mem on a read and a write access, respectively. These values can be determined from memory datasheets available from vendors. Otherwise, they can be also derived from the following instruction level energy model.

$$E(inst, imem, dmem) = E_{if}(imem) + E_{ex}(inst) + E_{da}(dmem) \tag{4.3}$$

The energy function $E(inst, imem, dmem)$ returns the total energy dissipated by the system during the execution of the instruction ($inst$) fetched from the instruction memory ($imem$) and possibly accessing data from the data memory ($dmem$). An in-depth discussion of the energy model in presented Chapter 3.

$$E_{read}(mem) = \left[E(load, MM, mem) - E(mov, MM, mem)\right] \tag{4.4}$$
$$E_{write}(mem) = \left[E(store, MM, mem) - E(mov, MM, mem)\right] \tag{4.5}$$

Using the Equation 4.3, the energy dissipated during a read access $E_{read}(mem)$ (*cf.* Equation 4.4) to a variable in the memory (mem) can estimated to be the difference between the energy consumption of a $load$ instruction accessing the memory mem and that of a register-move mov instruction. The energy dissipated during a write access $E_{write}(mem)$ can be computed in a similar manner.

$$E_{inst}(mo, mem) = inst\_fetch(mo) * [E_{if}(mem) + E_{ex}(inst)] \tag{4.6}$$
$$= inst\_fetch(mo) * E(mov, mem, MM) \tag{4.7}$$

The above function computes the energy dissipated during the execution of a memory object $mo \in T \cup V$ belonging to the instruction segments. The product of the number of instruction fetches $inst\_fetch(mo)$ originating from the memory object mo and the average energy dissipated during the execution of an instruction, estimates the energy $E_{inst}(mo, mem)$ dissipated due to the execution a memory object. An average value is used because for ARM processors all, except multiply and shift, instructions dissipate almost the same amount of energy upon execution. The energy dissipated by a register-move mov instruction is used as the average energy value in our model.

4.4.3 Problem Formulation

The formal definitions of the non-overlayed scratchpad allocation (SA) and fractional scratchpad allocation (Frac. SA) problems are presented in the following:

Problem 4.1 (Scratchpad Allocation (SA)). Given the set of memory objects MO, the energy model $E(mo, mem)$, and a memory hierarchy consisting of a scratchpad memory (SPM) and a main memory (MM). The problem is to determine a subset $MO_{SPM} \subseteq MO$ of the set of memory objects MO such that the allocation of memory objects $mo_i \in MO_{SPM}$ to the scratchpad memory minimizes the total energy consumption of the system.

$$E_{Total} = \left[\sum_{mo_i \in MO_{SPM}} E(mo_i, SPM) \right] + \left[\sum_{mo_i \in MO/MO_{SPM}} E(mo_i, MM) \right] \quad (4.8)$$

The minimization of the total energy consumption E_{Total} is to be performed under the following scratchpad size constraint:

$$\sum_{mo_i \in MO_{SPM}} size(mo_i) \leq size(SPM) \quad (4.9)$$

The minimization version of the SA problem can be converted to a maximization problem which closely resembles the knapsack problem. In order to convert the problem, we subtract the objective function value E_{Total} from a high threshold value $E_{Threshold}$ and name the difference as E_{Profit} shown in the following:

$$E_{Profit} = E_{Threshold} - E_{Total} \quad (4.10)$$

Next, we define $E_{Threshold}$, shown below, as the energy consumed by the system when all memory objects are allocated to the main memory.

$$E_{Threshold} = \sum_{mo_i \in MO} E(mo_i, MM) \quad (4.11)$$

The above definition of $E_{Threshold}$ and the definition of E_{Total} (*cf.* Equation 4.8) are used to reformulate E_{Profit} as follows:

$$\begin{aligned} E_{Profit} = & \left[\sum_{mo_i \in MO} E(mo_i, MM) \right] - \left[\sum_{mo_i \in MO_{SPM}} E(mo_i, SPM) \right] \\ & - \left[\sum_{mo_i \in MO/MO_{SPM}} E(mo_i, MM) \right] \end{aligned} \quad (4.12)$$

$$= \left\lfloor \sum_{mo_i \in MO_{SPM}} E(mo_i, MM) - E(mo_i, SPM) \right\rfloor \tag{4.13}$$

$$= \left\lfloor \sum_{mo_i \in MO_{SPM}} \Delta E(mo_i) \right\rfloor \tag{4.14}$$

where $\Delta E(mo_i)$ represents the difference in energy consumption values for a memory object mo_i when it is present in the main memory and in the scratchpad memory. The profit function E_{Profit} needs to be maximized in order to minimize the objective function E_{Total}. The maximization problem resembles the knapsack problem which is known to be an NP-complete problem [43].

Problem 4.2 (Fractional Scratchpad Allocation (Frac. SA)). Given the set of memory objects MO, the energy model $E(mo, mem)$, and a memory hierarchy consisting of a scratchpad memory (SPM) and a main memory (MM). The problem is to determine a subset $MO_{SPM} \subseteq MO$ of the set of memory objects MO such that the full or partial allocation of memory objects $mo_i \in MO_{SPM}$ to the scratchpad memory maximizes the total energy profit E_{Profit}.

$$E_{Profit} = \left\lfloor \sum_{mo_i \in MO_{SPM}} \left(\frac{spmsize(mo_i)}{size(mo_i)} \right) * \Delta E(mo_i) \right\rfloor \tag{4.15}$$

where, $spmsize(mo_i)$ represents the amount of space a memory object occupies on the scratchpad memory. The maximization of the total energy profit E_{Profit} is to be performed under the following constraints:

(a) The aggregate space occupied by the memory objects on the scratchpad memory should be less than the size of the memory.

$$\sum_{mo_i \in MO_{SPM}} spmsize(mo_i) \leq size(SPM) \tag{4.16}$$

(b) A maximum of one memory object should be partially allocated on the scratchpad.

In the following section, we present the ILP based SA approach and the greedy algorithm based Frac. SA approach.

4.5 Non-Overlayed Scratchpad Allocation

The current section presents an integer linear programming (ILP) based optimal approach to solve the SA problem and a greedy algorithm based fractional scratchpad allocation approach. In the following, we start by presenting the ILP formulation of the SA problem.

4.5.1 Optimal Non-Overlayed Scratchpad Allocation

Let us define the following binary variable $l(mo_i)$ to denote the location of the memory object mo_i in the memory hierarchy.

$$l(mo_i) = \begin{cases} 1 \text{ if memory object } mo_i \text{ is present in the SPM} \\ 0 \text{ otherwise} \end{cases} \tag{4.17}$$

We use the binary variable $l(mo_i)$ to reformulate the objective function (*cf.* Equation 4.8) of the SA problem as follows:

$$E_{Total} = \sum_{mo_i \in MO} [l(mo_i) * E(mo_i, SPM) + (1 - l(mo_i)) * E(mo_i, MM)] \tag{4.18}$$

The binary variable $l(mo_i)$ is also used to reformulate the scratchpad size constraint (*cf.* Equation 4.9) as shown below:

$$\sum_{mo_i \in MO} l(mo_i) * size(mo_i) \leq size(SPM) \tag{4.19}$$

The ILP formulation solves the non-overlay scratchpad allocation (SA) problem, as it determines the subset of memory objects ($MO_{SPM} = \{mo_i | l(mo_i) = 1\}$) which minimizes the total energy consumption of the system. A commercial ILP solver [32] is used to obtain an optimal solution for the problem. Despite the fact that the knapsack problem is an NP-complete problem, the solver required less than a second to compute the optimal solution for each of the benchmarks. Next, we present the greedy algorithm based fractional scratchpad allocation approach.

4.5.2 Fractional Scratchpad Allocation

Figure 4.3 presents the greedy algorithm to solve the fractional scratchpad allocation problem. First, the algorithm sorts the memory objects according to their valence $V(mo_i) = E(mo_i, MM)/size(mo_i)$ to compute a sorted list of memory objects $SortedMO$. Then,

```
FractionalScratchpadAllocation(MO, SPMSize)
/* sort memory objects according to valence $V(mo_i) = \frac{E(mo_i, MM)}{size(mo_i)}$ */
$SortedMO$ =SortAccordingValence($MO$)
RemSPMSize = SPMSize /* default values */
PrevAddress = SPMBaseAddress
$MO_{SPM} = \{\}$
while ( RemSPMSize $> 0$ ) do
  /* select the memory object with maximum valence */
  $mo_i$ = head($SortedMO$)
  RemSPMSize = RemSPMSize - $size(mo_i)$
  /* add the memory object to the set $MO_{SPM}$ */
  $MO_{SPM} = MO_{SPM} \bigcup mo_i$
  AddressVector[i] = PrevAddress
  PrevAddress += $size(mo_i)$
end-while
return $MO_{SPM}$
```

Fig. 4.3. Greedy Algorithm for Fractional Scratchpad Allocation Problem

Benchmark	Code Size (bytes)	Data Size (bytes)	System Architecture
adpcm	804	4996	uni-processor ARM
edge detection	908	7792	uni-processor ARM
histogram	704	133156	uni-processor ARM
mpeg4	1524	58048	uni-processor ARM
multisort	636	2020	uni-processor ARM
dsp	2784	61272	uni-processor ARM
media	3280	75672	uni-processor ARM
multi-process edge detection	4484	23820	multi-processor ARM
complex multiply		24576	M5 DSP
fir		3688	M5 DSP
fir2dim		3380	M5 DSP

Table 4.4. Benchmark Programs for the Evaluation of Non-Overlayed Scratchpad Allocation Approaches

it removes a memory object mo_i from the head of the list $SortedMO$ and assigns it to the scratchpad memory. This process is repeated till the scratchpad space is completely allocated to memory objects. The algorithm maintains the set of memory objects MO_{SPM} which are chosen for scratchpad allocation and an array (AddressVector) which stores their starting addresses.

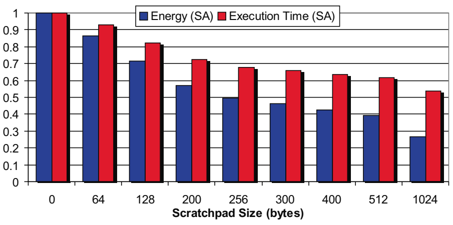
The algorithm requires $O\left(|MO|log(|MO|)\right)$ time for sorting the list of memory objects and $O\left(|MO|\right)$ time to determine the set of memory objects MO_{SPM} for allocation onto the scratchpad memory. If the last memory object chosen for scratchpad allocation has a constant valence for each of its elements, then the algorithm computes a lower energy consuming scratchpad allocation than that computed by the ILP based approach. However, the above premise in not valid for some of our benchmarks for which the algorithm computes a slightly worse scratchpad allocation. We did not investigate theoretical approximation bounds for the proposed greedy algorithm when memory objects have non-homogenous valences. Though, for our set of benchmarks we found that the energy consumption values due to the two approaches lie within 5% of each other. In the following section, we discuss the experimental results obtained for the two approaches.

4.6 Experimental Results

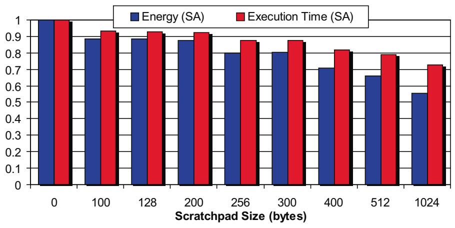
The efficacy of the proposed scratchpad allocation approaches is evaluated for uni-processor ARM, multi-processor ARM and M5 DSP based systems. The workflow employed to conduct the experiments is described in detail in Chapter 3. The benchmarks which are used to evaluate the proposed approaches are summarized in Table 4.4. The table also reports code and data sizes for the benchmarks. The energy values reported in this section are computed using accurate energy models for each of systems. In the following, we start by presenting the experimental results for the uni-processor ARM based system.

4.6.1 Uni-Processor ARM

The evaluation of the scratchpad allocation approaches for the uni-processor ARM based system is organized as follows: First, the benefit of the scratchpad memories allocated using the scratchpad allocation (SA) approach is evaluated. Then, a comparison of optimal and fractional scratchpad allocation approaches is presented.



(a) Edge Detection



(b) DSP

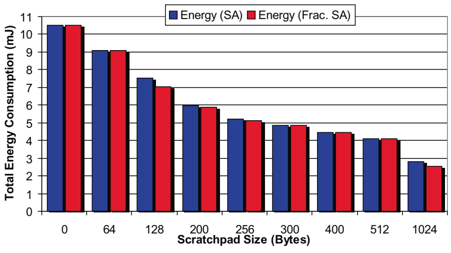
Fig. 4.4. Normalized Energy Consumption and Execution Time for Opt. SA Approach

Scratchpad Benefit:
Figure 4.4 presents the normalized total energy consumption and execution time values of the *edge detection* and *dsp* benchmarks for varying scratchpad sizes. The optimal non-overlayed scratchpad allocation (SA) approach is used to allocate memory objects onto the scratchpad memories. The energy and execution time values are normalized against the corresponding values for a system without a scratchpad memory. A few important observations can be made from the figure.

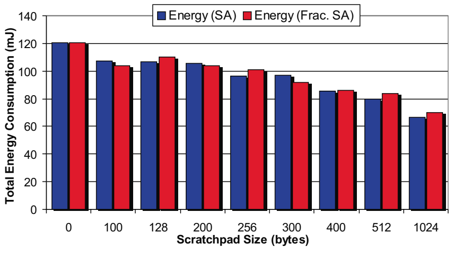
First, the energy consumption as well the execution time values monotonically decrease with the increase in the size of the scratchpad memory. This is because the larger the scratchpad, the more memory objects are allocated and the less are the accesses to the slow and energy inefficient main memory. On a closer look, it is observed that the energy and execution values decrease in a stepwise manner. In Figure 4.4(b), the energy consumption values remain the same for 100 bytes and 128 bytes scratchpad. This behavior emerges due to granularity of the SA approach. For uni-processor ARM setup, the SA approach allocates traces, functions and global variables to the scratchpad. Even though the approach is sufficiently fine grained, it is unable to find a different allocation when increasing the scratchpad size from 100 bytes to 128 bytes.

Second, it is observed that for each scratchpad size the normalized energy values are lower than the corresponding execution time values. This implies that the utilization of the scratchpad has more impact on reducing the total energy consumption of the system than on reducing the execution time of the application. This observation is justified because the difference in energy per access to the main memory and the scratchpad is larger than the difference in their access times. Lastly, it is observed that significant energy savings of more than 70% could be achieved by the introduction of a scratchpad in the memory hierarchy of the system.

Comparison of the Scratchpad Allocation Approaches:
Next, we present a comparison of the ILP based optimal non-overlayed scratchpad allocation (SA) approach and the greedy algorithm based fractional scratchpad allocation (Frac. SA) approach. Figures 4.5(a) and 4.5(b) present the comparison of the two approaches for *edge*



(a) Edge Detection



(b) DSP

Fig. 4.5. Energy Comparison of Scratchpad Allocation Approaches

detection and *dsp* benchmarks, respectively. The Frac. SA approach is allowed to allocate one memory object across the boundary of the scratchpad such that it is partially present in the scratchpad space and partially in the main memory space. Therefore, the approach always utilizes the entire scratchpad space to allocate memory objects. This should result in more energy efficient scratchpad allocations for the Frac. SA approach than those for the SA approach. The expected behavior can be observed from Figure 4.5(a), as the energy values for the Frac. SA approach are smaller than or equal to those for the SA approach.

However, as discussed earlier, the Frac. SA approach cannot always determine a better allocation than the SA approach because the energy valence for each element within a memory object is not always a constant. From Figure 4.5(b), we observe that at only three points (100, 200 and 300 bytes) the Frac. SA approach achieves a more energy efficient allocation than that of the SA approach. In contrast, the SA approach achieves energy efficient allocations for all the other scratchpad sizes.

The other important observation is that the energy values for the Frac. SA are always very close to those for the SA approach. A maximum difference of 2% is observed for the *dsp* benchmark at 512 bytes scratchpad sizes. Therefore, if the system architecture permits allocation of a memory object across the boundary of a scratchpad, then the Frac. SA approach having a polynomial time complexity should be used to replace the SA approach.

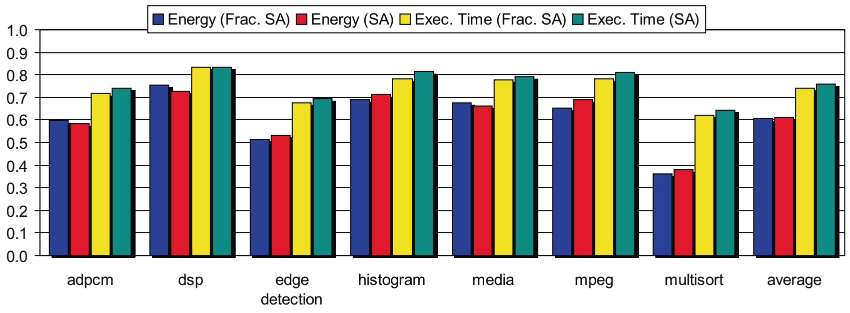


Fig. 4.6. Overall Comparison of the Scratchpad Allocation Approaches

Next, a comparison of the two scratchpad allocation approaches over all benchmarks is presented. Figure 4.6 presents the normalized average total energy consumption and

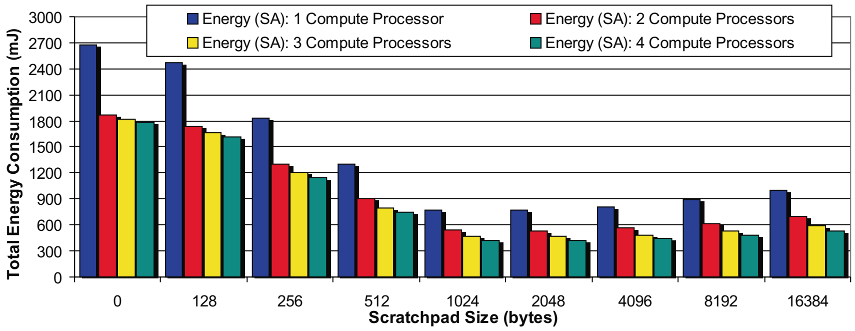


Fig. 4.7. Multi-Process Edge Detection: Energy Consumption for Varying Compute Processors and Scratchpad Sizes (Cycle Latency = 1 Master Cycle)

execution time values for the Frac. SA approach and the SA approach. The average energy value for each benchmark is computed by averaging over all energy values obtained for scratchpad memories of 128, 256, 512 and 1024 bytes. Subsequently, the value is normalized against the energy value obtained for a system without a scratchpad. The same procedure is used to compute the normalized averaged execution time values for the benchmarks.

The reduction in the average energy consumption due to the SA approach varies between 61% for the *multisort* and 28% for the *dsp* benchmark. For the Frac. SA approach, similar reductions in the average energy consumption of the benchmarks are observed. From the figure, we observe that the average energy values for the Frac. SA approach are smaller than those of the SA approach for *edge detection*, *histogram*, *mpeg* and *multisort* benchmarks, whereas the opposite is observed for *adpcm*, *dsp* and *media* benchmarks. The average energy savings across all the benchmarks is about 40% for both the allocation approaches. A similar albeit smaller reduction in the execution times of the benchmarks is also observed. The average reduction in the execution times due to the SA approach varies between 39% for the *multisort* and 18% for the *dsp* benchmark. A reduction of 25% and 24% in the average execution time is observed for the SA and Frac. SA approaches, respectively.

4.6.2 Multi-Processor ARM

In the current subsection, the ILP based non-overlayed scratchpad allocation (SA) approach is evaluated for a multi-process edge detection benchmark. The multi-process benchmark consists of an initiator process, a terminator process and a variable number of compute processes. The benchmark is simulated on the homogenous multi-processor ARM based system such that each process is mapped to a unique ARM processor. The processors in the multi-procesor system are named according to the mapped process. Each processor has its own local scratchpad memory, while all of them access a shared main memory. A detailed description of the benchmark and the multi-processor system can be found in Section 3.2.

Figure 4.7 presents the total energy consumption values for the benchmark when the number of the compute processors and the size of the local scratchpad memory is varied. The SA approach analyzes each process executing on the processor and accordingly allocate

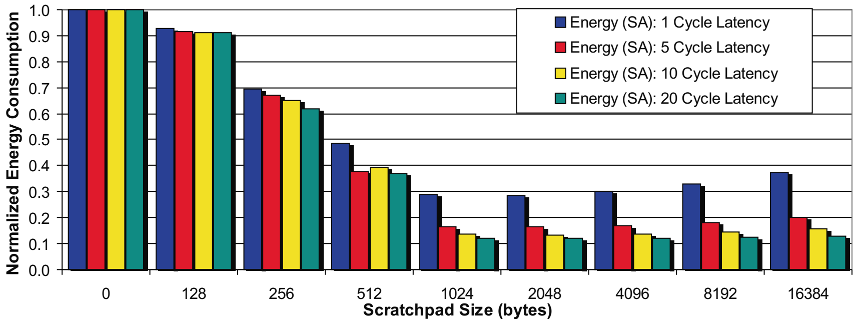


Fig. 4.8. Multi-Process Edge Detection: Normalized Energy Consumption for Varying Memory Access Times (#Compute Processors = 2)

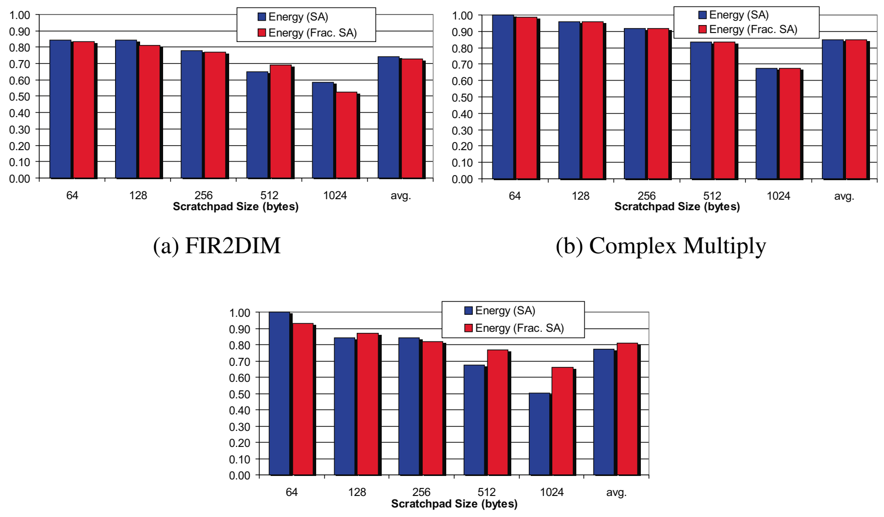
memory objects to its scratchpad memory. The Frac. SA approach could not be used, as the multi-processor ARM based system does not support partial allocation of the scratchpad memories.

A few observations could be made from Figure 4.7. First, the total energy consumption of the application shows a steep decrease when the number of compute processors is increased from one to two. A further increase in the number of compute processors has miniscule impact on reducing the energy consumption of the system. Therefore, we believe that two compute processors are the ideal choice for the current benchmark.

The second observation is that the energy consumption of the system initially decreases with the increase in the size of the scratchpad memories till it reaches the minimum value at 2048 bytes. Thereafter, an increase in the scratchpad size also increases the energy consumption of the system. The reason for the observation is that the per access energy to the scratchpad memories increases exponentially with the increase in their sizes. Therefore, once the most frequently accessed memory objects are allocated onto the scratchpad memory, any further increase in the scratchpad size contributes positively to energy consumption of the system.

Next, we evaluate the impact of the SA approach on the energy consumption when the latency to access the main memory is varied. The scratchpad memory has a zero cycle latency, while cycle latencies of 1, 5, 10 and 20 master clock cycles are assumed for the main memory. Figure 4.8 presents the normalized energy consumption values of the edge detection benchmark executing on a multi-processor system containing two compute processors along with one initiator and one terminator processor. The energy values are normalized against the energy values of the system containing a zero byte scratchpad memory.

It is observed that the normalized energy values are smaller for the systems with a slower main memory. This is because the slower the main memory, the longer the processor has to wait for the requested data and the higher is the energy consumed by the processor. Even though the energy consumed by the memory remains invariant, the total energy consumption being the sum of processor and memory energy consumption increases. The SA approach reduces the number of accesses to the main memory by allocating memory objects to the scratchpad. The slower the main memory, the larger is the difference in their access times



(a) FIR2DIM (b) Complex Multiply

(c) FIR

Fig. 4.9. Normalized Energy Comparison of Scratchpad Allocation Approaches

and therefore, the larger is the reduction in the total energy consumption of the system. The SA approach achieves a maximum total energy reduction of 71%, 84%, 87% and 88% for main memories with 1, 5, 10 and 20 master clock latencies, respectively.

4.6.3 M5 DSP

The M5 DSP in its default configuration contains a large onchip group memory to hold the data variables. The energy dissipation of the data memory hierarchy is improved by inserting a small and energy efficient L1 scratchpad or group memory. The approaches presented in this chapter and in the subsequent chapters reduce the energy dissipation through the improved utilization of the L1 scratchpad memory. An accurate energy model from UMC is used to compute the energy consumed by the data memory hierarchy. However, due to copyright reasons, we are forbidden from reporting exact energy values. Therefore, we will report normalized energy values for the data memory subsystem of the M5 DSP.

Figure 4.9 presents the comparison of normalized energy consumption values when the L1 scratchpad is allocated using the SA approach or the Frac. SA approach. The unit valued baseline represents the energy consumed by the default data memory subsytem of M5 DSP. From the figure, we make a few observations. First, the normalized energy values for the SA approach at 64 bytes (*cf.* Figures 4.9(b) and 4.9(c)) is the unit value because the scratchpad space is too small to allocate any memory object. In contrast, the normalized energy values for the Frac. SA allocation is smaller than the unit value as it allocates partial memory objects onto the scratchpad. For *fir2dim* benchmark, 64 bytes are already sufficient to achieve energy savings with both the approaches.

Second, as also observed in the previous figures, energy values for both the approaches decrease monotonically with increase in the scratchpad size. Third, for the *complex* *multiply* benchmark, the energy values for both approaches are equal for all scratchpad sizes between 128 and 1024 bytes. This is because for all these sizes, the SA approach selects the same memory objects for allocation as the Frac. SA approach and both the approaches entirely utilize the scratchpad space. Last, the energy reductions for the SA approach at 1024 bytes scratchpad are 42%, 33% and 50% for *fir2dim*, *complex multiply* and *fir* benchmarks, respectively. Similar energy reductions of 48%, 33% and 34% are observed for the same benchmarks using the Frac. SA approach.

4.7 Summary

In this chapter, non-overlayed scratchpad allocation approaches to minimize the energy consumption of the system were proposed. The problem of non-overlayed scratchpad allocation was shown to be the NP-complete knapsack problem. An ILP based scratchpad allocation approach and a greedy algorithm based fractional scratchpad allocation approach were proposed. The fractional scratchpad allocation approach allocates memory objects such that a maximum of one memory object can be partially present in the scratchpad and partially in the main memory. It was demonstrated that under certain conditions the scratchpad allocations determined by the fractional approach are more energy efficient than those computed by the ILP based approach.

The proposed approaches are evaluated for uni-processor ARM, multi-processor ARM and M5 DSP based systems. It is reported that the approaches achieve significant reductions of more than 70% in the total energy consumption of the system through the utilization of the scratchpad memory. It is also observed that the fractional allocation approach achieves very close to optimal allocations.

Some of the results presented in this chapter are published in [122, 123].

5

Non-Overlayed Scratchpad Allocation Approaches for Main / Scratchpad + Cache Memory Hierarchy

The previous chapter presented scratchpad allocation approaches for systems containing scratchpad memories as the only L1 memories in the memory hierarchy. However, a large number of high-end embedded microprocessors contain instruction and data caches as well as scratchpad memories as the L1 memories. The application of the previously presented non-overlayed scratchpad allocation (SA) approach for this memory hierarchy leads to unpredictable results as it does not model the behavior of caches. In this chapter, approaches to perform optimal and near-optimal non-overlayed cache aware allocation of the scratchpad memory are discussed. The approaches model the interaction of memory objects accessed through the cache memory and then perform non-overlayed allocation of memory objects onto the scratchpad memory.

The rest of this chapter is organized as follows: Section 5.1 gives a brief introduction to the proposed allocation approaches and a survey of the related work is presented in Section 5.2. Section 5.3 demonstrates the shortcomings of two previous approaches with the help of a motivating example. Section 5.4 describes the memory objects, the conflict graph based cache model, the energy model and formally defines the cache aware scratchpad allocation (CASA) problem. Section 5.5 presents the description of the proposed optimal and near-optimal approaches to solve the problem. In Section 5.6, the experimental results for uni-processor and multi-processor ARM based systems are presented. Finally, the chapter ends with a brief summary.

5.1 Introduction

Caches and scratchpad memories represent two contrasting memory architectures. Caches, under the control of hardware logic, automatically exploit temporal and spatial locality present in the program. Therefore, they need two additional hardware components, *viz.* the tag memory and the address comparison logic, beside the data memory for their autonomous operation [137]. The tag memory is required for storing information regarding the valid addresses, while the comparison logic is used to distinguish between cache hits and misses. These additional components dissipate a significant amount of energy per access to the cache irrespective whether the access translates to a hit or a miss.

On the other end of the spectrum are the scratchpad memories, consisting of just data memory and address decoding circuitry. Due to the absence of the tag memory and

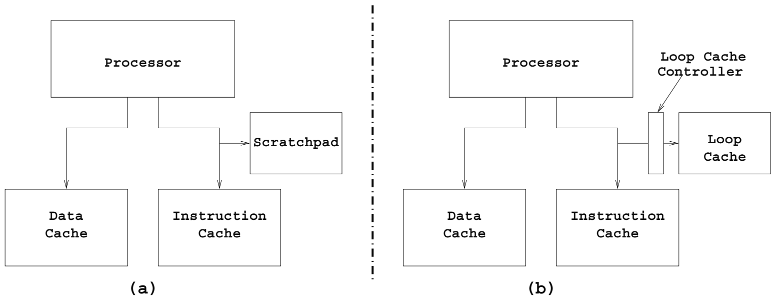


Fig. 5.1. System Architecture: (a) Scratchpad (b) Loop Cache

the comparison hardware, scratchpad memories require considerably less energy than a cache [16]. In addition, scratchpad memories require less onchip area. However unlike caches, scratchpads require complex program analysis and explicit support from the compiler. In order to strike a balance between these contrasting approaches, most of the high-end embedded microprocessors (*e.g.* ARM11 [9], ColdFire MCF5 [92]) include both onchip caches and scratchpads.

In this chapter, we first assume a memory hierarchy as shown in Figure 5.1(a) and utilize the scratchpad for storing instructions. The decision to store only instructions is motivated by the following observations:

(a) For RISC architectures, the number of instruction fetches is much higher than the number of data accesses.
(b) Instruction fetches demonstrate a high locality, hence a small scratchpad can be used to serve a high number of instruction accesses.
(c) High-end microprocessors (*e.g.* ARM11 [9]) feature separate dedicated scratchpad memories for data and instructions.

The previous scratchpad allocation approach assigns access frequency based benefits to memory objects without considering their interaction with other memory objects in the cache memory. Therefore, it fails to produce optimal results when applied to the current memory architecture. In Section 5.3, we demonstrate with the aid of an example that the previous approach may even lead to the problem of *cache thrashing* [52]. This argument is further strengthened by observing a similar behavior, in Section 5.6 for two real-life benchmarks.

In this chapter, the interaction of memory objects accessed through the cache memory is modeled as a conflict graph. The nodes of the conflict graph represent the memory objects present in the application, while the edges represent the conflict-miss relationship between two memory objects. The misses in a cache memory can be classified into *cold misses*, *capacity misses* and *conflict misses* [52]. The conflict graph is used to capture only the conflict miss relationship among the memory objects.

The goal of the cache aware scratchpad allocation (CASA) problem is to minimize the energy consumption of the application through the non-overlayed allocation of memory objects onto the scratchpad memory while considering their conflict relationships in the cache memory. As shown later, the problem of finding the best set of objects to be allocated on the scratchpad memory can be formulated as a non-linear optimization problem.

Under simplifying conditions, it can be reduced to either a Weighted Max-Cut [43] or a Knapsack [43] problem, both of which are known to be NP-complete problems. An optimal solution is obtained by formulating the CASA problem as an *integer linear programming* (ILP) problem. For all our experiments, an insignificant compute time was required to solve the ILP formulation. However, the problem with ILP formulations is that we do not know which formulation will require an exponential compute time. Therefore, we also propose a heuristic based approach with a polynomial time complexity to obtain a near-optimum solution.

The chapter also presents a comparison of the scratchpads with preloaded loop caches [48], as the utilization of the scratchpad in the current setup (see Figure 5.1) is similar to a loop cache. Preloaded loop caches are architecturally more complex than scratchpads, but are less flexible as they can be preloaded with only a limited number of loops/functions. The goal of this study is to demonstrate that by using the proposed approaches, scratchpads can outperform their complex counterparts.

5.2 Related Work

The work related to the optimization of the instruction memory hierarchy can be classified into the three broad categories as shown below:

(a) Code reorganization approaches
(b) Hardware controlled instruction buffers
(c) Software controlled instruction buffers

Code Reorganization Approaches:

Application code placement approaches change the layout of the program code to reduce the number of instruction cache misses. This in turn improves the CPI (cycles per instruction) and the performance of the processor. The approaches perform the code layout transformations at different levels of granularity, from basic blocks [101, 119] to procedures [46].

The approach by Pettis et al. [101] is one of the earliest application code reorganization techniques to improve the instruction cache performance. It groups frequently executed basic blocks to create *traces* and then reorganizes traces within the function boundaries according to the "closest is the best" strategy. Therefore, the most frequently executed traces are placed close to each other at the top of the function, whereas the least frequently executed or never executed traces are placed at the bottom of the function.

Tomiyama et al. [119] present one of the most complete profile-guided basic block reordering techniques. Similar to the previous approach, the proposed approach generates traces using the execution profile of the application and formulates the code reorganization problem as an *integer linear programming* (ILP) problem with the objective to minimize the number of instruction cache misses. One of the advantages of the approach [119] is that it generates a globally optimized placement of the traces, as they are allowed to be placed even across the function boundaries. However, as noted by the authors, the ILP solver requires a substantial time to compute the optimal solution.

The authors in [46] propose a weighted graph coloring based approach to map procedures to cache lines to reduce the number of instruction cache misses. The approach constructs the call graph of the application and assigns execution frequency edge weights. It then accesses the edges in the descending order of their weights and assigns colors or cache lines to the nodes. The approach only eliminates the first-generation cache conflicts, which are the conflicts between a node and its parents or the node and its children. This may be a serious limitation, as cache conflicts between two unrelated nodes of the call graph occur frequently in real-life benchmarks.

Hardware Controlled Instruction Buffers:

Numerous hardware controlled instruction buffers to reduce the energy dissipation of the instruction memory hierarchy have been proposed in the literature. Here, we discuss two representative instruction buffers, *viz.* filter caches and dynamically loaded loop caches. A filter cache [68] is a small direct-mapped cache introduced between the processor and the L1 instruction cache. On an instruction fetch, first the filter cache is accessed. If the access results in a hit, the value is returned immediately, otherwise the L1 instruction cache is accessed resulting in one cycle miss penalty. A 256 byte filter cache [68] was shown to have a hit rate of 60-85% for Mediabench [87] benchmarks. Hence, it achieves more than 50% reduction in the energy dissipation at the expense of a 20% degraded performance of the benchmarks.

Authors in [74] proposed the use of a small memory called *dynamically loaded loop caches* to buffer the loop body. The dynamically loaded loop cache, as shown in Figure 5.1(b), is present at the same horizontal level as the L1 instruction cache. Therefore, it can be accessed in parallel to the instruction cache. The loop cache does not cache all the instruction fetches made by the processor but caches only those instructions which belong to a loop. The loop cache controller identifies loops at runtime whenever a short branch backwards is taken, which happens at the end of the first iteration of the loop. During the second iteration, the controller copies the loop body instructions to the loop cache. From the third iteration onwards, the instructions are fetched from the energy-efficient loop cache and not from the instruction cache.

The limitation of loop caches is that they cache only those loops which do not contain control flow changing (*e.g.* branch) instructions in the loop body. The approach improves upon the filter cache as the performance of the application is not degraded. This is because the loop cache is filled non-intrusively and accessed only when a hit is guaranteed. Authors in [74] report a 38% reduction in energy consumption for a 64 byte dynamically loaded loop cache.

Software Controlled Instruction Buffers:

The software controlled instruction buffers refer to the instruction memories which require either compile-time or runtime support from the software. The typical examples of these buffers are preloaded loop caches [48] and scratchpad memories. Preloaded loop caches (*cf.* Figure 5.1(b)) as well as scratchpad memories require both compile-time and runtime support from the software.

The *preloaded loop cache* was proposed by Ross et al. [48] to overcome the limitations of the dynamically loaded loop cache. A preloaded loop cache can be statically loaded with pre-identified code segments before the start of the application. Start and end address of the code segments are stored in the controller, which on every instruction fetch determines whether to access the loop cache or the instruction cache. Therefore, unlike their dynamically loaded variants, preloaded loop caches can be loaded with complex loops as well as functions.

In order to keep the energy consumed by the controller within bounds, only a small number of memory objects (typically 2-6) can be preloaded. The property of being able to store only a fixed number of memory objects in the preloaded loop cache is too restrictive for large applications with several hot spots. Moreover, code segments are greedily chosen according to their execution time valence (execution time per unit size). As described below and also as shown in Subsection 5.4.4, execution time or frequency based models do not accurately model energy dissipation of memory objects in a cache based memory hierarchy.

Most of the research on scratchpad allocation [14, 23, 66, 97] has focused on allocating data elements onto the scratchpad. Authors in [14, 97] statically allocated global/local variables on the scratchpad, whereas authors in [23, 66] looked at the possibility of dynamically copying the data elements from the main memory onto the scratchpad.

Approaches [103, 113, 115, 128] are among the few approaches that allocate code segments onto the scratchpad. The approaches [103, 113] perform overlay based allocation of the scratchpad memory for only code segments, while the approach [128] performs scratchpad memory overlay for both code segments and data variables. A detailed discussion of the scratchpad overlay approaches is presented in Chapter 6. The approach in [115] and the SA approach presented in Chapter 4 perform non-overlayed allocation of code segments and data variables (memory objects) onto the scratchpad. Both the approaches assume a memory hierarchy consisting of only scratchpad and main memory. Profit values are assigned to the code segments and data variables according to their execution and access counts, respectively. Then, the problem of finding the best of memory objects to be moved to the scratchpad is formulated as an integer linear programming (ILP) problem. The problem is solved optimally using a commercial ILP solver [32].

Even though the SA approach is sufficiently accurate for its memory hierarchy, it is not suitable for the current setup, which includes caches in the memory hierarchy. The assumption that execution counts are sufficient to represent energy consumption by a memory object fails in the presence of a cache, where execution counts have to be decomposed into cache hits and misses. The energy consumption of a cache miss is significantly larger than that of a cache hit. Consequently, two memory objects can have the same execution counts, yet have substantially different cache hit/miss ratio and hence the energy consumption. Additionally, the previous approach does not maintain the conflict relationships between memory objects during the code placement step. Memory objects are moved instead of copying them from the main memory to the scratchpad. As a result, the program layout is changed, which may cause a completely different cache access pattern and thus lead to erratic results.

In the wake of the above discussion we enumerate the contributions of the work presented in this chapter.

(a) It for the first time studies the combined effect of a scratchpad and an I-cache on the memory system's energy consumption.
(b) It demonstrates the inefficiencies of previous approaches when applied to the present architecture and stresses the need for an advanced allocation approach.
(c) It demonstrates that scratchpad memories along with an allocation algorithm can replace complex loop caches.

Please note that in the rest of this chapter, the energy consumption refers to the energy consumption of the instruction memory subsystem and loop cache refers to preloaded loop cache. In the following section, we present a motivating example demonstrating the inadequacies of the previous approaches and the efficacy of the proposed approach.

5.3 Motivating Example

We execute an example application on a system whose instruction memory hierarchy in the base configuration consists of an instruction cache and a main memory. Later, we insert a scratchpad or a preloaded loop cache and study the behavior of the instruction memory hierarchy. The scratchpad is allocated using the non-overlayed SA approach as well as the CASA approach, while the preloaded loop cache is allocated using Ross [48] approach. For all the above scenarios, we assume that we are given a weighted control flow graph (CFG), a layout of CFG nodes in the main memory and an execution trace of the example application. Based on the given information, we analyze the behavior of the instruction memory hierarchy and compute its energy consumption.

We would like to make a couple of simplifying assumptions before presenting the motivating examples. First, the size of each basic block as well as that of each cache line is assumed to be 1 word. Second, that moving a basic block within the memory hierarchy does not modify the size of any other basic block. We would like to state that these assumptions are added for the sake of clarity and that they apply only to the motivating examples. Next, we describe the motivating example for the base configuration of the instruction memory subsystem.

5.3.1 Base Configuration

Figure 5.2 shows a weighted control flow graph (CFG), main memory layout, an instruction cache and an execution trace for an example application. The nodes of the CFG represent the corresponding basic blocks, while the edges represent the possible flow of control during

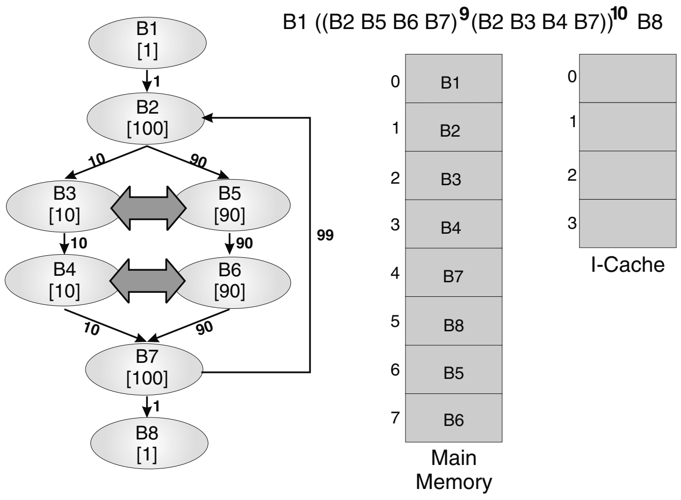


Fig. 5.2. Example: Base Configuration

Memory	Access Type	Symbol	Energy
Cache	Hit	E_{Cache_hit}	1.0
Cache	Miss	E_{Cache_miss}	10.0
SPM	Hit	E_{SP_hit}	0.5
Loop Cache	Hit	E_{LC_hit}	0.5

Table 5.1. Energy Values for Different Memories

BB	I-Cache Accesses	I-Cache Misses	Energy
BB2	100	0	100
BB3	10	10	100
BB4	10	10	100
BB5	90	10	180
BB6	90	10	180
BB7	100	0	100
Total	400	40	760

Table 5.2. Energy Values for Base Configuration

the execution of the application. The nodes and edges of the CFG are weighted according to the corresponding execution frequencies during a typical execution of the application. The gray bar in the middle of the figure presents the layout of the application code in the main memory as well as the absolute address of each basic block. The instruction cache, in this setup, is a direct mapped cache of 4 words in size. The execution trace, shown at the top of the figure, represents an example execution of the application at the granularity of basic blocks.

The execution trace reveals that the right arm (*i.e.* B2, B5, B6, B7) of the loop is executed 9 times before the left arm (*i.e.* B2, B3, B4, B7) is executed once. The executions of the right arm followed by that of the left arm is repeated for 10 times before the end of the execution. According to the main memory layout and the modulo addressing (*i.e.* $2 \; mod \; 4 \equiv 6 \; mod \; 4 \equiv 2$) employed by caches, we observe that nodes B3 and B5 will share the same cache line, so does nodes B4 and B6.

During the program execution, nodes B3 and B4 will constantly replace nodes B5 and B6 in the cache leading to an aggregate 40 cache misses. Table 5.1 presents the assumed energy values for a cache hit and a cache miss as well as the energy per access values for a scratchpad memory and a loop cache. Table 5.2 summarizes the access and the miss count as well as the energy consumption of each basic block. The energy values and the access and miss counts are used to compute the total energy consumption of the application which amounts to 760 units.

5.3.2 Non-Overlayed Scratchpad Allocation Approach

Now, we introduce a scratchpad (*cf.* Figure 5.3) into the instruction memory hierarchy of the system. The scratchpad is allocated using the previously presented non-overlayed scratchpad allocation (SA) approach which that the energy consumption of a basic block depends solely upon its execution frequency. Therefore, in order to minimize the energy consumption of the system the approach selects the memory objects with the highest execution frequencies to be moved onto the scratchpad.

For the unit sized scratchpad, the approach has to choose between node B2 or B7, as both nodes have the highest execution frequencies among all the nodes. Figure 5.3 displays the modified main memory layout when node B7 is moved to the scratchpad memory. From the figure, we observe that in the modified memory layout, nodes B2 and B5 share the same cache line. Therefore, during the execution, nodes B2 and B3 will constantly replace nodes B5 and B6 in the cache, respectively. This results in a total of 200 conflict cache misses,

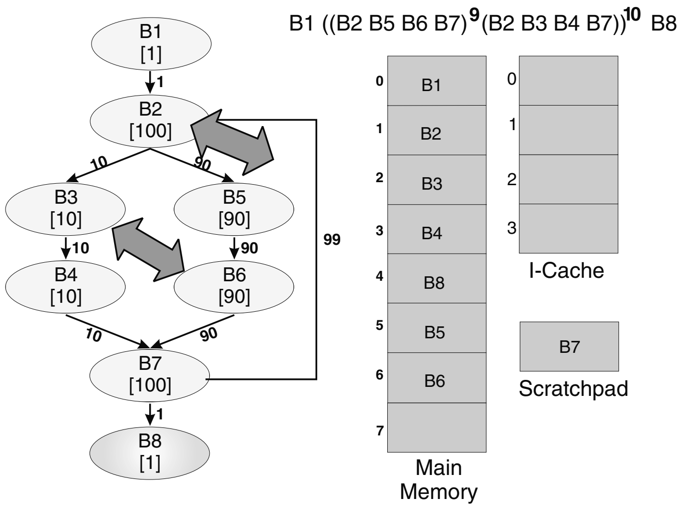


Fig. 5.3. Example: Non-Overlayed Scratchpad Allocation Approach

BB	I-Cache Accesses	I-Cache Misses	SPM Accesses	Energy
BB2	100	90	0	910
BB3	10	10	0	100
BB4	10	0	0	10
BB5	90	90	0	900
BB6	90	10	0	180
BB7	0	0	100	50
Total	300	200	100	2150

Table 5.3. Energy Values for Scratchpad (1 Word) Based System

BB	I-Cache Accesses	I-Cache Misses	SPM Accesses	Energy
BB2	0	0	100	50
BB3	10	10	0	100
BB4	10	10	0	100
BB5	90	10	0	180
BB6	90	10	0	180
BB7	0	0	100	50
Total	200	40	200	660

Table 5.4. Energy Values for Scratchpad (2 Words) Based System

up from 40 in the base configuration. Consequently, the energy consumption (*cf.* Table 5.3) for the scratchpad based instruction memory hierarchy rises steeply to 2150 units, which is about 3 times larger than that for the base configuration (760 units) of the system.

The reason for the exaggerated energy consumption is that the allocation approach did not consider the instruction cache present in the hierarchy. The current situation in which frequently executed basic blocks (B2 and B5) recurrently replace each other in the cache is known as *cache thrashing*. For a 2 word sized scratchpad, the allocation approach maps nodes B2 and B7 to the scratchpad. Though the number of scratchpad access doubles in this case, the number of cache misses remains constant at 40. The energy consumption (*cf.* Table 5.4) for the scratchpad based memory hierarchy is 660 units.

5.3.3 Loop Cache Approach

Next, we assume that a preloaded loop cache (*cf.* Figure 5.4) is present in the instruction memory hierarchy of the system. The loop cache is restricted to accommodate only one loop or function. According to the loop cache allocation strategy [48], only loops and functions

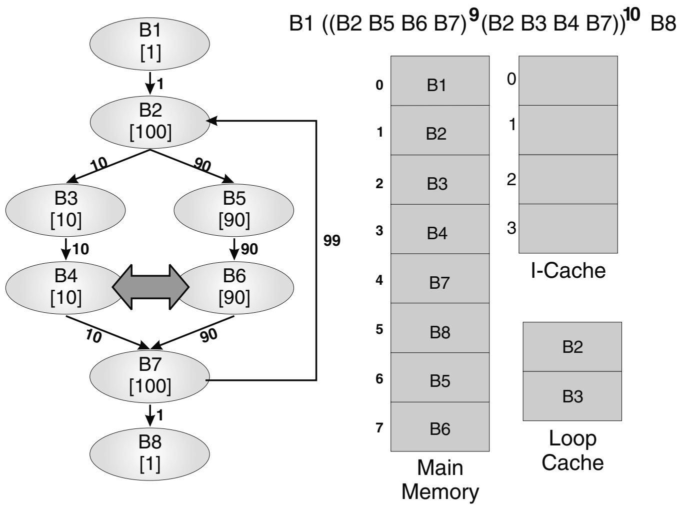


Fig. 5.4. Example: Loop Cache Approach

BB	I-Cache Accesses	I-Cache Misses	Loop Cache Accesses	Energy
BB2	0	0	100	50
BB3	10	10	0	100
BB4	10	10	0	100
BB5	90	10	0	180
BB6	90	10	0	180
BB7	100	0	0	100
Total	300	40	100	710

Table 5.5. Energy Values for Loop Cache (1 Word) Based System

BB	I-Cache Accesses	I-Cache Misses	Loop Cache Accesses	Energy
BB2	0	0	100	50
BB3	0	0	10	5
BB4	10	10	0	100
BB5	90	0	0	90
BB6	90	10	0	180
BB7	100	0	0	100
Total	290	20	110	525

Table 5.6. Energy Values for Loop Cache (2 Words) Based System

can be fully or partially allocated onto the loop cache. Moreover, allocation of loops or functions can only begin from the starting basic block and can only be extended to the next contiguous basic block in the memory. Finally, loops and functions which are to be preloaded are copied instead of being moved to the loop cache. Thus, the program memory layout remains invariant upon loop cache allocation.

The unit word sized loop cache is preloaded with node B2. The energy consumption of the memory hierarchy (*cf.* Table 5.5) reduces to 710 units, while the number of cache misses remain the same. In the next scenario, when a loop cache of size 2 words is present in the system, nodes B2 and B3 are allocated onto the loop cache. Consequently, the number of cache misses and the energy consumption reduces to 20 and 525 units, respectively.

5.3.4 Cache Aware Scratchpad Allocation Approach

We will now demonstrate the effectiveness of the proposed approach for an instruction memory hierarchy containing a scratchpad and an instruction cache. The proposed approach uses a precise energy model based on cache hits and misses. Therefore, it can identify the most energy consuming basic blocks for scratchpad allocation. Moreover, it keeps the

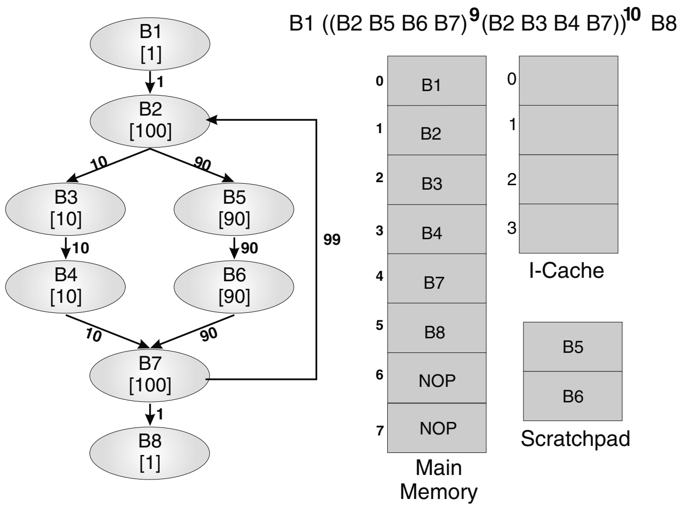


Fig. 5.5. Example: Cache Aware Scratchpad Allocation Approach

BB	I-Cache Accesses	I-Cache Misses	Scratchpad Accesses	Energy
BB2	100	0	0	100
BB3	10	0	0	10
BB4	10	10	0	100
BB5	0	0	90	45
BB6	90	10	0	180
BB7	100	0	0	100
Total	310	20	90	535

Table 5.7. Energy Values for Scratchpad (1 Word) Based System

BB	I-Cache Accesses	I-Cache Misses	Scratchpad Accesses	Energy
BB2	100	0	0	100
BB3	10	0	0	10
BB4	10	0	0	10
BB5	0	0	90	45
BB6	0	0	90	45
BB7	100	0	0	100
Total	220	0	180	310

Table 5.8. Energy Values for Scratchpad (2 Words) Based System

program memory layout invariant by copying basic blocks on the scratchpad and replacing them in the main memory by NOP instructions.

For a unit word sized scratchpad, the proposed approach chooses the most energy consuming node (B6 or B5), refer Table 5.2, to be allocated onto the scratchpad. Table 5.7 demonstrates that the allocation of node B6 to the scratchpad reduces the number of caches misses by half and also reduces the energy consumption to 535 units. For 2 word sized scratchpad, both the nodes B5 and B6 are copied to the scratchpad. Consequently, the proposed approach eliminates all cache misses and minimizes the energy consumption. The energy consumption of the system is 310 units, the minimum compared with all the previously presented approaches.

5.4 Problem Formulation and Analysis

The cache aware scratchpad allocation approach computes the energy consumption of the code segments using an accurate cache conscious energy model. It then maps code segments to the non-cacheable scratchpad region such that the aggregate energy consumption of the instruction memory subsystem is minimized. The application code layout in the main memory is kept invariant by replacing the code segments mapped to the scratchpad memory by NOP instructions. Therefore, it can be guaranteed that the number of cache misses will not increase after the scratchpad allocation.

In the remainder of this section, we first describe the underlying architecture for which the optimization technique is developed, followed by the description of memory objects. The interaction of memory objects within the cache is represented using a conflict graph (*cf.* Subsection 5.4.3), which forms the basis of the proposed energy model (*cf.* Subsection 5.4.4) and the allocation approaches.

5.4.1 Architecture

For the current research work, we assume a Harvard architecture (*cf.* Figure 5.1(a)) with the scratchpad at the same horizontal level as the L1 instruction cache. The scratchpad is mapped to a region in the processor's address space and acts as an alternative non-cacheable location for fetching instructions. Instruction fetches which do not access the scratchpad are accessed through the cache. Figure 5.1(b) demonstrates, the system architecture containing a loop cache. It can be observed from Figure 5.1 that both the architectures are quite similar to each other.

5.4.2 Memory Objects

A memory object is the smallest granularity object in the application through which the memory optimizations are performed. It can be a code fragment (*e.g.* function) or a data variable (*e.g.* global variable). For the uni-processor ARM based system, the cache aware scratchpad allocation (CASA) approach assumes that the memory objects consist of traces and functions. Due to the implementation constraints, for multi-processor ARM based system, the memory objects are restricted to contain only functions present in the application. Table 5.9 summarizes the memory objects for the CASA approach for different system architectures.

Traces, like functions, are an atomic unit of instructions which can be placed anywhere in the memory without modifying the other memory objects. Moreover, traces allow fine grained scratchpad allocation as they are smaller in size than functions and efficiently envelop the hot-spots or tight loops in the application. The formal definition of a trace can be found in Subsection 3.1.3 on page 23. Figure 5.6 illustrates the CFG of the example (*cf.* Figure 5.2) at the granularity of traces which were restricted to a maximum size of 2 words. Figure 5.6 also presents the main memory layout and the execution trace of the

Memory Optimization	System Architecture	Memory Objects	Explanation
Non-overlayed Scratchpad Allocation for MM/SPM + Cache Hierarchy (Chapter 5)	Uni-processor ARM	$MO \subseteq T \cup F$	traces and functions
	Multi-processor ARM	$MO \subseteq F$	functions

Table 5.9. Memory Objects for Non-Overlayed Scratchpad Allocation Approach

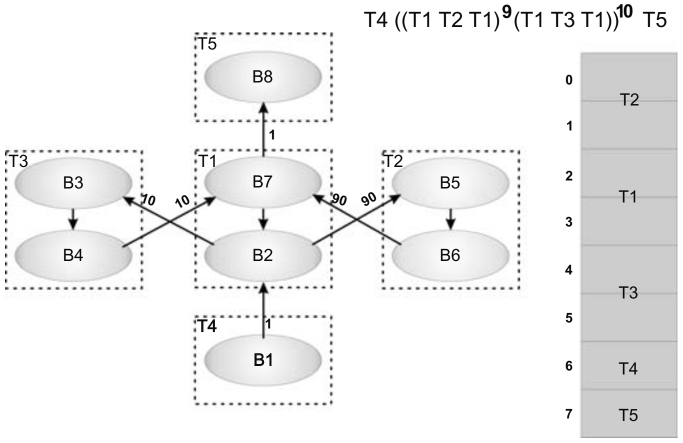


Fig. 5.6. Example: Application after Trace Generation Step

example application at the granularity of traces. In the following subsection, we model the interaction of memory objects within the cache memory by a conflict graph.

5.4.3 Cache Model (Conflict Graph)

The cache maps an instruction to a cache line according to the following function:

$$Map(address) = address \textbf{ mod } \frac{CacheSize}{Associativity * WordsPerLine} \tag{5.1}$$

where $CacheSize$ and $Associativity$ represent the size and the associativity of the cache, respectively. The variable $WordsPerLine$ refers to the size of the cache line in terms of 4 bytes or words. The above equation remains valid for all caches irrespective of their size, associativity or replacement policy. A memory object is mapped to cache line(s) depending upon its start address and size. Two memory objects potentially cause conflicts in the cache if they are mapped to at least one common cache line. This relationship can be represented by a conflict graph G (*cf.* Figure 5.7), which is defined as follows:

Definition 5.1 (Conflict Graph). *The* Conflict Graph $G(N,E)$ *is an edge and node weighted directed graph with node set* $N = \{n_1, \ldots, n_n\}$ *and is defined as follows:*

(a) $n_i \in N$ *node* n_i *corresponds to the memory object* $mo_i \in MO$.

(b) $e_{ij} \in E$ *directed edge* e_{ij} *from node* n_i *to node* n_j *is present if a cache line belonging to memory object* mo_i *is replaced by memory object* mo_j *due to the cache replacement policy.*

(c) $w(n_i)$ *weight of the node* $n_i \in N$ *represents the total number of instruction fetches within memory object* mo_i.

(d) $w(e_{ij})$ *weight of the edge* $e_{ij} \in E$ *represents the number of cache lines that needs to be fetched if a cache miss of* mo_i *occurs due to* mo_j.

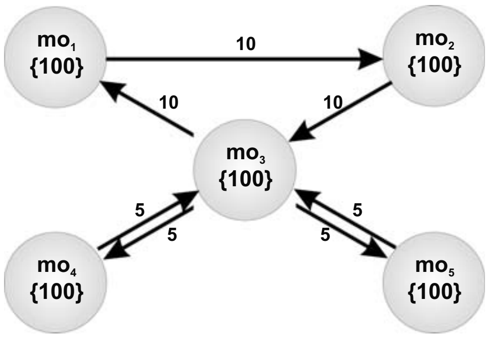


Fig. 5.7. Conflict Graph

In other words, an edge $e_{ij} \in E$ from node n_i to node n_j is present in the conflict graph G if there occurs a cache miss of memory object mo_i is caused by memory object mo_j. A conflict graph is built using both static and dynamic analysis of a program. A node is created for every memory object identified in the program code. The address range of every memory object is analyzed. For all pairs of memory objects which share a common cache line, two directed edges are created, connecting the corresponding nodes in the conflict graph. The weight of each node and each edge in the conflict graph is initialized to zero.

Dynamic profiling of the application is performed to compute the real weight of each node and each edge. The total number of instruction fetches of a memory object mo_i is attributed as the weight of the corresponding node n_i. The number of conflict misses of memory object mo_i caused due to memory object mo_j is attributed as the weight of the directed edge e_{ij} from node n_i to node n_j. Finally, the graph is pruned to remove nodes and edges with zero weights. In order to minimize the influence of the chosen input data set on the conflict graph, average values generated by using several distinct input vectors can be used.

The conflict graph as shown in Figure 5.7 is a directed graph because the conflict relationship is *antisymmetric*. The conflict graph has the advantage that it can precisely model a wide range of cache memories. Any cache with a fixed set of parameters (*e.g.* associativity, size, replacement policy etc.) can be represented using a conflict graph. This is because of the fact that all caches follow Equation 5.1 to assign memory objects to cache lines and that weights are attributed to the edges based on dynamic profiling of the application. Therefore, each edge correctly correlates a cache miss with the two conflicting memory objects. The conflict graph G and the energy values of the cache memory are used to compute the energy consumption of a memory object according to the energy model presented in the following subsection.

5.4.4 Energy Model

As described in Chapter 3, the energy function $E(inst, imem, dmem)$, shown below, returns the energy dissipated during the execution of instruction $inst$ fetched from instruction memory $imem$ and possibly accessing data memory $dmem$.

$$E(inst, imem, dmem) = E_{if}(imem) + E_{ex}(inst) + E_{da}(dmem) \tag{5.2}$$

In this chapter, the focus of the optimization is to reduce the energy consumed by the instruction memory hierarchy. Therefore, the energy dissipated by the memory hierarchy during the execution of an instruction $inst$ is reduced to the following:

$$E(inst, imem, dmem) = E_{if}(imem) \tag{5.3}$$

The energy consumed by a memory object mo_i in the considered memory hierarchy configuration can be presented as follows:

$$E(mo_i) = \begin{cases} E(mo_i, SPM) & \text{if MO } mo_i \text{ is present on the scratchpad} \\ E(mo_i, Cache) & \text{otherwise} \end{cases} \tag{5.4}$$

where the energy dissipated by a memory object fetched through the cache memory $E(mo_i, Cache)$ is computed as shown below:

$$E(mo_i, Cache) = Hit(mo_i) * E_{Cache\_hit} + Miss(mo_i) * E_{Cache\_miss} \tag{5.5}$$

where functions $Hit(mo_i)$ and $Miss(mo_i)$ return the number of cache hits and misses, respectively while fetching the instructions of memory object mo_i. E_{Cache_hit} is the energy of a read hit and E_{Cache_miss} is the energy of a read miss in the cache.

$$E_{Cache\_hit} = E_{if}(Cache) \; = \; E_{read}(Cache) \tag{5.6}$$

$$\begin{aligned} E_{Cache\_miss} &= 2 * E_{read}(Cache) \; + \\ &\quad linesize(Cache) * (E_{read}(MM) + E_{write}(Cache)) \end{aligned} \tag{5.7}$$

where $E_{read}(Cache)$ and $E_{write}(Cache)$ is the energy consumed by the cache memory for a read or a write access, respectively. These values can be found in the data sheets available from the memory vendors. A read hit translates to a single read access to the cache. Therefore, the energy of a read hit E_{Cache_hit}, as shown in Equation 5.6, is equal to that of a read access $E_{read}(Cache)$ to the cache.

In contrast, a read miss results in a read access to the cache, followed by a series of write accesses to the cache to refill the cache line and finally another read access to the cache. The first read access determines that the current read access results in a miss, whereas the second read access represent the access required to send the data requested by the processor once it has been brought into the cache. A cache line refill requires a series of read accesses to the main memory and write access to cache memory, respectively, to read and write the requested data. The energy consumed due to a cache miss is presented in Equation 5.7.

The total number of cache misses to a memory object mo_i can be decomposed as shown below:

$$Miss(mo_i) = \sum_{mo_j \in N(mo_i)} Miss(mo_i, mo_j) \quad \text{with} \tag{5.8}$$

$$N(mo_i) = \{mo_j \; : \; \forall j \;\; n_j \in N \; \& \; e_{ij} \in E\}$$

where $Miss(mo_i, mo_j)$ denotes the number of cache misses of memory object mo_i caused due to the conflicts with memory object mo_j. The neighbor set of memory object mo_i is represented as $N(mo_i)$ in the above equation. We know that the sum of the number of hits and misses of a memory object mo_i is equal to the number of instruction fetches within the memory object which is also represented as the weight $w(n_i)$ of the corresponding node n_i in the conflict graph.

$$w(n_i) = Hit(mo_i) + Miss(mo_i) \tag{5.9}$$

For a given input data set, the number of instruction fetches within a memory object mo_i is a constant and is independent of the memory hierarchy. Substituting the terms $Miss(mo_i)$ from Equation 5.8 and $Hit(mo_i)$ from Equation 5.9 in Equation 5.5 and rearranging derives the following equation:

$$\begin{aligned} E(mo_i, Cache) = {} & w(n_i) * E_{Cache\_hit} + \\ & \sum_{mo_j \in N(mo_i)} Miss(mo_i, mo_j) * (E_{Cache\_miss} - E_{Cache\_hit}) \end{aligned} \tag{5.10}$$

The first term in the above equation is a constant while the second term, which is variable, depends on the overall program code layout and the memory hierarchy. We would like to point out that the approach [48] only considered just the constant term in its energy model and thus, could not optimize for the overall memory energy consumption. The energy consumed by a memory object mo_i when accessed from the scratchpad is shown in the following energy equation:

$$E(mo_i, SPM) = w(n_i) * E_{SPM\_hit} = w(n_i) * E_{if}(SPM) \tag{5.11}$$

where $E_{SPM\_hit} = E_{if}(SPM)$ is the energy for performing an instruction fetch from the scratchpad memory.

5.4.5 Problem Formulation

Problem 5.2 (Cache Aware Scratchpad Allocation (CASA)). Given the set of memory objects MO, the instruction memory hierarchy consisting of a scratchpad memory, an instruction cache and a main memory and a conflict graph $G(N, E)$ representing the behavior of the instruction cache. The problem is to determine a subset $MO_{SPM} \subseteq MO$ of the set of memory objects MO such that the allocation of memory objects $mo_i \in MO_{SPM}$ to the scratchpad memory minimizes the total energy consumption E_{Total} of the instruction memory subsystem.

$$E_{Total} = \sum_{mo_i \in MO} E(mo_i) \tag{5.12}$$

The energy consumption function $E(mo_i)$ of a memory object mo_i is formally defined in the previous subsection. The minimization of the total energy consumption E_{Total} is to be performed under the following scratchpad size constraint:

$$\sum_{mo_i \in MO_{SPM}} size(mo_i) \leq size(SPM) \tag{5.13}$$

The above optimization problem is related to two NP-complete problems, *viz.* Knapsack [43] and Weighted Max-Cut [43] problem. Let's make the simplifying assumption that the cache present in the system is large or high-associative enough to hold all the memory objects such that not a single conflict cache miss occurs. The energy consumption of a memory object under the above assumption becomes independent of other memory objects and the CASA problem is reduced to a Knapsack problem with each node having constant weights.

On the other hand, if we assume that the energy of an access to the scratchpad E_{SP_hit} is equal to the energy of a cache hit E_{Cache_hit} and that the conflict graph is an undirected graph, the problem is reduced to the Bi-Criteria Max-Cut problem. In this problem, the cost of the cut is defined as the sum of the weight of the nodes present in side the cut and the weight of the edges crossing the cut. A simple transformation of the objective function (*cf.* Equation 5.12) to the objective function of the Max-Cut problem can be easily constructed. The maximization of the objective function of the Max-Cut problem needs to be performed while respecting the scratchpad size constraint. In the following section, we present ILP based optimal and greedy heuristic based near-optimal approaches for solving the cache aware scratchpad allocation problem.

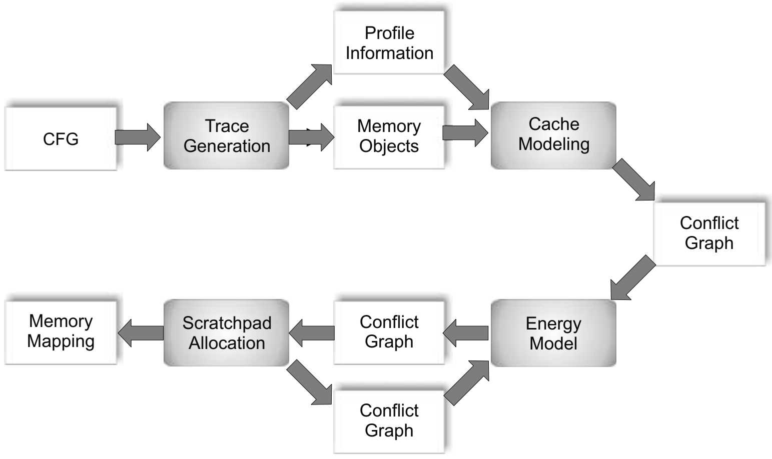


Fig. 5.8. Workflow of Scratchpad Allocation Approaches

5.5 Cache Aware Scratchpad Allocation

The non-overlayed cache aware scratchpad allocation approaches presented in the following subsections are based upon the workflow shown in Figure 5.8. The first step of the proposed approaches is the memory object determination step. It uses the compiler optimization, *viz.*, trace generation, to identify traces from functions larger in size than a threshold value. Functions along with traces are considered as memory objects in the proposed approaches. In the first step, NOP instructions are also appended to the memory objects to align them to the cache line boundaries. The alignment of memory objects is done as it helps in correctly correlating a cache miss to the culprit memory object.

The second step analyzes the interaction of memory objects within the cache. The interaction is represented as a conflict graph with nodes denoting the memory objects and edges between nodes denoting the conflict relationship between the corresponding memory objects. In the third step, the energy model computes the energy consumption for each memory object and attributes it to each node in the conflict graph. Finally, based upon the conflict graph, the scratchpad allocation approach determines a mapping of memory objects to the non-cacheable scratchpad memory in order to minimize the energy consumption of the system.

The ILP based optimal allocation approach selects the memory objects to be mapped to the scratchpad in a single step. In contrast, the greedy heuristic iterates over the conflict graph, every time removing the most energy consuming memory object from the conflict graph and mapping it onto the scratchpad memory. Next, we describe the ILP based optimal approach for solving the CASA problem. After that, we present a greedy heuristic which solves the CASA problem near-optimally with a polynomial runtime complexity.

5.5.1 Optimal Cache Aware Scratchpad Allocation

Once we have created the conflict graph G annotated with vertex and edge weights, the energy consumption of memory objects can be computed. Now, the problem is to select a subset of memory objects which minimizes the overall energy consumed by the instruction memory hierarchy of the system. The subset should be bounded in size by the size of the scratchpad memory. We define a couple of binary variables before presenting the ILP formulation. The binary variable $l(mo_i)$ denotes the location of memory object mo_i in the memory hierarchy and is defined as follows:

$$l(mo_i) = \begin{cases} 0 \text{ if memory object } mo_i \text{ is present in the SPM} \\ 1 \text{ otherwise} \end{cases} \tag{5.14}$$

The miss function $Miss(mo_i, mo_j)$, presented below, returns the number of conflict cache misses of memory object mo_i caused by memory object mo_j:

$$Miss(mo_i, mo_j) = \begin{cases} 0 & \text{if memory object } mo_j \text{ is present in the SPM} \\ w(e_{ij}), & \text{otherwise} \end{cases} \tag{5.15}$$

where $w(e_{ij})$ is the weight of the edge e_{ij} connecting node n_i to node n_j. The miss function $Miss(mo_i, mo_j)$ returns zero if memory object mo_j is present in the scratchpad, because the memory objects present in the scratchpad do not conflict with those present in the cache. The miss function is reformulated using the location variable $l(mo_j)$ as:

$$Miss(mo_i, mo_j) = l(mo_j) * w(e_{ij}) \tag{5.16}$$

The location variable $l(mo_i)$ is also used to reformulate the energy function $E(mo_i)$ (*cf.* Equation 5.4) which denotes the energy consumed by the memory object mo_i.

$$E(mo_i) = [1 - l(mo_i)] * E(mo_i, SPM) + l(mo_i) * E(mo_i, Cache) \tag{5.17}$$

The energy functions $E(mo_i, Cache)$ and $E(mo_i, SPM)$ presented in Equation 5.10 and Equation 5.11, respectively, are substituted into the above equation. After rearranging the terms, the above equation representing the energy consumption is transformed to the following equation.

$$\begin{aligned} E(mo_i) = {} & w(n_i) * E_{SP\_hit} + w(n_i) * [E_{Cache\_hit} - E_{SP\_hit}] * l(mo_i) \\ & + [E_{Cache\_miss} - E_{Cache\_hit}] * \left[\sum_{mo_j \in N(mo_i)} l(mo_j) * l(mo_i) * w(e_{ij}) \right] \end{aligned} \tag{5.18}$$

We find the last term is a quadratic degree term, since the number of misses of a memory object mo_i not only depends on its location but also on the location of the conflicting memory objects mo_j. In order to formulate a 0-1 Integer Linear Programming problem, we need to linearize the above equation. This can be achieved by replacing the non-linear term $l(mo_i) * l(mo_j)$ of Equation 5.18 by an additional binary variable $L(mo_i, mo_j)$:

$$\begin{aligned} E(mo_i) = {} & w(n_i) * E_{SP\_hit} + w(n_i) * [E_{Cache\_hit} - E_{SP\_hit}] * l(mo_i) \\ & + [E_{Cache\_miss} - E_{Cache\_hit}] * \left[\sum_{mo_j \in N(mo_i)} L(mo_i, mo_j) * w(e_{ij}) \right] \end{aligned} \tag{5.19}$$

In order to prevent the linearizing variable $L(mo_i, mo_j)$ from assuming arbitrary values, the following linearization constraints are added to the set of constraints:

$$l(mo_i) - L(mo_i, mo_j) \geq 0 \tag{5.20}$$

$$l(mo_j) - L(mo_i, mo_j) \geq 0 \tag{5.21}$$

$$l(mo_i) + l(mo_j) - 2 * L(mo_i, mo_j) \leq 1 \tag{5.22}$$

The objective function E_{Total} to be minimized, as shown in the following, denotes the total energy consumed by the instruction memory subsystem.

$$E_{Total} = \sum_{mo_i \in MO} E(mo_i) \tag{5.23}$$

The objective function is to be minimized while maintaining the scratchpad size constraint.

$$\sum_{mo_i \in MO} [1 - l(mo_i)] * size(mo_i) \leq size(SPM) \tag{5.24}$$

The size of memory object $size(mo_i)$ is computed without considering the appended NOP instructions which are stripped away from the memory objects prior to allocating them to the scratchpad. The underlying assumption in the problem formulation is that no new edges in the conflict graph or no new conflict relationships are created when a memory object is mapped to the non-cacheable scratchpad. This assumption can be partially fulfilled by keeping the program memory layout invariant. However, the assumption may fail for graphs with circular edge dependencies. For example, in Figure 5.7 memory objects mo_1, mo_2 and mo_3 have circular edge dependencies. Thus, the allocation of memory object mo_3 to the scratchpad may create a new edge from mo_2 to mo_1.

The above ILP formulation solves the cache aware scratchpad allocation (CASA) problem, as it determines the subset of memory objects ($MO_{SPM} = \{mo_i | l(mo_i) = 0\}$) which minimizes the energy consumption of the instruction memory subsystem. The number of

```
CacheAwareScratchpadAllocation-Heuristic(G(N,E), SPMSize)
1  RemSPMSize = SPMSize /* default values */
2  $MO_{SPM} = \{\}$
3  while ( $\exists mo_i \in MO : size(mo_i) \leq$ RemSPMSize ) do
4    /* select the memory object with maximum energy consumption */
5    /* and which is smaller than the remaining scratchpad size */
6    select $n_i \in N$ such that $size(mo_i) \leq$ RemSPMSize and
     $E(mo_i) > E(mo_k)\ \forall n_k \in N : size(mo_k) \leq$ RemSPMSize
7    /* remove node $n_i$ from the graph $G$ */
8    $E = E - \{e_{ij} | \forall j : j \in N(n_i)\} - \{e_{ji} | \forall i : i \in N(n_j)\}$
9    $N = N - \{n_i\}$
10   RemSPMSize = RemSPMSize - $size(mo_i)$
11   /* add the memory object to the set of memory objects $MO_{SPM}$ */
12   $MO_{SPM} = MO_{SPM} \bigcup mo_i$
13   /* Recompute the energy consumption value for each node */
14   UpdateEnergyValue($G$)
15 end-while
16 return $MO_{SPM}$
```

Fig. 5.9. Greedy Heuristic for Cache Aware Scratchpad Allocation Problem

vertices $|N|$ of the conflict graph G is equal to the number of memory objects, which is bounded by the number of basic blocks in the program code. The number of linearizing variables is equal to the number of edges $|E|$ in the conflict graph G. Therefore, the number of variables in the ILP formulation is equal to $|N| + |E|$ and is bounded by $O(|N|^2)$.

The actual runtime of the commercial ILP solver [32] was found to be less than a second on a Sun Sparc 1300 MHz compute blade server for a conflict graph containing a maximum of 455 vertices. However, we cannot guarantee that ILP based approach will scale efficiently for benchmarks with large conflict graphs. Therefore, in the following subsection, we describe a polynomial time heuristic for obtaining near-optimum solutions.

5.5.2 Near-Optimal Cache Aware Scratchpad Allocation

The proposed greedy heuristic takes as input the conflict graph G and the scratchpad size and returns the set of memory objects MO_{SPM} to be allocated onto the scratchpad memory. The pseudo-code of the heuristic is presented in Figure 5.9.

The heuristic iterates over the conflict graph and for each node of the graph computes the energy consumption of the corresponding memory object. The energy model (*cf.* Subsection 5.4.4) uses the execution count and the conflict cache misses to compute the energy consumption of the memory objects. In each iteration, the heuristic selects the maximum energy memory object which can fit in the available scratchpad memory and adds the memory object to the set of memory objects MO_{SPM} marked for scratchpad allocation. It then removes the corresponding node from the conflict graph and appropriately reduces the unallocated (RemSPMSize) scratchpad size. At the end of each iteration the heuristic recomputes and stores the energy consumption values for each memory object.

The heuristic iterates as long as there exists a memory object which can be placed on the scratchpad without violating the scratchpad size constraint. On termination, the set of

Benchmark	Code Size (bytes)	I-Cache Size (bytes)	System Architecture
adpcm	944	128 B DM	uni-processor ARM
epic	12 kB	1 kB DM	uni-processor ARM
g721	4.7 kB	1 kB DM	uni-processor ARM
media	4 kB	1 kB DM	uni-processor ARM
mpeg2	21.4 kB	2 kB DM	uni-processor ARM
multi-process edge detection	4484	4 kB DM	multi-processor ARM

Table 5.10. Benchmark Programs for the Evaluation of CASA Approaches

memory objects MO_{SPM} marked for scratchpad allocation is returned. The time complexity of the heuristic is $O(|N| * (|N| + |E|))$. As shown in the following section, the heuristic achieves close to optimal results for most of the experiments.

5.6 Experimental Results

In this section, the cache aware scratchpad allocation approaches are evaluated for a uni-processor ARM and a multi-processor ARM based system. Table 5.10 presents the benchmarks used for the evaluation of the approaches. It also presents the code size of the benchmarks as well as the size of the instruction cache used in the experimental setup. Experiments were conducted by varying the size of the scratchpad/loop cache as well as the size and the associativity of the instruction cache present in the system. The size of the instruction cache line, though, was kept constant at 16 bytes. The application is executed and profiled to compute the number and the type of accesses to each memory in the hierarchy. Based on the profile information and the energy model, the energy consumption of the system is computed.

The multi-processor ARM simulator allows us to compute the energy consumption of the multi-processor system for different memory hierarchies, therefore, total energy consumption values are reported. For the uni-processor ARM based setup, the simulator (ARMulator [12]) is used to simulate the application with a flat memory hierarchy, whereas the memory hierarchy simulator [89] is used to simulate the memory hierarchy consisting of instruction and data caches, a scratchpad and a loop cache. The energy consumption of the instruction memory subsystem and the number of CPU cycles spent during the execution of the benchmarks are computed. A detailed description of the experimental setup can be found in Chapter 3.

5.6.1 Uni-Processor ARM

The evaluation of the cache aware scratchpad allocation approaches for the uni-processor ARM based system is presented in the following order:

(a) Benefits of a cache and a scratchpad memory based memory hierarchy for the *mpeg* benchmark.
(b) Comparison of the scratchpad memory allocation approaches.

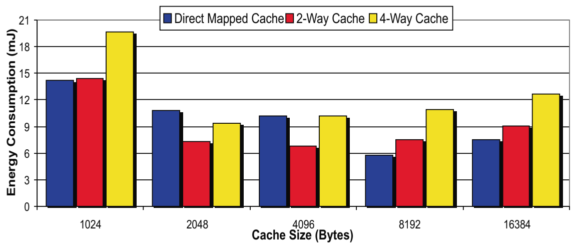


Fig. 5.10. MPEG: Instruction Memory Energy Consumption

(c) Determination of the optimal scratchpad memory size for the *mpeg* benchmark.
(d) Comparison of preloaded loop cache and scratchpad memory.

First, we present the benefits of an instruction cache and scratchpad based instruction memory hierarchy. To demonstrate the benefit of the current memory hierarchy, a comparison of its energy consumption and that of the least energy consuming instruction cache based memory hierarchy is presented. Second, we compare the optimal and near-optimal cache aware scratchpad allocation approaches with the non-overlayed scratchpad allocation approach proposed in Chapter 4. Third, we determine the most energy efficient instruction cache and scratchpad based memory hierarchy for the *mpeg* benchmark. Last, a comparison of the scratchpad allocated with the proposed optimal approach with the preloaded loop cache is presented.

Benefit of Instruction Cache and Scratchpad Based Memory Hierarchy:
First, we determine the least energy consuming instruction cache based memory hierarchy for the *mpeg* benchmark. Experiments are conducted by varying the size and the associativity of the instruction cache present in the system. Figure 5.10 displays the energy consumption of the instruction memory hierarchy as a function of the size for three different set associative instruction caches.

We observe from the figure that for each associativity, the energy consumption of the instruction memory subsystem monotonically decreases to the minimum point with the increase in the size of the instruction cache. Thereafter, the energy consumption increases with any further increase in the cache size. The reason for the increase in the energy consumption is that large energy per access values for large caches offset the energy savings achieved due to lower cache misses. The minimum energy points represent the most energy efficient instruction cache size for each associativity. It can be observed that 2k, 4k and 8k bytes caches result in the minimum energy consumption for 4-way, 2-way set associative and direct mapped instruction cache based memory hierarchies, respectively. Moreover, from Figure 5.10, we observe that the 8k byte direct mapped instruction cache forms the least energy consuming instruction memory hierarchy for the *mpeg* benchmark.

Now, we assume that the instruction memory hierarchy consists of an instruction cache and a scratchpad memory and conduct the experiments to determine the energy consumption of the current memory hierarchy. The scratchpad memory is allocated using the optimal cache aware scratchpad allocation approach presented in Subsection 5.5.1. Figure 5.11 shows the relative energy consumption of the memory hierarchies consisting of 512 or 1024 bytes scratchpads and instruction caches of various sizes and associativity. The energy

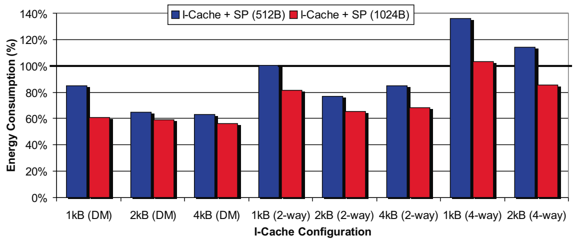


Fig. 5.11. MPEG: Comparison of Energy Consumption of I-Cache + Scratchpad with 8 kB DM I-Cache

consumption of the least energy consuming memory hierarchy, *i.e.* 8k bytes direct mapped instruction cache, found previously is represented as the 100% baseline.

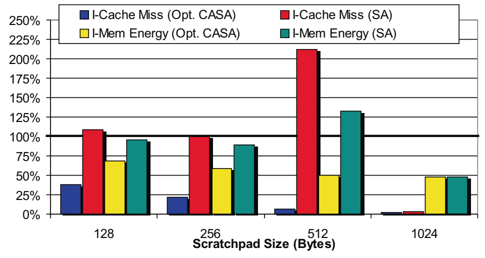
There are a couple of important observations to be made from the figure. First, in most cases the energy consumption of the scratchpad and cache based memory hierarchy is lower than the least energy consuming cache-only memory hierarchy. The exception cases occur for small and high associative caches, as the energy consumption of the memory hierarchies based on these caches is much higher than the least energy consumption values. For example, the energy consumption of a 1k bytes 4-way set associative instruction cache based memory hierarchy (*cf.* Figure 5.10) consumes more than 3 times as much energy as that consumed by the least energy consuming memory hierarchy. Hence, even the addition of a scratchpad to the hierarchy could not reduce their energy consumption below the 100% baseline.

Second, the energy consumption of the memory hierarchy consisting of a 1k bytes direct mapped cache and a 1k bytes scratchpad relative to the least energy consuming memory hierarchy is about 60%. Consequently, the efficacy of the considered memory hierarchy is demonstrated by the fact that not only does it consume only 60% of the energy consumed by the least energy consuming memory hierarchy but also accounts for a mere (2k bytes) 25% of its onchip size.

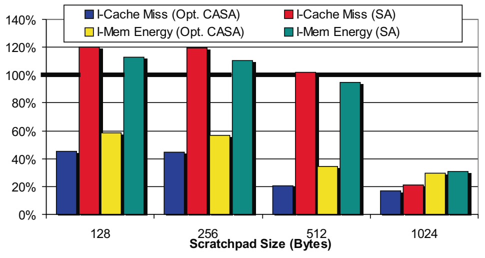
Comparison of Scratchpad Allocation Approaches:

We will start by demonstrating the negative implications of not modeling a cache by the non-overlayed scratchpad allocation (SA) approach. Figure 5.12(a) displays the number of instruction cache misses and the energy consumption of the *epic* benchmark using the SA approach and the proposed optimal cache aware scratchpad allocation (Opt. CASA) approach. In order to display both the cache misses and energy consumption values in the same figure, they are presented as percentages of a 100% baseline. The baseline in Figure 5.12(a) represents the number of cache misses and the energy consumption of an instruction memory hierarchy without a scratchpad. The 100% baseline is independent of the scratchpad size and is pegged at a constant value for all the experiments.

From Figure 5.12(a), we observe that the number of cache misses for the SA approach demonstrate unpredictable behavior. At 128 bytes scratchpad, the cache misses are slightly higher than the baseline, but are lower for 256 bytes. However at 512 bytes, a steep rise in the number of cache misses is observed. This phenomenon of excessive increase in cache misses is known as *cache thrashing* and it also substantially increases the energy consumption of the system. On the other hand, for the Opt. CASA approach, cache misses and energy consumption values monotonically decrease with the increase in scratchpad



(a) EPIC



(b) MPEG

Fig. 5.12. Cache Behavior: Comparison of Opt-CASA and SA Approaches

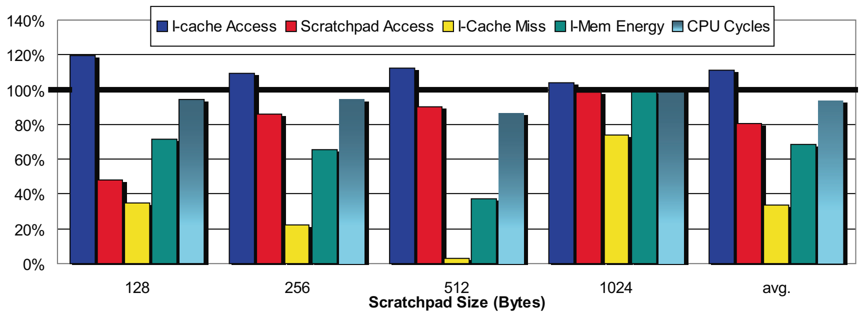


Fig. 5.13. EPIC: Comparison of Opt. CASA and SA Approaches

size. This characteristic behavior of our approach originates from the precise cache and energy models.

Figure 5.12(b) shows a similar comparison of the allocation approaches, and the SA approach again demonstrates an unpredictable behavior for the *mpeg* benchmark. The baseline, again, represents the number of cache misses and the energy consumption of the *mpeg* benchmark on a system without a scratchpad. The energy consumption for the SA approach is higher than the baseline for scratchpad sizes of 128 bytes and 256 bytes. The high number of cache misses due the SA approach for both the benchmarks nullifies the energy reductions achieved due to the utilization of an energy efficient scratchpad.

A detailed comparison of all the parameters of the instruction memory hierarchy is presented to enable a better appreciation of the results: Figures 5.13 and 5.14 display the energy consumption of the instruction memory hierarchy with all its respective parameters (*i.e.* scratchpad accesses, cache accesses and cache misses) for the *epic* and the *mpeg* benchmarks, respectively, allocated using the Opt. CASA approach. The last column of the figures displays the execution time of the two benchmarks in terms of CPU cycles. A direct mapped instruction cache of size 1k and 2k bytes is assumed to be present in the system for the *epic* and the *mpeg* benchmarks, respectively.

The results of the Opt. CASA are compared with the corresponding results of the SA approach. Unlike the baseline of Figure 5.12(a), the 100% baseline in Figure 5.13 represents the varying experimental values achieved by the SA approach for each scratchpad size.

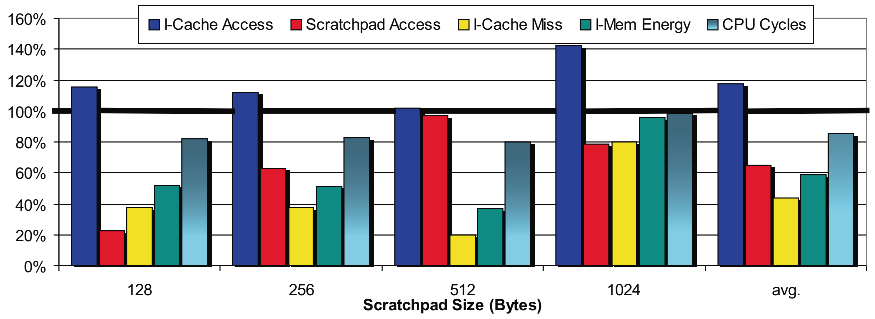


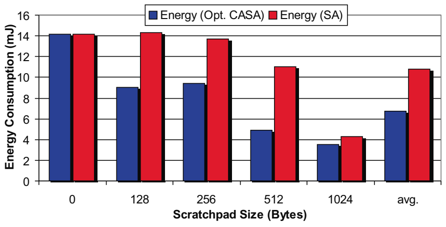
Fig. 5.14. MPEG: Comparison of Opt. CASA and SA Approach

However, it enables a direct comparison of the Opt. CASA approach with the SA approach for each scratchpad size. It should be noted that the cache misses and the energy consumption of the system (*cf.* Figure 5.12(a)) for the Opt. CASA approach decreases monotonically with the increase in the scratchpad size.

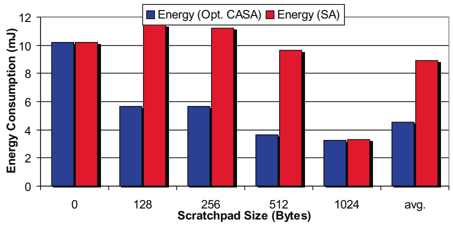
From Figures 5.13 and 5.14, it is interesting to note that in spite of the higher instruction cache accesses (see the first bar) and lower scratchpad accesses (see the second bar), the Opt. CASA approach reduces energy consumption compared with the SA approach. The reason for this behavior is that the SA approach reduces the energy consumption by increasing the number of scratchpad accesses. In contrast, the Opt. CASA approach reduces the energy consuming cache misses by assigning conflicting memory objects to the scratchpad memory. Since on every cache miss, the slow and power hungry main memory is accessed, avoiding cache misses is beneficial both in terms of energy and performance. Therefore, the Opt. CASA approach with substantially lower instruction cache misses is able to over-compensate for higher cache access and results in reduced energy consumption values. For the *mpeg* benchmark, the Opt. CASA approach achieves up to 80% reduction in instruction cache misses and as a consequence achieves substantial energy reductions of more than 60%. On the average, the Opt. CASA average conserves 31% and 42% energy compared with the SA approach for the *epic* and the *mpeg* benchmarks, respectively. A reduction of 14% in the CPU cycles is also reported for the *mpeg* benchmark. However, at 1024 bytes the reduction in energy consumption and the execution time using our algorithm is minimal. This is due to the fact that a 1024 bytes scratchpad is large enough to hold all important memory objects and as a consequence the solution sets of the SA approach as well as the Opt. CASA approach are fairly similar. However, the SA approach modifies the program layout, therefore, may also lead to erratic results.

Up to this point, we compared the scratchpad allocation algorithms for *epic* and *mpeg* benchmarks for memory hierarchies consisting of direct mapped instruction caches of 1k and 2k bytes, respectively. Now, we compare the Opt. CASA approach with the SA approach for the *mpeg* benchmark and for systems with different instruction cache configurations. The considered cache configurations include direct-mapped 1k and 4k bytes caches and 2-way and 4-way set associative caches of size 1k and 2k bytes.

Figures 5.15(a) and 5.15(b) illustrates the energy consumption of the *mpeg* benchmark for memory hierarchies containing 1k bytes and 4k bytes of direct mapped instruction caches, respectively. The proposed Opt. CASA approach always leads to a more energy

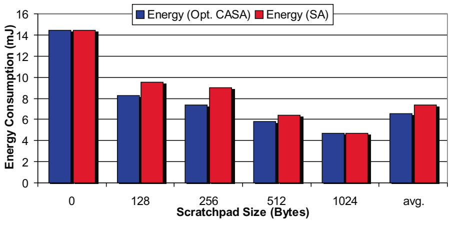


(a) 1 kB DM I-cache

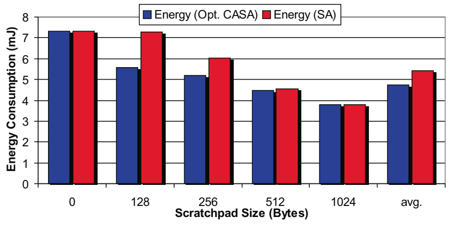


(b) 4 kB DM I-cache

Fig. 5.15. MPEG: Energy Comparison of Opt. CASA and SA Approaches for Direct Mapped I-Caches



(a) 1 kB 2-way I-cache



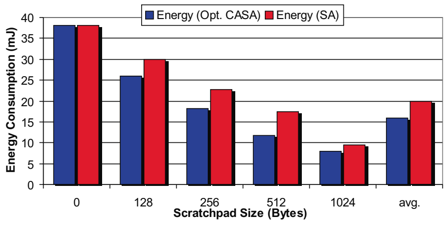
(b) 2 kB 2-way I-cache

Fig. 5.16. MPEG: Energy Comparison of Opt. CASA and SA Approaches for 2-Way Set-Associative I-Caches

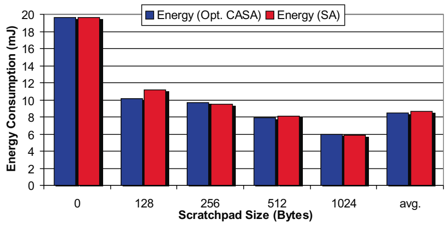
efficient allocation than that obtained by the SA approach. For the 4k bytes instruction cache, the SA approach again causes the *cache thrashing* problem and results in energy consumption values higher than those for a system without a scratchpad memory. On the average, the Opt. CASA approach leads to an energy reduction of 35% over the SA approach for a system with 1k byte direct mapped instruction cache. An even higher average energy reduction of 41% is reported for the system with a 4k byte direct mapped instruction cache.

Figures 5.16(a) and 5.16(b) present the comparison of the proposed Opt. CASA approach and the SA approach for the systems with 1k and 2k bytes of 2-way set associative instruction cache, respectively. In Figure 5.16(b), the SA approach again displays an unpredictable behavior, as it leads to an increase in cache misses for 128 byte scratchpad. The Opt. CASA approach performs better than the SA approach, although the reduction in energy consumption is less than that achieved for direct mapped instruction caches. This behavior is justified as the 2-way set associative instruction caches result in a hit ratio of more than 99% for the *mpeg* benchmark. Nevertheless, the algorithm achieves energy savings of upto 19% and 24% compared with the SA approach for 1k and 2k bytes instruction caches, respectively.

The last set of Figures 5.17(a) and 5.17(b) presents the comparison of the allocation approaches for 1k and 2k bytes 4-way set associative instruction cache based memory hierarchy, respectively. For high associative caches, the number of conflict cache misses is substantially lower than that for low associative or direct mapped caches. Consequently, very few conflict edges are present in the conflict graph used to model the behavior of memory objects present in a cache. As discussed in Section 5.4, our allocation problem reduces to the



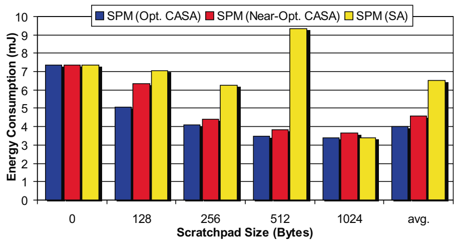
(a) 1 kB 4-way I-cache



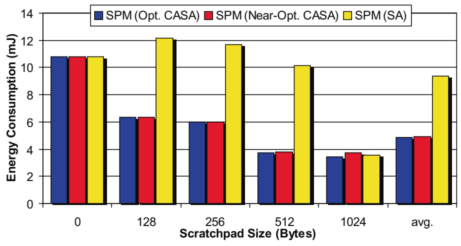
(b) 2 kB 4-way I-cache

Fig. 5.17. MPEG: Energy Comparison of Opt. CASA and SA Approaches for 4-Way Set-Associative I-Caches

Knapsack problem, which also forms the basis of the SA approach. Under these conditions, the allocation of the memory objects achieved by the Opt. CASA approach is similar to that achieved by the SA approach. This is also corroborated by energy consumption values presented in Figure 5.17.



(a) EPIC

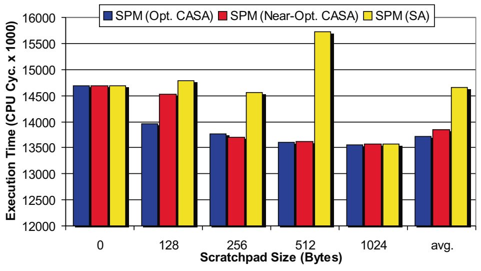


(b) MPEG

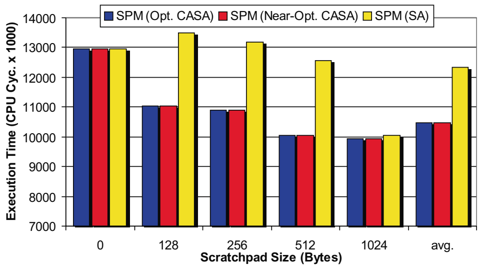
Fig. 5.18. Energy Comparison of Opt. CASA, Near-Opt. CASA and SA Approaches

Figures 5.18 and 5.19 present a comparison of the scratchpad allocation approaches for the *epic* and *mpeg* benchmarks, respectively. Figure 5.18 displays the energy consumed by the benchmarks while Figure 5.19 presents the execution time of the benchmarks when optimized using the scratchpad allocation approaches. The scratchpad allocation approaches include the optimal (Opt. CASA) and the near-optimal (Near-Opt. CASA) cache aware scratchpad allocation approaches and the non-overlayed scratchpad allocation (SA) approach.

A few interesting points can be noted from the figures. Firstly, the Opt. CASA and Near-Opt. CASA approaches result in a monotonically decreasing energy consumption and execution time behavior of the benchmarks. However, the reduction in the energy consumption is larger than that in execution time. The reason for this behavior is the difference in energy per access to a main memory and a scratchpad is much larger than the difference in the access times of the two memories. Table 3.2 on Page 22 summarizes the energy per access and the access times of the two memories. Secondly, the energy consumption of the benchmarks due to the Near-Opt. CASA approach is fairly close to the optimal energy consumption achieved by the Opt. CASA approach.



(a) EPIC



(b) MPEG

Fig. 5.19. Execution Time Comparison of Opt. CASA, Near-Opt. CASA and SA Approaches

Thirdly, we observe that the execution time of the *epic* benchmark for the Near-Opt. CASA approach is slightly better than the Opt. CASA approach at 256 bytes of scratchpad. In contrast, the energy consumption of the benchmark due the Near-Opt. CASA approach is larger than the Opt. CASA approach at the same scratchpad size. This reason for this behavior is that the objective of the allocation approaches is to optimize the energy consumption, while the performance of benchmarks is improved as a side effect. Therefore, the energy optimal solution is not always the performance optimal solution. Finally, we observe that the average energy consumption due to the Near-Opt. CASA approach is about 30% and 47% better than that due to SA approach for *epic* and *mpeg* benchmark, respectively.

Figure 5.20 summarizes the average energy consumption of various benchmarks for instruction cache and scratchpad based systems. For the experiments, the size of the instruction cache was chosen according to the code size of the benchmark. We expect that for real-life embedded applications the code size is about 8-10 times larger than the instruction cache size. Consequently, the instruction cache size was set to 128, 1k, 1k, 1k and 2k bytes (*cf.* Table 5.10 on Page 68) for benchmarks *adpcm*, *epic*, *g721*, *media* and *mpeg*, having program size of 1k, 4.7k, 12k, 4k and 21.4k bytes, respectively. The energy and performance values, shown in the figure, are the average values obtained by varying the scratchpad size in the range of 64 bytes and 1k bytes.

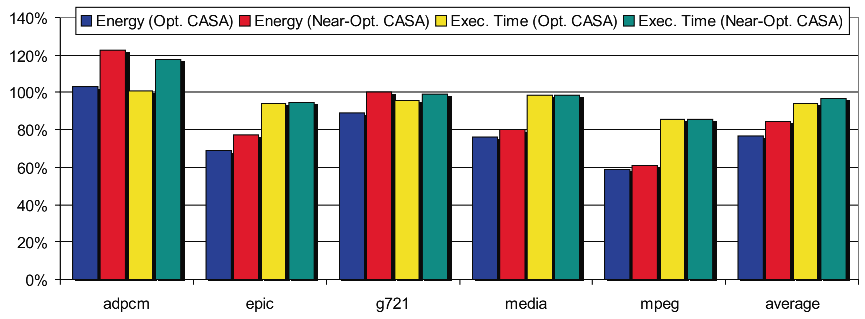


Fig. 5.20. Overall Comparison of Opt. CASA, Near-Opt. CASA and SA Approaches

Benchmark	Code Size (wo NOPs) bytes	Code Size (w NOPs) bytes	Appl. Size (wo NOPs) bytes	Appl. Size (w NOPs) bytes	% inc. (Code Size)	% inc. (Appl. Size)
adpcm	804	944	7736	7876	17.41	1.81
epic	9268	12132	91152	94016	30.90	3.14
g721	4376	4556	7388	7568	4.11	2.44
media	3280	4808	78950	80478	46.59	1.94
mpeg	18284	21896	50320	53932	19.75	7.18

Table 5.11. Code and Application Sizes of Benchmarks without and with Appended NOP Instructions

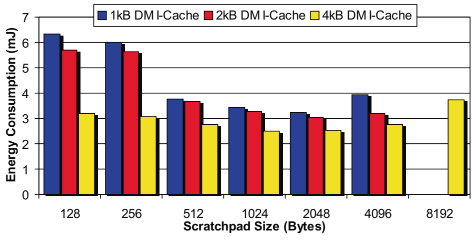
For Figure 5.20, we make a few important observations. Firstly, the energy consumption due to the Opt. CASA approach is lower than the SA approach for all but one benchmark. The exception case occurs for the *adpcm* benchmark which contains two simple encode and decode routines. The SA approach moves one of the traces of the benchmark to the scratchpad and, thereby accidentally modifies the code layout such that the conflict caches misses are minimized. On the other hand, the Opt. CASA approach does not change the code layout and thus could not minimize the cache misses for this benchmark. Secondly, the Near-Opt. CASA approach performs nearly as good as the optimal approach. On an average, the Near-Opt. CASA approach is only 6% worse than the Opt. CASA approach. Finally, the Opt. CASA and Near-Opt. CASA approaches achieve overall average energy reductions, across all benchmarks and all scratchpad sizes, of 22% and 18%, respectively, compared to the SA approach.

The cache aware scratchpad allocation approaches append NOP instructions to memory objects such that the memory objects are aligned to cache line boundaries. The insertion of NOP instructions causes an increase in the code size and the aggregate size (*i.e.* code size + data size) of the application. Table 5.11 summarizes the code size and the aggregate size of the benchmarks with and without the appended NOP instructions. The table also presents the percentage increase in the sizes of benchmarks. The increase in the code size of benchmarks range between 4% and 46%. The high increase in the code size does not translate into a corresponding high increase in the aggregate application size, as the code size accounts for a relatively small fraction of the aggregate application size. We observe (*cf.* Table 5.11) a maximum increase of only 7% in the aggregate application size over all benchmarks.

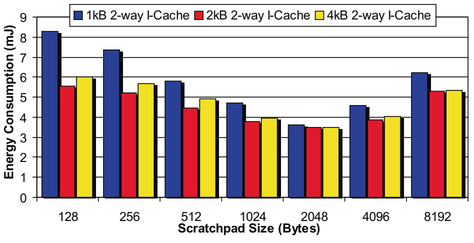
Determining the Optimal Scratchpad Size:

In the experiments presented so far, memory objects were allocated onto the given memory hierarchy, consisting of a scratchpad memory and an instruction cache. Now, we present the results of the experiments conducted to determine the optimal scratchpad size for the *mpeg* benchmark. The experimental results also determine the set of Pareto-optimal scratchpads for the *mpeg* benchmark. The experiments were conducted by increasing the scratchpad size from 128 bytes upto 8k bytes and the energy consumption of the system was computed. The scratchpad present in the memory hierarchy is allocated using the Opt. CASA approach.

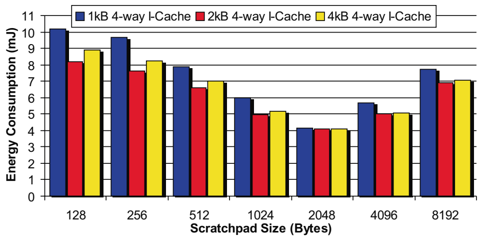
Figure 5.21(a) shows the energy consumption of the instruction memory hierarchy with a direct-mapped instruction cache and a scratchpad memory of varying sizes. Additional experiments were conducted by varying the size and the associativity of the instruction cache.



(a) DM I-cache



(b) 2-way I-cache



(c) 4-way I-cache

Fig. 5.21. MPEG: Determining the Optimal Scratchpad Size

Figures 5.21(b) and 5.21(c) present the energy consumption of the systems consisting of a 2-way and a 4-way set associative instruction cache, respectively.

From Figure 5.21, we observe that the energy consumption of the memory hierarchy decreases as we increase the scratchpad size until it reaches the minimum point. Any further increase in the scratchpad size also increases the energy consumption of the hierarchy. As shown in Figure 5.21(a), the minimum energy point occurs at 1k bytes of scratchpad memory for a 4k bytes direct mapped instruction cache based system. However, for systems with 1k bytes and 2k bytes direct mapped instruction cache, 2k bytes of scratchpad memory lead to the minimum energy consumption.

For 2-way and 4-way set associative instruction cache based systems (*cf.* Figures 5.21(b) and 5.21(c)), the minimum energy consumption occurs when 2k bytes of scratchpad memory is present in the system. Scratchpad memories larger than the minimum energy configuration scratchpad memory are an unattractive option compared to the instruction cache present in the system. The high energy per access to the large scratchpad memory offsets the gains that can be achieved by allocating more memory objects to the scratchpad. Consequently, the approach allocates more memory objects to the instruction cache and less to the large scratchpad memory.

The benefits of performing the above set of experiments are threefold. Firstly, we are able to study the variation in the energy consumption of the system with the increase in scratchpad size. It is interesting to observe that 2k bytes of scratchpad memory form the minimum energy configuration in all but one cache configurations. However, the energy consumption of the remaining configuration (1k bytes scratchpad and 4k bytes instruction cache) is the global minimum energy consumption value.

Secondly, we are able to determine the range of scratchpad sizes which would be interesting for a system designer to perform design-space exploration. For example, a memory hierarchy composed of 2k bytes scratchpad and 2k bytes direct mapped cache is 20% smaller but results in 20% more energy consumption than the globally minimum energy consuming memory hierarchy. For the *mpeg* benchmark, the minimum energy consuming scratchpad size is 1k bytes or 2k bytes. Consequently, scratchpad memories between 128 bytes and 1k or 2k bytes form the set of energy efficient (Pareto-optimal) scratchpad sizes, which allow a trade-off between the onchip area and the energy consumption of the system. Scratchpads larger than 2k bytes consume more onchip area and also result in increased energy consumption of the system. Hence, they are not part of the energy efficient range of scratchpad sizes. Finally, an iterative or a binary search based algorithm can be employed for determining the optimal scratchpad size for a given instruction cache based memory hierarchy.

5.6.2 Comparison of Scratchpad and Loop Cache Based Systems

Now, we compare the energy savings achieved by a scratchpad based system with those achieved by a preloaded loop cache based system. The scratchpad is allocated using the Opt. CASA approach while the preloaded loop cache is allocated using the Ross approach [48]. The loop cache is bounded by the aggregate size and the number of memory objects which can be allocated. In current setup, the loop cache controller is assumed to have 4 pairs of registers for storing the start and end addresses of memory objects. Therefore, a maximum of 4 non-contiguous memory objects can be stored onto the loop cache.

We extended the Ross approach [48] to coalesce memory objects which occupy adjacent locations in the main memory and are marked for allocation onto the loop cache. The coalescing enabled us to store more memory objects onto the loop cache as the coalesced memory object requires only a single pair of registers to store its start and end addresses.

Figures 5.22 and 5.23 display the energy consumption of the instruction memory hierarchy and the CPU cycles for the *epic* and the *mpeg* benchmarks, respectively. The number of scratchpad accesses, instruction cache accesses as well as the instruction cache misses are also displayed in the figures. All results are shown as percentages of the corresponding parameters of the Ross approach. Similar to Figure 5.13, the values of the 100% baseline vary for each loop cache size. The size of the loop cache was assumed to be equal to the size

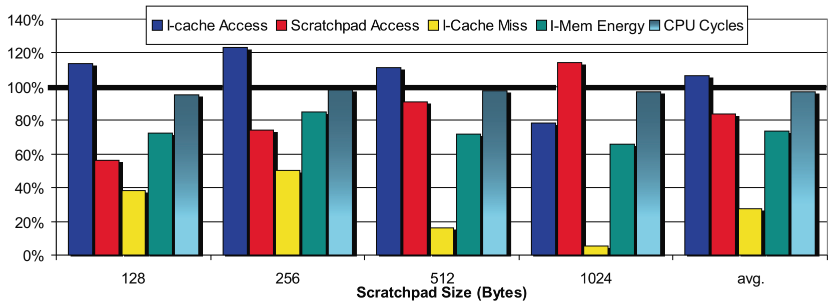


Fig. 5.22. EPIC: Comparison of (SPM) Opt. CASA and (Loop Cache) the Ross Approach

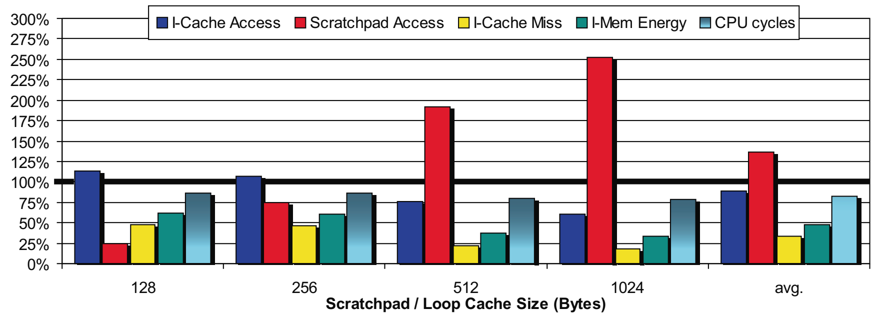


Fig. 5.23. MPEG: Comparison of (SPM) Opt. CASA and (Loop Cache) the Ross Approach

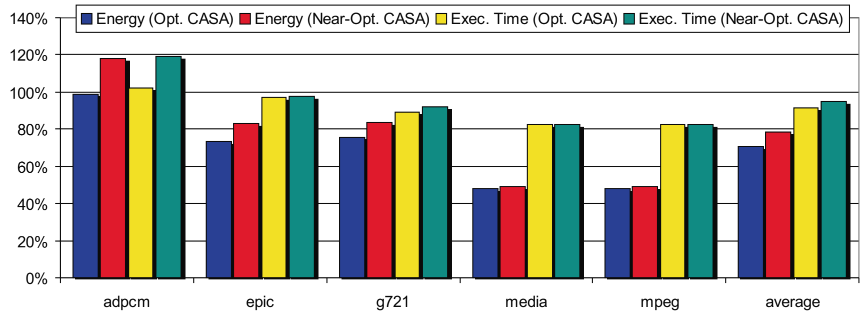


Fig. 5.24. Overall Comparison of (SPM) Opt. CASA, (SPM) Near-Opt. CASA and (Loop Cache) the Ross Approach

of the scratchpad, even though a loop cache requires more onchip area than a scratchpad due to the presence of a controller.

For small scratchpad/loop cache sizes (128 and 256 bytes), the number of accesses to the loop cache is higher than that to scratchpad. However, as we increase the size, the loop cache's performance is restricted by the maximum number of pre-loadable memory objects. The scratchpad, on the other hand, can be preloaded with any number of memory objects as long as their aggregate size is less than the scratchpad size. Moreover, the number of instruction cache misses is substantially lower for all sizes if a scratchpad allocated with our technique is used instead of a loop cache. Consequently, on the average a scratchpad based system reduces energy consumption by 24% and 52% compared with a loop cache based system for the *epic* and the *mpeg* benchmarks, respectively. Average reductions of 6% and 18% in the execution time are also reported for the *epic* and the *mpeg* benchmarks, respectively.

Finally, we present an overall comparison of the scratchpad based system with the loop cache based system for all benchmarks. The scratchpad present in the system is allocated using the Opt. CASA and Near-Opt. CASA approaches. Figure 5.24 displays the energy consumption of scratchpad based systems relative to that of the loop cache based systems, which is represented as the 100% baseline. Similar to Figure 5.20, the instruction cache size for the current results is set to 128, 1k, 1k, 1k and 2k bytes (*cf.* Table 5.10) for benchmarks *adpcm*, *epic*, *g721*, *media* and *mpeg*, respectively. The energy and performance values (*cf.* Figure 5.24) are the average values obtained by varying the scratchpad size and the loop cache size in the range of 64 bytes and 1k bytes.

We make a few observations from Figure 5.24. Firstly, except for the *adpcm* benchmark, the scratchpad based memory hierarchies fare better in terms of energy consumption than those based on the loop cache. The *adpcm* benchmark is a small benchmark with only two frequently executed loops. The loop cache is able to store all the memory objects belonging to these loops. The Opt. CASA approach identifies the same objects, whereas the greedy heuristic based Near-Opt. CASA fails to identify the correct memory objects.

Secondly, the scratchpad based memory hierarchy consumes much less energy for benchmarks with large code sizes (*viz. mpeg, epic, media* etc.) compared to that consumed by the loop cache based memory hierarchy. The energy consumption of the prior for the *mpeg* benchmark is about 40% of that of the latter. The reason for this behavior is that large benchmarks contain several frequently accesses memory objects, all of which can be stored in the scratchpad. Unlike the loop cache, the scratchpad memory does not impose any constraint on the number of memory objects that can be stored. Additionally, the number of conflict cache misses is higher for larger benchmarks. These misses are optimized by the proposed allocation approaches, providing substantial energy savings compared to a loop cache based system.

Finally, the overall average energy savings for the scratchpad based system with the loop cache based system are about 30% and 20% for the Opt. CASA and Near-Opt. CASA approaches, respectively. In the following subsection, we discuss the experimental results for the multi-processor ARM based system.

5.6.3 Multi-Processor ARM

In this subsection, we present the evaluation of the Opt. CASA approach for the multi-process edge detection benchmark. The benchmark is simulated on the multi-processor ARM simulation platform which allows simulation of a variable number of ARM processors as well as that of a wide variety of memory hierarchies local to each processor. In the present setup, the local memory hierarchy of each ARM processor consists of a 4k bytes direct mapped instruction cache, a 16k bytes 4-way set associative data cache and a scratchpad memory of varying sizes. The multi-processor edge detection benchmark consists of an initiator process, a terminator process and a variable number of compute processes. Each of the processes is mapped to an independent processor and therefore, the processors are named according to the mapped process. The benchmark and simulation framework are described in detail in Section 3.2.

Figure 5.25 presents the total energy consumption values of the system for the multi-process edge detection benchmark when the number of compute processors and the size of the scratchpad memory is varied. The energy consumption of the system without a scratchpad is also shown in the figure. The scratchpad present in the system is allocated using the Opt. CASA approach.

A few important observations can be made from Figure 5.25. Firstly, we observe that increasing the number of compute processors from 1 to 2 leads to a substantial reduction in the energy consumption of the system. However, any further increase in the number of compute processor does not result in substantial energy reductions.

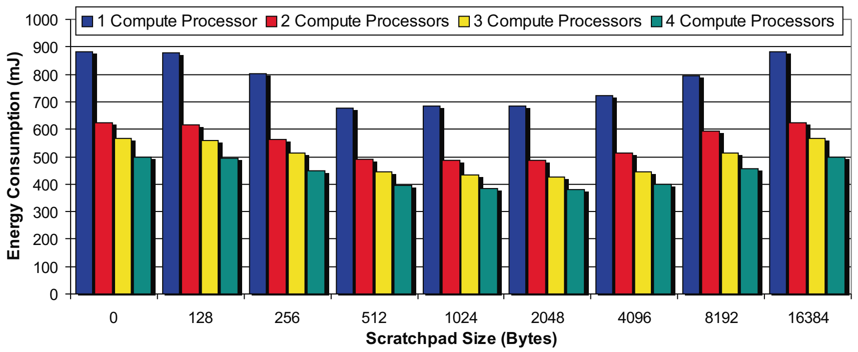


Fig. 5.25. Multi-Process Edge Detection: Energy Consumption for Varying Compute Processors and Scratchpad Sizes (Cycle Latency = 1 Master Cycle)

Secondly, we observe that the energy consumption values for all processors decrease with the increase in scratchpad size until they reach 512 bytes for 1 Compute Processor and 2k bytes for the 2, 3 or 4 compute processors based systems. These scratchpad sizes lead to the minimum energy consumption for the system, any further increase in the scratchpad size also increases the energy consumption of the system.

Thirdly, we observe that the energy consumption of the systems with large scratchpad memories (*e.g.* 16k bytes) is equal to that of the system without a scratchpad. The reason for this behavior is that large scratchpad memories consume high energy per access and therefore, are an unattractive option compared to caches of smaller size for storing memory objects. The Opt. CASA approach does not allocate any memory object to 16k byte scratchpads in the present setup. Finally, it is observed that a 2k bytes scratchpad achieves a reduction of 25% in the total energy consumption of the system compared with a system without a scratchpad.

5.7 Summary

The essence of this chapter is that equal emphasis should be given to both the novel memories and also to the allocation algorithms which lead to their proper utilization. In this chapter, we demonstrated that the addition of a scratchpad to an instruction cache leads to substantial savings in the energy consumption and the execution time of the application for a uniprocessor ARM based system given that an appropriate allocation approach is used.

We reported energy and onchip area reductions of 40% and 75% over the least energy consuming instruction cache configuration found for one benchmark. In addition, the cache aware scratchpad allocation problem was modeled as a generic non-linear optimization problem and was solved optimally using an ILP based approach as well as near-optimally using a heuristic. The near-optimal solutions obtained by the heuristic were on an average 6.0% and 4.0% worse than the optimal solutions, in terms of energy consumption and performance, respectively. The proposed cache aware scratchpad allocation approaches reduce both then energy consumption of the system and the execution time of the applications compared to the scratchpad allocation presented in the previous chapter. Average reductions of 23.4% and 7.0% in energy consumption and execution time were reported for the ILP based approach. We also determined the Pareto-optimal scratchpad sizes for one benchmark.

In addition, we demonstrated that the simple scratchpad memory allocated with the presented approaches outperforms a preloaded loop cache. Average reductions of 29.4% and 8.7% in energy consumption and execution time, respectively, were also reported. The presented approaches were also used to allocate both instruction segments and data variables onto the scratchpad memories of a multi-processor ARM based system. Our experiments for the proposed approach report up to 25% reduction in the total energy consumption of the multi-processor system.

The approaches presented in this chapter were published in [127], [129], and [130].

6

Scratchpad Overlay Approaches for Main / Scratchpad Memory Hierarchy

In the previous two chapters, the proposed allocation approaches assigned an optimal set of memory objects to disjoint address regions on the scratchpad memory. These memory objects then remain assigned to their respective address regions for the entire execution time of the application. In contrast, the allocation approaches presented in this chapter assign memory objects to the address regions such that two or more memory objects may be assigned to overlapping (non-disjoint) address regions if they are never used at the same execution time instant. In other words, memory objects are overlayed on the scratchpad memory if they have disjoint live-ranges. The current chapter presents the allocation approaches for a simple memory hierarchy consisting of an L1 scratchpad memory and a background main memory.

In the following, a brief introduction to the scratchpad overlay approaches is presented, followed by the presentation of a motivating example in Section 6.2. Section 6.3 presents a survey of work related to register and memory allocation approaches. In Section 6.4, the formal definition of the scratchpad overlay problem and the description of the preliminaries is presented. Section 6.5 presents optimal and near-optimal solutions to the scratchpad overlay problem. Experimental results to evaluate the scratchpad overlay approaches for three different system architectures are presented in Section 6.6. Section 6.7 concludes the chapter with a short summary.

6.1 Introduction

The scratchpad overlay approaches reduce the energy consumption of the system by overlaying memory objects with non-conflicting live-ranges onto the scratchpad. In addition, the approaches orchestrate the movement of memory objects within the memory hierarchy through the insertion of spill instructions at appropriate locations in the application code. Consider the application code fragment containing two equal sized arrays `A` and `B`, presented in Figure 6.1. For the sake of simplicity, assume that the allocation approaches are restricted to assign only arrays onto the scratchpad and that the scratchpad is large enough to contain only one of the two array variables at the same time.

```
   #define SIZE 10
1  for (i=0;i<SIZE;i++) {
2    for (j=0;j<SIZE;j++) {
3      A[i]= ...; }}
4  for (i=0;i<SIZE;i++) {
5    for (j=0;j<SIZE;j++) {
6      ... = A[i]; }}
7  for (i=0;i<SIZE;i++) {
8    for (j=0;j<SIZE;j++) {
9      B[i]= ...; }}
10 for (i=0;i<SIZE;i++) {
11   for (j=0;j<SIZE;j++) {
12     ... = B[i]; }}
```

```
   #define SIZE 10
1  spill_load(A);
2  for (i=0;i<SIZE;i++) {
3    for (j=0;j<SIZE;j++) {
4      A[i]= ...; }}
5  for (i=0;i<SIZE;i++) {
6    for (j=0;j<SIZE;j++) {
7      ... = A[i]; }}
8  spill_store(A);
9  spill_load(B);
10 for (i=0;i<SIZE;i++) {
11   for (j=0;j<SIZE;j++) {
12     B[i]= ...; }}
13 for (i=0;i<SIZE;i++) {
14   for (j=0;j<SIZE;j++) {
15     ... = B[i]; }}
16 spill_store(B);
```

Fig. 6.1. Example and Overlayed Application Code Fragments

The non-overlayed scratchpad allocation approach presented in Chapter 4 will allocate one of the two arrays on to the scratchpad. In contrast, the scratchpad overlay approach will assign both arrays to the scratchpad, realizing that the two arrays have disjoint live-ranges and are never accessed at the same time. For the example code fragment presented in Figure 6.1, the live-range of array `A` spans from the statement on line 1 to the statement on line 6. Similarly, the live-range of array `B` spans from the statement on line 7 to the statement on line 12.

Figure 6.1 also presents the application code fragment modified by the overlay approach. The `spill_load` (`spill_store`) routines copy the arrays to (from) the SPM from (to) the main memory, respectively. The overlay approach achieves energy benefit through the improved utilization of the scratchpad. On the other hand, it also incurs an energy overhead due to execution of the spill routines. The approach reduces the overall energy consumption of the system if the energy benefit is greater than the energy overhead due to spill routines.

An important observation that should be made at this point is that the scratchpad overlay problem is similar to the well known *global register allocation problem* [52]. Similar to the scratchpad overlay problem, the register allocation problem assigns the compiler generated symbolic variables to the registers in the register file of the processor while considering the live-ranges of the variables. A detailed comparison of the two problems is presented in Section 6.4.

In this chapter, optimal and near-optimal approaches to solve the scratchpad overlay problem for a memory hierarchy composed of the scratchpad and the main memory. Optimal Scratchpad Overlay (Opt. SO) and Near-Optimal Scratchpad Overlay (Near-Opt. SO) approaches overlay instruction segments as well as global variables onto the scratchpad memory. The approaches generate overlays [52] or memory objects from the application code and divide the scratchpad overlay problem in two smaller problems.

The goal of the first problem *viz. Memory Assignment Problem* is to determine the assignment of memory objects to the scratchpad memory such that the energy consumption of the system is minimized. The problem also determines spill locations which cause the least energy overhead. Both Opt. SO and Near-Opt. SO approaches use an 0-1 ILP based approach to determine the best set of memory objects assigned to the scratchpad memory. The motivation for such an approach is based on the empirical observation that the optimal solution to the ILP formulation of the memory assignment problem can be determined in $O(n^{1.3})$ time [8]. This observation was also corroborated with our experiments.

The second problem *viz. Address Assignment Problem*, computes the addresses of the memory objects assigned to the scratchpad memory such that the scratchpad space is shared by memory objects which are not accessed at the same execution time instance. The second problem is solved optimally through an ILP formulation by the Opt. SO approach and near-optimally through a first-fit heuristic by the Near-Opt. SO approach. The ILP formulation of the address assignment problem requires considerable time to compute the solution for large scratchpad sizes and large benchmarks. Therefore, the Near-Opt. SO approach replaces the ILP formulation by a first-fit heuristic based approach. This is motivated by the observation that the first-fit heuristic achieves close to optimal allocation for real-life benchmarks [62]. For our set of benchmarks, the Near-Opt. SO approach achieved near optimal results in negligible computation time. In the following section, we present the benefit of overlaying the scratchpad memory with help of a real-life example.

6.2 Motivating Example

We start by presenting a motivating example to demonstrate that real-life applications consist of multiple hot-spots. These hot-spots have non-conflicting live ranges and can be overlayed on the scratchpad to reduce the energy consumption of the application. Figure 6.2 presents the workflow of the Edge Detection application which determines edges in a tomographic image. The application consists of three sequential steps called GaussBlur, ComputeEdges and DetectRoots. Each of these steps processes a given input image and writes the resulting image as output which is then passed to the next stage in the workflow. A detailed description of the application can be found on page 33 of Section 4.2.

The execution profile of the Edge Detection application is presented in Figure 6.3. We have scaled down the input image to speedup the profiling of the application. A point (x,y) in the figure represents that the x^{th} executed instruction in the instruction trace of the application was fetched from the address y in the memory. The dark regions in Figure 6.3 correspond to the execution of the stages of the Edge Detection application. For example, the largest region in the center of the figure corresponds to the execution of the ComputeEdges stage of the application. From Figure 6.3, we observe that each stage of the application is a hot-spot and that the stages do not interfere with each other. Hence, the

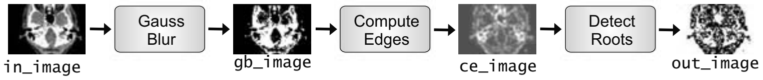


Fig. 6.2. Workflow of Edge Detection Application

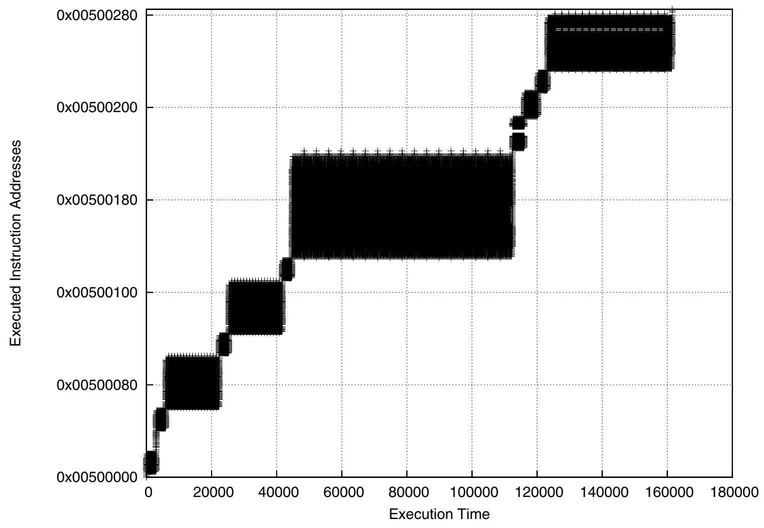


Fig. 6.3. Execution Profile of Edge Detection Application (without ReadImage and WriteImage Routines)

contents of each stage, *i.e.* code segments and variables, can be overlayed onto the scratchpad. However, they need to copied on and off the scratchpad before entering and after leaving the stages.

In Section 4.2 of Chapter 4, we demonstrated that the set of memory objects should contain both code segments and variables in order to achieve the minimize the total energy consumption. Therefore, in the present chapter as well we will consider both code segments and traces as memory objects.

6.3 Related Work

Global Register Allocation is one of the most researched and fundamental topics in code optimization and compiler construction [52]. A compiler initially generates code assuming an infinite number of *symbolic registers* which have to be assigned to the limited number of the processor's *real registers*. Global register allocation attempts to find an assignment of the symbolic registers to the processor's real registers such that a maximum number of *symbolic registers* is assigned to the *real registers*. The allocation problem was proven to be NP-complete [43]. Most of the register allocators [22, 109] are based on the graph coloring heuristic [25]. In the recent past, optimal approaches [42, 47] to solve the register allocation problem have been proposed. Although global register allocation is NP-complete for arbitrary graphs, coloring of graphs found in real-life programs [31] has been demonstrated to be easier. A study by the authors [47] empirically demonstrated that it takes $O(n^3)$ to optimally solve the register allocation problem for real-life benchmarks.

Dynamic Storage Allocation (DSA) has been a fundamental part of operating systems for allocating memory to applications [69]. Applications either make requests for memory or release some of the already allocated memory. The job of the allocator is to satisfy requests for memory from applications such that the total amount of memory required is minimized. The DSA problem has also been proven to be NP-complete [43]. Several heuristic based allocation approaches, *e.g.* first-fit, next-fit, best-fit have been proposed. The authors of [62] present a good survey and comparison of the various approaches.

The research on scratchpad utilization for single process applications can be classified into two broad categories *viz.* non-overlay and overlay based allocation techniques. In the former, the scratchpad is loaded once at the start and its contents remain invariant during the entire execution period of the application. In contrast, overlay based allocation techniques partition the application into overlays [52]. These overlays are copied on and off the scratchpad during the execution to capture the dynamic behavior of the application. A detailed survey of non-overlayed scratchpad allocation techniques can be found in Section 4.3 and Section 5.2.

The scratchpad overlay approaches can be classified into two broad categories, the approaches which overlay only data elements [23, 64, 96] and those which overlay only instructions [6, 103, 115]. The approaches presented in this chapter overlay both variables and instructions onto the scratchpad memory. First, we present a discussion on data-only overlay approaches, followed by that on instruction-only overlay approaches. In the end, a brief comparison of the approaches proposed in this chapter with the related work is presented.

Authors [64] and [23] present two similar approaches for overlaying array tiles on the scratchpad memory. Both approaches are able to optimize the application code for a multi-level memory hierarchy. The approach [64] utilizes reuse vectors and reuse matrices to determine the appropriate array tile sizes and to manage the movement of these array tiles within the memory hierarchy. In addition, it can also generate a scratchpad based memory hierarchy optimized for the input application. However, the approach does not consider the overhead due to the copying of array tiles within the memory hierarchy. The other disadvantage of the approach is that it independently optimizes each loop nest, therefore can not overlay array tiles across loop nests.

The second approach [23] generates all potential tiles or copy-candidates from the arrays present in the application. The approach generates a mapping of copy-candidates to memories after taking into consideration their live-ranges, the benefits obtained by mapping them to different memories and the overhead caused due to the movement of these copy-candidates within the memory hierarchy. The limitation of the approach presents a data centric view of an application and the effect of the approach on instruction memory or processor energy consumption is not evaluated.

Ozturk et al. [96] presented an interesting approach in which data compression is used to create space on the scratchpad. Thus, upon the assignment of an array tile to the scratchpad, the array tiles presently occupying the scratchpad space are either spilled to the main memory or compressed and stored back on the scratchpad. The precondition to compression is that the array tiles have to be decompressed prior to accessing them from the scratchpad. The approach selects compression over spilling if the overhead due to decompressing the array tile is smaller than accessing the uncompressed array tile from the main memory. However, the main drawback with this approach is that the size of the compressed array tiles can not be estimated at compile-time.

The first work on overlaying instruction segments onto the scratchpad is by Steinke et al. [115]. The approach considered overlaying of frequently accessed code segments *viz.* hot-spots and preselects certain points, *e.g.* loop-entry points in the application as potential copy-points. An ILP based approach determines the mapping of hot-spots to the scratchpad and also selects the copy-points required to copy the hot-spots onto the scratchpad.

Angiolini et al. [6] proposed a link-time approach to overlay code segments identified from the application binary. The authors present a dynamic programming based pseudo-polynomial time algorithm to determine the overlay of code segments. The advantages of the approach are that code-segments can be selected at any arbitrary granularity and it can also optimize proprietary applications for which the source code may be unavailable. However, the executable patching tools are heavily dependent on the instruction set architecture of the processor.

The approach [103] describes a heuristic based approach for overlaying instruction segments onto the scratchpad memory. The approach [103] and the approaches presented in this chapter are very similar in their analysis of instruction segments. The approaches generate traces from the application code and analyze the inter-procedural control flow graph of the application to overlay these traces onto the scratchpad memory. Therefore, the approaches are able to determine spill locations across function boundaries causing the least possible energy overhead.

A recent approach [35] proposes a memory architecture consisting of a scratchpad memory along with a dedicated memory management unit (MMU). In this setup, the processor issues virtual addresses and the scratchpad memory is assigned to the physical address space. The MMU stores the virtual-to-physical memory mappings for the application and maps virtual address accesses to physical memory accesses. The MMU allocates the scratchpad memory with 1 kB pages from code segments. In addition, the authors propose a profile guided annotation strategy to mark the pages which should be copied to the scratchpad memory at runtime.

The scratchpad overlay approaches presented in this chapter are the only approaches which can overlay both instruction segments and data variables on to the scratchpad memory. Unlike the other approaches, the presented approaches accurately model the energy benefits due to overlaying and the energy overhead due to insertion of copy-routines. Experiments are conducted to report the total energy consumption of the system, unlike the other approaches which present a piecemeal view of the system.

The overlay approaches [23, 64, 96] are superior than the approaches proposed in this chapter if they are restricted to overlaying only array variables. However, the proposed overlay approaches are far better than the data centric overlay approaches in minimizing the total energy consumption of the system. Nevertheless, an extension of the proposed approaches to overlay array tiles along with instruction segments will definitely improve their efficacy and is part of the future work. In the following section, the preliminaries are described and the scratchpad overlay problem is formally defined.

6.4 Problem Formulation and Analysis

The objective of the scratchpad overlay problem is to minimize the energy consumption of the application through the overlay of memory objects onto the scratchpad. Memory objects which are not required at the same time during the execution of the application are overlayed or assigned to overlapping address regions on the scratchpad memory. During the execution of the application, the memory objects are copied on and off the scratchpad memory. Therefore, an additional objective of the problem is to determine locations in the application code to insert copy routines such that the energy overhead due to the copy routines is minimized.

The scratchpad overlay problem is similar to the global register allocation problem for CISC microprocessors. Unlike RISC microprocessors, CISC microprocessors are relaxed such that one or more operands of an instruction can be a memory location. This reduces the pressure on the register file and also reduces the number of unnecessary load and store instructions. The scratchpad memory in the current setup is similar to the register file of a CISC microprocessor, as besides the main memory, the scratchpad memory acts as an alternative location for storing and retrieving both instructions and data. The high cost of accessing the main memory serves as a reason for the optimal utilization of the register file or the scratchpad memory.

The scratchpad overlay problem is also different from the global register allocation problem in a few small aspects. First, the register allocation problem requires all symbolic registers to be unit sized, while the overlay problem assigns memory objects of different sizes to the scratchpad memory. Second, the overlay problem allocates both instructions and data variables onto the scratchpad, whereas the register allocation problem assigns symbolic registers to the real register. In spite of the differences, the scratchpad overlay problem can be considered as the weighted version of the global register allocation problem, which has been proved NP-complete [8].

6.4.1 Preliminaries

As described earlier, the overlay problem is defined for a inter-procedural control flow graph $G(N,E)$, created by combining the local control flow graph $LG_i(N_i,E_i)$ of every function f_i constituting the application. Additional edges representing the call and the return relationship between every caller and callee functions are added to the graph $G(N,E)$. These edges model the flow of control between the caller and the callee functions by connecting their local control flow graphs.

Definition 6.1 (Inter-Procedural Control Flow Graph). *The Inter-Procedural Control Flow Graph (IPCFG) $G(N,E)$ is the representation of the entire application. The graph $G(N,E)$, an edge and node weighted directed graph, is created by combining control flow graph $G_i(N_i,E_i)$ of every function f_i constituting the application and is defined as follows:*

(a)	N	*node set is the union of node sets of all $G_i(N_i,E_i)$* $N = N_1 \cup N_2 \cup \cdots \cup N_n$
(b)	E	*set of inter-procedural control flow edges* $E = E_1 \cup E_2 \cup \cdots \cup E_n \cup E^{CALL} \cup E^{RET}$
(c)	$e_{ij} \in E^{CALL}$	*a directed edge from the calling node n_i of the caller function to the source node* ***source***$_j$ *of the called function*
(d)	$e_{ij} \in E^{RET}$	*a directed edge from the sink node* ***sink***$_i$ *of the called function to the calling node n_j of the caller function*

(e)	***source***	*source node of the entry routine (main function) is represented as the source node of the flow graph $G(N,E)$*
(f)	***sink***	*sink node of the entry routine (main function) is represented as the sink node of the flow graph $G(N,E)$*
(g)	$w(n)$	*weight of node $n \in N$ which represents the execution count of the node*
(h)	$w(e)$	*weight of edge $e \in E$ which represents the execution count of the edge*

6.4.2 Memory Objects

The memory optimization approaches proposed in this chapter overlay both code segments and variables onto the scratchpad memory in the system. The memory objects for the uni-processor ARM based system consist of the following:

(a) Global variables (V) including *scalar* and *non-scalar* variables.
(b) Code segments including *traces* (T) and *functions* (F).

The following are the memory objects for the multi-processor ARM based system:

(a) Global variables (V) including *scalar* and *non-scalar* variables.
(b) Code segments including only *functions* (F).

The scratchpad overlay approach for the multi-processor ARM based system is implemented as a source-level transformation. Therefore, the memory objects include code segments at the coarse granularity of *functions*. For an M5 DSP based system, the overlay approach optimizes the data memory hierarchy. Consequently, the memory objects consist of only *global data arrays* found in DSP applications. Table 6.1 summarizes the memory objects for the scratchpad overlay approach applied to the different system architectures. The set of memory objects for the uni-processor ARM based system is the super-set of memory objects for the other systems. Therefore, the scratchpad overlay approaches will be described for the uni-processor ARM based system.

6.4.3 Liveness Analysis

Definition 6.2 (Liveness). *A memory object mo is* live *at an edge $e \in E$ of the control flow graph $G(N,E)$ if there exists a back path from the edge e to a node $n \in N$ where the memory object is defined without being redefined at any other node along the path. $Live(mo) \subseteq E$ represents the set of edges on which the memory object mo is live.*

Informally, a memory object being *live* at a point implies that it will be required at some point later during the execution of the application, so the analysis is called *liveness analysis*.

Memory Optimization	System Architecture	Memory Objects	Explanation
Scratchpad Overlay for MM / SPM Hierarchy (Chapter 6)	Uni-processor ARM	$MO \subseteq V \cup T \cup F$	global variables, traces and functions
	M5 DSP	$MO \subseteq V$	global data arrays
	Multi-processor ARM	$MO \subseteq V \cup F$	global variables and functions

Table 6.1. Memory Objects for Scratchpad Overlay Approach

A *Live-Range* $Live(mo) \subseteq E$ of a memory object mo is a set of edges at which the memory object is *live*.

Basic blocks present in the application code contain statements which refer (*i.e.* modify or use) to the memory objects. This information, *viz.* reference R, is attached to the nodes of the control flow graph of the application. The precondition to compute the live-range for a memory object mo is that each reference R to the memory object should be classified as a DEF, MOD or USE reference. The classification procedure, presented below, for references to global variables is a little different than that to code segments.

A reference R = (mo, s_r, s_w, n_{rs}, n_{ws}, $class$) is a *6-tuple* or a *sextuple* and contains the name of the memory object, the read (s_r) and write (s_w) subreferences, the number of distinct read (n_{rs}) and write (n_{ws}) subreferences referred on each access and the classification ($class \in$ {DEF, MOD, USE}). A read s_r and write s_w subreference to a global variable V is the set of scalar elements of the variable V read and written by the reference, respectively. A pair of subreferences may be disjoint (*i.e.* refer to distinct scalar elements of variable V) or conjoint (*i.e.* not disjoint). A scalar variable is assumed to be a non-scalar variable consisting of a single element. A reference R is classified into one of the following categories:

(a) DEF : if the write subreference s_w definitely changes all the scalar elements of the variable V (*i.e.* $s_w = V$).
(b) MOD : if the write subreference s_w changes some but not all scalar elements of the variable V (*i.e.* $s_w \subset V$).
(c) USE : if the subreference uses scalar elements of the variable V.

If a node has more than one reference to the same variable then the following priority order is used to determine the appropriate classification.

$$DEF > MOD > USE$$

The priority order guarantees that the liveness of a variable is correctly preserved. Next, we explain the classification of references to variables with the help of an example.

Example 6.3. Please refer to the example application code and its control flow graph presented in Figure 6.4. The example application processes two array variables `A[SIZE]` and `B[SIZE]`. The nodes of the control flow graph contain references (R1,...,R6) to array variables `A` and `B`. Table 6.2 presents the attributes and the classification of the write subreferences. From the table, we observe that reference R1 = (`A`, s_{r1}, s_{w1}, 0, 1, DEF) is classified as a DEF reference because the write subreference $s_{w1} = \{A[0],\ldots,A[SIZE-1]\} =$ `A` assigns values to all the scalar elements of variable `A`. Similarly, reference R2 = (`B`, s_{r2}, s_{w2}, 0, 1, DEF), defining variable `B`, is also classified as a DEF reference. The number of write subreferences per access n_{ws} is 1 in both of the above scenarios.

Reference R3 = (`A`, s_{r3}, s_{w3}, 0, 1, MOD) is classified as a MOD reference, as the write subreference $s_{w3} = \{A[0],A[2],\ldots\} \subset$ `A` assigns values to only the even elements of variable `A`. Similarly, reference R4 is also classified as a MOD reference, the write subreference $s_{w4} = \{B[1],B[3],\ldots\} \subset$ `B` modifies the odd elements of array `B`. The number of distinct read and write subreferences per access is 1 for reference R4. Finally, references R5 and R6

```
#define SIZE 10
for (i=0;i<SIZE;i++) {
  A[i]=i;
  B[i]=i;
}
if (even) {
  for (i=0;i<SIZE;i+=2) {
     A[i]=2*i;
  }
} else {
  for (i=1;i<SIZE;i+=2) {
     B[i]=B[i-1]+i;
  }
}
for (i=0;i<SIZE;i++) {
  sum=sum+A[i]*B[i];
}
```

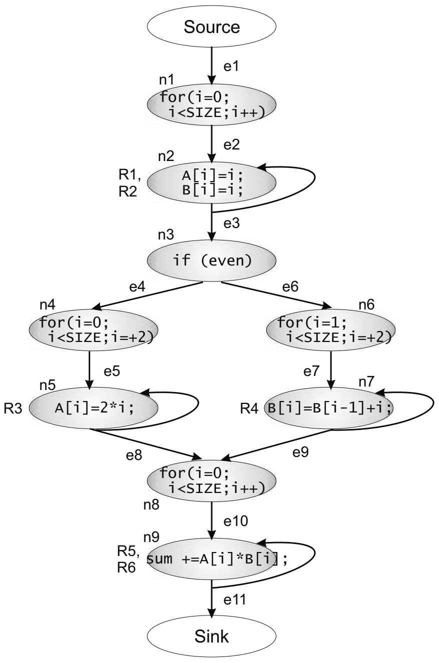


Fig. 6.4. Example Application Code and the Corresponding Control Flow Graph

Reference	Variable	Write Subreference	n_{rs}	n_{ws}	Classification
R1	A	$s_{w1} = \{A[0], \ldots, A[SIZE-1]\} = \mathtt{A}$	0	1	DEF
R2	B	$s_{w2} = \{B[0], \ldots, B[SIZE-1]\} = \mathtt{B}$	0	1	DEF
R3	A	$s_{w3} = \{A[0], A[2], \ldots\} \subset \mathtt{A}$	0	1	MOD
R4	B	$s_{w4} = \{B[1], B[3], \ldots\} \subset \mathtt{B}$	1	1	MOD
R5	A	$s_{r5} = \{A[0], \ldots, A[SIZE-1]\} = \mathtt{A}$	1	0	USE
R6	B	$s_{r6} = \{B[0], \ldots, B[SIZE-1]\} = \mathtt{B}$	1	0	USE

Table 6.2. Attributes of References for Global Variables

are classified as USE references since read subreferences s_{r5} and s_{r6} utilize the values of scalar elements of variables A and B, respectively.

Every basic block present in the application contains a single reference R = (mo, s_r, s_w, n_{rs}, n_{ws}, $class$) to the corresponding code segment *i.e.* trace and function. Please refer to Section 3.1.3 for a formal definition of a trace. The read subreference s_r to a code segment is the set of instructions belonging to the basic block containing the reference R. The classification procedure of references to code segments is much simpler than that for global variables. The code segments reside in the read-only region of the application and the processor only executes, or in other words uses, the instructions present in the code segments. Therefore, all references to code segments are classified as USE references. Example 6.4 explains the classification of references to code segments.

Example 6.4. Figure 6.5 presents the control flow graph of the example application code presented in Example 6.3. Additionally, the figure displays the traces (T1,...,T5) and the references (R7,...,R15).

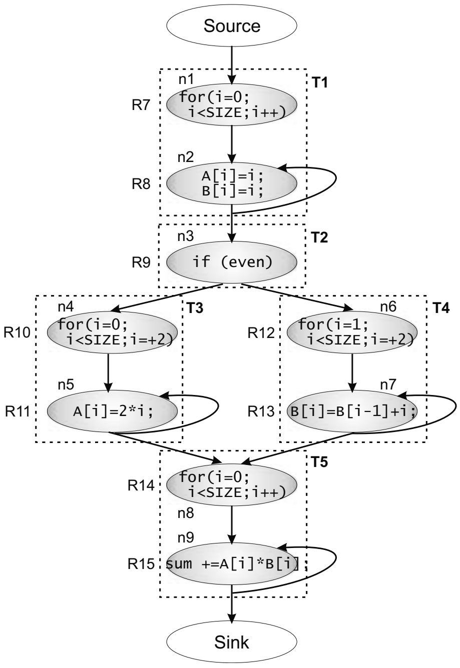


Fig. 6.5. Control Flow Graph Displaying Traces

Reference	Trace	Read Subreference	n_{rs}	n_{ws}	Classification
R7	T1	$s_{r7} = inst(\text{BB1}) \subset inst(\text{T1})$	$\|inst(\text{BB1})\|$	0	USE
R9	T2	$s_{r9} = inst(\text{BB3}) = inst(\text{T2})$	$\|inst(\text{BB3})\|$	0	USE
R10	T3	$s_{r10} = inst(\text{BB4}) \subset inst(\text{T3})$	$\|inst(\text{BB4})\|$	0	USE
R12	T4	$s_{r12} = inst(\text{BB6}) \subset inst(\text{T4})$	$\|inst(\text{BB6})\|$	0	USE
R13	T4	$s_{r13} = inst(\text{BB7}) \subset inst(\text{T4})$	$\|inst(\text{BB7})\|$	0	USE
R14	T5	$s_{r14} = inst(\text{BB8}) \subset inst(\text{T5})$	$\|inst(\text{BB8})\|$	0	USE

Table 6.3. Attributes of References for Traces

From the figure and Table 6.3, we observe that each node contains a single reference to the enclosing trace. Node BB4 contains reference R10 = (T3, s_{r10}, s_{w10}, $|inst(\text{BB4})|$, 0, USE) to trace T3 as the node BB4 is contained within trace T3. The read subreference $s_{r10} = inst(\text{BB4}) \subset inst(\text{T3})$ is the set of instructions belonging to node BB4. The reference R10 is classified as a USE reference as the instructions belonging to node BB4 are only executed (or used) by the processor. Moreover, the number of distinct read subreferences n_{rs} for reference R10 is the number of instructions in node BB4 $|inst(\text{BB4})|$. The number of write subreferences for a reference to a code segment is always is 0.

In order to compute the liveness of a memory object mo, *live-in* and *live-out* attributes, attached to the nodes of the control flow graph, are computed for the memory object. The formal definition of *live-in* and *live-out* attributes is presented below.

Definition 6.5 (Live-In). *Live-in attribute $LiveIn(n) \subseteq MO$ for node n is the set of memory objects mo which are live at any of the edges entering the node n.*

Definition 6.6 (Live-Out). *Live-out attribute $LiveOut(n) \subseteq MO$ for node n is the set of memory objects mo which are live at any of the edges exiting the node n.*

The dataflow equations presented below are used to determine the live-in and live-out attributes for all the nodes. Traditional dataflow analysis for scalar variables does not model MOD attributes required for computing the live-in and live-out attributes involving non-scalar variables. Thus, the standard dataflow equations [93] are extended to incorporate the MOD attribute. The extended dataflow equations for node n are the following:

$$LiveIn(n) = USE(n) \cup MOD(n) \cup (LiveOut(n) - DEF(n)) \tag{6.1}$$

$$LiveOut(n) = \bigcup_{s \in Succ(n)} LiveIn(s) \tag{6.2}$$

where $USE(n) \subseteq MO$ is the set of memory objects with USE reference at node n. Similarly, $DEF(n)$ and $MOD(n)$ are the set of memory objects with DEF-reference and MOD-reference at node n, respectively. A backward iterative dataflow analysis algorithm [93] was used to compute $LiveIn(n)$ and $LiveOut(n)$ for all nodes n of the control flow graph.

The following corollary presents the relationship between the liveness of a memory object mo at an edge e and the *live-in* and *live-out* attributes defined at the destination and source nodes of the edge e, respectively.

Corollary 6.7. *A memory object mo is* live *at an edge e iff the memory object mo is* live-out *at the source node of the edge e and is* live-in *at the destination node of the edge e.*

Example 6.8. Figure 6.6 displays the live-in and live-out attributes for the nodes present in the control flow graph of the example application. The live-in attribute $LiveIn(source)$ for the source node and the live-out attribute $LiveOut(sink)$ for the sink node of the control flow graph are empty sets. This is because of the fact that no memory object can live before the start of the application or live after the end of the application.

The backward dataflow analysis algorithm sets the live-in $LiveIn(n_9) = \{\mathtt{A}, \mathtt{B}, T5\}$ of the node n_9 in the first pass over the control flow graph. Similarly, live-in attributes of nodes n_7, n_5 and n_2 are set to $\{\mathtt{A}, \mathtt{B}, T4, T5\}$, $\{\mathtt{A}, \mathtt{B}, T3, T5\}$ and $\{T1, T2, T3, T4, T5\}$, respectively, during the first pass of the dataflow analysis algorithm. The algorithm then iterates over the control flow graph until all live-in and live-out attributes stabilize to a constant value. Figure 6.6 shows the live-in and live-out attributes for the nodes after the dataflow analysis algorithm has finished.

The corollary 6.7 and the above dataflow equations are used to compute the live-range of the memory objects. The live-range of array A, $live(\mathtt{A}) = \{e3, e4, e5, e6, e7, e8, e9, e10\}$, extends from the edge exiting the defining node until the edge entering the last use node.

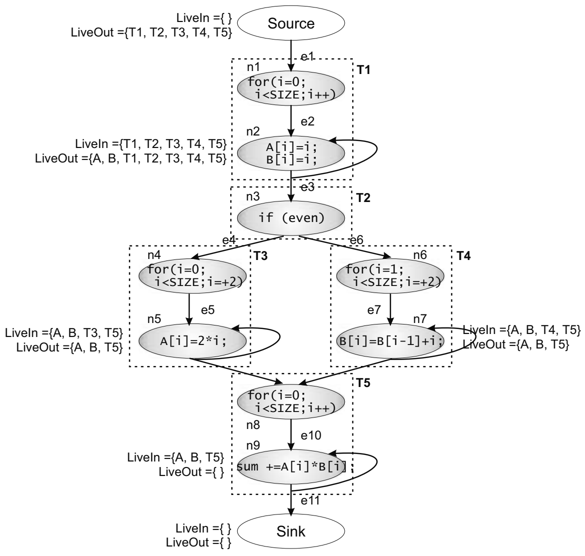


Fig. 6.6. Control Flow Graph Displaying $LiveIn$ and $LiveOut$ Attributes

Variables/Traces	Memory Object (mo)	Live-Range $live(mo)$
A	mo_1	e3, e4, e5, e6, e7, e8, e9, e10
B	mo_2	e3, e4, e5, e6, e7, e8, e9, e10
T1	mo_3	e1, e2
T2	mo_4	e1, e2, e3
T3	mo_5	e1, e2, e3, e4, e5
T4	mo_6	e1, e2, e3, e6, e7
T5	mo_7	e1, e2, e3, e4, e5, e6, e7, e8, e9, e10

Table 6.4. Live-Ranges of Memory Objects

Table 6.4 summarizes the mapping of variables and traces to memory objects and live-ranges of memory objects.

6.4.4 Energy Model

As described in Chapter 3, the energy model [114] is used to compute the energy function $E(inst, imem, dmem)$ which returns the total energy dissipated by the system during the execution of the instruction $inst$ fetched from the instruction memory $imem$ and possibly accessing data from the data memory $dmem$. The components of the energy function $E(inst, imem, dmem)$ are presented as the following:

$$E(inst, imem, dmem) = E_{if}(imem) + E_{ex}(inst) + E_{da}(dmem) \tag{6.3}$$

Complete details regarding the above equation can be found in Subsection 3.1.1. These energy values are used to compute the energy dissipation function $E(mo, R, mem)$ of the memory object mo. The energy function $E(mo,R,mem)$, presented in the following, returns the total system energy dissipated during a single execution of basic block represented by IPCFG node n which contains reference R to the memory object mo.

$$E(mo,R,mem) = \begin{cases} E_{var}(mo,R,mem) & \text{if } mo \in V \\ E_{inst}(mo,R,mem) & \text{if } mo \in T \cup F \end{cases} \tag{6.4}$$

The above equation implies that the total energy dissipated by accessing variables is different than that dissipated through the execution of instruction segments (*e.g.* traces and functions). The energy functions $E_{var}(mo,R,mem)$ and $E_{inst}(mo,R,mem)$ as shown in the following, represent the energy dissipated for memory objects belonging to the set of global variables V and to instruction segments $T \cup F$, respectively:

$$\begin{aligned} E_{var}(mo,R,mem) &= n_{rs} * [E(load,MM,mem) - E(mov,MM,mem)] \\ &\quad + n_{ws} * [E(store,MM,mem) - E(mov,MM,mem)] \end{aligned} \tag{6.5}$$

$$E_{inst}(mo,R,mem) = n_{rs} * inst(mo) * E(mov,mem,MM) \tag{6.6}$$

where, n_{rs} and n_{ws} are the number of distinct read and write subreferences to the memory object mo referred on each access or execution, respectively. The energy dissipated due to reading of a variable is equal to the data read access energy $E_{da}(mem)$. In our energy model [114], it is assumed to be the difference in the energy dissipated by a load and a register-move instruction. Similarly, the energy dissipated for writing a variable is the difference in the energy dissipated by a store instruction and a register-move instruction. The total energy dissipated by a variable is the sum of the products of energy for reading and writing the variable with the number of distinct reads n_{rs} and writes n_{rs}, respectively. Energy dissipation due to the execution of a code segment is the product of the number of instructions present in the code segment and the energy dissipation of a register-move instruction. A single average energy value is used to represent the energy dissipation of all CPU instructions because for ARM processors they dissipate almost equal energy.

The product of weight $w(n)$ of the node n and the energy function $E(mo,R,mem)$ results in the total energy dissipated by the memory object mo through the reference R at node n in the IPCFG.

$$E(mo,mem) = \sum_{R} (w(n) * E(mo,R,mem)) \tag{6.7}$$

Spilling a memory object mo requires copying the memory object from the source memory $smem$ to the destination memory $dmem$. The spill energy function $E_{spill}(mo, smem,dmem)$, presented in the following, computes the energy dissipated during the execution of the spill routine.

$$\begin{aligned} E_{spill}(mo,smem,dmem) &= size(mo) * [E(load,MM,smem) - E(mov,MM,smem)] \\ &\quad + size(mo) * [E(store,MM,dmem) - E(mov,MM,dmem)] \\ &\quad + inst(spill(size(mo))) * E(mov,MM,MM) \end{aligned} \tag{6.8}$$

where, $inst(spill(size(mo)))$ returns the number of instructions of the spill routine that are executed to spill a memory object mo of size $size(mo)$. The first two parameters of the above equation represent the energy dissipated due to reading and writing data values, while the third parameter represents the energy dissipated in the processor for executing the spill routine. The following subsection presents the formal definition of the scratchpad overlay problem.

6.4.5 Problem Formulation

Problem 6.9 (Scratchpad Overlay (SO)). Given the set of memory objects MO, the control flow graph $G(N,E)$ of the application, the live-range $Live(mo_i)$ of each memory object $mo_i \in MO$ and two memories (SPM, MM) with known start and end address regions. The problem is to assign a contiguous address range $[a_j^i, a_j^i + size(mo_i)]$ to memory object mo_i on edge $e_j \in E$ such that the total energy profit E_{Total}, which depends upon the address range of memory object mo_i, is maximized.

$$\begin{aligned} E_{Total} = & \sum_{mo_i \in MO} \sum_{e_j \in E} E_{profit}\left(mo_i, e_j, a_j^i\right) \\ & - \sum_{mo_i \in MO} \sum_{e_j \in E} E_{spill}(mo_i, e_j, MM, SPM) \\ & - \sum_{mo_i \in MO} \sum_{e_j \in E} E_{spill}(mo_i, e_j, SPM, MM) \end{aligned} \tag{6.9}$$

The energy profit function $E_{profit} : MO \times E \times \mathbb{N} \rightarrow \mathbb{R}$ as well as the spill energy function $E_{spill} : MO \times E \times [SPM, MM] \times [SPM, MM] \rightarrow \mathbb{R}$ are formally defined in Subsection 6.4.4. The maximization of the total energy profit E_{Total} is to be performed under the following constraints:

(a) The address region $[a_j^i, a_j^i + size(mo_i)]$ assigned to a memory object mo_i should be contained within the address space of the corresponding memory.

$$\begin{aligned} Start_{SPM} \leq a_j^i \leq End_{SPM} - size(mo_i) \text{ XOR} \\ Start_{MM} \leq a_j^i \leq End_{MM} - size(mo_i) \end{aligned} \tag{6.10}$$

(b) If the live-ranges of two memory objects mo_i and mo_j overlap, then the memory objects should be assigned to disjoint address regions on the edges where the overlap of live-ranges occur.

$$\forall e_k \in Live(mo_i) \cap Live(mo_j) : [a_k^i, a_k^i + size(mo_i)] \cap [a_k^j, a_k^j + size(mo_j)] = \emptyset \tag{6.11}$$

(c) A memory object mo can be moved from one memory to another at any edge $e \in Live(mo)$ in its live range without destroying the semantics of the application. A penalty in terms of energy E_{spill} is incurred whenever a memory object is moved at any edge $e \in Live(mo)$.

The scratchpad overlay problem can be simplified by disallowing the spilling of memory objects during their live-ranges. This can be achieved by setting the spill energy function

$E_{spill}(mo_i, e_j, src, dst) = \infty$ to return a very high penalty value irrespective of the input parameters. Such situations arise for systems where movement of data within the memories is expensive or prohibited by design. Systems with flash memory based main memory serve as an example of systems with the above restriction. The problem of scratchpad overlay without spilling (SOWOS) is an NP-complete problem as a reduction of the optimal cost chromatic partitioning (OCCP) problem to the SOWOS problem can be constructed. The OCCP problem has been proven to be NP-complete [61] for a large class of graphs. Thus, we deduce that the SO problem being the generalization of SOWOS problem is also an NP-complete problem.

6.5 Scratchpad Overlay Approaches

In the previous section, the scratchpad overlay (SO) problem was formally presented and also shown to be an NP-complete problem. Our initial experiments concluded that significant computational effort is required to compute an optimal solution for the SO problem in its current form. Therefore, we decided to break the SO problem into two smaller subproblems *viz.* Memory Assignment Problem and Address Assignment Problem. The goal of the memory assignment problem is to maximize the energy profit by an optimal assignment of memory objects to the memories and by the determination of energy optimal locations for the insertion of spill routines. The goal of the address assignment is to compute overlayed addresses of the memory objects assigned to the scratchpad memory.

The Opt. SO approach computes the optimal solution for both the subproblems by using ILP-based approach. Thus, the Opt. SO approach can compute an optimal solution to the SO problem as long as it can determine a valid solution to the second subproblem. However, it may fail to find a valid solution for the address assignment. Our experiments so far have shown that it never failed to compute addresses for memory objects assigned to the scratchpad. The Near-Opt. SO approach computes a near-optimal solution to the SO problem. It uses an ILP-based approach to compute the optimal solution to the first subproblem and uses a first-fit heuristic based approach to compute a near-optimal solution for the second subproblem.

The scratchpad overlay problem is solved using the workflow shown in Figure 6.7. In the first step, variables and code segments from the application code are identified as memory

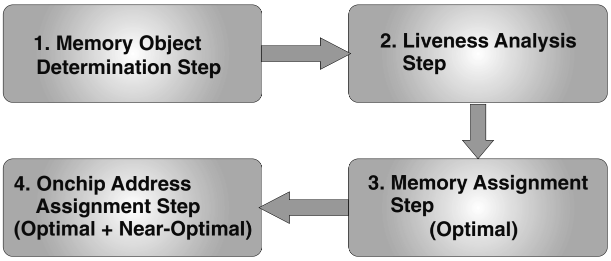

1. Memory Object Determination Step
2. Liveness Analysis Step
3. Memory Assignment Step (Optimal)
4. Onchip Address Assignment Step (Optimal + Near-Optimal)

Fig. 6.7. Workflow of the Scratchpad Overlay Approaches

objects. Liveness analysis is performed in the second step to determine the live ranges of these memory objects. In the third step, the optimal solution to the memory assignment is computed. The final step, depending upon the overall approach, computes an optimal or a near-optimal solution to the address assignment problem.

6.5.1 Optimal Memory Assignment

The memory assignment problem is formulated such that the decision to assign memory objects to the memories is taken at the edges rather than at the nodes of the IPCFG. The edge based formulation enables the determination of the energy optimal points for the spilling of memory objects to the different memories. Prior to presenting the ILP-formulation of the memory assignment problem, we define the following flow-attribute $\mathit{flow}_j^i \in \mathit{Attrib}_{\mathit{flow}} \bigcup \{\mathit{NOFLOW}\} = \{\mathit{DEF}, \mathit{MOD}, \mathit{USE}, \mathit{CONT}, \mathit{NOFLOW}\}$ for every memory object mo_i on each edge e_j of the IPCFG.

$$flow_j^i = \begin{cases} \mathit{DEF} & \text{if } R = \mathit{Reference}(mo_i, Src(e_j)) \text{ and } \mathit{R.class} = \mathit{DEF} \\ \mathit{MOD} & \text{if } R = \mathit{Reference}(mo_i, Dst(e_j)) \text{ and } \mathit{R.class} = \mathit{MOD} \\ \mathit{USE} & \text{if } R = \mathit{Reference}(mo_i, Dst(e_j)) \text{ and } \mathit{R.class} = \mathit{USE} \\ \mathit{CONT} & \text{if } e \in live(mo_i) \\ \mathit{NOFLOW} & \text{if } e \notin live(mo_i) \end{cases} \tag{6.12}$$

Flow-attributes represent the flow of liveness of a memory object on the edges of the IPCFG. A flow-attribute $flow_j^i$ for a memory object mo_i on an edge e_j is classified as a DEF attribute if the reference R to the memory object on the source node of the edge is classified as a DEF reference. Similarly, a flow attribute $flow_j^i$ is classified as a MOD or a USE attribute if the reference R on the destination node of the edge e_j is classified as a MOD or a USE reference, respectively. If a memory object mo_i is live on an edge e_j, then the flow-attribute $flow_j^i$ is classified as a CONT attribute, otherwise as a NOFLOW attribute. If an edge potentially carries more than one flow attributes, then the following priority order is used to determine the appropriate attribute.

$$\mathit{DEF} > \mathit{MOD} > \mathit{USE} > \mathit{CONT} > \mathit{NOFLOW} \tag{6.13}$$

In addition to the flow attributes, spill-load attribute $sl_j^i \in \{\mathit{LOAD}, \mathit{NOSPILL}\}$ and spill-store attribute $ss_j^i \in \{\mathit{STORE}, \mathit{NOSPILL}\}$ are defined on the edges to model appropriate spilling of memory objects. Prior to presenting the classification procedure of spill-load and spill-store attributes, we would like to define *merge* and *diverge* nodes as follows:

Definition 6.10 (Merge Node). *A merge node is a node whose in-degree is greater than one, without counting the self-loop edges.*

Definition 6.11 (Diverge Node). *A diverge node is a node whose out-degree is greater than one, without counting the self-loop edges.*

We will start by presenting the classification of the spill-load attribute, followed by that for the spill-store attribute. The spill-load attribute can be classified as a LOAD or as a

NOSPILL attribute. In order to generate optimal spill code, the following theorem is used to classify a spill-load attribute:

Theorem 1 *For energy optimal spill code generation, it is sufficient to classify the spill-load attribute sl_j^i for memory object mo_i on edge $e_j \in E$ as a LOAD attribute if the edge e_j satisfies any of following two constraints:*

(a) The flow attribute $flow_j^i$ for memory object mo_i on the edge e_j is already classified as a MOD, USE or CONT attribute.
(b) The memory object mo_i is live on the edge e_i whose destination node is a merge node.

The spill-load attribute sl_j^i is classified as a NOSPILL attribute if none of the above two conditions is satisfied.

Proof. By contradiction. Assume that there exists an $e_k \in E$ whose spill-load attribute sl_k^i is classified as a NOSPILL attribute by the theorem and that to generate energy optimal spill code, memory object mo_i must be spill-loaded on the edge e_k. From the edge e_k move in the direction of the next reference R to the memory object, until either an edge e_j whose spill-load attribute is classified as a LOAD attribute by the theorem is reached or a diverge node is reached. Assume that edge e_j whose spill-load attribute is classified as a LOAD attribute is reached. The overhead, quantified in terms of energy dissipated by the spill code, to generate the spill code to load the memory object mo_i at e_j is the same as that to generate the spill code at edge e_k. Moreover, the generation of spill code at edge e_j reduces the time for which the memory object mo_i is assigned to the scratchpad and thereby increases the scratchpad overlay opportunity. Therefore, the edge e_j will be chosen instead of the edge e_k for generating the optimal spill code. Suppose that a diverge node is reached, then the total energy overhead of generating spill code on every edge exiting the diverge node is less than or equal to that of generating spill code at the edge e_k. Therefore, we classify the spill-load attributes on all the exiting edges as LOAD attributes and recursively apply the proof to these edges. □

The mathematical representation of spill-load attribute sl_j^i, as specified by Theorem 1, is presented below:

$$sl_j^i = \begin{cases} \mathit{LOAD} & \text{if } flow_j^i \in \{\mathit{MOD}, \mathit{USE}, \mathit{CONT}\} \text{ or} \\ & (e_j \in live(mo_i) \text{ and } Merge(Dst(e_j))) \\ \mathit{NOSPILL} & \text{otherwise} \end{cases} \tag{6.14}$$

The classification of the spill-load attribute sl_j^i as a LOAD attribute implies that the memory object mo_i can be spill loaded or copied from the main memory to the scratchpad on the edge e_j. In contrast, a NOSPILL attribute implies that spill-loading of the memory object mo_i at the edge e_j will not result in an energy optimal solution. Similar to Theorem 1, the following theorem is used to classify the spill-store ss_j^i into a STORE attribute or a NOSPILL attribute:

Theorem 2 *For energy optimal spill code generation, it is sufficient to classify the spill-store attribute ss_j^i for memory object mo_i on edge $e_j \in E$ as a STORE attribute if the edge e_j satisfies any of following two constraints:*

(a) The flow attribute $flow_j^i$ for memory object mo_i on the edge e_j is already classified as a DEF attribute.
(b) The memory object mo_i is live on the edge e_i whose source node is a diverge node.

The spill-store attribute ss_j^i is classified as a NOSPILL attribute if none of the above two conditions is satisfied.

Proof. By contradiction. Assume that there exists an $e_k \in E$, whose spill-store attribute ss_k^i is classified as a NOSPILL attribute by the theorem and that to generate energy optimal spill code, memory object mo_i must be spill-stored on the edge e_k. From the edge e_k move in the direction of the previous reference R to the memory object mo_i, until either an edge e_j whose spill-store attribute is classified as a STORE attribute by the theorem is reached or a merge node is reached. Suppose that edge e_j whose spill-store attribute is classified as a STORE attribute is reached. The energy overhead to generate the spill code to store the memory object mo_i at e_j is the same as that to generate the spill code at edge e_k. Moreover, the generation of spill store code at edge e_j reduces the time for which the memory object mo_i remains assigned to the scratchpad and thereby increases the opportunity for scratchpad overlay. Therefore, the edge e_j will be chosen instead of the edge e_k for generating optimal spill code. Assume that a merge node is reached, then the total overhead of generating spill code at all the edges entering the merge node is less than or equal to that of generating spill code at the edge e_k. Therefore, we classify the spill-store attributes on all the entering edges as STORE attributes and recursively apply the proof to these edges. □

The classification of the spill-store attribute ss_j^i as a STORE attribute implies that memory object mo_i can be copied from the scratchpad to the main memory on edge e_j. The formal definition of spill-store attribute ss_j^i is presented as the following:

$$ss_j^i = \begin{cases} STORE & \text{if } flow_j^i \in \{DEF\} \text{ or} \\ & (e_j \in live(mo_i) \text{ and } Diverge(Src(e_j))) \\ NOSPILL & \text{otherwise} \end{cases} \tag{6.15}$$

The following example, describes the classification procedure of flow and spill attributes for the application code presented in Example 6.3.

Example 6.12. In this example, Equations 6.12, 6.14 and 6.15 are used to define flow and spill attributes for global variable `A` (*cf.* Example 6.3) and trace T4 (*cf.* Example 6.4). The analysis is performed on the control flow graph of the application code presented in Example 6.3.

Table 6.5 presents the classification of flow, spill-load and spill-store attributes for variable `A` and trace T4. For variable `A`, a NOFLOW flow-attribute and a NOSPILL spill-attribute are defined on edges e_1 and e_2, as the variable is not live on these edges. The trace T4 is live on the same edges and there are no references R to trace T4 on the nodes at the ends of the edges. Therefore, the flow-attributes ($flow_1^6$ and $flow_2^6$) are classified as CONT attributes. The spill-load attributes (sl_1^6 and sl_2^6) are classified as LOAD attributes, whereas, the spill-store attributes (ss_1^6 and ss_2^6) are classified as NOSPILL attributes as none of the conditions for STORE attribute, presented in Equation 6.15, are satisfied.

Edge (e_i)	Attributes for variable A (mo_1)			Attributes for Trace T4 (mo_6)		
	Flow ($flow_i^1$)	Load Spill (ls_i^1)	Store Spill (ss_i^1)	Flow ($flow_i^6$)	Load Spill (ls_i^6)	Store Spill (ss_i^6)
e_1	NOFLOW	NOSPILL	NOSPILL	CONT	LOAD	NOSPILL
e_2	NOFLOW	NOSPILL	NOSPILL	CONT	LOAD	NOSPILL
e_3	DEF	NOSPILL	STORE	CONT	LOAD	NOSPILL
e_5	MOD	LOAD	NOSPILL	NOFLOW	NOSPILL	NOSPILL
e_6	CONT	LOAD	STORE	USE	LOAD	STORE
e_8	CONT	LOAD	NOSPILL	NOFLOW	NOSPILL	NOSPILL
e_{10}	USE	LOAD	NOSPILL	NOFLOW	NOSPILL	NOSPILL

Table 6.5. Definition of Flow and Spill Attributes for Global Variable A and Trace T4

Next, we define a couple of binary variables $x^i_{j\ k}$ and $y^i_{j\ k}$ representing the assignment of memory object mo_i to the scratchpad and the spilling of the memory object mo_i on edge e_i, respectively. The binary variable $x^i_{j\ k}$ represents the flow of memory object mo_i on the scratchpad and is defined as the following:

$$x^i_{j\ k} = \begin{cases} 1 \text{ if } mo_i \text{ is present on the scratchpad memory at edge } e_j \text{ and} \\ \quad flow^i_j \text{ attribute is classified as } at_k \in Attrib_{flow} \\ 0 \text{ otherwise} \end{cases} \tag{6.16}$$

where $mo_i \in MO, e_j \in E$ and $at_k \in Attrib_{flow}$. For example, if the value of binary variable $x^i_{j\ USE}$ is 1, then memory object mo_i is present on the scratchpad on edge e_j and is also being used (USE) on the edge. The binary variable $y^i_{j\ k}$ to represent the spilling of memory object mo_i on edge e_j is defined as the following:

$$y^i_{j\ k} = \begin{cases} 1 \text{ if operation corresponding to the spill attributes} \\ \quad at_k \in LOAD, STORE \text{ is performed for } mo_i \text{ at edge } e_j \\ 0 \text{ otherwise} \end{cases} \tag{6.17}$$

Similarly, if the value of $y^i_{j\ LOAD}$ is equal to 1, then the memory object mo_i is spill-loaded onto the scratchpad at edge e_j. Next, we describe the objective function and the constraints comprising the proposed ILP formulation.

Objective Function:

The objective function represents the energy savings achieved by overlaying memory objects onto the scratchpad. The function, as shown in the following, needs to be maximized in order to minimize the energy consumption of the system:

$$E = \sum_{mo_i \in MO} \sum_{e_j \in E} \left\{ \begin{array}{ll} E_{profit}(mo_i, e_j, at_k) & * \ x^i_{j\ k} \\ - E_{load\_cost}(mo_i, e_j) & * \ y^i_{j\ LOAD} \\ - E_{store\_cost}(mo_i, e_j) & * \ y^i_{j\ STORE} \end{array} \right\} \tag{6.18}$$

where, $E_{profit}(mo_i, e_j, at_k)$ is the energy saving achieved by assuming that memory object mo_i is present on the scratchpad at edge e_j. $E_{load\_cost}(mo_i, e_j)$ and $E_{store\_cost}(mo_i, e_j)$

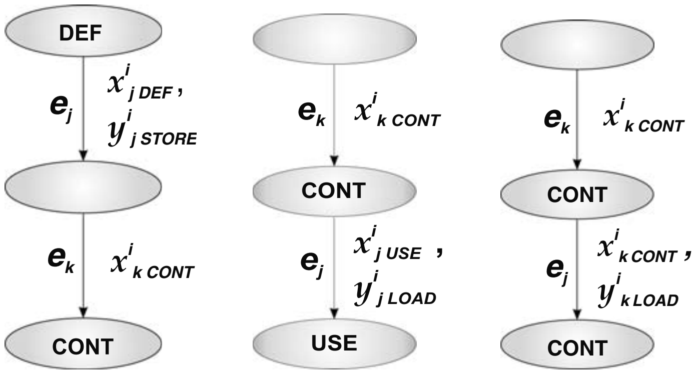
are the energy overheads of spilling memory object mo_i to and from the scratchpad at edge e_j, respectively. The energy savings function $E_{profit}(mo_i, e_j, at_k)$ is computed using the energy dissipation function $E(mo, R, mem)$ defined in Subsection 6.4.4 and is shown as follows:

$$E_{profit}(mo_i, e_j, at_k) = E(mo_i, R_j, MM) - E(mo_i, R_j, SPM) \tag{6.19}$$

where, R_j is the reference to memory object mo_i at edge e_j and is determined using Equation 6.12. The spill overhead functions $E_{load\_cost}(mo_i, e_j)$ and $E_{store\_cost}(mo_i, e_j)$ are computed as follows:

$$E_{load\_cost}(mo, e) = E_{spill}(mo, e, SPM, MM) \tag{6.20}$$
$$E_{store\_cost}(mo, e) = E_{spill}(mo, e, MM, SPM) \tag{6.21}$$

where $E_{spill}(mo, src\_mem, dst\_mem)$ is the spill energy function (*cf.* Equation 6.8) defined in Subsection 6.4.4.



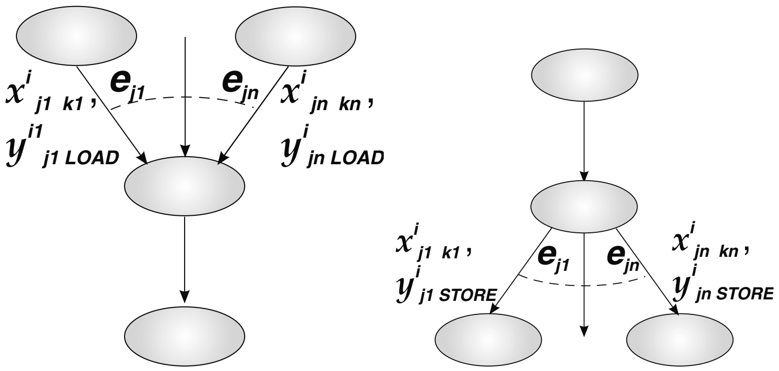
(a) DEF (b) USE (c) CONT

Fig. 6.8. Flow Constraints: (a) DEF (b) USE and (c) CONT Constraint

Constraints:
Constraints are added to the ILP-formulation to prevent the binary variables $x^i_{j\,k}$ and $y^i_{j\,k}$ from assuming arbitrary values and to obtain a legitimate solution to the memory assignment problem. We first explain the flow constraints that are added to maintain a legal flow of liveness of memory objects. The following is a *DEF-constraint* which is added for all edges with a DEF attribute:

$$x^i_{j\,DEF} - x^i_{k\,CONT} - y^i_{j\,STORE} = 0 \quad \forall mo_i \in MO \tag{6.22}$$

In the above constraint, edge e_j (refer Figure 6.8(a)) contains a DEF attribute while edge e_k is chosen such that the source node of edge e_k is same as the target node of edge e_j. Informally, the DEF-constraint states that if a memory object mo_i is defined (DEF) on the scratchpad on an edge e_j, then it can continue (CONT) to remain assigned to the scratchpad


$x^i_{j1\ k1}$, $y^{i1}_{j1\ LOAD}$
e_{j1}
e_{jn}
$x^i_{jn\ kn}$, $y^i_{jn\ LOAD}$
$x^i_{j1\ k1}$, $y^i_{j1\ STORE}$
$x^i_{jn\ kn}$, $y^i_{jn\ STORE}$

(a) Merge-Node Constraint (b) Diverge-Node Constraint

Fig. 6.9. Flow Constraints: (a) Merge-Node (b) Diverge-Node Constraint

on the following edge e_k or it can be spill-stored (STORE) to the main memory on the edge e_j. Similarly, *MOD-constraints* and *USE-constraints* are added for edges e_j with MOD and USE attribute defined, respectively.

$$x^i_{j\ USE} - x^i_{k\ CONT} - y^i_{j\ LOAD} = 0 \quad \forall mo_i \in MO \tag{6.23}$$

$$x^i_{j\ MOD} - x^i_{k\ CONT} - y^i_{j\ LOAD} = 0 \quad \forall mo_i \in MO \tag{6.24}$$

In both the above constraints, edge e_k (refer Figure 6.8(b)) is chosen such that the source node of edge e_j is same as the target node of edge e_k. Informally, the USE-constraint states that if a memory object mo_i is being used (USE) from the scratchpad on an edge e_k, then it was already present (CONT) on the scratchpad on the previous edge e_j or it was spill loaded (LOAD) on the edge e_k. A similar explanation exists for the MOD-constraint. The following constraint is added for edges with CONT attribute.

$$x^i_{j\ CONT} - x^i_{k\ CONT} - y^i_{j\ LOAD} = 0 \quad \forall mo_i \in MO \tag{6.25}$$

Similar to the USE-constraint, edge e_j (refer Figure 6.8(c)) contains a CONT attribute while edge e_k is a previous edge such that the source node of edge e_j is the target node of edge e_k. The CONT constraint implies that if a memory object mo_i continues (CONT) to remain assigned to the scratchpad on edge e_k, then it was already continuing (CONT) on a previous edge e_j or it was spill loaded (LOAD) onto the scratchpad on the current edge e_k. The following flow constraints are added to ensure the legality of flow of liveness on merge and diverge nodes. More importantly, they ensure an energy optimal spill code placement. It should be clarified that the code for spill routines is always allocated on the main memory. The following *merge-node constraints* are added for all merge nodes.

$$y^i_{j\ LOAD} - x^i_{j\ k} \le 0 \qquad \forall e_j \ \in \{e_{j1} \cdots e_{jn}\} \ at_k \in \{at_{k1} \cdots at_{kn}\} \tag{6.26}$$

$$x^i_{j1\ k1} = \cdots = x^i_{jn\ kn} \qquad s.t.\ at_{k1} \cdots at_{kn} \in Attrib_{flow} \ \forall mo_i \in MO \tag{6.27}$$

In the above constraints, edges $e_{j1} \cdots e_{jn}$ (refer Figure 6.9(a)) constitute all the edges entering a merge node. The first constraint (Equation 6.26) ensures that if a memory object

mo_i is spill loaded (LOAD) on an edge e_j, then it must be assigned to the scratchpad on the same edge. The second constraint (*cf.* Equation 6.27) ensures that if a memory object mo_i is assigned to the scratchpad on one of the edges entering the merge node, then it must be assigned to the scratchpad on each of the remaining edges. For all the diverge nodes, the following constraints, *viz. diverge-node constraints* are added.

$$y^i_{j\ STORE} - x^i_{j\ k} \leq 0 \quad \forall e_j \ \in \{e_{j1} \cdots e_{jn}\} \ at_k \in \{at_{k1} \cdots at_{kn}\} \tag{6.28}$$

$$x^i_{j1\ k1} = \cdots = x^i_{jn\ kn} \quad s.t.\ at_{k1} \cdots at_{kn} \in Attrib_{flow} \ \forall mo_i \in MO \tag{6.29}$$

As shown in Figure 6.9(b), edges $e_{j1} \cdots e_{jn}$ denote the edges emerging from a diverge node. In order to maintain the legality of liveness flow, if a memory object mo_i is assigned to the scratchpad on one of the edges exiting a diverge node, then it must be assigned to the scratchpad or spill-stored (STORE) to main memory on each of the remaining edges. Finally, we append the *scratchpad size constraint* which ensures that the aggregate size of all memory objects assigned to the scratchpad memory on an edge should be less than the scratchpad size. The following constraint is added to all edges where a memory object mo_i is being defined (DEF) or spill-loaded (LOAD).

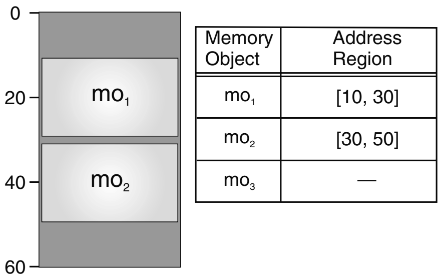
$$\sum_{mo_i\ \in\ MO} x^i_{j\ k} * size(mo_i) \leq size(SPM) \ \forall e_j \in E \tag{6.30}$$

A commercial ILP solver [32] is used to obtain an optimal assignment of memory objects to the scratchpad memory which maximizes the energy savings while satisfying the above constraints. The total number of binary variables used in the 0-1 ILP formulation of the Memory Assignment Problem is $O(|MO| * |E|)$. The maximum and the average runtimes of the ILP solver for all the experiments were found to be 1333.0 and 9.9 CPU seconds on a Sun Sparc 1300 MHz machine, respectively. We have computed the assignment of the memory objects onto the scratchpad memory. However, the scratchpad overlay problem is solved only when the addresses of the memory objects assigned to the scratchpad memory are computed. In the following section, we present the ILP formulation to compute the addresses of memory objects.

6.5.2 Optimal Address Assignment

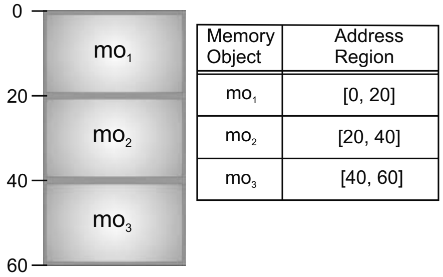
The previous step makes an implicit assumption that if the aggregate size of the memory objects assigned to the scratchpad on each edge was less than the scratchpad size, then the overlayed addresses for those memory objects can be computed. This assumption can fail due to a bad address assignment strategy which causes the fragmentation of the scratchpad address space. As a result, memory objects cannot be assigned addresses, despite the scratchpad size constraint being satisfied. The problem of address assignment is trivial if all the memory objects are of the same size. However, the problem becomes NP-complete when the memory objects are of different sizes [43]. The following example demonstrates that a bad assignment can fragment the scratchpad space into smaller address regions.

Example 6.13. The problem is to assign memory address regions to 3 equal sized memory objects mo_1, mo_2 and mo_3. The size of each memory object is 20 bytes while the size of the memory is 60 bytes.



Memory Object	Address Region
mo_1	[10, 30]
mo_2	[30, 50]
mo_3	—

Fig. 6.10. Incorrect Address Assignment



Memory Object	Address Region
mo_1	[0, 20]
mo_2	[20, 40]
mo_3	[40, 60]

Fig. 6.11. Correct Address Assignment

Figure 6.10 demonstrates a bad assignment of address regions to memory objects mo_1 and mo_2 which leads to the fragmentation of the memory address space. Memory objects mo_1 and mo_2 are assigned the address region $[10, 30]$ and $[30, 50]$. Thus, memory object mo_3 could not be assigned an address region, despite the memory being large enough to hold all three memory objects together. A correct assignment of address regions to memory objects is presented in Figure 6.11.

Every memory object mo_i on each edge e_j is assigned an address region $[a^i_j, a^i_j + size(mo_i)]$ which is contained within the address space of the appropriate memory. The integer variable a^i_j represents the start address of the memory object mo_i at the edge e_j and it satisfies the following constraint:

$$\begin{aligned} Start_{SPM} \leq a^i_j \leq End_{SPM} - size(mo_i) \ \text{XOR} \\ Start_{MM} \leq a^i_j \leq End_{MM} - size(mo_i) \end{aligned} \tag{6.31}$$

The assumption that the main memory is always large enough to hold all memory objects in disjoint address regions is used to simplify the above inequations. The assumption requires setting the variables $Start_{MM} = End_{SPM}$ and $End_{MM} = \infty$ such that the main memory is of infinite size and also does not overlap with the address space occupied by the scratchpad memory. The simplified constraint for the integer variable a^i_j is the following:

$$Start_{SPM} \leq a^i_j \leq End_{SPM} - size(mo_i) \tag{6.32}$$

The next set of constraints is added to the ILP formulation to ensure that the address ranges of no two memory objects (mo_i and mo_j) defined at the same edge e_k overlap with each other.

$$a^i_k - a^j_k \geq size(mo_j) \quad \text{XOR} \tag{6.33}$$

$$a^j_k - a^i_k \geq size(mo_i) \tag{6.34}$$

The first constraint (*cf.* Equation 6.33) of the above set of constraints implies that on edge e_k the start address $\left(a^i_k\right)$ of the memory object mo_i is greater than the end address $(a^j_k + size(mo_j))$ of memory object mo_j. The second constraint (Equation 6.34) implies

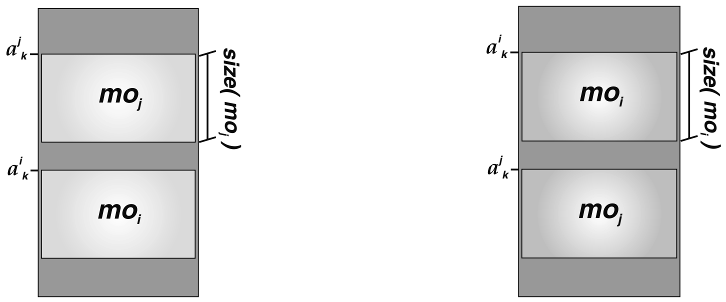


Fig. 6.12. Two Potential Placements of Memory Objects

the reversed placement of the memory objects. Figure 6.12 demonstrates the two possible assignments of address regions of the memory objects in the scratchpad. The XOR operator implies that only one of the two constraints can be satisfied at the same time. However, the XOR operator can not be modeled using linear programming and therefore, a binary variable $u_k^{i\,j}$ is added to linearize the above set of constraints.

$$u_k^{i\,j} = \begin{cases} 0 \text{ constraint (6.33) is to be satisfied} \\ 1 \text{ constraint (6.34) is to be satisfied} \end{cases} \tag{6.35}$$

The following is the linearized form of the set of constraints (Equation 6.33 and Equation 6.34) with **L** being a sufficiently large constant.

$$a_k^i - a_k^j + \mathbf{L} * u_k^{i\,j} \geq size(mo_j) \qquad \forall e_k \in E \tag{6.36}$$

$$a_k^j - a_k^i - \mathbf{L} * u_k^{i\,j} \geq size(mo_i) - \mathbf{L} \quad \forall e_k \in E \tag{6.37}$$

The above set of constraints is repeated for all pairs of memory objects which are assigned to the scratchpad memory on edge e_k. Subsequently, they are also repeated for all edges $e_k \in E$ with more than one memory object assigned to the scratchpad memory. Next, a constraint is added to restrict that the start address of a memory object mo_i is the same on two edges e_j and e_k which are connected by a node.

$$a_j^i - a_k^i = 0 \tag{6.38}$$

In the above constraint, edges e_j and e_k are chosen such that source node of edge e_k is the target node of edge e_j. Any change in the offsets of the memory object mo_i on edges e_j and e_k is captured using the following binary variable.

$$v_{j\,k}^i = \begin{cases} 1 \text{ if } a_j^i \neq a_k^i \\ 0 \text{ otherwise} \end{cases} \tag{6.39}$$

The unit value of the variable $v_{j\,k}^i$ would require a reorganization of the memory objects present in the scratchpad at the node connecting edge e_j and e_k. The reorganization would require additional instructions and cause energy overhead, not taken into account by the memory assignment problem. Therefore, a unit value of the variable $v_{j\,k}^i$ would imply an invalid solution to the address assignment problem. Equation 6.38 is transformed to the following form after the insertion of the binary variable $v^i_{j\,k}$.

$$a^i_j - a^i_k - \mathbf{L} * v^i_{j\,k} = 0 \quad \forall e_j, e_k \in E \tag{6.40}$$

The above constraint is repeated for all memory objects assigned to the scratchpad on both the edges $e_j, e_k \in E$ and also for all such valid pair of edges. A valid solution is characterized by the fact that the offsets of memory objects on all pairs of edges remain invariant. The address assignment problem is a decision problem which has to be converted to an optimization problem because the ILP solver [32] can only solve optimization problems. The summation of the binary variable $v^i_{j\,k}$ for all valid pairs of edges and for all memory objects is denoted as the objective function of the ILP formulation.

$$\sum_i \sum_j \sum_k v^{ij}_k \tag{6.41}$$

For a valid solution, the value of the objective function should be zero which is achieved by minimizing the objective function. The presented ILP formulation consists of both binary and integer variables. The number of integer variables in the above formulation is $O(|MO| * |E|)$ while the number of binary variables is $O(|MO| * |E|^2)$.

The problem is solved using the *branch and bound* technique of the commercial ILP solver [32], which can take substantial time for certain problem instances. For one instance, the ILP solver failed to find the result in 3 weeks. If we exclude the three problem instances for which the ILP solver took 35, 69 and 92 hours to compute the solution, the remaining problem instances were solved in small amounts of time. The maximum and the average runtime of the ILP solver is 7304 and 479 CPU seconds on a Sun Sparc 1300 MHz machine, respectively. Nevertheless, the computation time needs to be reduced so that the overall scratchpad overlay approach could be implemented within a compiler based framework. In the following subsection, we describe a first-fit heuristic based near-optimal approach for the solving the address assignment problem.

6.5.3 Near-Optimal Address Assignment

The First-Fit heuristic [43] based algorithm is used for assigning addresses to the memory objects on each edge of the IPCFG. The heuristic is chosen ahead of the well known alternatives (*e.g.* next-fit, best-fit, segregated-list, binary-buddy) because the study [62] demonstrated that it performs close to best-fit and is faster than best-fit for real-life workloads.

Figure 6.13 presents the pseudo-code for the first-fit based near-optimal address assignment algorithm. The algorithm assigns the same address to a memory object if it has been assigned an address on a previous edge. Otherwise, it uses first-fit heuristic to determine an available address region for the memory object.

We implemented the variant of the first-fit heuristic which divides the scratchpad address space into $|MO|$ variable sized regions. In order to reduce unused address space, the start boundary of an empty region is adjusted to the end address of its previous region which is assigned to a memory object. The first-fit heuristic assigns a memory object the first empty region which can accommodate the memory object. This might lead to the problem of fragmentation as some scratchpad memory space may remain unused. However, the

```
void AddressAssignment() {
  for (k=1;k<=|MO|;k++) {
    for (i=1;i<=|E|;i++) {
      /* Is MO $mo_k$ mapped to the SPM on edge $e_i$? */
      if ( $x^i_{DEF\,k}$ || $x^i_{LOAD\,k}$ ) {
        /* Does MO $mo_k$ already have a valid address? */
        if (AddressVector[k]==INVALID) {
          /* No, compute a valid address using first-fit heuristic. */
          address = FirstFitAddress(i,k);
          if (address == INVALID)
            /* First-Fit couldn't find an address, make $mo_k$ offchip. */
            SetMemoryObjectOffchip(k);
          else
            /* Assign the valid address to $mo_k$. */
            AddressVector[k] = address;
        }
      } else if ($x^i_{MOD\,k}$ || $x^i_{USE\,k}$|| $x^i_{CONT\,k}$ ) {
        /* Does MO $mo_k$ already have a valid address? */
        if (AddressVector[k]==INVALID)
          /* Get Address from one of the previous edges. */
          AddressVector[k] = AddressFromPreviousEdge(i,k);
      } else
        AddressVector[k] = INVALID;
  }}
```

Fig. 6.13. First-Fit Heuristic Based Address Assignment Algorithm

study [62] pointed out that only minimal memory space is lost due to fragmentation by the first-fit heuristic. This observation is also corroborated by our experiments.

The address assignment algorithm calls the first-fit heuristic on an edge only when a memory object is either defined or loaded onto the scratchpad and the memory object has not been assigned a valid address on the same edge. If the first-fit heuristic fails to assign an address to the memory object, then the memory object is assigned to the main memory for its entire lifetime. If a memory object is used, modified or continued on an edge and does not have a valid address, then it is assigned an address from one of the immediately previous edges. The `AddressFromPreviousEdge` routine also checks if the addresses on all the immediately previous edges to the current are equal, otherwise the error flag is set. Our implementation of the first-fit heuristic has the runtime complexity of $O(|E| * |MO|)$. Consequently, the runtime complexity of the address assignment algorithm is $O(|E|^2 * |MO|^2)$. The algorithm required negligible (less than 1 CPU sec.) time to compute solutions which are very close to the optimal solutions for all our experiments. This fact has been validated by the experimental results presented in the following section.

6.6 Experimental Results

The experiments were conducted for uni-processor ARM, multi-processor ARM and M5 DSP based systems according to their corresponding experimental workflows presented

Benchmark	Code Size (bytes)	Data Size (bytes)	System Architecture
adpcm	804	4996	uni-processor ARM
edge detection	908	7792	uni-processor ARM
histogram	704	133156	uni-processor ARM
mpeg4	1524	58048	uni-processor ARM
multisort	636	2020	uni-processor ARM
dsp	2784	61272	uni-processor ARM
media	3280	75672	uni-processor ARM
multi-process edge detection	4484	23820	multi-processor ARM
complex multiply		24576	M5 DSP
fir		3688	M5 DSP
fir2dim		3380	M5 DSP

Table 6.6. Benchmark Programs for the Evaluation of Scratchpad Overlay Approaches

in Chapter 3. The scratchpad overlay approaches for uni-processor ARM and M5 DSP based are implemented as memory optimizations performed within the backends of the corresponding compilers. In contrast, the overlay approaches for the multi-processor ARM based system are implemented as pre-compiler source-level optimizations. Accurate energy models for these systems are used to compute the energy consumption values presented in this section. Table 6.6 presents the benchmarks along with their code and data sizes which are used to evaluate the scratchpad overlay approaches for different system architectures. In the following subsection, we start with the evaluation of the overlay approaches for the uni-processor ARM based system.

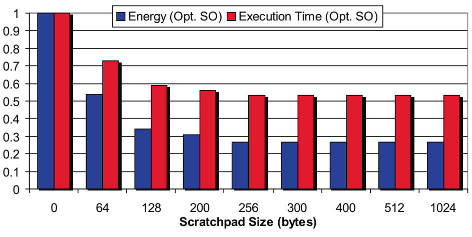
6.6.1 Uni-Processor ARM

In this subsection, the evaluation of the scratchpad memory and the overlay approaches is presented in the following order:

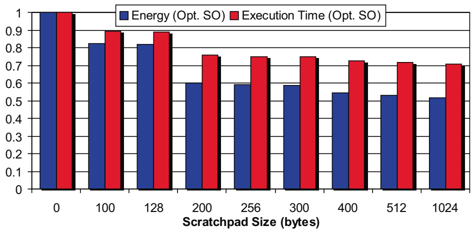
(a) Benefits of the scratchpad memory
(b) Comparison of the different scratchpad memory allocation approaches
(c) Comparison of the cache and the scratchpad memory

First, the benefit of the scratchpad in terms of both energy dissipation and execution time is evaluated. Second, a comparison of the various scratchpad allocation approaches is presented. Finally, a comparison of cache and scratchpad memories of the same sizes for the energy dissipation and the execution time metric is presented.

Scratchpad Benefit:
Figures 6.14(a) and 6.14(b) present the normalized energy consumption and execution time values for systems with varying scratchpad sizes and executing the *edge detection* and *dsp* benchmarks, respectively. The scratchpad is allocated using the Optimal Scratchpad Overlay (Opt. SO) approach presented in the previous section. In order to quantify the benefit due to a scratchpad memory, all energy consumption and execution time values are normalized according to those values obtained for the system without a scratchpad.

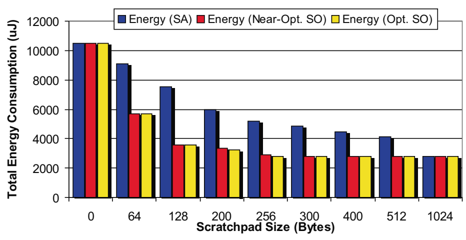


(a) Edge Detection

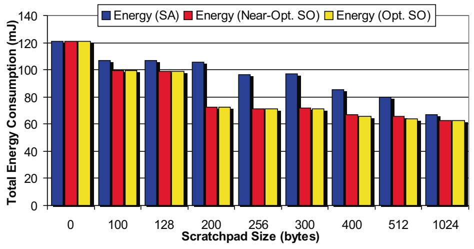


(b) DSP

Fig. 6.14. Normalized Energy Consumption and Execution Time for Opt. SO Approach



(a) Edge Detection



(b) DSP

Fig. 6.15. Energy Comparison of Scratchpad Allocation Approaches

Both the figures clearly demonstrate the efficacy of including a scratchpad into the memory hierarchy. For the *edge detection* benchmark, we observe that a 64 bytes scratchpad (*cf.* Figure 6.14(a)) leads to 48% reduction in the energy dissipation of the system and to 32% reduction in the execution time of the benchmark compared to the system without a scratchpad. For the same application, a 256 bytes scratchpad having a size of only 2% compared to the total application size, leads to more than 70% reduction in terms of energy dissipation. The *dsp* benchmark, being 9 times larger than the *edge detection* benchmark, demonstrates similar benefits for the scratchpad based systems. A 1024 bytes scratchpad, again accounting for 2% of the total application size, presents 48% reduction in energy dissipation and 30% reduction in the execution time of the *dsp* benchmark.

Comparison of the Scratchpad Allocation Approaches:

A comparison of the scratchpad allocation techniques *viz.* Non-Overlayed Scratchpad Allocation (SA), Optimal Scratchpad Overlay (Opt. SO) and Near-Optimal Scratchpad Overlay (Near-Opt. SO) for the *edge detection* and *dsp* benchmarks is presented in Figure 6.15(a) and Figure 6.15(b), respectively. The figures present the total energy consumption of the ARM based systems when the scratchpad is allocated using the different allocation techniques. From Figures 6.15(a) and 6.15(b), we make the following observations:

First, we observe that the energy consumption values for the SA approach monotonically decrease with the increase in scratchpad size. The energy consumption values for Opt. SO and Near-Opt. SO approaches decrease faster than those for the SA approach. The energy values for the scratchpad overlay approaches (cf. Figure 6.15(a)) reach a threshold value at 256 bytes and thereafter remain constant. For the SA approach, a larger scratchpad size implies that more memory objects can be statically allocated onto the scratchpad and lower the energy consumption. This justifies the monotonically decreasing energy values for the SA approach. In contrast, the overlay approaches share the scratchpad among memory objects with non-conflicting life-times and result in lower energy values than for the SA approach. The energy consumption values become constant when no additional memory objects can be overlayed on the scratchpad. For 1024 bytes scratchpad, all the allocation approaches assign the same memory objects to the scratchpad and therefore result in the same energy consumption values.

Second, we observe that the energy values for Near-Opt. SO and Opt. SO approaches (*cf.* Figure 6.15(a)) at 256 bytes scratchpad is equal to that for the SA approach at 1024 bytes. Similarly, for the *dsp* benchmark (refer Figure 6.15(b)) the energy consumption for the overlay approaches at 400 bytes of scratchpad is equal to that for the SA approach at 1024 bytes of scratchpad. We can, therefore, conclude that the scratchpad overlay approaches efficiently utilize the scratchpad memory. Finally, from both the figures we observe that the energy consumption values for the Near-Opt. SO are very close to those for the Opt. SO. This implies that the proposed Near-Opt. SO approach performs fairly close to the Opt. SO approach.

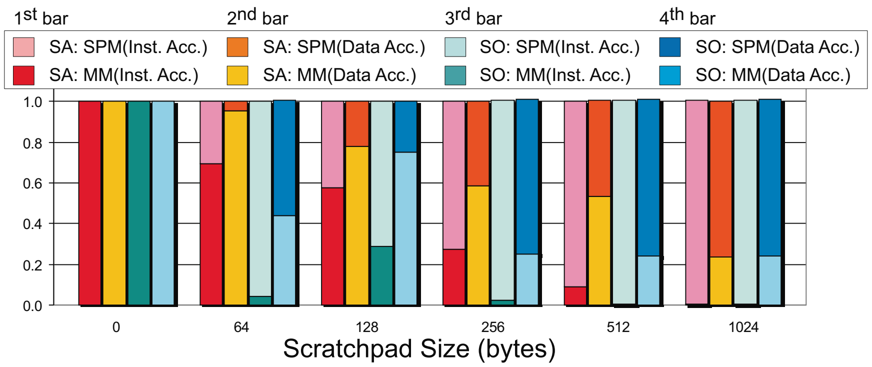


Fig. 6.16. Edge Detection: Comparison of SA and Near-Opt. SO Approaches for Memory Accesses

Next, we compare the SA and Near-Opt. SO approaches in terms of memory accesses for the *edge detection* benchmark. Figure 6.16 presents the comparison for the two scratchpad allocation approaches. The memory accesses are normalized according to the main memory accesses of the appropriate type (*i.e.* instruction or data access) for a system without a scratchpad. The memory accesses for each allocation approach are classified into 4 categories, depending on the access type and accessed memory. The labels of Figure 6.16 are defined according to the following regular expression.

$$\{AllocationApproach\} : \{Memory\}(\{AccessType\}) \tag{6.42}$$

where, $AllocationApproach \in \{\text{SA, SO}\}$, $Memory \in \{\text{MM, SPM}\}$ and $AccessType \in \{\text{Inst. Acc., Data Acc.}\}$ represent the allocation approach, accessed memory and the access type. For example, the label SA: MM(Inst. Access) implies that the bar represents normalized instruction accesses to the main memory when the SA approach is used to allocate the scratchpad present in the system. The normalized memory accesses to the main memory and the scratchpad are stacked to distinguish between aggregate instruction and data access. In the above figure, the first and the second bars represent the instruction and data memory accesses for the SA approach, respectively. The third and the fourth bars present the same for the Near-Opt. SO approach. From the figure, we make the following observations:

First, the third and the fourth bars for the SO approach, representing normalized instruction and data accesses, are larger than the unit value. This is because additional instruction and data accesses are required for copying the memory objects on and off the scratchpad. Second, the SO approach very efficiently utilizes the scratchpad for caching instruction segments. For example, a 256 byte scratchpad allocated using the SO approach caches 98% of all instruction accesses. In contrast, the same sized scratchpad allocated through the SA approach is able to cache around 40% of the total instruction accesses.

Third, the scratchpad allocated using the SO approach can also cache a higher percentage of data accesses than that by the scratchpad allocated using the SA approach. However, the percentage of cached data accesses is lower than that of cached instruction accesses for the both the allocation approaches. This is due to the fact that instruction segments (*i.e.* traces and functions) are smaller and more frequently accessed than data variables. Finally, we conclude that the SO approach more efficiently utilizes the scratchpad than the SA approach.

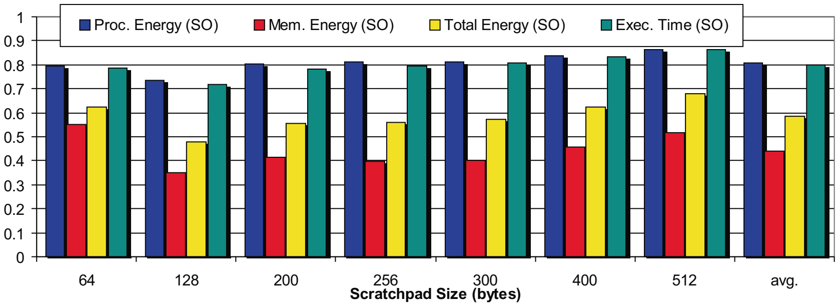


Fig. 6.17. Edge Detection: Comparison of SA and Near-Opt. SO Approaches

Figure 6.17 presents the normalized energy consumption values and performance values of the *edge detection* benchmark allocated using the Near-Opt. SO approach. The components of total energy consumption, processor and memory energy consumption are also shown in the figure. The energy and performance values for the benchmark allocated using the SA approach are denoted as the unit valued baseline.

From Figure 6.17, we observe that both the energy and performance values for the Near-Opt. SO approach are less than the corresponding values for the SA approach. Despite the fact that the processor executes additional instructions for copying memory objects on and off the scratchpad, the overlay approaches are able to save both energy and execution time. The accesses to the scratchpad are cheaper than the main memory both in terms of access time and energy per access. This enables the Near-Opt. SO approach to over-compensate for the copying overhead which results in significant savings compared to the SA approach. The Near-Opt. SO approach leads to a maximum reduction of 65% in the memory energy consumption for a 128 bytes scratchpad. The total energy consumption, being the sum of the processor energy and the memory energy, shows an average reduction of 42%. The application on an average requires 21% less CPU cycles for execution.

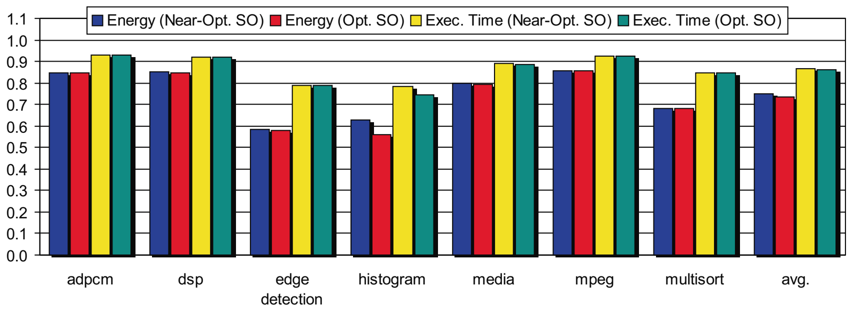
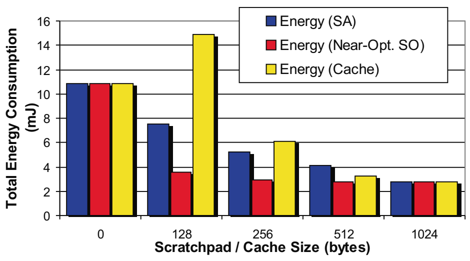


Fig. 6.18. Overall Comparison of Near-Opt. SO, Opt. SO and SA Approaches

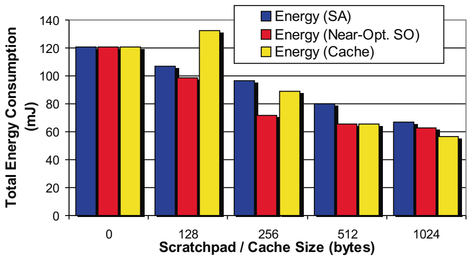
Figure 6.18 presents the comparison of the scratchpad overlay approaches against the non-overlayed scratchpad allocation (SA) approach across all benchmarks. The total energy consumption and execution time values for the overlay approaches are averaged across all scratchpad sizes and are normalized against the corresponding average energy and execution time values for the SA approach. Moreover, the average energy consumption and execution time values across all scratchpad sizes and across all benchmarks are normalized and are presented as the last set of bars in the figure.

From the figure, it is observed that the minimum energy and execution time savings are for the *adpcm* benchmark. This is justified as the *adpcm* benchmark consists of only one encoder and decoder routines and provides little opportunity for overlay. For the *histogram* benchmark, the maximum average energy savings of 44% are reported due to the Opt. SO approach. Maximum performance improvements of 22% and 26% due to the Near-Opt. SO and the Opt. SO approaches, respectively, are reported for the same benchmark. The *histogram* benchmark comprises several hot-spots and data arrays with non-conflicting life-times. As a consequence, the overlay approaches achieve the maximum savings for the benchmark.

For the same benchmark, we observe a noticeable difference between the energy consumption and execution time values for the Near-Opt. SO and the Opt. SO approach. This is because the assignment step failed to assign an address in the scratchpad space to one of the most energy consuming memory object. The Near-Opt. SO approach, otherwise, achieved very close to the optimal results. Finally, the overall average energy consumption and execution time savings for the Near-Opt. SO approach are reported to be 25% and 13%, respectively. The overall average energy and execution time savings for the Opt. SO approach are merely 1% better than those for the Near-Opt. SO approach.



(a) Edge Detection



(b) DSP

Fig. 6.19. Comparison of Cache with SA and Near-Opt. SO Approaches

Comparison of the Cache and the Scratchpad Memory:
The operation of a scratchpad under the control of the scratchpad overlay approaches is similar to that of a cache. Hence, it would be appropriate to present a comparison between a scratchpad and a cache. Figure 6.19(a) and Figure 6.19(b) compare caches against scratchpads allocated using the SA and the Near-Opt. SO approaches. The cache experiments were conducted for a 4-way set associative unified cache of varying sizes.

From both the figures, we find that the energy consumption of the system with a 128 bytes cache is much worse than that with a 128 bytes scratchpad. This is because a small cache causes a high number of misses which results in high number of expensive main memory accesses. In contrast, the best set of memory objects is assigned to the scratchpad memory, reducing the number of main memory accesses. Consequently, the small scratchpads, irrespective of the allocation approaches, are significantly better than a small cache.

The energy consumption for the cache based systems improves significantly for larger sizes. The reason being that the number of energy hungry cache misses is a lot lower for large caches and that caches can perform a fine grained caching of variables than the proposed scratchpad overlay approaches. For 1024 bytes, the energy consumption of the cache based system is a bit better than that of the scratchpad memory based systems. Though, we should note that for the *edge detection* benchmark the energy value for a 1024 bytes scratchpad allocated using the Near-Opt. SO approach is same as that for a 256 bytes scratchpad. Hence, we conclude that the energy values for a small scratchpad based system with Near-Opt. SO approach is comparable to that for a significantly larger cache based system.

Figure 6.20 presents a similar comparison of the scratchpad overlay approaches for the scratchpad memory based systems with cache based systems. The total energy consumption and execution time values for a scratchpad based system are normalized against those for a same sized cache based system. These values are then averaged across all scratchpad and cache sizes for each benchmark. The scratchpad allocated using the overlay approaches clearly outperforms the cache for all benchmarks. However, the total energy and execution time savings due to overlay approaches are lower than those presented in Figure 6.18. Maximum energy savings of 35% each due to the optimal and near-optimal overlay approaches, are reported for the *edge detection* benchmark. Overall average energy savings of 19% and 20% are reported due to the Near-Opt. SO and the Opt. SO approaches, respectively. Moderate average performance improvements of 9% and 10% are reported against a cache-based

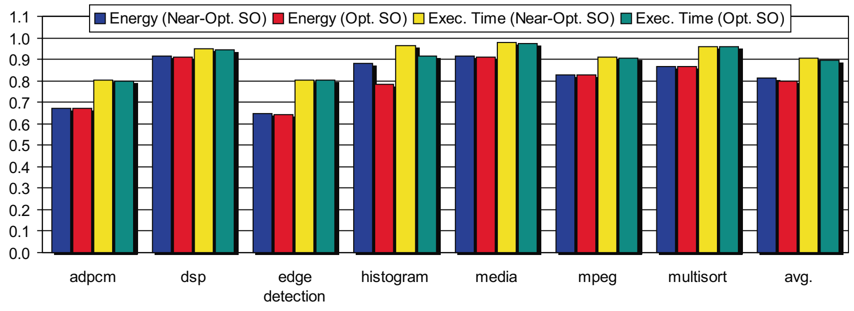


Fig. 6.20. Overall Comparison of the Cache and Scratchpad Overlay Approaches

approach. In the following subsection, we present the evaluation of the scratchpad overlay approaches for the multi-processor ARM setup.

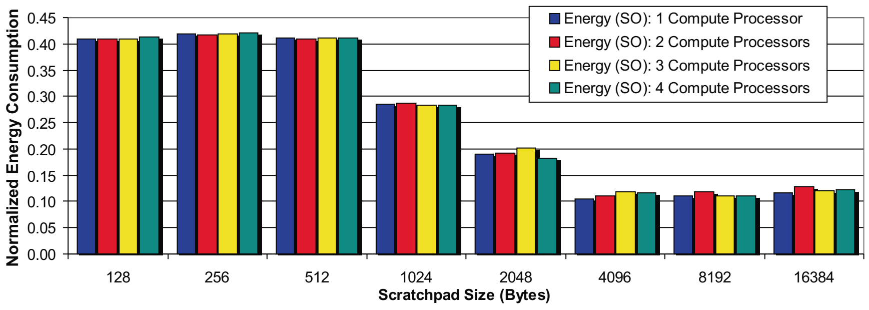


Fig. 6.21. Multi-Process Edge Detection: Normalized Energy Consumption for Varying Compute Processors and Scratchpad Sizes (Cycle Latency = 1 Master Cycle)

6.6.2 Multi-Processor ARM

In this subsection, the evaluation of the near-optimal scratchpad overlay approach for Multi-Process *edge detection* benchmark is presented. The benchmark is simulated on the multi-processor ARM based system. It consists of an initiator process, a terminator process and a variable number of compute processes. Since each process is mapped to a separate processor in the multi-processor ARM based system, the processors are named according to the mapped process. Each ARM processor has its own local scratchpad memory, while all of them access a shared main memory. Additional details regarding the Multi-Process Edge Detection benchmark can be found in Section 3.2 of Chapter 3.

Figure 6.21 presents the normalized energy consumption values for the Multi-Process Edge Detection benchmark when the number of compute processors and the size of scratchpad is varied. The energy values are normalized against the energy values of a system without a scratchpad. The near-optimal scratchpad overlay approach is used to allocate the scratchpad attached to the processors.

The first observation from the figure is that the energy consumption values for a system with 128 bytes of scratchpad is around 40% of that for a system without a scratchpad. This result clearly indicates the benefit of using a scratchpad for energy minimization. The total energy consumption of the system decreases with the increase in the size of the scratchpad until it reaches the minimum energy consuming scratchpad memory size. Thereafter, any increase in scratchpad size also increases the energy consumption of the system. The minimum energy consuming scratchpad size for 1 and 2 compute processors is 4k bytes and for 3 and 4 compute processors is 8k bytes.

The energy consumption values increase after the minimum energy value since the benefit of a larger scratchpad is negated by its higher energy per access. The minimum energy values are about 10% of those for the system without a scratchpad. We observe that the normalized energy values are at little bit higher at 256 bytes than those at 128 bytes. This is because the memory objects that are chosen for allocation are same for 128 bytes and 256 bytes scratchpad. However, the latter consumes more energy per access than the former. Therefore, the energy consumption slightly increases at 256 bytes. Finally, we observe that the normalized energy values across different compute processors but for the same scratchpad size are fairly close to each other.

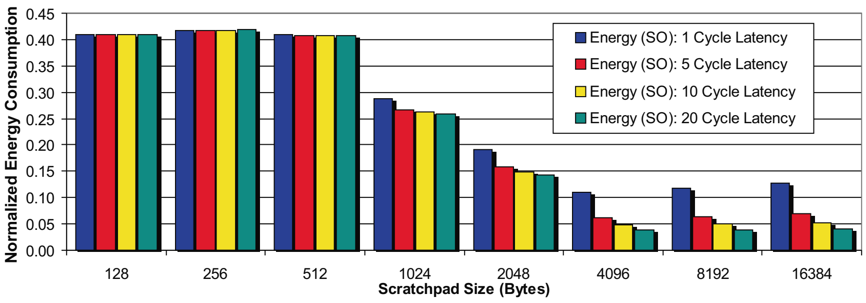


Fig. 6.22. Multi-Process Edge Detection: Normalized Energy Consumption for Varying Memory Access Times (#Compute Processors = 2)

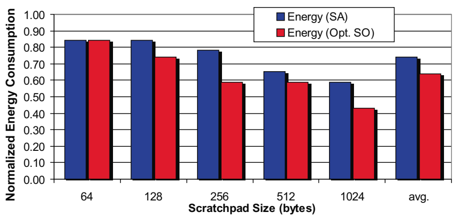
The previous figure presented the energy comparison for the case when the main memory has a latency of 1 master clock cycle for each access. This scenario is not true for most of the contemporary embedded systems. Therefore, we would like to evaluate the scratchpad approach under realistic latency values of the main memory. Figure 6.22 presents the normalized energy consumption values of the benchmark when the main memory has a latency of 1, 5, 10 and 20 master clock cycles. The scratchpad is assumed to have a latency of zero clock cycles. The energy values are normalized against those for the systems without a scratchpad and with corresponding main memory latencies. Also, the multi-processor ARM system is assumed to consist of 2 compute processors.

As expected, the effect of scratchpad overlay in reducing the energy consumption increases as the main memory becomes slower. The effect of slower memory is more prominent for larger scratchpads as it captures a larger fraction of the total number of accesses. For a 2k bytes scratchpad, the normalized energy values are about 20% and 15% for systems whose main memory has a latency of 1 and 20 master clock cycles, respectively. It is expected that as the difference in access times of the main memory and the scratchpad will increase, the benefit of utilizing the scratchpad will also improve.

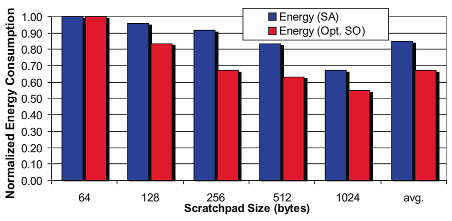
6.6.3 M5 DSP

The experiments for the M5 DSP were conducted by assuming that the L1 group memory or the scratchpad of varying sizes can be synthesized and that only global variables can be assigned to the scratchpad. The optimal scratchpad overlay (Opt. SO) algorithm is used for overlaying memory objects on the scratchpad. The Opt. SO approach requires minimal computation time as the number of memory objects for the DSP routines are fairly small. The Near-Opt. SO approach is not considered due to the low computation time of the Opt. SO approach. Array slicing [93] approach was used to create small slices of the data arrays present in the applications.

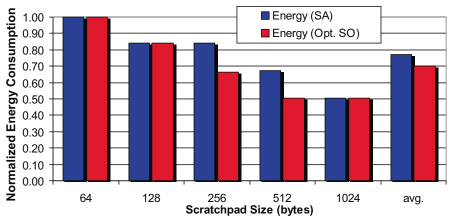
A comparison of the SA and the Opt. SO approaches for *fir2dim*, *complex multiply* and *fir* routines is presented in Figures 6.23(a), 6.23(b) and 6.23(c), respectively. Normalized energy values of the data memory subsystem of the M5 DSP are shown in the figure. The energy consumption value of the data memory system without a scratchpad is considered as the unit valued baseline. From the figures, we make a few observations. First, we observe that the energy values for the Opt. SO approach are always better than or equal to those for the SA approach. Second, we observe that the insertion of a small scratchpad into the data memory hierarchy decreases its energy consumption. Energy reductions between 40% and 50% can be observed for a system with 512 bytes scratchpad memory allocated using the Opt. SO approach. Third, the Opt. SO approach results in the maximum energy savings of 54%, 44% and 50% for *fir2dim*, *complex multiply* and *fir* benchmarks, respectively. Finally,



(a) FIR2DIM



(b) Complex Multiply



(c) FIR

Fig. 6.23. Normalized Energy Comparison of Scratchpad Allocation Approaches

the Opt. SO approach results in average energy savings of 31%, 32% and 30% for *fir2dim*, *complex multiply* and *fir* benchmarks, respectively.

6.7 Summary

In this chapter, overlay based techniques for the utilization of the scratchpad memory were presented. The problem of overlaying both data and instructions onto the scratchpad memory was shown to be similar to the problem of global register allocation. Both optimal and near-optimal approaches to solve the overlay problem were presented.

The approaches solved the problem in a two step process. While the first step is common, the second step is different for both the techniques. In the first step, the approaches assign memory objects to the scratchpad memory and also determine optimal locations to insert spill instructions. In the second step, the approaches compute addresses for the memory objects assigned to the scratchpad memory. The optimal approach uses an ILP formulation to determine optimal solutions, whereas the near-optimal approach uses a first-fit heuristic based approach.

The proposed approaches result in reduced energy consumption of the system against the non-overlayed scratchpad allocation approach and also against a cache based system. The scratchpad overlay approaches were evaluated for uni-processor ARM, multi-processor ARM and DSP based systems. For an uni-processor ARM based system, the average energy reductions of 24% and 20% are reported against the previous approach and the cache based system, respectively. For a multi-processor ARM based system, a 4k bytes scratchpad memory reduces the energy consumption to 10% of that of a scratchpad-less system. Additional experiments for the low power DSP report average savings of 31% in the energy consumption of the data memory hierarchy.

The approaches presented in this chapter were published in [127], [130], and [124].

7

Data Partitioning and Loop Nest Splitting

In the previous chapters, the proposed scratchpad allocation approaches allocated aggregate variables along with code fragments onto the scratchpad memory. The property of allocating only aggregate variables may restrict the optimization potential of the approaches. Most of the applications found in embedded system domain consist of large data arrays which are not properly optimized by the previously presented allocation approaches. In this chapter, we present a data partitioning approach which divides a large array into two smaller partitions and then allocates one of these partitions along with application code fragments onto the scratchpad memory in a non-overlayed manner. It is observed that partitioning the data arrays degrades the control flow of the application, as *if*-statements are inserted in the application code for appropriately accessing array partitions. Therefore, an approach which combines the data partitioning with an already known loop nest splitting is also proposed in this chapter. The objective of the combined approach is to improve the control flow degraded by the data partitioning approach.

In the following, a brief introduction into the data partitioning and loop nest splitting approaches is presented with the help of a motivating example. Section 7.2 presents a survey of the related work and enumerates the novel points of the data partitioning approach. Section 7.3 describes the preliminaries and presents a formal definition of the data partitioning problem. Section 7.4 and Section 7.5 present the data partitioning and the loop nest splitting approaches, respectively. The evaluation of the experimental results for the proposed approach is presented in Section 7.6. Section 7.7 ends the chapter with a short summary.

7.1 Introduction

Most of the applications running on embedded devices contain several large arrays. Therefore, considering aggregate array variables for allocation onto the scratchpad may not be an ideal decision, as it may lead to the under-utilization of the scratchpad. The proposed data partitioning approach rectifies the aforementioned problem by partitioning an array present in the application into two smaller arrays. It then uses non-overlayed allocation to assign one of the two array partitions along with application code fragments to the scratchpad. The approach uses an accurate energy model [114] to compute the energy overhead

```
#define SIZE 100
int A[SIZE];


for (i=0; i<SIZE/2; i++)
  for (j=0; j<i; j++) {
    data = A[i+j];
    ... }
```

```
#define SIZE 100
#define SPLIT 70
#define READ_ACCESS(value,index)
  if (index < SPLIT)
    value = Aleft[index];
  else
    value = Aright[index-SPLIT];
int Aleft[SPLIT],Aright[SIZE-SPLIT];

for (i=0; i<SIZE/2; i++)
  for (j=0; j<i; j++) {
    READ_ACCESS(data,i+j);
    ... }
```

Fig. 7.1. Example Code Fragment before and after Data Partitioning

for accessing the array partitions. Therefore, it partitions an array if it can determine that the energy benefit due to data partitioning over-compensates the energy overhead caused due to modified access functions to the array partitions. Otherwise, the approach allocates aggregate variables and the code segments onto the scratchpad in a non-overlayed manner.

Figure 7.1 presents the application code before and after data partitioning. We can observe that array `A` is replaced by two smaller arrays `Aleft` and `Aright` in the partitioned application code. Additionally, the read access to array `A` in the loop is replaced by an access macro (`READ_ACCESS`). The access macro accesses the appropriate partitioned array depending on the index value (`i+j`). A similar access macro is used for write accesses but is not shown in the figure for the sake of simplicity. The use of access macros for accessing array variables allows us to optimize application code with both regular (affine) and irregular index functions. An additional advantage of our approach, highlighted by the second for-loop, is its ability to optimize loop-nests with non-constant loop bounds. Most of the array partitioning approaches work on the premise of perfect loops and affine index functions. In the above example, the data partitioning approach increases the access to the scratchpad by 1015 if array `Aleft` is allocated to the scratchpad.

The data partitioning approach inserts access macros containing *if*-statements in the application code for accessing the appropriate array partition. Given that array references typically occur in the innermost loops of embedded applications, these *if*-statements lead to overhead w.r.t. both runtime and energy consumption. If the index function to the unpartitioned array in the application is an affine function, we combine data partitioning with loop nest splitting [39] to substantially reduce the overhead caused by the inserted *if*-statements.

Figure 7.2 depicts the array partitioned application code from Figure 7.1, before and after loop nest splitting. Please note that the read access macro (`READ_ACCESS`) shown in Figure 7.1 has been expanded here. The loop nest splitting transformation determines the ranges of loop iterations where all the *if*-statements in the loop nest are provably satisfied. Using this information, the transformation rewrites the loop nest such that no *if*-statements are executed for these iteration ranges.

In Figure 7.2, the transformation detects that the outer `i` loop iterates from 0 to 49 and the inner `j` loop iterates from 0 to the current value of `i`. Considering the condition (`i+j<70`) inserted by data partitioning, it recognizes that this condition is true for all values

```
for (i=0; i<50; i++)
  for (j=0; j<i; j++) {
    if (i+j<70)
      data = Aleft[i+j];
    else
      data = Aright[i+j-70];
    ... }
```

```
for (i=0; i<50; i++)
  if (i<=35) /* splitting-if */
    for (; i<=35; i++)
      for (j=0; j<i; j++) {
        data = Aleft[i+j];
        ... }
  else
    for (j=0; j<i; j++) {
      if(i+j<70)
        data = Aleft[i+j];
      else
        data = Aright[i+j-70];
      ... }
```

Fig. 7.2. Example Code Fragment before and after Loop Nest Splitting

of `i<=35`. The transformation inserts a new *if*-condition, called splitting-if, in the loop nest to check for the condition `i<=35`. The *then*-part of the splitting-if contains the loop nest containing only those statements which were executed when the original condition (`i+j<70`) was true. The *else*-part of the splitting-if is an exact copy of the original loop body. The loop nest splitting transformation reduces the execution of the *if*-statement from 1,225 down to 610 for the example code shown in Figure 7.2.

7.2 Related Work

Data partitioning approaches have been extensively studied in the high-performance computing domain. The approaches propose loop-nest and data layout transformations to enable parallel processing or to improve the cache locality of the applications. Banerjee [17] showed that unimodular matrices could be used to represent loop transformations, *viz.*, *permutation*, *reversal* and *skewing*, in a mathematically elegant form. Wolf et al. [138] presented a loop transformation theory that combined unimodular matrix based representations of loop transformations with direction vectors. This unification allows the determination of a single compound transformation, as opposed to series of transformations, that maximizes an objective function, while satisfying a set of constraints. In addition, the theory enables a single test to determine the legality of the compound transformation.

In the domain of memory constrained embedded systems, numerous approaches [5, 23, 58, 64, 65, 132] to perform loop transformations and data partitioning have been proposed. The goal of the approaches is to efficiently utilize the scratchpad based memory hierarchies and to reduce the energy consumption of the embedded processors.

The authors of [5] propose a data partitioning approach to partition data arrays into disjoint partitions depending on the footprint associated with each of the references. Profit values based on the number of distinct accesses are assigned to the partitions, then a knapsack algorithm is used to perform non-overlayed allocation of the array partitions on the scratchpad memory. The authors of [132] extended this approach by combining loop-nest transformations (*e.g. loop fusion*) with data partitioning to optimize across multiple loops.

Loop-nest transformations are applied to improve the locality and reuse of the data arrays while preserving the semantics of the application.

Kandemir et al. [64, 65] presented the overlayed allocation of array parts on the scratchpad. In [64], authors proposed to partition data arrays into equal sized parts called *tiles*, whereas in [65], data arrays are partitioned into parts of different shapes and sizes called *slabs*. The application is modified such that array parts are swapped in and out of the scratchpad at runtime. The previous two approaches did not consider the reuse of the array elements prior to copying them onto the scratchpad. Therefore, the approaches reduce energy consumption only for those benchmarks for which the energy benefit due to high data reuse can offset the energy overhead caused due to the swapping of array parts.

Authors in [23] proposed a so-called memory hierarchy layer assignment (MHLA) approach which performs the partitioning and the allocation of arrays while considering the data reuse. The approach generates disjoint array parts from the application arrays and then performs the mapping of the array parts to scratchpad memories present in the multi-level memory hierarchy. The approach also modifies the application code such that arrays parts are moved within the memory hierarchy at execution time. An additional advantage of the MHLA approach is that it can also generate an energy efficient memory hierarchy optimized for a given application.

Authors in [58] extended the MHLA approach to generate overlapping (non-disjoint) array parts. Therefore, the approach not only reduces the scratchpad memory requirements of the application but also reduces the overhead caused due to movement of the array parts within the memory hierarchy.

All the data partitioning approaches presented above share the same set of limitations. All the approaches are completely data centric and pay scant regard to the control-flow of the application. In order to reference array parts, the index functions to the unpartitioned array are replaced by significantly complex index functions to the array parts. These index functions [23] are composed of complex mathematical operators even containing *mod* and *div* operators. The instruction sets of various processors do not natively support division and modulo operations and result in costly calls to the runtime library routines. In [124], we demonstrated that application of the MHLA approach for the uni-processor ARM based system leads to worse total energy consumption values of the system compared to the overlayed scratchpad allocation approach presented in the previous chapter.

Another limitation of the approaches is that they are only applicable under the simplifying constraint of perfectly nested loops, exactly known loop bounds and affine index functions. The authors of [59] noted that a significant percentage of applications, even from the embedded multi-media domain, are not written in a form that complies with the above constraints. This limits the applicability of the approaches for applications running on embedded processors. Next, we describe the work related to loop nest transformations for embedded systems.

Loop nest transformations, *e.g.* loop fusion, loop fission, loop unrolling, loop interchange etc., are now a part of all state-of-the-art optimizing compilers. Their impact on many facets of computer performance, such as instruction and data cache performance, regularity of the control flow and the code size of the application have been thoroughly studied. [4] and [15] provide a thorough insight into the loop transformations integrated within optimizing compilers.

Authors in [67] studied the effect of several loop transformations (*loop unrolling*, *loop interchange*, *loop fusion* and *loop tiling*) on the memory system energy caused due to instruction and data accesses. The authors evaluated the loop transformation for four *motion estimation* codes which are an integral part of the MPEG4 video coding standard. They observed that the loop transformations reduced the energy consumed by the data accesses, but significantly increased (by up to 466%) the energy consumed by the instruction fetches. Therefore, the authors concluded that the energy optimizing compilers need to consider both instruction and data locality in a unified optimization framework.

```
for (i=0; i<n; i++) {
  a[i] = a[i]+c;
  if (x<7)
    b[i] = a[i]*c[i];
  else
    b[i] = a[i-1]*b[i-1];
}
```

```
if (x<7)
  for (i=0;i<n; i++) {
    a[i] = a[i]+c;
    b[i] = a[i]*c[i]; }
else
  for (i=0;i<n; i++) {
    a[i] = a[i]+c;
    b[i] = a[i-1]*b[i-i]; }
```

Fig. 7.3. Loop Unswitching

Classical *loop unswitching*, as shown in Figure 7.3, is the loop transformation that is closest to the *loop nest splitting* transformation [39]. The transformation [15] is applied when a loop contains an *if*-statement with a loop-invariant test condition. The loop then is replicated inside each branch of the conditional, saving the overhead of conditional branching inside the loop. Additionally, the transformation enables a loop to be cached by a dynamically loaded loop cache, as the *if*-statement is moved out of the loop. The main disadvantage of *loop unswitching* is that the control flow modifying *if*-statement must not depend on the index variables. The *loop nest splitting* transformation removes this constraint as it allows the splitting of the loop nest containing index variable dependent *if*-statements.

We would like to discuss the advantages and limitations of the proposed approaches. The following are the advantages of our approach.

(a) The approach is very generic and can partition arrays even if they are accessed through irregular access functions. It optimizes benchmarks containing imperfectly nested loops or loop nests with non-constant bounds.
(b) The approach optimizes the total energy consumption of the system. It uses accurate energy functions to determine the energy overhead of additional instruction accesses caused by data partitioning. Consequently, it can determine when it is not beneficial to partition an array.
(c) The approach considers both instruction segments and data array variables for allocation onto the scratchpad.

Our approach has the following limitations which we would like to remove in the future.

(a) The modified access function to partitioned arrays is naive and can be optimized for benchmarks containing perfectly nested loops and affine array accesses.
(b) The approach performs non-overlayed allocation of array parts onto the scratchpad memory which we believe can be improved to overlayed allocation.

7.3 Problem Formulation and Analysis

The goal of the data partitioning approach is to minimize the energy consumption of the application. In order to minimize the energy consumption, the approach decides whether to partition or not an array present in the application. Additionally, it selects the energy optimal set of memory objects which should be allocated in an non-overlayed manner onto the scratchpad memory. If the approach decides to partition an array, then it must select the split point that one of two array partitions will belong to the energy optimal set of memory objects marked for scratchpad allocation. Otherwise, the energy overhead is larger the energy benefit due to data partitioning. In this case, the unpartitioned application consumes less energy than the partitioned application.

In the this section, we present the data partitioning problem along with the details required for understanding the problem. The following subsection presents a discussion on the selecting a candidate array for partitioning, followed by a discussion on splitting point. Subsection 7.3.3 describes the set of memory objects used for the current approach, followed by a brief discussion on the energy model. In the end, we present the formal definition of the data partitioning problem.

7.3.1 Partitioning Candidate Array

The proposed data partitioning approach selects from all the arrays in the application one candidate array for partitioning. The candidate array for partitioning is the highest valence (*i.e.* energy consumption per array element) array which could not be allocated onto the scratchpad when only aggregate variables and code segments are considered for allocation. It can be proved by contradiction that the candidate array provides the maximum potential for energy reduction among all the remaining non-scratchpad allocated arrays. The selection procedure also implies that the candidate arrays can vary for different scratchpad sizes.

7.3.2 Splitting Point

The splitting point refers to the point which bi-partitions the candidate array. Figure 7.4 depicts two possible splitting points for the candidate array `A` and their corresponding four partitioned array variables. If the splitting point is allowed to be placed at any array element, the number of possible array partitions become unmanageable for large arrays. Therefore, we combine a constant number $bsize$ of adjacent array elements of the array `A` to form a basis element b. Now, the array `A` instead of having $size$ elements has $\lceil size/bsize \rceil$ basis elements and the number of possible splitting points is reduced to $\lceil size/bsize \rceil - 1$. A total of $2 * (\lceil size/bsize \rceil - 1)$ partitioned array variables PV are generated, as each splitting point generates two partitioned array variables. This reduces the complexity of the problem and the accuracy of the solution against the solution where each array element is a basis element (*i.e.* $bsize = 1$).

From Figure 7.4 and from the discussion so far, it becomes evident that a two-way partitioning of the array `A` is allowed by the proposed data partitioning approach. Though the approach can support n-way partitioning of the array, it is restricted to consider only two-way partitioning. For the uni-processor ARM based setup, we realized that the n-way

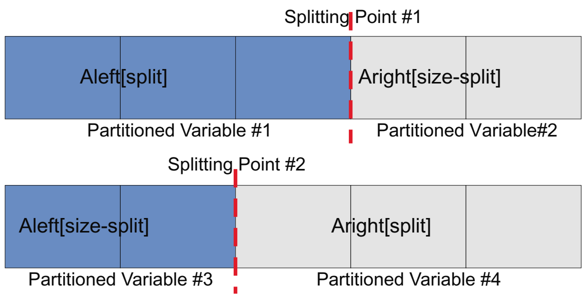


Fig. 7.4. Splitting Points and Partitioned Variables

Memory Optimization	System Architecture	Memory Objects	Explanation
Data Partitioning and Loop Nest Splitting	Uni-processor ARM	$MO \subseteq V \cup BB \cup F$	global variables, basic blocks, functions, partitioned variables, referenced basic blocks, referenced functions

Table 7.1. Memory Objects for Data Partition Approach

partitioning is not beneficial as the resulting complex *if-then-else* structures over-compensate for the energy savings.

7.3.3 Memory Objects

The set of memory objects include functions (F), basic blocks (BB) and aggregate variables (V). In addition, it includes a subset of functions and basic blocks and all partitioned array variables of the application. The functions and basic blocks which contain a reference R to the candidate array are known as *referenced functions* (RF) and *referenced basic blocks* (RBB), respectively. It should be noted that only these functions and basic blocks will get modified if the partitioning approach decides to partition the candidate array. Therefore, they are included in the set of memory objects to represent the transformed application. The set of memory objects also contain $2 * (\lceil size/bsize \rceil - 1)$ partition variables PV generated from the candidate array. Table 7.1 summarizes the memory objects for the proposed data partitioning approach.

7.3.4 Energy Model

The energy function $E(inst, imem, dmem)$ shown below returns the energy dissipated by the system during the execution of an instruction ($inst$) which is fetched from the instruction memory ($imem$) and possibly accesses a data value from the data memory ($dmem$). The energy function is derived from the energy model presented in Chapter 3.

$$E(inst, imem, dmem) = E_{if}(imem) + E_{ex}(inst) + E_{da}(dmem) \qquad (7.1)$$

The energy function $E(\mathit{inst},\mathit{imem},\mathit{dmem})$ is used to compute the energy $E(mo,mem)$ dissipated by a memory object $mo \in MO \subseteq V \cup BB \cup F \cup PV \cup RBB \cup RF$ which is executed from the instruction memory (*mem*) or accessed from the data memory (mem).

$$E(mo,mem) = \begin{cases} E_{bb}(mo,mem) & \text{if } mo \in BB \cup RBB \\ E_{fn}(mo,mem) & \text{if } mo \in F \cup RF \\ E_{var}(mo,mem) & \text{if } mo \in V \\ E_{pv}(mo,mem) & \text{if } mo \in PV \end{cases} \tag{7.2}$$

The energy function $E(mo,mem)$ (*cf.* Equation 7.2) classifies the memory objects into four disjoint sets and computes the energy dissipation values differently for each of the set. The classification of the memory objects into sets is presented in the following:

(a) basic blocks and referenced basic blocks $(BB \cup RBB)$
(b) functions and referenced functions $(F \cup RF)$
(c) global variables including the partitioning candidate array (V)
(d) partitioned array variables for the partitioning candidate array (PV)

The energy function $E_{bb}(mo,mem)$ presented in the following computes the energy dissipated by memory object $mo \in \{BB \cup RBB\}$ belonging to the set of basic blocks and referenced basic blocks:

$$E_{bb}(mo,mem) = n_r * \mathit{inst}(mo) * E(mov,mem,MM) \tag{7.3}$$

where n_r is the number of executions of the memory object mo and $\mathit{inst}(mo)$ returns the number of instructions contained within the memory object. A basic block is characterized by the property that each instruction is executed if the execution flow enters the basic block. Therefore, the energy dissipated during a single execution of the basic block can be computed as the product of the number of instructions belonging to the basic block and the average energy dissipated by an instruction. The above average energy value simplification works for ARM based setups since all instructions dissipate almost the same amount of energy. In the present setup, a register-move instruction is used to represent the instruction with average energy dissipation.

The energy dissipation function $E_{fn}(mo,mem)$ of a memory object belonging to set of functions and referenced functions is the following:

$$E_{fn}(mo,mem) = \sum_{bb_i \in mo} E_{bb}(bb_i,mem) \tag{7.4}$$

As shown in the above equation, the energy dissipated by a function $mo \in F \cup RF$ is the sum of the energy dissipated by the basic blocks contained within the function. The energy function $E_{var}(mo,mem)$ presented below computes the energy dissipated by a variable:

$$\begin{aligned} E_{var}(mo,mem) = {} & n_r(mo) * [E(\mathit{load},MM,mem) - E(mov,MM,mem)] \\ & + n_w(mo) * [E(\mathit{store},MM,mem) - E(mov,MM,mem)] \end{aligned} \tag{7.5}$$

where, $n_r(mo)$ and $n_w(mo)$ return the number of read and write accesses to the variable, respectively. Using the energy model, we compute that the energy dissipated for a read access is equal to the difference in the energy dissipation of a load instruction and a register-move instruction. Similarly, the energy dissipated due to a write access is computed to be the difference in the energy dissipation of a store and a register-move instruction. The above equation sums the energy dissipated for all read and write access to compute the aggregate energy dissipated by the variable.

The energy dissipated by a memory object mo belonging to the set of partitioned variables PV is computed as follows:

$$\begin{aligned} E_{pv}(mo, mem) &= n_r(mo) * [E(load, MM, mem) - E(mov, MM, mem)] \\ &\quad + n_w(mo) * [E(store, MM, mem) - E(mov, MM, mem)] \quad (7.6) \\ n_r(mo) &= \sum_{b_i \in pv} n_r(b_i) \quad (7.7) \\ n_w(mo) &= \sum_{b_i \in pv} n_w(b_i) \quad (7.8) \end{aligned}$$

where, $n_r(mo)$ and $n_w(mo)$ return the number of read and write accesses to the partitioned variable, respectively. The number of the read accesses $n_r(mo)$ to a partitioned variable (*cf.* Equation 7.7) is the sum of the number of read accesses $n_r(b_i)$ to basis elements $b_i \in PV$ belonging to the partitioned variable. The number of write accesses $n_w(mo)$ to a partitioned variable can be computed in a similar manner.

The following equation presents the energy overhead caused due to the partitioning of the candidate array A:

$$E_{overhead} = n_r(A) * E(\texttt{READ\_ACCESS}) + n_w(A) * E(\texttt{WRITE\_ACCESS}) \quad (7.9)$$

where $E(\texttt{READ\_ACCESS})$ and $E(\texttt{WRITE\_ACCESS})$ return the energy dissipated during a single execution of the `READ_ACCESS` and `WRITE_ACCESS` macros, respectively. As discussed in Section 7.1, these access macros (*cf.* Figure 7.1) are required for accessing the correct array partition. The read and write access macros are used, respectively, on each read and write access to the partitioning candidate array. Therefore, $E_{overhead}$ represents the energy overhead caused due to the partitioning of the array A. In the following subsection, we formally define the data partitioning problem.

7.3.5 Problem Formulation

Problem 7.1 (Data Partitioning (DP)). Given the set of memory objects MO, a candidate array A marked for partitioning and a memory hierarchy consisting of a scratchpad and a main memory. The problem is to determine a subset $MO_{SPM} \subseteq MO$ of the set of memory objects such that the the total energy profit E_{Total}, achieved due to allocation of memory objects $mo_i \in MO_{SPM}$ to the scratchpad memory, is maximized.

$$E_{Total} = \sum_{mo_i \in MO} E_{profit}(mo_i) - p(A) * E_{overhead} \quad (7.10)$$

where, $E_{profit}(mo)$, shown below, computes the energy profit obtained by allocating a memory object to the scratchpad memory.

$$E_{profit}(mo) = E(mo, MM) - E(mo, SPM) \quad (7.11)$$

Moreover, $p(A)$ is a binary variable to represent the partitioning of the array A.

$$p(A) = \begin{cases} 1, \text{ if the array variable } A \text{ is partitioned} \\ 0, \text{ otherwise} \end{cases}$$

The maximization of the total energy profit E_{Total} is to be performed under the following constraints:

(a) The aggregate size of the memory objects marked for scratchpad allocation must be less than the size of the scratchpad memory.

$$\sum_{mo_i \in MO_{SPM}} size(mo_i) \leq size(SPM) \tag{7.12}$$

(b) Only one of the partitioned array variable or the candidate array should be allocated onto the scratchpad memory.

(c) If the candidate array A is not partitioned (*i.e.* $p(A) = 0$), then none of the referenced basic blocks or the referenced functions should be allocated onto the scratchpad memory.

(d) If the candidate array A is partitioned (*i.e.* $p(A) = 1$), then all basic blocks and functions for which referenced basic blocks or referenced functions, respectively, exist should not be allocated onto the scratchpad memory.

The data partitioning problem needs to decide between the original application and the modified application. One approach is to compare the energy consumption values of the applications and choose the one with lower energy consumption. The other approach is to consider their energy difference to a high energy value, name the difference as *energy savings* and choose the one with higher energy savings. We choose the latter approach as it aids the problem formulation. The high energy value corresponds to the energy consumed by a system for which all memory objects are assigned to the main memory.

The problem can be viewed as a decision problem, to choose between referenced basic blocks, referenced functions and a partitioned array on one side and original basic blocks, functions and the unpartitioned array on the other side. The decision is to maximize the energy savings while ensuring that the combined size of the chosen memory objects does not exceed the scratchpad size. The problem can also be viewed as a variant of the knapsack problem [43] where objects to be packed in the knapsack have mutual exclusivity constraints with other objects. Therefore, the data partitioning problem is also an NP-hard problem. In the following section, we describe an ILP based approach for solving the data partitioning problem.

7.4 Data Partitioning

The data partitioning approach extends the previously known non-overlayed scratchpad allocation approach [115] which allocated aggregate variables along with code segments onto the scratchpad memory. The proposed data partitioning approach partitions the data array and also computes the energy optimal set of memory objects MO_{SPM} to be allocated onto the scratchpad memory. The approach, as shown in Figure 7.5, works in the following stepwise manner:

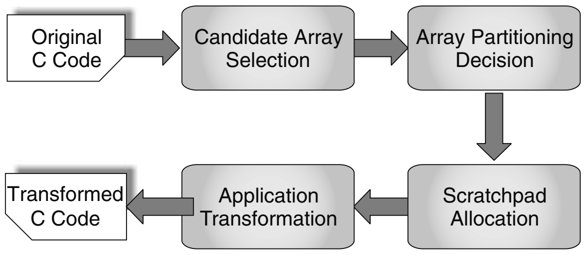

Original C Code
Candidate Array Selection
Array Partitioning Decision
Scratchpad Allocation
Application Transformation
Transformed C Code

Fig. 7.5. Workflow of the Data Partitioning Approach

1. It chooses a candidate array A among all the possible arrays for partitioning.
2. It decides whether partitioning the array A will result in reduced energy consumption of the application.
3. If the array A is partitioned, then it computes the splitting point `split` for partitioning the array and determines the set of memory objects MO_{SPM}, including one of the partitioned array, for energy optimal scratchpad allocation. If the array A is not partitioned, then it determines the set of memory objects MO_{SPM}, containing only the aggregate variables and original basic blocks and functions, for scratchpad allocation.
4. Given the partitioning decision and the splitting point, it modifies the original application according to the splitting point.

The procedures to select the candidate array (*cf.* Step 1) and to modify the application (*cf.* Step 4) have already been presented in Subsection 7.3.1 and Section 7.1, respectively. Step 2 and Step 3 are solved simultaneously in a phase coupled manner using an ILP based approach. The ILP formulation of the data partitioning approach is presented in the following subsection.

7.4.1 Integer Linear Programming Formulation

In order to formulate the data partitioning problem (*cf.* Subsection 7.3.5) as an *integer linear programming* problem, we need to define a couple of binary variables. The first binary variable is used to represent the location of a memory object $mo_i \in MO$ in the memory hierarchy and is defined as follows:

$$l(mo_i) = \begin{cases} 1, \text{ if the memory object } mo_i \text{ is present in the SPM} \\ 0, \text{ otherwise} \end{cases} \tag{7.13}$$

The second binary variable, also defined in Subsection 7.3.5, represents the partitioning decision of the candidate array A.

$$p(A) = \begin{cases} 1, \text{ if the array variable } A \text{ is partitioned} \\ 0, \text{ otherwise} \end{cases} \tag{7.14}$$

Objective Function:
The above variable definitions (*cf.* Equations 7.13 and 7.14) are used to formulate the objective function which needs to maximized. The objective function represents the total

energy profit achieved through the mapping of memory objects onto the scratchpad and is defined as follows:

$$E_{Total} = \left[\sum_{mo_i \in MO} E_{profit}(mo_i) * l(mo_i) \right] - p(A) * E_{overhead} \tag{7.15}$$

where, the energy function $E_{profit}(mo_i)$ shown below computes the difference between the energy values when the memory object mo_i is mapped to the main memory $E(mo_i, MM)$ and to the scratchpad $E(mo_i, SPM)$.

$$E_{profit}(mo) = E(mo, MM) - E(mo, SPM) \tag{7.16}$$

Energy function $E_{overhead}$ (*cf.* Equation 7.9) denotes the energy overhead incurred due to the partitioning the array A.

Constraints:
We add a few constraints to the ILP formulation such that it is restricted to generate only valid solutions. The first constraint *viz.*, scratchpad size constraint, restricts the aggregate size of memory objects mapped to the scratchpad to be less than the size of the scratchpad.

$$\sum_{mo_i \in MO} l(mo_i) * size(mo_i) \leq size(SPM) \tag{7.17}$$

A second constraint is added to ensure that if the array A is not partitioned (*i.e.* $p(A) = 0$), then memory objects $mo_i \in RBB \cup RF \cup PV$ belonging to the set of referenced basic blocks, referenced functions and partitioned variables are not selected for scratchpad allocation.

$$\left[\sum_{mo_i \in RBB \cup RF \cup PV} l(mo_i) \right] - C * p(A) \leq 0 \tag{7.18}$$

The third constraint ensures that the opposite of the above constraints also does not occur. It ensures that if the array A is partitioned (*i.e.* $p(A) = 1$), no memory object belonging to a subset of memory objects MO_{org} containing original basic blocks and original functions is selected for scratchpad allocation. The constraint, as shown in the following, also ensures that even the array A is not allocated to the scratchpad.

$$\left[\sum_{mo_i \in MO_{org}} l(mo_i) \right] - C * [1 - p(A)] \leq 0 \tag{7.19}$$

$$MO_{org} = \{bb_i | rbb_i \in MO\} \cup \{f_i | rf_i \in MO\} \cup A \tag{7.20}$$

A basic block or a function belongs to this memory object subset MO_{org} (*cf.* Equation 7.19) iff there exists a corresponding referenced basic block or a referenced function, respectively. Only the memory objects belonging to the set MO_{org} are restricted by the above constraint while the remaining memory objects are unrestricted to be allocated onto the scratchpad.

In the above constraints, C is any sufficiently large constant to ensure that Equations 7.18 and 7.19 always remain less than or equal to zero.

The final constraint is added to restrict that a maximum of one array partition is allocated to the scratchpad memory.

$$\sum_{mo_i \in PV} l(mo_i) \leq 1 \tag{7.21}$$

The solution to ILP formulation of the data partitioning (DP) problem determines the subset of memory objects $(MO_{SPM} = \{mo_i | l(mo_i) = 1\})$ which maximizes the objective function representing the energy benefit due to the allocation of memory objects $mo_i \in MO_{SPM}$ to the scratchpad memory.

A commercial ILP solver [32] is used to solve the ILP formulation. The number of program code memory objects is bounded by the number of basic blocks $O(|BB|)$ in the application, and the number of data variable memory objects is bounded by the sum of global variables and partitioned variables $O(|V| + |PV|) = O(|V| + \lceil size(A)/bsize \rceil)$. Therefore, the total number of memory objects is $O(|BB| + |V| + \lceil size(A)/bsize \rceil)$. The ILP solver [32] requires only a minimal runtime of less than a second to compute the solution of the ILP formulation. The following section presents the loop nest splitting approach to improve the control flow of the partitioned application.

7.5 Loop Nest Splitting

This section presents a brief overview of the *loop nest splitting* transformation. The interested reader is referred to [39] for a detailed description and a comprehensive analysis of the loop nest splitting approach. As described in Section 7.1, the data partitioning approach improves scratchpad utilization but impairs the control flow of the application. The partitioning approach inserts *if*-statements in the loop nests and thus incurs significant performance and energy penalty. The loop nest splitting transformation minimizes the penalty by rewriting the loop nest such that the execution of *if*-statements is minimized. We start by describing the basics of loop nest splitting.

Basics of Loop Nest Splitting:
Given a loop nest $\Lambda = \{L_1, \ldots, L_N\}$ of depth N, L_l denotes a single loop l with its index variable i_l and the lower and upper bounds, lb_l and ub_l, respectively. Initially, the transformation could only optimize loop nests with constant loop bounds. Later, it was extended to process loop nests for which all loops L_l except the outermost loop can have lb_l and ub_l as the affine functions of the enclosing loops.

Each loop L_l can contain one or more *if*-statements whose conditions depend on the index variables of the loop nest Λ. Such conditions are said to be loop-dependent. The *if*-statements must have the format `if` $(C_1 \oplus C_2 \oplus \ldots)$, where each C_i can be a loop-dependent or a loop-invariant condition. These conditions, however, must be combined using logical operators $\oplus \in \{\wedge, \vee\}$. Every loop-dependent condition C must be an affine expression of the index variables i_l and can thus be represented as $C = \sum_{l=1}^{N} (c_l * i_l) + c \geq 0$ for constants $c_l, c \in \mathbb{Z}$.

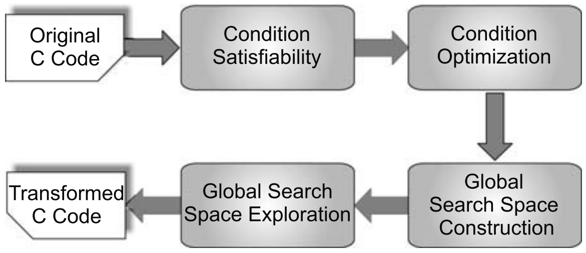


Fig. 7.6. Workflow of the Loop Nest Splitting Transformation

Loop Nest Splitting Transformation:
The loop nest splitting transformation, as shown in Figure 7.6, consists of the following four different steps:

1. Condition Satisfiability.
2. Condition Optimization.
3. Global Search Space Construction.
4. Global Search Space Exploration.

The *condition satisfiability* step independently analyzes all conditions C_i of the *if*-statements contained within the loop nest Λ. Each condition defines a subset of the *iteration space* of the loop nest and is represented as a polytope. The condition satisfiability step, based on the generated polytope, determines if a condition C_i is either always satisfied or always unsatisfied for all iterations of the loop nest. Such conditions are redundant conditions and are replaced by their corresponding truth values in the *if*-statement. These conditions are also pruned from the subsequent analysis steps.

In the second step *viz.*, *condition optimization*, all non-redundant conditions are separately analyzed and optimized. During this step, each condition C_i is independently optimized assuming that the loop nest Λ contains a single *if*-statement with a single condition, namely C_i. For each C_i, a locally optimized solution of minimized *if*-statement executions, represented as a polytope $P_{C_i}\left(\left[lb'_{C_i,l}, ub'_{C_i,l}\right]\right)$, is determined.

An ILP based approach is deemed unsuitable for the current optimization due to the non-linearity of the objective function. Therefore, a *genetic algorithm* (GA) based approach is used to compute the polytope P_{C_i}. The fitness function computes the inverse of the number of executed *if*-statements after loop nest splitting, *i.e.* the lower the number of executed *if*-statements, the higher is the fitness of an individual. The fitness function has a linear time complexity for loop nests with constant loop bounds, but an exponential time complexity for loop nests with affine bounds.

The *global search space construction* step combines the local solution for each condition C_i to generate a *global search space* G. In order to generate G, first the polyhedra P_{C_i} associated with the conditions of a single *if*-statement are combined according to the binary logical operators $\oplus \in \{\wedge, \vee\}$. For example, if two conditions C_i and C_j are connected using the $\wedge$ operator, then their corresponding polyhedra P_{C_i} and P_{C_j} are intersected. For the $\vee$ operator, a union of the polyhedra is generated. This way, a polyhedron is generated representing those iterations for which a single *if*-statement is satisfied. Since all *if*-statements in a loop nest need to be satisfied for loop nest splitting, all the polyhedra for the different *if*-statements are intersected. The resulting polyhedron called the *global search space G* represents those iterations for which all *if*-statements are satisfied.

It was observed that the global search space G is composed of a finite union of polyhedra: $G = R_1 \cup R_2 \cup \cdots \cup R_M$, where each R_i represents a region in the iteration space where all the *if*-statements in the loop nest are satisfied. The goal of the loop nest splitting transformation is to minimize the execution of *if*-statements. However, it was observed that by using all the regions R_i of G to perform loop nest splitting the resulting loop nest executed more *if*-statements than the minimum value. Therefore, the final step *viz.*, *Global Search Space Exploration*, is used to determine better solutions. The step uses a GA based approach to prune G and restricts it to those regions R_i which lead to the globally optimized solution representing the minimum number of executed *if*-statements. Based on the regions R_i present in the pruned search space G, the loop nest in the original application is transformed.

7.6 Experimental Results

In this section, the data partitioning and loop nest splitting approaches are evaluated for the uni-processor ARM based setup. The data partitioning approach is implemented only for ARM processors, however it can be easily extended to optimize benchmarks for M5 DSP. The loop nest splitting approach is an architecture independent source-level transformation and therefore is applicable to all processor architectures. The experiments are conducted according to the experimental workflow (*cf.* Section 3.1) for the uni-processor ARM based setup.

For the experiments, benchmarks from different application domains were selected to demonstrate the efficacy of the proposed approaches. First, a 40 order *fir* filter routine representing a typical embedded DSP algorithm is used. Second, the sorting algorithms *bubble sort*, *insertion sort* and *selection sort* were analyzed. Finally, a complete MPEG4 *motion estimation* routine was optimized. Table 7.2 presents the benchmarks along with their code and data sizes. The *fir* and *motion estimation* benchmarks feature perfectly nested loops and affine access functions. Therefore, they can also be optimized by the affine function based data partitioning approaches. The novelity of our approach is that it can as well optimize the sorting routines which do not satisfy the conditions for optimization required by the other data partitioning approaches.

The experiments in this section compare the two proposed approaches and a non-overlay based scratchpad allocation (SA) [115] approach. The SA approach allocates aggregate array

Benchmark	Code Size (bytes)	Data Size (bytes)	System Architecture
bubble sort	92	1412	uni-processor ARM
fir	92	3712	uni-processor ARM
insertion sort	88	1416	uni-processor ARM
motion estimation	644	176140	uni-processor ARM
selection sort	76	1412	uni-processor ARM

Table 7.2. Benchmark Programs for the Evaluation of Data Partitioning and Loop Nest Splitting

variables along with code segments onto the scratchpad. The proposed data partitioning (DP) approach divides a candidate array into two array partitions and then allocates an array partition along with code segments onto the scratchpad in an non-overlayed manner. The other proposed approach combines data partitioning with loop nest splitting (DP+LS) such that overhead caused by the data partitioning approach is minimized.

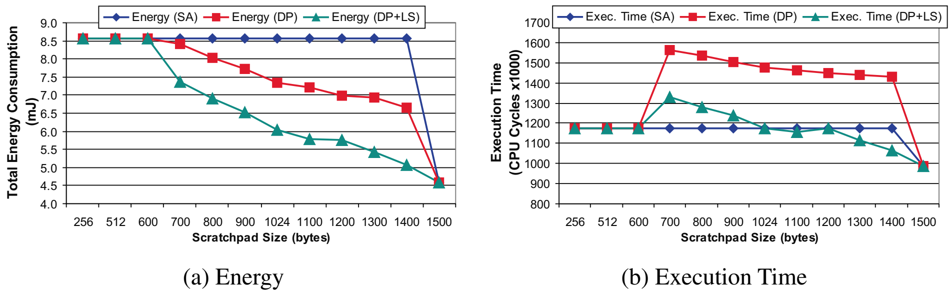


Fig. 7.7. Selection Sort: Comparison of Data Partitioning, Data Partitioning + Loop Nest Splitting and Scratchpad Allocation Approaches

Figure 7.7 compares the two proposed approaches (DP, DP+LS) with the scratchpad allocation (SA) approach for the *selection sort* benchmark. A number of important observations can be made from the figure. First, the energy consumption and the execution time of the benchmark for the SA approach demonstrate a step-wise decrease. This is because the SA approach allocates the scratchpad at a coarse granularity due to aggregate array variables. Therefore, the energy consumption and the execution time values for the SA approach (*cf.* Figure 7.7) do not change when the scratchpad size is increased from 256 bytes upto 1400 bytes, as the large array in the benchmark could not be allocated in its entirety onto the scratchpad. In contrast, the proposed approaches smoothen the energy consumption curve as they perform scratchpad allocation at a finer granularity.

The second observation is that the data partitioning (DP) approach can decide, depending on the scratchpad size, if it is energy efficient to partition the candidate array. For example, the DP approach decides not to partition the array (*cf.* Figure 7.7) for scratchpad sizes between 256 and 600 bytes. This is because for small scratchpad sizes, the energy overhead due to array partitioning is larger than the energy gain achieved due to allocating a small array partition onto the scratchpad. The array is again not partitioned for large scratchpads, *e.g.* at 1500 bytes in Figure 7.7, as the aggregate array variable can be allocated onto the large scratchpad. For all scratchpads of size between 700 bytes and 1400 bytes, the array is partitioned into increasingly larger partitions to achieve energy reduction.

The third observation is that the partitioning the array (*cf.* 700 bytes in figures 7.7(a) and 7.7(b)) reduces the energy consumption and increases the execution time for the DP approach when compared to the corresponding values for the SA approach. The reason for this behavior is that the DP approach reduces energy consumption by utilizing the scratchpad space for array partitions. However, it inserts access macros to reference the correct array partition into the application source code, and the execution of the inserted access macros causes an increase in the execution time of the benchmark. Therefore, we conclude that the data partitioning approach optimizes the benchmark for energy consumption and not for performance which is not a common phenomenon in energy aware code optimizations.

Last, the combined (DP+LS) approach significantly improves the energy consumption as well as the performance of the benchmark compared to the results obtained by the DP approach. The loop nest splitting (LS) approach minimizes the execution of *if*-statements which were inserted by the DP approach. For scratchpads larger than 1024 bytes, the execution time of the benchmark optimized by the combined (DP+LS) approach is even smaller than that obtained by the SA approach. Next, an overall evaluation of the data partitioning and the combined approach across all benchmarks is presented.

Figure 7.8 and Figure 7.9 depict the relative processor energy, memory energy, total energy and execution time values for all benchmarks optimized using the DP approach and the combined (DP+LS) approach, respectively. The energy and performance values due to the proposed approaches are relative to the corresponding values due to the SA approach, which are shown as the unit value baseline in the figures. For the current experiments, the scratchpad size for the sorting benchmarks and the *fir* routine is set to 1400 bytes and 1800 bytes, respectively. The *motion estimation* benchmark with its large video frames is analyzed for a 119k bytes scratchpad memory. From Figure 7.8, a couple of observations can be made for the benchmarks optimized using the DP approach.

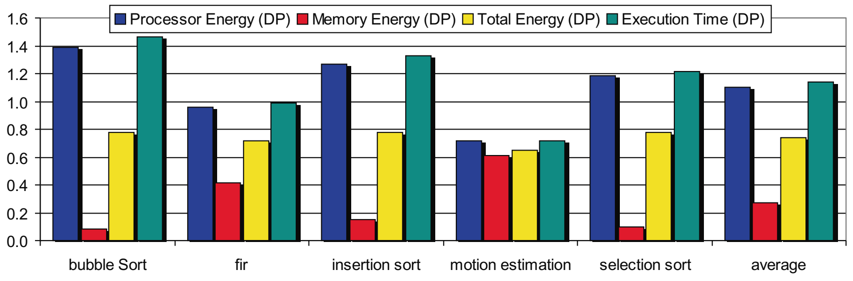


Fig. 7.8. Overall Comparison of Data Partitioning and Scratchpad Allocation Approaches

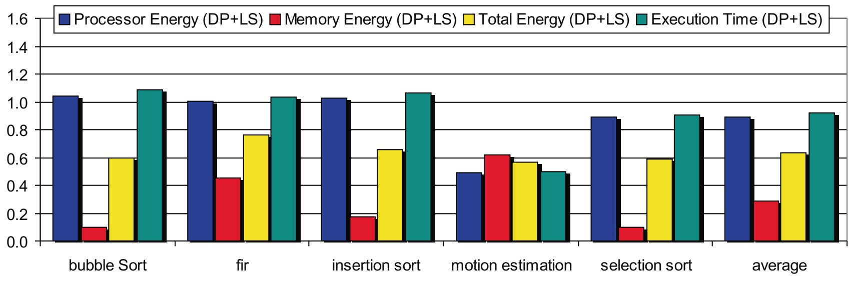


Fig. 7.9. Overall Comparison of the Combined Data Partitioning and Loop Nest Splitting Approach and Scratchpad Allocation Approach

First, the DP approach reduces the memory energy consumption to a large extent compared with that for the SA approach. The reductions in the memory energy consumption values for the sorting routines range between 82% and 93%. For the *fir* and *motion estimation* benchmarks, the reduction in the memory energy consumption is 59% and 39% compared with that for the SA approach, respectively.

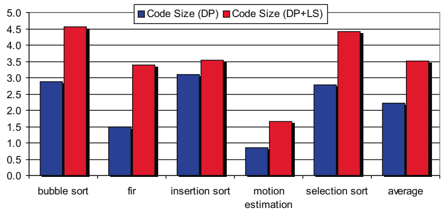
Second, the relative processor energy consumption values for the sorting routines are higher than the unit baseline, while they are lower for *fir* and *motion estimation* benchmarks. The same behavior is also observed for the relative execution time values for the benchmarks. The reason for improved performance and processor energy consumption is that an access to the scratchpad is both energy efficient and faster than an access to the main memory. Therefore, for data dominated benchmarks, the overhead for executing additional instructions is over-compensated by the energy and performance efficient accesses to the partitioned array located on the scratchpad memory. An increase of 39% and 47% in the processor energy consumption and the execution time, respectively, is observed for the *bubble sort* benchmark. In contrast, a reduction of 29% in both the processor energy and the execution time is observed for the *motion estimation* benchmark. The total energy, being the sum of the processor energy and the memory energy, demonstrates a reduction of between 22% and 35%. On average, the DP approach saves a quarter (25%) of the total energy compared to that for the SA approach.

From Figure 7.9, it can be observed that the combined (DP+LS) approach substantially improves both the energy and the execution time values for all but one benchmark compared with those for the DP approach. For the *fir* benchmark, the combined (DP+LS) approach leads to slightly degraded (by about 4%) total energy consumption and execution time values. This is because loop nest splitting increases the code size and in this case modifies the scratchpad allocation achieved by the data partitioning approach. On the other hand, a substantial reduction of about 35% in both the total energy consumption and the execution time is observed for the *bubble sort* benchmark when compared to those values achieved by the DP approach.

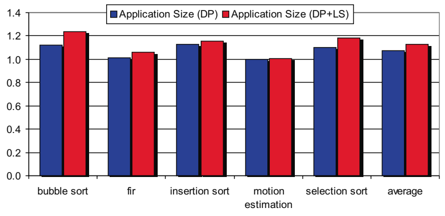
The comparison of the combined (DP+LS) approach with the SA approach, in Figure 7.9, reveals that for the *motion estimation* benchmark, the execution time and the total energy consumption values are reduced by 51% and 43%, respectively. On average, a reduction of 37% in the total energy consumption is observed. The average execution time values are also better than those obtained for the SA approach.

Both the DP and combined (DP+LS) approaches cause an increase in the code size of the benchmark. Figure 7.10 evaluates the increase in the code size and the application size (*i.e.* sum of the code size and the data size) of the benchmarks due to the two approaches compared with the sizes of the original unoptimized benchmark. From Figure 7.10(a), it can be easily observed that for small sorting benchmarks, the increase in the code due to the DP approach is fairly substantial and ranges between 180% and 210%. For the *motion estimation* benchmark, a reduction of only 16% in the code size is measured which is caused by the insertion of less code for register spilling after data partitioning.

The combined (DP+LS) approach results in an even higher increase in the code sizes of the benchmarks. The code size increase for the combined (DP+LS) approach ranges between 66% and 350% over the code size of the original benchmark. The loop nest splitting approach replicates the body of the loop nest and therefore, leads to a substantial increase in the code size. However, the increase in the total application size, which is the measure of the size of



(a) Code Size



(b) Application Size

Fig. 7.10. Code and Application Size Comparison for Data Partitioning and Loop Nest Splitting Approaches

the main memory required to hold the entire application, is moderate. A maximum increase of 22% in the application size due to the combined approach is observed for the *bubble sort* benchmark. On the average, the DP approach and the combined (DP+LS) approach lead to a modest increase of 7% and 13% in the application size, respectively.

7.7 Summary

In this chapter, we proposed data partitioning based scratchpad allocation approaches to optimize embedded applications. The data partitioning approach partitioned the large arrays present in the application and allocated code segments and one of the array partitions onto the scratchpad memory. The approach partitioned the array when the overhead due to partitioning is lower than the energy gains achieved by allocating the partitioned array onto the scratchpad memory. The combined loop nest splitting and data partitioning approach minimized the overhead caused by data partitioning.

Both approaches achieved significant benefits in terms of energy consumption compared with a scratchpad allocation approach which could allocate only aggregate array variables and code segments onto the scratchpad memory. The data partitioning approach leads to a reduction between 20% and 50% in the total energy consumption of the benchmarks. However, the approach causes an average increase of 18% in the execution time. The combined approach not only achieved higher energy savings but also reduced the execution time of the benchmarks. On average, the combined approach reduced the total energy consumption and the execution time by 37% and 9%, respectively.

The approaches presented in this chapter were published in [40] and [126].

8

Scratchpad Sharing Strategies for Multiprocess Applications

In the previous chapters, scratchpad allocation approaches to optimize single process applications were proposed. However, most of the contemporary embedded devices like mobile phones, execute complex multiprocess applications. Applying the previously presented scratchpad allocation approaches to a multiprocess application will result in non-optimal energy reductions, as the approaches can consider only a single process for allocation. Therefore, in order to circumvent this problem we propose a set of three strategies to share the scratchpad memory among the processes of a multiprocess application with the objective to minimize its energy consumption. The first strategy is beneficial for large scratchpads, while the second is beneficial for small scratchpads. The third strategy is relatively complex but outperforms the previous two strategies.

The focus of this chapter is to propose approaches which are analyzable at design time and which preserve the predictable characteristic of the scratchpad memories. In that respect, this is the first work on fully-analyzable scratchpad sharing approaches for multiprocess applications. The strategies are proposed for systems with a memory hierarchy consisting of an L1 scratchpad and a background main memory.

The rest of the chapter is organized as follows: The following section provides an introduction to the proposed approaches, followed by the presentation of a motivating example. Section 8.3 presents a brief survey of the related work and Section 8.4 presents the preliminaries for the approaches. Sections 8.5, 8.6 and 8.7 define the problems and also describe the optimal approaches to solve the problems. The experimental setup is described in Section 8.8 and the experimental results to evaluate the strategies are described in Section 8.9. Section 8.10, ends the chapter with a short summary.

8.1 Introduction

The approaches proposed in this chapter enable sharing of the scratchpad memory among the processes of a multiprocess application under the objective to minimize the energy consumption of the application. A set of three scratchpad sharing strategies, *viz.*, Scratchpad Non-Saving/Restoring Context Switch (Non-Saving), Scratchpad Saving/Restoring Context Switch (Saving) and Hybrid Scratchpad Saving/Restoring Context Switch (Hybrid), are proposed. The names of the strategies are representatives of the activity that occurs at

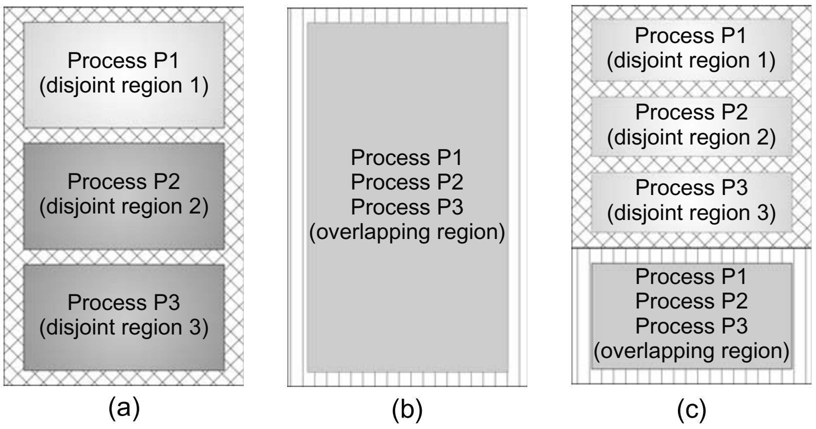


Fig. 8.1. Scratchpad Sharing Strategies: (a) Non-Saving (b) Saving and (c) Hybrid

context switch time for these strategies. The proposed strategies use the non-overlayed allocation approach [115] to allocate memory objects of each process to its assigned scratchpad memory region.

The Non-Saving approach partitions the scratchpad into disjoint regions such that each process is exclusively assigned a region. The approach then uses the non-overlayed allocation approach to allocate memory objects of each process onto its corresponding region. Figure 8.1(a) shows the distribution of the scratchpad into three disjoint regions. The benefit of partitioning the scratchpad into disjoint regions is that the contents of the regions need not be updated at context switch time. The Non-Saving approach is beneficial for large scratchpad memories, as they can be partitioned into adequately sized regions for the processes of the application.

In contrast to the previous approach, the Saving approach shares the scratchpad (*cf.* Figure 8.1(b)) as a common overlapping region and each process assumes that the entire scratchpad is exclusively allocated to itself. The memory objects for each process are copied to the scratchpad when the process is scheduled to execute on the processor. They are copied back to the main memory when the process is scheduled off the processor. However, during the time the process is executing, its memory objects are not swapped in and out of the scratchpad, as they are allocated using a non-overlayed allocation approach. The approach reduces the energy consumed by the application when the copy overhead at each context switch is less than the energy reduction achieved due to the improved scratchpad utilization. For small scratchpads, the Saving approach is better than the Non-Saving approach, as the energy savings are larger than the relatively small copy overhead.

The Hybrid approach, as the name suggests, combines the previous two approaches for the shared utilization of the scratchpad. As shown in Figure 8.1(c), the approach distributes the scratchpad into many disjoint regions and a single overlapping region. The disjoint regions are exclusively assigned to the processes, while the overlapping region is shared by all the processes. The approach ensures that at every context switch, only the overlapping region is updated with the memory objects of the executing process. The hybrid approach minimizes the energy consumption of the application for all scratchpad sizes, as it can partition the scratchpad into both disjoint and overlapping regions. However, the approach requires a complex analysis of the application and a longer computation time to partition the scratchpad. The following section demonstrates the benefit of the proposed approaches with help of a motivating example.

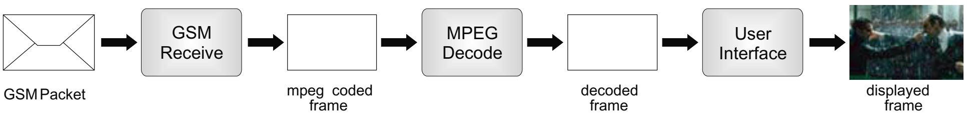


Fig. 8.2. Workflow of a Video Phone Application

Process	Energy fn.	0 kB	1 kB	2 kB	3 kB	4 kB
GSMReceive	$f^N_{receive}$	40	25	20	18	18
MPEGDecode	f^N_{decode}	100	84	60	60	57
UserInterface	f^N_{ui}	20	9	8	8	8

Table 8.1. Energy Functions (Abstract Units) for Video Phone Application

8.2 Motivating Example

We now introduce a motivating example of a video phone application consisting of three simultaneously running processes, *viz.*, GSMReceive, MPEGDecode and UserInterface. Figure 8.2 depicts the workflow of the video phone application. The GSMReceive process receives the GSM packets over the network containing the MPEG video frames. The process then unpacks the packets and stores coded MPEG video frames in an intermediate buffer. The MPEGDecode process decodes the frames from the intermediate buffer and stores the decoded frames in a decode frame buffer. These decoded frames are then displayed on the screen by the UserInterface process.

Table 8.1 presents the energy consumption values of the processes when each process is exclusively assigned scratchpad regions of varying sizes. It can be observed from the table, all the processes have different energy consumption values and have different scratchpad memory requirements. The memory objects of the processes are allocated in a non-overlayed manner onto the assigned scratchpad region and the energy consumption of each process is computed independently. The energy values presented in Table 8.1 are representative of the real-life energy values. However, for the sake the simplicity they are proportionally reduced and are presented in abstract units. For example, the MPEGDecode process consumes 84 units of energy when it utilizes a 1 kB scratchpad region.

Let us assume that the video-phone application is executed on our uni-processor ARM based system having a 4 kB scratchpad memory. Let us also assume that an allocation approach to allocate memory objects of a single process onto the scratchpad is used. The approach selects memory objects from the MPEGDecode process, as it is leads to the minimum energy consumption of the application under the assumption that the scratchpad can be allocated to a single process. In this scenario, when the MPEGDecode process receives 4 kB scratchpad and the other processes receive no scratchpad, the energy consumed by the application is $40 + 57 + 20 = 117$ units. The other potential assignments are when

GSMReceive or UserInterface receive 4 kB scratchpad. The energy consumption values for these scenarios are $18+100+20=138$ or $40+100+8=148$ units, respectively.

The energy consumed by the video-phone application can be reduced further, if the restriction to assign the scratchpad memory to a single process is removed. For example, if 1k, 2k and 1k bytes scratchpad regions are assigned to the GSMReceive, MPEGDecode and UserInterface processes, respectively, then the energy consumed by the application is reduced to $25+60+9=94$ units. On the other hand, an improper assignment of 0, 1k and 3k bytes scratchpad regions to GSMReceive, MPEGDecode and UserInterface processes, respectively, can also lead to an increased energy consumption of $40+84+8=132$ units. Therefore, there is a need to perform a careful assignment of scratchpad regions to processes. In this chapter, we propose approaches to share the scratchpad among the processes under the objective to minimize the energy consumption of a multiprocess application.

8.3 Related Work

Computers in 1960's and 1970's were severely memory constrained. Therefore, parallels can be drawn between the approaches proposed in this chapter and the memory management approaches in the operating systems of early computers. The only difference is that the proposed approaches consider multiprocess applications with a fixed number of periodic processes or tasks. The extension of these approaches for aperiodic tasks is a part of our immediate future work.

In the OS/360 operating system by IBM, the memory was divided into contiguous address spaces called *regions* or *partitions*. The operating system [100] featured memory management techniques called *multiple contiguous fixed partition allocation* (MFT) and *multiple contiguous variable partition allocation* (MVT). The MFT approach divided the memory into regions of fixed sizes and each arriving process is assigned a multiple of fixed sized regions. In contrast, the MVT approach creates regions of the exact size as that of the arriving process. The MFT approach causes both internal and external fragmentation, while the MVT approach only causes external fragmentation but is more complex. The Non-Saving approach is similar to the MVT approach, as it divides the scratchpad into disjoint regions of variable sizes for each process.

The resident monitor with swapping based memory management scheme [100] found in *Compatible Time Sharing System* (CTSS) is similar to the proposed Saving approach. The memory management scheme copies the contents of the process into the user memory when it is scheduled to execute on the processor. When the processor switches to a next process, the contents of the previously executing process are swapped to a backing store (a disk or drum) and the contents of the next process are copied into the user memory.

Upon analysis of the recent related work, we realized that there are not many approaches to share the scratchpad memories of the contemporary memory constrained embedded devices. Only one approach [41] to share the scratchpad among processes is known. However, the approach is not designed to allocate multiprocess application on the scratchpad. It is based on a dynamic memory allocator [83] which handles memory allocation and de-allocation requests from the application. Therefore, the approach can allocate only dynamically created data variables and not code segments onto the scratchpad.

The main disadvantage of the approach is that the programmer or a timing analyzer cannot be sure if a dynamically created variable which is supposed to be allocated onto the scratchpad, is actually mapped on the scratchpad by the dynamic allocator. This can severely degrade the predictability w.r.t. worst case execution time (WCET) bounds for a scratchpad based system. Moreover, the approach does not have a mechanism to ensure that the scratchpad always contains the most energy efficient variables. The other disadvantage is that the approach is not automated and the programmer needs to manually insert API calls at the appropriate locations in the application code to utilize the scratchpad memory. This manual intervention can easily lead to error-prone or sub-optimal results.

In the wake of the above discussion, we would like to enumerate the advantages of the proposed scratchpad sharing approaches:

(a) They are fully automated and require no intervention of the programmer.
(b) They allocate both code segments and data variables onto the scratchpad.
(c) They are fully analyzable at design time, *i.e.* the locations of the memory objects are decided and fixed at compile time.
(d) They minimize the energy consumption of the application and also generate pareto-optimal energy consumption curves which enable exploration of the design space for energy vs. scratchpad size tradeoffs.

In the following section, we describe the preliminaries associated with the scratchpad sharing approaches.

8.4 Preliminaries for Problem Formulation

The scratchpad sharing approaches, *viz,* Non-Saving, Saving and Hybrid, minimize the energy consumption of a multiprocess application by sharing the scratchpad among its processes. The Non-Saving approach partitions the scratchpad into disjoint regions such that each process is assigned a region. On the other hand, the Saving approach shares the scratchpad as the overlapping region common for all the processes. The hybrid approach which is a combination of the two approaches, divides the scratchpad into disjoint regions and a common overlapping region.

The proposed approaches partitions the scratchpad into regions and then use a non-overlayed allocation approach [115] to allocate memory objects of each process to its assigned scratchpad region. As will be shown in the following sections, the proposed approaches are independent of the underlying allocation approach, which can be easily replaced by a complex overlay based approach (*cf.* Chapter 6). In this work, we chose the non-overlayed allocation approach because of its simplicity.

The rest of the section is structured as the following: The following subsection describes the notation of the functions used in this chapter, followed by the definitions of the system variables. Subsection 8.4.3 describes the memory objects and Subsection 8.4.4 describes the energy model used by the scratchpad sharing approaches.

8.4.1 Notation

Several notations to represent functions are commonly used. The rigorous notation [136] $f : x \rightarrow f(x)$ specifies that f is a function acting upon a single number x and returning a value $f(x)$. In addition, the notation $f(x)$ is used to refer to the function f. In this chapter, unless indicated otherwise, the notation $f(x)$ refers to the rigorous notation $f\colon x \rightarrow f(x)$, whereas the notation $f(x_i)$ with a subscripted argument x_i refers to the value of the function f for the input number x_i. Similar, notations $f(x,y)$, $f(x_i,y_j)$ are used for bi-variate functions.

8.4.2 System Variables

For the presented work, we assume a statically scheduled system with periodic processes such that the execution profile (*i.e.* execution time and energy consumption) of each process is known or can be estimated a priori. We also assume that the multiprocess application consists of n processes $P_1 \cdots P_n$ running on a system with an M byte scratchpad memory and that the non-saving $f_k^N(x)$, the saving $f_k^S(x)$ and the hybrid $f_k^H(x,y)$ energy functions are known for each process P_k. The energy function $CE(x, smem, dmem)$ returning the energy overhead caused by the copying routines, is also assumed to be known. Furthermore, we assume that schedule count s_k is the number of times a process P_k is scheduled for execution.

Definition 8.1 (Non-Saving Energy Function $f_k^N(x)$). *The Non-Saving energy function $f_k^N\colon [0,M] \rightarrow \mathbb{R}$ takes the size of the disjoint scratchpad region as input and returns the energy consumed by the process P_k when it is allocated onto the disjoint region.*

Definition 8.2 (Saving Energy Function $f_k^S(x)$). *The Saving energy function $f_k^S\colon [0,M] \rightarrow \mathbb{R}$ takes the size of the overlapping scratchpad region as input and returns the energy consumed by the process P_k when it is allocated onto the overlapping region. It also includes the energy consumed by copy routines for swapping the contents of the process at the context switch.*

Definition 8.3 (Hybrid Energy Function $f_k^H(x,y)$). *The Hybrid energy function $f_k^H\colon [0,M] \times [0,M] \rightarrow \mathbb{R}$ takes the sizes of the disjoint and the overlapping scratchpad regions as input and returns the energy consumed by the process P_k when it utilizes the disjoint and the overlapping regions.*

Definition 8.4 (Copy Energy Function $CE(x, smem, dmem)$). *The Copy energy function $CE\colon [0,M] \times \{SPM, MM\} \times \{SPM, MM\} \rightarrow \mathbb{R}$ returns the energy consumed in copying x bytes from the source memory $smem$ to the destination memory $dmem$.*

Definition 8.5 (System Variables). *The system variables, used to represent the scratchpad sharing strategies, are defined as follows:*

(a)	n	*Number of processes.*
(b)	M	*Size of the SPM present in the system.*
(c)	$\{P_1 \cdots P_n\}$	*Set of processes in the application.*
(d)	s_k	*Schedule count for process P_k.*
(e)	$f_k^N(x_i)$	*Non-Saving energy function for process P_k.*
(g)	$f_k^S(x_i)$	*Saving energy function for process P_k.*
(i)	$f_k^H(x_i, y_j)$	*Hybrid energy function for process P_k.*
(k)	$CE(x,s,d)$	*Copy energy function.*

Memory Optimization	System Architecture	Memory Objects	Explanation
Scratchpad Sharing Strategies for Multiprocess Applications	Uni-Processor ARM	$MO = \bigcup_{P_k} MO_k$ $MO_k \subseteq V \cup BB \cup F$	global variables, basic blocks functions

Table 8.2. Memory Objects for Scratchpad Sharing Strategies

8.4.3 Memory Objects

The scratchpad sharing approaches utilize the non-overlayed scratchpad allocation approach [115] as the underlying approach for allocating memory objects onto the scratchpad memory. The non-overlayed approach considers global variables, basic blocks and functions of each process P_k as the memory objects MO_k for scratchpad allocation. Therefore, the set of memory objects MO for the scratchpad sharing approaches contain all memory objects MO_k of all processes P_k. Currently, the scratchpad sharing approaches are implemented only for a uni-processor ARM based system. At the time of writing, extensions to the proposed scratchpad sharing approaches are being implemented and evaluated for multi-processor ARM based systems. Table 8.2 summarizes the memory objects used for the proposed approaches.

8.4.4 Energy Model

The proposed scratchpad sharing approaches depend on the energy functions $f_k^N(x)$, $f_k^S(x)$ and $f_k^H(x,y)$ for each process P_k. The non-saving energy function $f_k^N(x_i)$ computes the energy dissipated by process P_k when it utilizes an x_i bytes scratchpad region. We solve the non-overlayed allocation (SA) problem (*cf.* page 39 of Chapter 4) for an x_i bytes scratchpad and obtain the minimized value of the objective function $E_{Total} = Objective(SA(MO_k, x_i))$. Using this information, the non-saving energy function $f_k^N(x)$ is computed as follows:

$$f_k^N(x_i) = Objective(SA(MO_k, x_i)) \ \ \forall x_i \in [0, M] \tag{8.1}$$

where, MO_k is the set of memory objects belonging to the process P_k. In order to compute the energy function $f_k^N(x)$, the non-overlayed scratchpad allocation (SA) problem should be solved for all scratchpad sizes between 0 and M bytes. However, for practical reasons the SA problem is solved at a granularity of 16 bytes.

The Saving approach assumes that the entire scratchpad is available to the process, while the dispatcher manages the scratchpad contents of the process during a context switch. As shown in the following, the saving energy function $f_k^S(x)$ computes the energy consumed by the process P_k which utilizes an overlapping scratchpad region and includes the energy consumed by the copy routines of the dispatcher.

$$f_k^S(x_i) = f_k^N(x_i) + s_k * [CE(x_i, SPM, MM) + CE(x_i, MM, SPM)] \ \ \forall x_i \in [0, M] \tag{8.2}$$

where s_k is the schedule count of process P_k and $CE(x, smem, dmem)$ is the copy energy function. The copy energy function $CE(x_i, SPM, MM)$ in the above equation results in a slightly overestimated energy value, as an intelligent approach would realize that program memory objects can never be modified and therefore should not copied from the scratchpad to the main memory at every context switch.

The hybrid scratchpad sharing approach assigns one disjoint and one overlapping region to each process P_k. Therefore, it depends on a variant of the non-overlayed allocation approach, *viz.* Bi-Scratchpad Allocation (BSA) approach. The BSA approach allocates memory objects of a given process P_k onto two given scratchpads. A formal definition of the Bi-Scratchpad Allocation problem is presented in Subsection 8.7.1. The hybrid energy function $f_k^H(x_i, y_j)$ is computed as follows:

$$f_k^H(x_i, y_j) = Objective(BSA(MO_k, x_i, y_j)) \;\; \forall x_i, y_j \in [0, M] \tag{8.3}$$

where, x_i and y_j are the sizes of the two scratchpads allocated by the BSA approach. Similar to the non-saving energy function $f_k^N(x)$, the value of the hybrid energy function $f_k^H(x_i, y_j)$ is equal to the minimized objective $E_{Total} = Objective(BSA(MO_k, x_i, y_j))$ of the BSA problem. In the following section, the Non-Saving approach for sharing the scratchpad memory is presented.

8.5 Scratchpad Non-Saving/Restoring Context Switch (Non-Saving) Approach

The Non-Saving approach partitions the scratchpad memory into n disjoint regions, one for each process, such that the non-overlayed allocation of each process P_k to its corresponding scratchpad region minimizes the total energy consumption of the multiprocess application. The approach generates a null sized region for a process, if it determines that it is not energy efficient to assign a scratchpad region to the process. In the following, the formal definition of the Non-Saving problem is presented.

8.5.1 Problem Formulation

Problem 8.6 (Non-Saving/Restoring Context Switch (Non-Saving)). Given a multiprocess application with n processes $P_1 \cdots P_n$, the non-saving energy function $f_k^N(x)$ for each process P_k and a memory hierarchy consisting of a scratchpad (SPM) and a main memory (MM). The problem is to partition the scratchpad into disjoint contiguous address regions of sizes $d_1 \cdots d_n$ one for each process P_k such that the total energy consumption of the application E_{Total}^N, defined below, is minimized.

$$E_{Total}^N = f_1^N(d_1) + \cdots + f_n^N(d_n) \tag{8.4}$$

The minimization of the total energy consumption E_{Total}^N is to be performed under the following constraint:

$$\sum_{P_k} d_k \leq size(SPM) \tag{8.5}$$

The constraint specifies that the aggregate size of all disjoint scratchpad regions should be less than the scratchpad size.

```
BinMin (f, g)
Require: Energy functions f(x) and g(x) st. $f, g : [0, M] \rightarrow \mathbb{R}$
Ensure: $h : [0, M] \rightarrow \mathbb{R}$, where $h(x_i) = min\{f(l_j) + g(m_k) | l_j + m_k \leq x_i\}$
1  $min = \infty$
2  for ($x_i = 0$ to $M$) do
3    for ($tmp = 0$ to $x_i$) do
4      $l_j = tmp$
5      $m_k = x_i - tmp$
6      if ($f(l_j) + g(m_k) < min$) then
7        $min = f(l_j) + g(m_k)$
8      end-if
9    end-for
10   $h(x_i) = min$
11 end-for
12 return h(x)
```

Fig. 8.3. Algorithm for Computing $binmin$ Function

It can be observed from Equations 8.4 and 8.5, that the energy consumption of the processes are inter-dependent as allocation of scratchpad region of size d_k to process P_k reduces the scratchpad size available to other processes. A pseudo-polynomial algorithm for the Non-Saving problem is presented in the following.

8.5.2 Algorithm for Non-Saving Approach

The Non-Saving problem can be formulated easily as an Integer Linear Programming (ILP) problem, as it optimizes of the objective function under the given constraints. However, a small modification to the Non-Saving problem enables the computation of the non-saving energy function $h_n^N(x)$ for the multiprocess application. The function $h_n^N(x)$ for the application allows the system designer to perform energy/scratchpad size tradeoffs. The non-saving energy function $h_n^N : [0, M] \rightarrow \mathbb{R}$, computed using the non-saving energy function $f_k^N(x)$ for each process P_k, is defined as follows:

$$h_n^N(x_i) = min\left\{f_1^N(d_1) + \cdots + f_n^N(d_n) | d_1 + \cdots + d_n \leq x_i\right\} \quad \forall x_i \in [0, M] \tag{8.6}$$

$$E_{Total}^N = h_n^N(M) \tag{8.7}$$

The value of the function $h_n^N(x)$ at $x = M$ is equal to the value of the optimized objective function of the Non-Saving problem. Computing the value $h_n^N(x_i)$ using the above equation requires $O\left((x_i + 1)^n\right)$ summation operations. Thus, a full-exhaustive algorithm to compute $h_n^N(x)$ would require an exponential $O(M^n)$ runtime. An efficient algorithm can utilize the following distributive property of the min operator:

$$\begin{aligned} h_3^N(x) &= min\left\{f_1^N(d_1) + f_2^N(d_2) + f_3^N(d_3) | d_1 + d_2 + d_3 \leq x\right\} \\ &= binmin\left(binmin\left(f_1^N, f_2^N\right), f_3^N\right) \end{aligned} \tag{8.8}$$

Definition 8.7 (Binary Minimum Function ($binmin(f,g)$)). *The binary minimum function $binmin(f,g)$ takes two functions f and g as input and returns another function h as output.*

$$binmin\colon ([0,M] \to \mathbb{R}) \times ([0,M] \to \mathbb{R}) \longrightarrow ([0,M] \to \mathbb{R}) \tag{8.9}$$

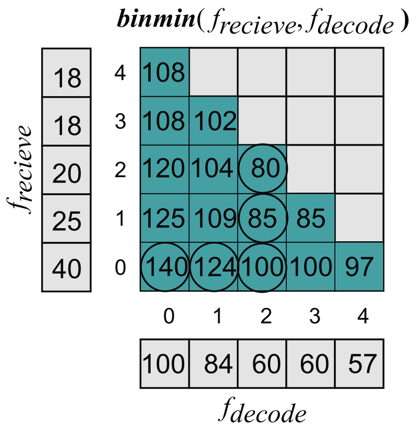
$$h(x) = binmin(f,g) \tag{8.10}$$

where the returned function $h(x)$ satisfies the following property:

$$h(x_i) = min\{f(l_j)+g(m_k)|l_j+m_k \le x_i\} \quad \forall x_i \in [0,M] \tag{8.11}$$

Figure 8.3 presents the algorithm to compute the function $h(x)$ returned upon the application of the binary minimum function $binmin$ to two input functions f and g. For each $x_i \in [0,M]$, the algorithm iterates over those values of l_j and m_k (*cf.* lines 4-5) which satisfy the summation constraint $l_j+m_k \le x_i$. The algorithm computes the sum of values returned by input functions f and g at l_j and m_k, respectively and stores the minimum computed value $f(l_j)+g(m_k)$ at $h(x_i)$. The algorithm requires $O(M^2)$ runtime to compute the binary minimum function $h(x) = binmin(f,g)$ for two energy functions. The working of the algorithm is explained with the help of an example in the following:

Example 8.8. Consider the Non-Saving energy functions shown in Table 8.1 for the example video phone application. In this example, we will compute function $h(x)$ by applying the binary minimum function (*cf.* Figure 8.3) to non-saving energy functions $f^N_{receive}$ and f^N_{decode} of the processes *GSMReceive* and *MPEGDecode*, respectively.



18	4	108				
18	3	108	102			
20	2	120	104	80		
25	1	125	109	85	85	
40	0	140	124	100	100	97
		0	1	2	3	4
		100	84	60	60	57

Fig. 8.4. Computation Matrix

Energy Fn.	0 kB	1 kB	2 kB	3 kB	4 kB
$f^N_{receive}$	40	25	20	18	18
f^N_{decode}	100	84	60	60	57
$binmin(f^N_{receive},$	140	124	100	85	80
$f^N_{decode})$	(0,0)	(0,1)	(0,2)	(1,2)	(2,2)

Table 8.3. Computed $binmin$ Function

Figure 8.4 presents the computation of the BinMin algorithm on energy functions $f^N_{receive}$ and f^N_{decode} in the form of a matrix. The computed function $binmin(f^N_{receive}, f^N_{decode})$ values are encircled in the computation matrix and are also presented in Table 8.3. From the algorithm in Figure 8.3, we observe that for each $x_i \in [0,1,2,3,4]$, only those values l_j and m_k which satisfy the summation constraints are considered. For example, for $x_i = 2$ there are three possible input arguments $(0,2)$, $(1,1)$ and $(2,0)$ to the functions $f^N_{receive}$ and

f^N_{decode}. For these input arguments $(0,2)$, $(1,1)$ and $(2,0)$ (*cf.* Figure 8.4), the summed values $f(l_j)+g(m_k)$ are 100, 109 and 120, respectively. The minimum of the three values *i.e.* 100, is the value $h(x_i)$ stored at $x_i = 2$. The function $h(x)$ shown in Table 8.3 is computed in a similar manner.

The definition of the binary minimum function and the distributive property of the min operator (*cf.* Equation 8.8) are used to convert the non-saving energy function $h^N_n(x)$ (*cf.* Equation 8.6) of the application into the following recurrence equation.

$$h^N_1(x) = binmin\left(f^N_1(x), Z(x)\right) \tag{8.12}$$

$$h^N_n(x) = binmin\left(h^N_{n-1}(x), f^N_n(x)\right) \tag{8.13}$$

```
**NonSaving($f^N_1, \ldots, f^N_n$)**
**Require:** Non-Saving Energy functions $f^N_1(x), \ldots, f^N_n(x)$
**Ensure:** $h^N_n : [0, M] \to \mathbb{R}$, where
        $h^N_n(x_i) = min\{f^N_1(d_1) + \cdots + f^N_n(d_n) \mid d_1 + \cdots + d_n \le x_i\}$
1 if ($n > 1$) then
2     $h^N_{n-1}(x) = NonSaving(f^N_1(x), \ldots, f^N_{n-1}(x))$
3     $h^N_n(x) = BinMin(h^N_{n-1}(x), f^N_n(x))$
4 else
5     $h^N_n(x) = BinMin(f^N_1(x), Z(x))$
6 end-if
7 return $h^N_n(x)$
```

Fig. 8.5. Recursive Algorithm for the Non-Saving Approach

where, $Z(x) = 0$ is a zero function which returns zero for all values of x. The correctness proof of the recurrence equations is presented in Appendix A and their implementation as a recursive algorithm is shown in Figure 8.5. The algorithm takes non-saving energy functions $f^N_1(x), \ldots, f^N_n(x)$ as input and computes the non-saving energy function $h^N_n(x)$ of the application. If there is a single process, then the algorithm computes the $binmin$ function for the non-saving energy function of the process and the zero function $Z(x)$. Otherwise, the algorithm recursively applies the $binmin$ function on energy functions $h^N_{k-1}(x)$ and $f^N_k(x)$ to compute the function $h^N_k(x)$. The algorithm requires $O(nM^2)$ runtime to compute non-saving energy function $h^N_n(x)$ of the multiprocess application. The application of the NonSaving algorithm to the video application is demonstrated in the following example.

Example 8.9. The application of the NonSaving algorithm (*cf.* Figure 8.5) to the non-saving energy functions of the example video phone application is shown in Figure 8.6.

In the first step, the algorithm computes $h^N_2(x)$ by applying the binary minimum function $binmin(f^N_{receive}, f^N_{decode})$ on energy functions $f^N_{receive}$ and f^N_{decode}. The computation of $h^N_2(x)$ is described in Example 8.8. In the next step, the algorithm applies the binary minimum function to the function $h^N_2(x)$ computed in the previous step and to the non-saving energy function f^N_{ui} of the process UserInterface. The application of the binary minimum

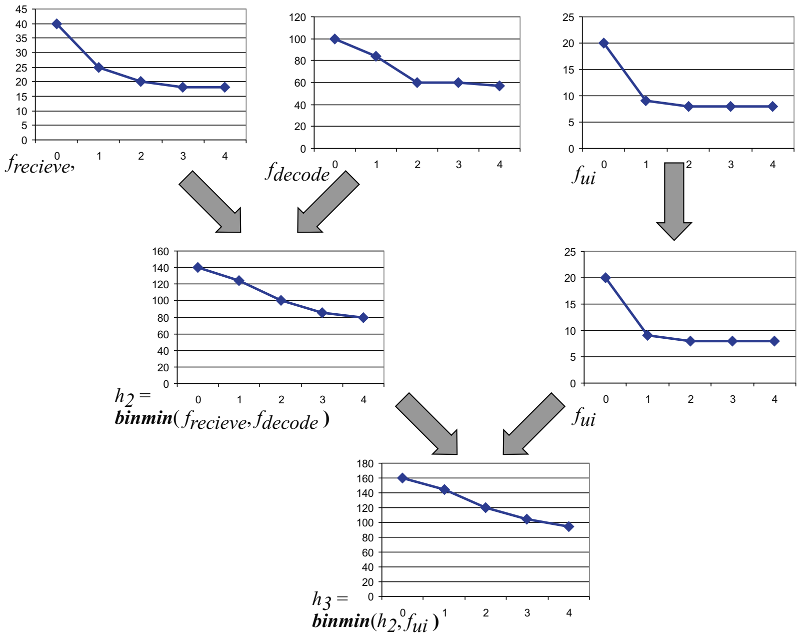


Fig. 8.6. Workflow of the NonSaving Algorithm for the Video Phone Application

Energy Fn. (Soln. Set)	0 kB	1 kB	2 kB	3 kB	4 kB
$h_2^N(x) =$ $binmin(f_{receive}^N, f_{decode}^N)$	140 (0,0)	124 (0,1)	100 (0,2)	85 (1,2)	80 (2,2)
$h_3^N(x) =$ $binmin\left(h_2^N, f_{ui}^N\right)$	160 (0,0)	144 (1,0)	120 (2,0)	105 (3,0)	94 (3,1)

Table 8.4. Computed Non-Saving Energy Functions

function results in function $h_3^N(x)$ which is the non-saving energy function of the multiprocess video phone application, as defined in Equation 8.6.

Table 8.4 presents energy functions $h_2^N(x)$ and $h_3^N(x)$ as well as the assignment of scratchpad regions to the processes. The video phone application dissipates 94 energy units for the 4 kB scratchpad shared using the Non-Saving approach, compared to 117 units when the scratchpad is not shared.

8.6 Scratchpad Saving/Restoring Context Switch (Saving) Approach

Unlike the Non-Saving approach, the Saving approach shares the scratchpad as a common region for all the processes. It assigns an overlapping scratchpad region to each process and then uses the non-overlayed allocation approach to allocate its memory objects onto the assigned scratchpad region. The approach needs support from the dispatcher, as it copies the memory objects of a process to the scratchpad everytime the process is scheduled to execute on the processor. The dispatcher is also responsible for copying them back when the process is taken off the processor. The formal definition of the Saving problem is presented in the following.

8.6.1 Problem Formulation

Problem 8.10 (Scratchpad Saving/Restoring Context Switch (Saving)). Given a multiprocess application with n processes $P_1 \cdots P_n$, the saving energy function $f_k^S(x)$ for each process P_k and a memory hierarchy consisting of a scratchpad (SPM) and a main memory (MM). The problem is to create overlapping scratchpad regions of sizes $o_1 \cdots o_n$ one for each process P_k such that the total energy consumption of the application E_{Total}^S, defined below, is minimized.

$$E_{Total}^S = f_1^S(o_1) + \cdots + f_n^S(o_n) \tag{8.14}$$

The minimization of the total energy consumption E_{Total}^S is to be performed under the following constraint:

$$\forall P_k : o_k \leq size(SPM) \tag{8.15}$$

The constraint specifies that the size of each overlapping scratchpad region should be less than the scratchpad size.

An important observation related to the Saving problem is that the energy consumption of each process allocated to the scratchpad memory in independent of the other processes. This is because the allocation of o_k bytes of scratchpad region to a process P_k does not limit the size of the scratchpad region that can be assigned to any other process.

The saving energy function $f_k^S(x)$ shown below for each process P_k has the interesting property that it contains a decreasing term and an increasing term.

$$f_k^S(x_i) = f_k^N(x_i) + s_k * [CE(x_i, SPM, MM) + CE(x_i, MM, SPM)] \;\; \forall x_i \in [0, M] \tag{8.16}$$

The non-saving energy function $f_k^N(x)$ is a monotonically decreasing function if the size x of the scratchpad region is less than the total size of the process. In contrast, the copy energy function $CE(x, smem, dmem)$ is a monotonically increasing function. The following example describes the saving energy functions for the example application.

Example 8.11. Table 8.5 displays the saving energy functions for the processes of the video phone application, computed using Equation 8.16. To compute saving function values, the non-saving energy function values are taken from Table 8.1, while the aggregate copy energy overhead is assumed to contribute 5 units of energy for each 1 kB of scratchpad region.

From Table 8.5, we observe that the saving energy function values initially decrease with the increase in the size of the scratchpad region till it reaches the minimum energy

Process	Energy fn.	0 kB	1 kB	2 kB	3 kB	4 kB
GSMReceive	$f^S_{receive}$	40	30	30	33	38
MPEGDecode	f^S_{decode}	100	89	70	75	77
UserInterface	f^S_{ui}	20	14	18	23	28

Table 8.5. Saving Energy Functions (Abstract Units) for Video Phone Application

value. Thereafter, the values increase with the increase in scratchpad regions. The saving energy functions $f^S_{receive}$, f^S_{decode} and f^S_{ui} reach minimum values of 30, 70 and 14 units for 2 kB, 2 kB and 1 kB of scratchpad regions, respectively.

8.6.2 Algorithm for Saving Approach

The Saving problem can also be formulated as an Integer Linear Programming (ILP) problem. However, we use a pseudo-polynomial algorithm to solve the problem because instead of just computing the assignment of scratchpad regions for an M byte scratchpad, we are interested in computing the saving energy function $h^S_n(x)$ for the application. The energy function $h^S_n : [0,M] \rightarrow \mathbb{R}$ uses the independence property of the saving functions $f^S_k(x)$ of

```
**Saving**($f^S_1,\ldots,f^S_n,s_1,\ldots,s_n,CE$)
**Require:** Saving Energy functions $f^S_1(x),\ldots,f^S_n(x)$
**Require:** Process schedule counts $s_1,\ldots,s_n$
**Require:** Copy Energy function $CE(x,smem,dmem)$
**Ensure:** $h^S_n : [0,M] \rightarrow \mathbb{R}$, where $h^S_n(x_i) = \sum_{P_k} min\left[f^N_k(o_j)|\ \forall o_j \le x_i\right]$
1  **for** ($k = 1$ to $n$) **do**
2      $prev\_min[k] = \infty$
3  **end-for**
4  **for** ($x_i = 0$ to $M$) **do**
5      **for** ($k = 1$ to $n$) **do**
6          $o_j = x_i$
7          $f^S_k(o_j) = f^N_k(o_j) + s_k * [CE(o_j,SPM,MM) + CE(o_j,MM,SPM)]$
8          **if** ($f^S_k(o_j) < prev\_min[k]$) **then**
9              $min[k] = f^S_k(o_j)$
10         **else**
11             $min[k] = prev\_min[k]$
12         **end-if**
13     **end-for**
14     **for** ($k = 1$ to $n$) **do**
15         $E_{min} = E_{min} + min[k]$, $prev\_min[k] = min[k]$
16     **end-for**
17     $h^S_n(x_i) = E_{min}$
18 **end-for**
19 **return** $h^S_n(x)$
```

Fig. 8.7. Algorithm for the Saving Approach

the processes and is defined as the follows:

$$h_n^S(x_i) = \sum_{P_k} min\left[f_k^S(o_j)|\ \forall o_j < x_i\right]\ \forall x_i \in [0, M] \tag{8.17}$$

$$E_{Total}^N = h_n^N(M) \tag{8.18}$$

The value of the saving energy function $f_n^S(x_i)$ at x_i bytes of scratchpad is the sum of the minimum value of the function $f_k^S(o_j)$ over the range $o_j \in [0, x_i]$ for each process P_k. Figure 8.7 presents the algorithm used to compute the saving energy function for an application consisting of n processes.

The algorithm (*cf.* Figure 8.7) iterates over all overlapping region sizes $x_i \in [0, M]$ and all processes $k \in [1, n]$ and computes the saving energy function values $f_k^S(x_i)$. For each overlapping region size x_i, the algorithm ensures that array $prev\_min$ and min contain the minimum values of the saving energy function $f_k^S(o_i)$ for every process P_k over the range $o_i \in [0, x_i - 1]$ and $o_i \in [0, x_i]$, respectively. The minimum values stored in the array variable min are then summed up to compute the saving energy function value $h_n^S(x_i)$ for each x_i. The algorithm requires $O(nM)$ to compute the saving energy function $h_n^S(x)$ for the application. The following example explains the computation of the saving energy function $h_3^S(x)$ for the video phone application.

Example 8.12. Table 8.6 presents the values of the min array variable when the algorithm iterates the variable x_i over the range $[0, M]$. The algorithm ensures that for each $x_i \in [0, M]$ the property of min variable $min[k] = min\left\{f_k^S(o_j)|\ \forall o_j \in [0, x_i]\right\}$ holds. In Table 8.6, $min[1]$, $min[2]$ and $min[3]$ represent the minimum values of saving functions $f_{receive}^S$, f_{decode}^S and f_{ui}^S, respectively.

Energy Fn. (Soln. Set)	0 kB	1 kB	2 kB	3 kB	4 kB
$min[1]$	40	30	30	30	30
$min[2]$	100	89	70	70	70
$min[3]$	20	14	14	14	14
$h_3^S(x) = Saving(f_{receive}^S, f_{decode}^S, f_{ui}^S)$	160 (0,0,0)	133 (1,1,1)	114 (2,2,1)	114 (2,2,1)	114 (2,2,1)

Table 8.6. Computed Saving Energy Function

The algorithm computes the value of the saving function $h_3^S(x_i)$ at iteration x_i by adding the values of the elements of the array min. For example, the value of function $h_3^S(x_i)$ at $x_i = 2$ kB (*cf.* Table 8.6) is computed to be $70 + 30 + 14 = 114$. The saving energy function $h_3^S(x)$ of the application and the assignment of the overlapping regions to the processes are presented in the last row of Table 8.6. For a 2 kB scratchpad, the algorithm assigns overlapping regions of sizes 2 kB, 2 kB and 1 kB to processes *GSMReceive*, *MPEGDecode* and *UserInterface*, respectively. Upon comparing, the saving $h_3^S(x)$ (*cf.* last row of Table 8.6) and the non-saving $h_3^N(x)$ (*cf.* last row of Table 8.4) energy function of the video phone application, we observe that for 1 kB and 2 kB scratchpads the saving function values are lower than the non-saving function values, while the opposite is observed for 3 kB and 4 kB scratchpads.

8.7 Hybrid Scratchpad Saving/Restoring Context Switch (Hybrid) Approach

The Hybrid Approach combines the two proposed scratchpad sharing approaches. It partitions the scratchpad into disjoint regions each allocated to one process, and one overlapping region which is commonly utilized by all processes. The most frequently accessed memory objects of a process are allocated in a non-overlayed manner to the dedicated disjoint scratchpad region. The other important memory objects are allocated to the overlapping region causing a tolerable copy overhead.

We should note that the overlapping region and their corresponding disjoint region cannot be adjacent for all the processes. The non-overlayed scratchpad allocation (SA) approach [115] is incapable of allocating memory objects to disjoint and overlapping region. Therefore, we use a Bi-Scratchpad Allocation (BSA) approach which determines the two energy optimal memory objects sets for allocation on two scratchpads with different energy consumption. The per access energy consumed by the overlapping region is higher than that by the disjoint region because of the copy overheads associated with the overlapping region. In the following, we present the formal definitions of the BSA problem and the Hybrid problem.

8.7.1 Problem Formulation

Problem 8.13 (Bi-Scratchpad Allocation (BSA)). Given the set of memory objects MO, a memory hierarchy consisting of two scratchpad memories (SPM_1 and SPM_2) and a main memory (MM). The problem is to determine three mutually disjoint subsets of memory objects $MO_{SPM_1}, MO_{SPM_2}, MO_{MM} \subseteq MO$ such that total energy consumption of the application E_{Total}, achieved due to non-overlayed allocation of memory objects $mo_i \in MO_{SPM_k}$ to the scratchpad memory SPM_k, is minimized.

$$\begin{aligned} E_{Total} = & \sum_{mo_i \in MO_{SPM_1}} E(mo_i, SPM_1) + \sum_{mo_i \in MO_{SPM_2}} E(mo_i, SPM_2) \\ & + \sum_{mo_i \in MO_{MM}} E(mo_i, MM) \end{aligned} \tag{8.19}$$

The minimization of the total energy consumption E_{Total} is to be performed under the following constraints:

(a) The aggregate size of memory objects assigned to the scratchpad memories should be the size of the corresponding scratchpad memory.

$$\sum_{mo_i \in MO_{SPM_1}} size(mo_i) \leq size(SPM_1) \tag{8.20}$$

$$\sum_{mo_i \in MO_{SPM_2}} size(mo_i) \leq size(SPM_2) \tag{8.21}$$

(b) Every memory object $mo_i \in MO$ should be allocated to atleast one of three memories.

$$MO_{SPM_1} \cup MO_{SPM_2} \cup MO_{MM} = MO \tag{8.22}$$

(c) Every memory object $mo_i \in MO$ should be allocated to only one of three memories.

$$MO_{SPM_1} \cap MO_{SPM_2} = \phi \tag{8.23}$$

$$MO_{SPM_1} \cap MO_{MM} = \phi \tag{8.24}$$

$$MO_{SPM_2} \cap MO_{MM} = \phi \tag{8.25}$$

The BSA problem is special case of the General Assignment Problem [43]. Authors [26] demonstrate that the BSA problem is an NP-Complete problem and that unlike the Knapsack problem it is APX-Hard implying that no polynomial time approximation scheme (PTAS) exist. The best known approximation algorithm for the BSA problem is a 2-approximation. In the current setup, the goal is to allocate memory objects to a disjoint and a overlapping region on the scratchpad memory. Therefore, we could reformulate $E(mo_i, SPM_1)$ and $E(mo_i, SPM_2)$ are follows:

$$E(mo_i, SPM_1) = E(mo_i, SPM) \tag{8.26}$$

$$\begin{aligned} E(mo_i, SPM_2) = E(mo_i, SPM) + s_k * [&CE(size(mo_i), SPM, MM) \\ &+ CE(size(mo_i), MM, SPM)] \end{aligned} \tag{8.27}$$

where, $E(mo_i, mem)$ returns the energy consumed by the memory object mo_i when it is allocated to the memory mem. Please refer to Section 4.4.2 on Page 37 for detailed information on energy function $E(mo_i, mem)$.

Problem 8.14 (Hybrid Scratchpad Saving/Restoring Context Switch (Hybrid)). Given a multiprocess application with n processes $P_1 \cdots P_n$, the hybrid energy function $f_k^H(x,y)$ for each process P_k and a memory hierarchy consisting of a scratchpad (SPM) and a main memory (MM). The problem is to generate disjoint regions of size $d_1 \ldots d_n$ and overlapping regions of sizes $o_1 \cdots o_n$ such that each process is assigned one disjoint and one overlapping region and that the total energy consumption of the application E_{Total}^H defined below is minimized.

$$E_{Total}^H = f_1^H(d_1, o_1) + \cdots + f_n^H(d_n, o_n) \tag{8.28}$$

The minimization of the total energy consumption E_{Total}^H is to be performed under the following constraint:

$$\forall P_k : \left(\sum_{P_i} d_i \right) + o_k \leq size(SPM) \tag{8.29}$$

The constraint specifies that the size of the overlapping scratchpad region plus the aggregate size of all disjoint regions should be less than the scratchpad size.

The Hybrid problem assumes that the hybrid energy function $f_k^H(x,y)$ for each process P_k is known. We solve the BSA problem for all $d_k, o_k \in [0,M]$ to compute the hybrid energy function $f_k^H(d_k,o_k)$ values.

8.7.2 Algorithm for Hybrid Approach

Similar to the Non-Saving and Saving approaches, we compute the hybrid energy function $h_n^H(x,y)$ of the application, rather than just computing an overlapping and disjoint regions which optimize the objective function E_{Total}^H of the Hybrid problem. The hybrid energy function $h_n^H : [0,M] \times [0,M] \to \mathbb{R}$ of the application depends upon the hybrid energy functions $f_1^H, \ldots, f_n^H$ of the processes and is defined as follows:

$$\begin{aligned} h^H(x_i,y_j) = min\{ & f_1^H(d_1,o_1) + \cdots + f_n^H(d_n,o_n) \mid \\ & d_1 + \cdots + d_n \le x_i \wedge \forall k \in [1,n] : o_k \le y_j\} \;\; \forall x_i, y_j \in [0,M] \end{aligned} \tag{8.30}$$

$$E_{Total}^H = min\{h^H(x_i,y_j) \; | x_i + y_j \le M\} \;\; \forall x_i, y_j \in [0,M] \tag{8.31}$$

The minimized objective function E_{Total}^H of the Hybrid problem can be computed using the above equation. Similar to the binary minimum function *binmin* of the Non-Saving approach, we define *hybridbinmin* as follows:

Definition 8.15 (Hybrid Binary Minimum Function ($hybridbinmin(f,g)$)). *The hybrid binary minimum function takes two bi-variate functions f and g as input and returns another bi-variate function h as output.*

```
HybridBinMin (f, g)
Require: Energy functions f(x,y) and g(x,y) st. $f,g : [0,M] \times [0,M] \to \mathbb{R}$
Ensure: $h : [0,M] \times [0,M] \to \mathbb{R}$, where
        $h(x_i,y_j) = min\{f(d_l,o_l) + g(d_m,o_m) | d_l + d_m \le x_i \wedge o_l \le y_j \wedge o_m \le y_j\}$
1  for ($y_j = 0$ to $M$) do
2    $min = \infty$
3    for ($x_i = 0$ to $M - y_j$) do
4      for ($tmp = 0$ to $x_i$) do
5        $d_l = tmp$, $o_l = y_j$, $d_m = x_i - tmp$, $o_m = y_j$
6        if ($f(d_l,o_l) + g(d_m,o_m) < min$) then
7          $min = f(d_l,o_l) + g(d_m,o_m)$
8        end-if
9      end-for
10     $h(x_i,y_j) = min$
11   end-for
12 end-for
13 return $h(x,y)$
```

Fig. 8.8. Algorithm for Computing *hybridbinmin* Function

$$\begin{aligned} hybridbinmin\colon & ([0,M]\times[0,M]\to\mathbb{R})\times([0,M]\times[0,M]\to\mathbb{R})\longrightarrow \\ & ([0,M]\times[0,M]\to\mathbb{R}) \end{aligned} \tag{8.32}$$

$$h(x) = hybridbinmin(f,g) \tag{8.33}$$

where the returned function $h(x)$ satisfies the following property:

$$\begin{aligned} h(x_i,y_j) = min\{f(d_l,o_l)+g(d_m,o_m)|d_l+d_m \le x_i \wedge o_l \le y_j \wedge o_m \le y_j\} \\ \forall x_i,y_j \in [0,M] \end{aligned} \tag{8.34}$$

Figure 8.8 presents the algorithm to compute the function $h(x,y)$ obtained by applying the $hybridbinmin$ function to two input functions $f(x,y)$ and $g(x,y)$. For each x_i and y_j, the algorithm iterates over all d_l, o_l, d_m and o_m which satisfy the condition $d_l+d_m \le x_i \wedge o_l \le y_j \wedge o_m \le y_j$ and computes the minimum sum of $f(d_l,o_l)$ and $g(d_m,o_m)$. The minimum value defines the value of the function $h(x,y)$ at $x=x_i$ and $y=y_j$. The algorithm requires $O(M^3)$ to compute the function $h(x,y)$.

Similarly to the Non-Saving approach, the definition of the hybrid binary minimum function $hybridbinmin$ and the distributive property of the min operator is used to convert the hybrid energy function $h_n^H(x)$ (*cf.* Equation 8.30) of the application to the following recurrence equations:

$$h_1^H(x,y) = hybridbinmin\left(f_1^H(x,y), Z(x,y)\right) \tag{8.35}$$

$$h_n^H(x,y) = hybridbinmin\left(h_{n-1}^H(x,y), f_n^H(x,y)\right) \tag{8.36}$$

where $Z(x,y)$ is the bi-variate zero function. Figure 8.9 presents the recursive algorithm to compute the hybrid energy function $h_n^H(x,y)$ of a multiprocess application consisting of n processes.

The algorithm presented in Figure 8.9 is similar to the algorithm for the Non-Saving approach (*cf.* Figure 8.5). It takes the bi-variate hybrid energy function $f_k^H(x,y)$ of each process P_k as input and computes the hybrid energy function $h_n^H(x,y)$ of the application in $O(nM^3)$ time. The minimum energy consumption of the application E_{Total}^H and the

```
Hybrid($f_1^H,\dots,f_n^H$)
Require: Hybrid Energy functions $f_1^H(x,y),\dots,f_n^H(x,y)$
Ensure: $h_n^H : [0,M]\times[0,M]\to\mathbb{R}$, where
        $h_n^H(x_i,y_j) = min\left\{f_1^H(d_1,o_1)+\cdots+f_n^H(d_n,o_n) \mid d_1+\cdots+d_n \le x_i \wedge \forall k\in[1,n] : o_k \le y_j\right\}$
1  if ($n>1$) then
2     $h_{n-1}^H(x,y) = Hybrid(f_1^H(x,y),\dots,f_{n-1}^H(x,y))$
3     $h_n^H(x,y) = HybridBinMin(h_{n-1}^H(x,y), f_n^H(x,y))$
4  else
5     $h_n^H(x,y) = HybridBinMin(f_1^H(x,y), Z(x,y))$
6  end-if
7  return $h_n^H(x,y)$
```

Fig. 8.9. Recursive Algorithm for the Hybrid Approach

assignment of disjoint d_k and overlapping o_k to each process P_k can be determined from $h_n^H(x,y)$ after a small processing step.

The experimental workflow used to evaluate the scratchpad sharing approaches is a bit different from the workflow of the previous scratchpad allocation approaches. Therefore, it is described separately in the following section.

8.8 Experimental Setup

The experiments were conducted for the uni-processor ARM based system. The memory hierarchy of the system consists of a 4 kB onchip scratchpad memory and a 512 kB SRAM based on-board main memory. The accurate energy model [114] (*cf.* Section 3.1 for additional details) is used to compute the energy consumed by the system. Additionally, a simplistic operating system was implemented to execute multiprocess applications on the uni-processor ARM system. We decide against using a standard operating system for embedded systems, *e.g.* RTEMS [104], because we wanted an operating system which causes the minimum overhead. The custom operating system consists of a Round-Robin scheduler, a dispatcher and provides an API to system calls for managing scratchpad contents. The operating system provides a time slice of 33000 CPU cycle or $1ns$ on the 33 MHz ARM processor to each process for executing on processor. For the current experiments, we assumed a statically scheduled system with all processes having equal priority.

The scratchpad sharing approaches require that the non-saving $f_k^N(x)$, the saving $f_k^S(x)$ and the hybrid $f_k^H(x,y)$ energy functions are precomputed for each process P_k. The non-saving energy function $f_k^N(x)$ is determined by computing the energy consumption values of process P_k utilizing scratchpad regions of sizes between 0 and 4096 bytes. The non-overlayed scratchpad allocation approach [115] is used for allocating the energy optimal set of memory objects to the scratchpad region. The saving energy function $f_k^S(x)$ is computed according to Equation 8.2 and uses the non-saving energy $f_k^N(x)$ and the schedule count s_k of the process P_k. Similar to the non-saving energy function $f_k^N(x)$, the hybrid energy function $f_k^H(x,y)$ for each process P_k is computed by solving the BSA problem described in Subsection 8.7.1. For our set of benchmarks, the computation of the non-saving and the hybrid energy function over the range of $[0,4096]$ bytes and at a granularity of 16 bytes required a maximum of 130 and 1000 CPU seconds, respectively. The experiments were conducted on a 1300 MHz Sun Sparc machine. The computation of the energy functions requires ample computation time, however, the database of energy functions can be reused later for many multiprocess applications.

The experiments were conducted according to the experimental workflow depicted in Figure 8.10. The energy functions of all the processes belonging to a multiprocess application are passed as input to the proposed scratchpad sharing algorithms. The algorithms process these energy functions and generate a file containing the assignment of a disjoint or an overlapping or both scratchpad region(s) to each process. In addition, the algorithm generates another file containing the energy function ($f_n^N(x)$ or $f_n^S(x)$ or $f_n^H(x)$) values of the multiprocess application. The source code of each process is then compiled using our energy optimizing research compiler (ENCC) and the optimal set of memory objects is marked for allocation onto the assigned scratchpad regions. The assembly codes of the processes and that of the operating system are then assembled and linked to create a single executable binary. The executable is executed on the processor simulator, *viz.* ARMulator [9],

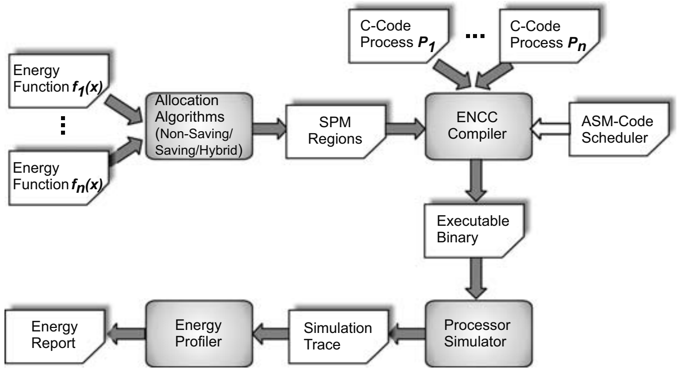


Fig. 8.10. Experimental Workflow

to generate the instruction trace which is then passed as input to the energy profiler. The profiler uses the 98% accurate energy model [114] to compute the total energy consumed by the application.

8.9 Experimental Results

The proposed scratchpad sharing approaches are evaluated for the set of benchmarks shown in Table 8.7. We converted two single process benchmarks *viz.*, Media and DSP, from our set of benchmarks into multiprocess applications, as no standard benchmark suite for multiprocess applications was available. The third benchmark is the video phone application which was used as example application throughout the chapter. The table also presents the total size and the processes constituting the application.

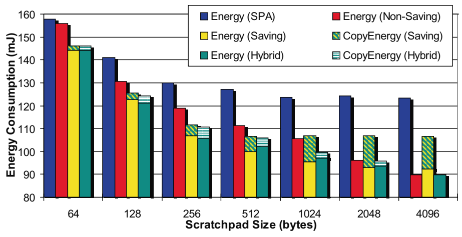
The evaluation of the scratchpad sharing approaches will be presented in the following order:

(a) Benefits of scratchpad sharing allocation approaches
(b) Comparison of cache and scratchpad memory based systems
(c) Pareto-optimal energy functions
(d) Location of copy routines

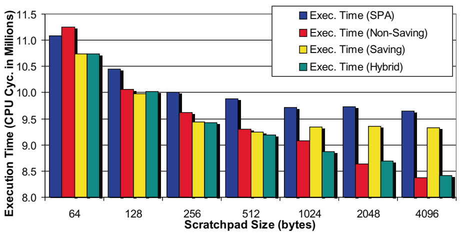
First, the benefits of the scratchpad sharing approaches are evaluated by comparing them to the single process allocation (SPA) approach. The SPA approach assigns the scratchpad to a process which leads to the maximum reduction in the energy consumption of the multiprocess application. Second, a comparison of the scratchpad utilized by the proposed

Application Name	Size (kB)	Processes (benchmarks)	System Architecture
Media	77	adpcm, g721, mpeg4, edge-detection	uni-processor ARM
Video Phone	80	gsm, mpeg4	uni-processor ARM
DSP	26	fast-idct, lattice-init, lattice-small, fft, fir	uni-processor ARM

Table 8.7. Multiprocess Applications



(a) Energy Consumption



(b) Execution Time

Fig. 8.11. Media: Comparison of SPA, Non-Saving, Saving and Hybrid Approaches

sharing approaches to the unified cache memory is presented. Third, the Pareto-optimal function generated by the approaches are briefly discussed. Finally, a small extension to store the copy routines of the Saving and Hybrid approaches onto the scratchpad memory is also presented.

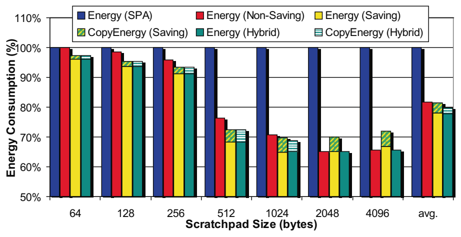
Benefits of Scratchpad Sharing Approaches:
Figures 8.11(a) and 8.11(b) evaluates the scratchpad sharing approaches *viz.*, Non-Saving, Saving and Hybrid, and the SPA approach w.r.t. energy consumption and execution time values of the Media benchmark, respectively. The energy overhead due to the copy routines for the Saving and Hybrid approaches is demarcated from the corresponding aggregate energy values in Figure 8.11(a). We make a few observations from the figures.

Firstly, we observe that energy consumption and execution time values resulting from the proposed approaches are always better than those for the SPA approach. This justifies sharing of the scratchpad memory among the processes of the application.

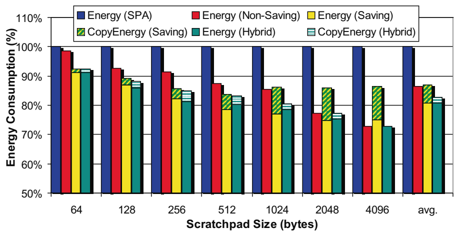
Secondly, the energy consumption as well as execution time values for the Non-Saving and Hybrid approaches decrease monotonically with the increase in the scratchpad size, as large scratchpads are partitioned into disjoint regions adequate for each process. However, the corresponding values for the Saving approach initially decrease till 512 bytes and then remain constant for any further increase in scratchpad sizes. The reason for such a behavior is that the Saving approach does not always assign the entire scratchpad as overlapping regions to processes because a high energy overhead is incurred due to copy routines for large overlapping regions.

Thirdly, for small scratchpad sizes between 64 and 512 bytes, the energy consumption values for the Saving approach are smaller than those for the Non-Saving approach, while the opposite is true for larger scratchpads. For small scratchpads, the improved utilization of the scratchpad due to the Saving approach and the small copy energy overhead result in the reduced energy consumption of the system.

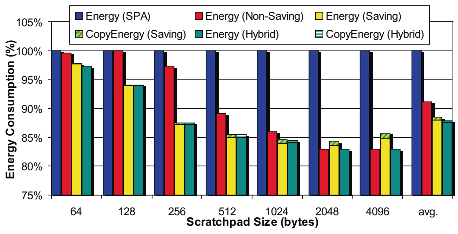
Finally, the Hybrid approach distributes the scratchpad into disjoint and overlapping regions and therefore achieves the minimum energy consumption compared to the Non-Saving and Saving approaches for all scratchpad sizes. The energy consumption values at 1024 bytes of scratchpad size in Figure 8.11(a) clearly demonstrate this behavior. The execution time values of the benchmark also show a similar behavior.



(a) DSP



(b) Media

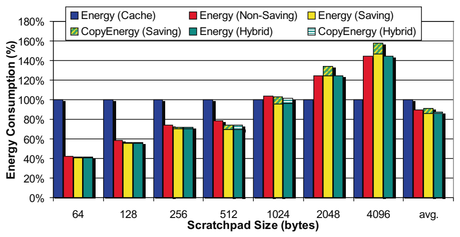


(c) Video Phone

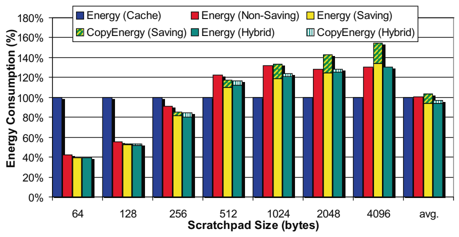
Fig. 8.12. Normalized Energy Consumption of Non-Saving, Saving and Hybrid Approaches with SPA Approach

Next, we compare the scratchpad sharing approaches to the SPA approach w.r.t. energy consumption of the Media, DSP and Video Phone applications. Figure 8.12 presents energy values of the proposed approaches relative to those of the SPA approach which are shown as 100% bars. From Figures 8.12(a), 8.12(b) and 8.12(c), we observe that the Hybrid approach reduces the energy consumption of the applications by upto 35%, 27% and 17% compared to the SPA approach, respectively. For the Media application, we report average energy reductions of 14%, 13% and 17% due to the Non-Saving, Saving and Hybrid approaches, respectively. Even higher average energy reductions of 18%, 19% and 20% due to Non-Saving, Saving and Hybrid approaches, respectively, are observed for the DSP application. The sharing approaches lead to the smallest average energy savings of 9%, 11% and 12% compared to the SPA approach for the Video Phone application. The MPEGDecode process of the application consumes significantly more energy than the GSMReceive process and therefore sharing the scratchpad results in small benefits.

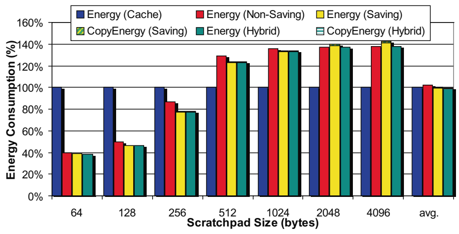
Comparison of the Cache and the Scratchpad Memory:
A comparison of scratchpad memories utilized by the proposed approaches and unified caches is presented below. In the present setup, the cache memory has a certain advantage over the scratchpad memory, as it stores memory objects at a much finer granularity of a cache line and also swaps the outdated memory objects with the relevant ones. In contrast, the non-overlayed scratchpad allocation approach utilized by the proposed approaches allocates code segments and aggregate array variables onto the scratchpad memory. We believe that the use of the advanced scratchpad allocation approaches presented in Chapter 5 and Chapter 6 would reduce the disparity between the scratchpad and the cache memory.



(a) DSP



(b) Media



(c) Video Phone

Fig. 8.13. Normalized Energy Comparison of Non-Saving, Saving and Hybrid Approaches with Cache

Figure 8.13 presents the energy consumption values of a scratchpad based system relative to the corresponding values (shown as 100% bars) for a unified cache based system of the same size. The comparison of scratchpad under the control of the Non-Saving, Saving and Hybrid approaches and the cache memory is presented for the DSP, Media and Video Phone applications. From Figure 8.13(a), we observe that for small sizes between 64 bytes and 512 bytes the energy consumption of a scratchpad based system is better than that of a cache based system executing the DSP application. For the Media and Video Phone applications, a similar behavior (*cf.* Figure 8.13(b) and 8.13(c)) is observed for sizes between 64 bytes and 256 bytes. The reason for this behavior is that for small cache sizes, a large number of conflict cache misses occur which result in excessive accesses to the power hungry main memory. On the other hand, the scratchpad is used to store the important memory objects of all the processes.

For larger sizes, the energy consumption of the cache based systems is better than that of the scratchpad based systems. This is because the scratchpad memory is not utilized to its potential by the underlying non-overlayed scratchpad allocation approach used by our scratchpad sharing approaches. Nevertheless, on the average the energy consumed by the scratchpad based system is either equal to or better than (by 13%) that consumed by the cache based system.

Pareto-Optimal Energy Functions:

Now, we would like to discuss the generation of Pareto-optimal curves for the multiprocess application by the proposed approaches. In addition to the shared allocation of the multi-process application onto an M bytes scratchpad, the energy functions ($h_n^N(x)$, $h_n^S(x)$ and

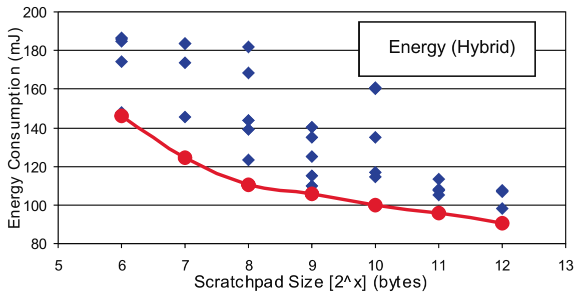


Fig. 8.14. Pareto-Optimal Curve for Media Application

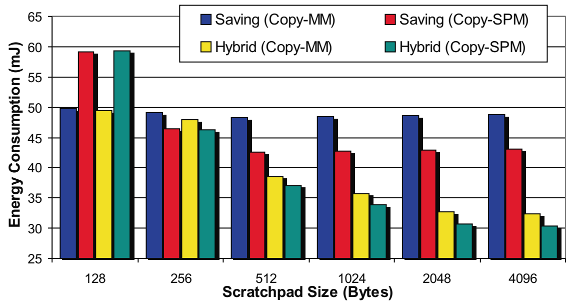


Fig. 8.15. DSP: Different Locations of Copy Routines

$h_n^H(x, y))$ defined over the range $[0, M]$ bytes of scratchpad sizes are also determined by the proposed allocation approaches. These energy functions represent the Pareto-optimal curves for the application.

Figure 8.14 presents the curve of the hybrid energy function $h_n^H(x)$ for the Media application. The values on the x-axis denote the scratchpad size and are presented in the logarithmic ($\log_2(x)$) scale. The remaining points represent the energy consumption values for promising allocations of scratchpad regions to the processes. The promising allocations, though, represent non-optimal solutions, as they consume more energy than those using the Hybrid approach. An easy indication for non-optimality is that all the remaining points lie above the Pareto-optimal curve. The Pareto-optimal curve aids the system designer in performing design-time scratchpad size vs. energy consumption tradeoffs. For example, a system with a (2^{12}) 4 kB scratchpad consumes just 10% less energy than a system with a (2^{10}) 2 kB scratchpad. Thus, a system designer can tradeoff 10% energy consumption for half of the onchip scratchpad size.

Location of Copy Routines:

We observed that for the Saving approach (*cf.* Figure 8.11(a)), the energy overhead due to copy routines accounts for significant portion of the total energy consumption of the system, as the copy routines by default are allocated to the energy inefficient main memory. Therefore, we conducted experiments to quantify the overhead when the copy routines are assigned to the scratchpad or to the main memory. Figure 8.15 presents the energy consumption values for the DSP application when the copy routines are assigned to the scratchpad (SPM) or the main memory (MM).

Energy values for the Saving and Hybrid approaches are presented in Figure 8.15, as they only utilize the copy routines. We observe that in all but one case the energy values for copy routines mapped to the scratchpad is lower than that for copy routines mapped to the main memory. The copy routines are executed quite often and their energy consumption gets reduced once they are assigned to the scratchpad. The exception case occurs for the smallest scratchpad size (128 bytes), for which the copy routines occupy most of the scratchpad space, and little space is left for the processes to utilize. From Figure 8.15, we observe that up to 13% energy can be saved for the Saving approach by assigning the copy routines on the scratchpad.

8.10 Summary

In this paper, a set of strategies *viz.*, Non-Saving, Saving and Hybrid were proposed for sharing the scratchpad among the processes of a multiprocess application. The Non-Saving approach partitioned the scratchpad into disjoint regions each allocated to a process of the application. In contrast, the Saving approach utilized the scratchpad as the common overlapping region for all the processes of the application. The Saving approach generated more energy efficient allocations than the Non-Saving approach for small scratchpads and vice-versa for large scratchpads. The Hybrid approach combines the two approaches and achieves the most energy efficient allocations for all scratchpad sizes, though it also required the longest computational time for the preprocessing step. The proposed approaches report average energy reductions of 9%-20% compared to a single process allocation approach. In addition to assigning energy optimal scratchpad regions to processes, the proposed approaches generated the Pareto-optimal curves allowing exploration of the design space.

The approaches presented in this chapter were published in [125].

9

Conclusions and Future Directions

In this book, memory optimization techniques were proposed to transform an application such that it efficiently utilizes the memory hierarchy of the underlying system. The objective of the proposed optimizations is to minimize the total energy consumption of system. A positive impact of the optimizations on reducing the execution time of system was also observed. In addition, it was discussed that the proposed optimizations improved the predictable behavior of the system. Consequently, it can be concluded that the proposed memory optimizations aid the system designer in meeting the previously stated design constraints on power, performance and predictability of the system.

The remainder of this chapter is organized as follows: The following section summarizes the contribution of the memory optimization techniques proposed in this work. After that, a discussion on important future directions for enhancing the proposed optimizations is presented.

9.1 Research Contributions

Memory subsystem has been identified as the bottleneck with respect to both the performance and energy consumption of the system. Consequently, a comprehensively optimized memory subsystem is considered to be critical for meeting the stringent design constraints on embedded devices, especially those belonging to the consumer electronics domain. In the past few years, code optimization techniques have assumed a considerable significance, since a majority of embedded software is being written in high-level languages. In addition, compiler based optimizations can fully exploit the optimization potential as they have a global view of the application.

Traditionally, caches were used to construct memory hierarchies in order to improve the performance of the memory subsystem. However, for embedded systems which execute only a limited set of applications in their entire lifetime, the limitations of caches in terms of energy consumption, performance and predictability are well documented. Therefore, energy efficient scratchpad memories now constitute the memory hierarchies found in most of the embedded processors. Unlike caches, scratchpad memories require support from the software or the compiler for their utilization.

In this work, compiler based optimization techniques to exploit the scratchpad based memory hierarchies of three orthogonal system architectures, *viz.* Uni-Processor ARM, Multi-Processor ARM and M5 DSP based systems, were proposed. The proposed optimizations were implemented within the backends of the corresponding compilers. In addition, they were implemented as “compiler-in-loop” source level transformations which enhanced their applicability to a wide variety of processor architectures. Unlike contemporary approaches, the proposed optimizations minimized the total energy consumption of the system as they allocate both instruction segments and data variables onto the scratchpad memory.

A more detailed description of the memory optimizations proposed in the book, is presented in the following:

Non-Overlayed Scratchpad Allocation for Main / SPM Memory Hierarchy is one of the basic memory optimization which allocated both instructions and variables onto the scratchpad memory in a non-overlayed manner. It was developed for a simple memory hierarchy consisting of an L1 scratchpad and a background main memory. The problem of non-overlayed scratchpad allocation was demonstrated to be either Knapsack or Fractional Knapsack problem under different pre-conditions of allocation. An Integer Linear Programming (ILP) based optimal approach was presented for the knapsack variant of the memory optimization, whereas a greedy algorithm based near-optimal approach was described for the fractional knapsack variant.

Experimental results reported near-identical energy consumption values for the two approaches. For the uni-processor ARM based system, average energy reductions between 29% and 64% compared to that for a scratchpad-less system were reported. For the multi-processor ARM based system, a maximum reduction of upto 90% (*cf.* Figure 4.8 on 45) in the total energy consumption of system was observed. Finally, for the M5 DSP based system, average reductions in between 15% and 22% in the data memory energy consumption of the system were observed.

Non-Overlayed Scratchpad Allocation for Main / Cache + SPM Memory Hierarchy is a memory optimization which utilized the scratchpad memory as an instruction buffer. It modeled the behavior of the cache memory as a conflict graph and allocated instruction segments onto the scratchpad memory such the energy consumption of the system is minimized. Again, both optimal and near-optimal approaches were proposed for the non-overlayed allocation of the scratchpad memory present in a cache based memory hierarchy. The need for a sophisticated allocation approach arose due to the observation that the previous non-overlayed allocation approach led to erratic and unpredictable results for the current memory hierarchy.

The impact of the proposed memory optimization is enumerated as follows: First, it was demonstrated that an I-Cache and a scratchpad based memory subsystem had a 40% lower energy consumption than the lowest energy I-Cache based memory subsystem. In addition, the scratchpad based memory subsystem required only a quarter (25%) of the onchip area required by the I-Cache based memory subsystem. Second, it was demonstrated that the scratchpad memory under the control of the proposed optimization outperforms the architecturally complex preloaded loop caches. Finally, for the uni-processor ARM based system, average energy reductions of 23% and 29% compared to the previous memory optimization and the preloaded loop cache, respectively, were reported. Moreover, a maximum reduction of 25% in the total energy consumption of the multi-processor ARM based system was also reported.

Scratchpad Overlay for Main / SPM Memory Hierarchy is a memory optimization which is based on the fact that all instruction segments and variables (memory objects) allocated onto the scratchpad memory are not accessed at the same time instant during the execution. Therefore, two or more memory objects with disjoint access times can be assigned to overlapping regions on the scratchpad memory. In addition, the approach identifies locations in the application code to insert spill routines for the copying of memory objects within the memory hierarchy.

It was demonstrated that the problem of scratchpad overlay is similar to the global register allocation problem. Similar to the previous optimizations, both optimal and near-optimal approaches were proposed for the scratchpad overlay problem. The problem was divided into two subproblems. The optimal approach used ILP formulations for the both subproblems, while the near-optimal approach used the same ILP formulation for the first subproblem, but used the first-fit heuristic for the second subproblem.

Data Partitioning and Loop Nest Splitting is a memory optimization which divided array variables into smaller array partitions and then performed a non-overlayed allocation of instruction segments and array partitions onto the scratchpad memory. The optimizations improved on previous optimizations which could allocate only aggregate array variables. Unlike known optimizations, the proposed optimization can partition array variables with regular as well as irregular index functions. In addition, the data partitioning approach divided an array variable when it can determine that such a division would reduce the overall energy consumption of the system.

The data partitioning approach inserts *if*-statements into the application source code to access the array partitions. The execution of the inserted *if*-statements degraded the control flow of the application and performance of the processor pipeline. Therefore, the data partitioning approach was combined with the loop nest splitting approach to improve the control flow of the application. The combined optimization improved the data memory energy consumption due to a fine grained allocation of array partitions onto the scratchpad and improved the execution time of the application by minimizing the number of executed *if*-statements. Experimental results report that the combined optimization achieved substantial reductions of upto 50% in the total energy consumption of the system and in the execution time of the application.

Scratchpad Sharing Strategies for Multiprocess Applications are based on the observation that most consumer centric embedded devices execute multiprocess applications, while all the known memory optimizations were proposed for a single process application. Therefore, a set of three approaches to share the scratchpad memory among the processes of the application were proposed. The approaches assigned disjoint and overlapping scratchpad regions to processes at the design time. The scratchpad sharing approaches reduced the total energy consumption of the system by about 30% compared to that for the optimization which assigned the entire scratchpad to the most energy consuming process of the application.

9.2 Future Directions

The dissertation ends with a brief outlook on the most relevant and promising future research directions, itemized below in no particular order:

Array Tiling:
The proposed optimizations allocate both instructions and variables onto the scratchpad memory and therefore optimize the total energy consumption of the system. However, the handling of array variables by the optimizations is a bit basic, as most of them consider aggregate variables for allocation onto the scratchpad. On the other hand, it was observed that an aggressive tiling of array variables severely degrades the control flow as well as the total energy consumption of the system. Therefore, the most promising step is to explore cost and size bounded approaches for array tiling which allocates both instruction segments and array tiles onto the scratchpad memory.

WCET Optimizations:
It has been observed that the predictability or the real-time responsiveness is becoming increasingly important design objective for consumer electronic devices. The scratchpad memory allows predictable and energy efficient accesses. Therefore, it is would a reasonable research direction to explore optimizations whose objective is to minimize worst case execution time (WCET) bounds on the system through the utilization of the scratchpad memory.

Multi-Process Scratchpad Sharing:
The scratchpad sharing approaches can be extended in numerous possible directions. The most promising extension is the integration of the scratchpad overlay as the underlying allocation approach to further reduce the energy consumption. The other promising extension is to consider multi-process applications composed of aperiodic tasks with different priorities.

Multi-Processor Scratchpad Allocation Approaches:
The approaches presented in this work represent the first such work on the utilization of scratchpad memories in multi-processor systems. The combination of memory allocation and optimizations with the mapping of tasks to processors in a homogenous or a heterogeneous multi-processor system opens a largely uncharted and yet, highly rewarding research direction.

A

Theoretical Analysis for Scratchpad Sharing Strategies

A.1 Formal Definitions

Definition A.1 (Non-Saving Energy Function $f_k^N(x)$). *The Non-Saving energy function $f_k^N\colon [0,M] \to \mathbb{R}$ for a process P_k takes the size of the disjoint scratchpad region as input and returns the energy consumed by the process P_k when it is allocated onto the disjoint region.*

Definition A.2 (Non-Saving Energy Function $h_n^N(x)$). *The Non-Saving energy function $h_n^N\colon [0,M] \to \mathbb{R}$ for a multi-process application with n independent processes is defined as the following:*

$$h_n^N(x_i) = min\left\{f_1^N(d_1) + \cdots + f_n^N(d_n) | d_1 + \cdots + d_n \leq x_i\right\} \ \forall x_i \in [0,M] \tag{A.1}$$

Definition A.3 (Binary Minimum Function ($binmin(f,g)$)). *The binary minimum function $binmin(f,g)$ takes two functions f and g as input and returns another function h as output.*

$$binmin\colon ([0,M] \to \mathbb{R}) \times ([0,M] \to \mathbb{R}) \longrightarrow ([0,M] \to \mathbb{R}) \tag{A.2}$$

$$h(x) = binmin(f,g) \tag{A.3}$$

where the returned function $h(x)$ satisfies the following property:

$$h(x_i) = min\{f(l_j) + g(m_k) | l_j + m_k \leq x_i\} \ \forall x_i \in [0,M] \tag{A.4}$$

A.2 Correctness Proof

Theorem 3 *The distributive property of the $\min$ operator over functions $f_1\colon [0,M] \to \mathbb{R}$, $f_2\colon [0,M] \to \mathbb{R}$ and $f_3\colon [0,M] \to \mathbb{R}$ can be shown as the following:*

$$\begin{aligned} &min\left\{f_1^N(d_1) + f_2^N(d_2) + f_3^N(d_3) | d_1 + d_2 + d_3 \leq x_i\right\} \ \forall x_i \in [0,M] \\ &\quad = binmin\left(binmin\left(f_1^N(l), f_2^N(m)\right),\ f_3^N(n)\right) \end{aligned}$$

Proof. Let us define function $h_3^N : [0,M] \rightarrow \mathbb{R}$ as the following:

$$h_3^N(x_i) = min\left\{f_1^N(d_1) + f_2^N(d_2) + f_3^N(d_3) | d_1 + d_2 + d_3 \leq x_i\right\} \ \ \forall x_i \in [0,M] \tag{A.5}$$

We know that if $x_1 + x_2 + x_3 \leq x$, then $x_1 + x_2 \leq x$. Using this property we define function $f_{12} : [0,M] \rightarrow \mathbb{R}$ as the following:

$$f_{12}^N(x_i) = min\left\{f_1^N(d_1) + f_2^N(d_2) | d_1 + d_2 \leq x_i\right\} \ \ \forall x_i \in [0,M] \tag{A.6}$$

We substitute the above equation into Equation A.5 to obtain the following equation:

$$h_3^N(x_i) = min\left\{f_{12}^N(d_1) + f_3^N(d_3) | d_1 + d_3 \leq x_i\right\} \tag{A.7}$$

From the definition of the *binmin* function (*cf.* Definition A.3) and Equation A.6, we obtain the following equation:

$$\begin{aligned} f_{12}^N(x_i) &= min\left\{f_1^N(l_j) + f_2^N(m_k) | l_j + m_k \leq x_i\right\} \ \ \forall x_i \in [0,M] && \text{(A.8)} \\ &= binmin\left(f_1^N(l), f_2^N(m)\right) && \text{(A.9)} \end{aligned}$$

The above equation is then substituted in Equation A.7 to obtain the following equation:

$$\begin{aligned} h_3^N(x_i) &= min\left\{f_{12}^N(d_1) + f_3^N(d_3) | d_1 + d_3 \leq x_i\right\} \ \ \forall x_i \in [0,M] \\ &= min\left\{binmin\left(f_1^N(l), f_2^N(m)\right) + f_3^N(d_3) | d_1 + d_3 \leq x_i\right\} && \text{(A.10)} \end{aligned}$$

Finally, we again substitute definition of *binmin* function into the above equation and obtain the following equality:

$$h_3^N(x) = binmin\left(binmin\left(f_1^N(l), f_2^N(m)\right), f_3^N(n)\right)$$

Theorem 4 *For any positive integer n, the static energy function of the multiprocess application $h_n^N(x)$*

$$h_n^N(x) = min\left\{f_1^N(x_1) + \cdots + f_n^N(x_n) | x_1 + \cdots + x_n \leq x\right\} \tag{A.11}$$

can be represented by the following recurrence equation:

$$\begin{aligned} \mathcal{H}_1^N(x) &= binmin\left(f_1^N(x), Z(x)\right) \\ \mathcal{H}_n^N(x) &= binmin\left(\mathcal{H}_{n-1}^N(x), f_n^N(x)\right) && \text{(A.12)} \end{aligned}$$

where, $Z(x) = 0$ is a zero function which returns zero for all values of x.

Proof. We will employ Structural Induction to prove the above theorem.
Basis of Induction: For $n = 1$,

$$\begin{aligned} h_1^N(x) &= min\{f_1^N(x_1) | x_1 \leq x\} && \text{(A.13)} \\ &= min\{f_1^N(x_1) + Z(x_2) | x_1 + x_2 \leq x\} && \text{(A.14)} \\ &= binmin\left(f_1^N(x), Z(x)\right) = \mathcal{H}_1^N(x) && \text{(A.15)} \end{aligned}$$

Inductive Hypothesis: Assume that for $n-1$ the following equality holds:

$$h_{n-1}^{N}(x) = \mathcal{H}_{n-1}^{N}(x)$$

Induction Step: From the definition of the $binmin$ operator (*cf.* Definition A.3) and Theorem 3, we derive the following:

$$\begin{aligned}
\mathcal{H}_{n}^{N}(x) &= binmin\left(\mathcal{H}_{n-1}^{N}(x), f_{n}^{N}(x)\right) \\
&= binmin\left(h_{n-1}^{N}(x), f_{n}^{N}(x)\right) \\
&= binmin\big(min\{f_{1}^{N}(x_1) + \cdots + f_{n-1}^{N}(x_{n-1})| \\
&\qquad\qquad x_1 + \cdots + x_{n-1} \leq x\}, f_{n}^{N}(x)\big) \\
&= min\left\{f_{1}^{N}(x_1) + \cdots + f_{n}^{N}(x_n) | x_1 + \cdots + x_n \leq x\right\} \\
&= h_{n}^{N}(x)
\end{aligned}$$

List of Figures

List of Tables

References

1. E. Aarts and R. Roovers. IC Design Challenges for Ambient Intelligence. In *Proceedings of Design Automation and Test in Europe (DATE'03)*, Munich, Germany, Mar. 2003.
2. AbsInt Angewandte Informatik GmbH. *aiT: Worst Case Execution Time Analyzers*. `http://www.absint.com/ait`, 2004.
3. R. Aitken, G. Kuo, and E. Wan. *Low-Power Flow Enable Multi-Supply Voltage ICs*. EETimes, `http://www.eetimes.com/news/design/showArticle.jhtml?articleID=15990221%6`, 2005.
4. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, San Francisco, California, 2002.
5. S. Anantharaman and S. Pande. An Efficient Data Partitioning Method for Limited Memory Embedded Systems. In *Proceedings of the ACM SIGPLAN'98 Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'98)*, Montreal, Canada, May 1998.
6. F. Angiolini, M. Francesco, F. Alberto, L. Benini, and M. Olivieri. A Post-Compiler Approach to Scratchpad Mapping of Code. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'04)*, Sep. 2004.
7. ANSI. *American National Standards Institute - ISO/IEC 9899:1999 (or: C99), "The ANSI C Standard"*. `http://www.ansi.org/`.
8. A. W. Appel and L. George. Optimal Spilling for CISC Machines with Few Registers. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'01)*, pages 243–253, Snowbird, Utah, USA, Jun. 2001.
9. ARM. *Advanced RISC Machines Ltd.* `http://www.arm.com/products/CPUs/ARM1156T2-S.html`.
10. ARM. *Advanced RISC Machines Ltd. - AMBA Homepage*. `http://www.arm.com/products/solutions/AMBAHomePage.html`.
11. ARM. *Advanced RISC Machines Ltd. - ARM7TDMI Reference Manual*. `http://www.arm.com/pdfs/DDI0210B_7TDMI_R4.pdf`.
12. ARM. *Advanced RISC Machines Ltd. - Development Tools*. `http://www.arm.com/products/DevTools/`.
13. ATMEL. *Atmel Corporation*. `http://www.atmel.com`.
14. O. Avissar, R. Barua, and D. Stewart. An Optimal Memory Allocation Scheme for Scratch-Pad Based Embedded Systems. *IEEE Transactions on Embedded Computing Systems (TECS)*, 1(1):6–26, Nov. 2002.
15. D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 26(4):345–420, 1994.

16. R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad Memory: A Design Alternative for Cache On-chip Memory in Embedded Systems. In *Proceedings of 10th International Symposium on Hardware/Software Codesign (CODES'02)*, Colorado, USA, May 2002.
17. U. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic Publisher, Boston u.a., 1. edition, 1993.
18. L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri. MPARM: Exploring the Multi-Processor SoC Design Space with SystemC. *Springer Journal of VLSI Signal Processing*, 41(2):169–182, Sep. 2005.
19. L. Benini, A. Bogliolo, G. Paleologo, and G. D. Micheli. Policy Optimization for Dynamic Power Management. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 18(6):813–833, 1999.
20. L. Benini and G. D. Micheli. *Dynamic Power Management - Design Techniques and CAD Tools*. Kluwer Academic Publishers, Massachusetts, 1998.
21. S. Borkar. Design Challenges of Technology Scaling. *IEEE Micro*, 19(4):23–29, 1999.
22. P. Briggs, K. D. Cooper, and L. Torczon. Improvements to Graph Coloring Register Allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):428–455, May 1994.
23. E. Brockmeyer, M. Miranda, H. Corporaal, and F. Cathoor. Layer Assignment Techniques for Low Energy in Multi-Layered Memory Organization. In *Proceedings of Design Automation and Test in Europe (DATE'03)*, Munich, Germany, Mar. 2003.
24. S. Carr. *Memory Hierarchy Management*. PhD Thesis, Rice University, Houston, Texas, USA, 1992.
25. G. J. Chaitin. Register Allocation & Spilling via Graph Coloring. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction (CC'82)*, pages 98–101, Boston, Massachusetts, USA, 1982.
26. C. Chekuri and S. Khanna. A PTAS for the Multiple Knapsack Problem. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'00)*, pages 213–222, San Francisco, California, USA, Jan. 2000.
27. D. Chiou, P. Jain, L. Rudolph, and S. Devadas. Application-Specific Memory Management for Embedded Systems Using Software-Controlled Caches. In *Proceedings of Design Automation Conference (DAC'00)*, Los Angeles, CA, USA, Jun. 2000.
28. G. Cichon, P. Robelly, H. Seidel, M. Bronzel, and G. Fettweis. Synchronous Transfer Architecture (STA). In *Proceedings of Fourth International Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS'04)*, Samos, Greece, Jul. 2004.
29. K. D. Cooper and T. J. Harvey. Compiler-Controlled Memory. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98)*, San Jose, CA, USA, Oct. 1998.
30. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill Book Company, New York, USA, 1990.
31. O. Coudert. Exact Coloring of Real-Life Graphs is Easy. In *Proceedings of Design Automation Conference (DAC'97)*, Anaheim, CA, USA, Jun. 1997. DAC.
32. CPLEX. *CPLEX Ltd.* `http://www.cplex.com`.
33. Dresden Silicon. *Samira Prototype DSP*. `http://www.dresdensilicon.com`, 2006.
34. J. Edler and M. D. Hill. *Dinero IV - Trace-Driven Uniprocessor Cache Simulator*. `http://www.cs.wisc.edu/~markhill/DineroIV/`.
35. B. Egger, J. Lee, and Heonshik Shin. Scratchpad Memory Management for Portable Systems with a Memory Management Unit. In *Proceedings of International Conference on Embedded Software (EMSOFT'06)*, Seoul, Korea, Oct. 2006.
36. A. E. Eichenberger, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind. Optimizing

Compiler for the Cell Processor. In *Proceedings of the The Fourteenth International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, Saint Louis, Missouri, USA, Sep. 2005.

37. ENCC. *University of Dortmund, Department of Computer Science XII.* `http://ls12-www.cs.uni-dortmund.de/research/encc`.
38. J. Fabri. *Automatic Storage Optimization*. UMI Research Press, Ann Arbor, Michigan, USA, 1982.
39. H. Falk and P. Marwedel. *Source Code Optimization Techniques for Data Flow Dominated Embedded Software*. Kluwer Academic Publishers, Norwell, MA, 2004.
40. H. Falk and M. Verma. Combined Data Partitioning and Loop Nest Splitting for Energy Consumption Minimization. In *Proceedings of Workshop on Software and Compiler for Embedded Systems (SCOPES'04)*, Amsterdam, The Netherlands, Sep. 2004.
41. P. Francesco, P. Marchal, D. Atienza, L. Benini, F. Catthoor, and M. J. Mendias. An Integrated Hardware/Software Approach for Run-Time Scratchpad Management. In *Proceedings of Design Automation Conference (DAC'04)*, Anaheim, California, USA, May 2004.
42. C. Fu and K. D. Wilken. A Faster Optimal Register Allocator. In *Proceedings of 31st International Microarchitecture Conference (MICRO'02)*, Istanbul, Turkey, Nov. 2002.
43. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide To the Theory of NP-Completeness*. Freeman, New York, USA, 1979.
44. GCC. *GNU Compiler Collection*. `http://gcc.gnu.org/`.
45. C. H. Gebotys. Low Energy Memory and Register Allocation Using Network Flow. In *Proceedings of Design Automation Conference (DAC'97)*, Anaheim, CA, USA, Jun. 1997.
46. N. Gloy, T. Blackwell, M. D. Smith, and B. Calder. Procedure Placement Using Temporal Ordering Information. In *Proceedings of 30th International Symposium on Microarchitecture (MICRO'97)*, Dec. 1997.
47. D. W. Goodwin and K. D. Wilken. Optimal and Near-optimal Global Register Allocation Using 0-1 Integer Programming. *Software-Practice and Experience*, 26(8):929–965, Aug. 1996.
48. S. C. A. Gordon-Ross and F. Vahid. Exploiting Fixed Programs in Embedded Systems: A Loop Cache Example. *Computer Architecture Letters*, 1, Jan. 2002.
49. F. Gruian. *Energy-Centric Scheduling for Real-Time Systems*. PhD Thesis, Lund University, Lund, Sweden, 2002.
50. GSI. *Geospatial Systems Inc.* `http://www.geospatialsystems.com/`.
51. M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proceedings of the 4th IEEE Annual Workshop on Workload Characterization*, Austin, Texas, USA, Dec. 2001.
52. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 3 edition, 2003.
53. IBM. *Cell Broadband Engine resource center*. `http://www-128.ibm.com/developerworks/power/cell/`.
54. ICD. *Informatik Centrum Dortmund (ICD e.V)*. `http://www.icd.de/es`.
55. Intel. *Microprocessor Hall of Fame*. `http://www.intel.com/museum/online/hist_micro/hof/tspecs.htm`.
56. Intel and Microsoft and Toshiba. *Advanced Configuration and Power Interface Specificaion*. `http://www.acpi.info`, 1996.
57. T. Ishihara and H. Yasuura. Voltage Scheduling Problem for Dynamically Variable Voltage Processors. In *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED'98)*, Monterey, CA, USA, Aug. 1998.
58. I. Issenin, E. Brockmeyer, M. Miranda, and N. Dutt. Data Reuse Analysis Technique for Software-Controlled Memory Hierarchies. In *Proceedings of Design Automation and Test in Europe (DATE'04)*, Feb. 2004.

59. I. Issenin and N. Dutt. FORAY-GEN: Automatic Generation of Affine Functions for Memory Optimizations. In *Proceedings of Design Automation and Test in Europe (DATE'05)*, Munich, Germany, Mar. 2005.
60. ITRS. *Information Technology Roadmap for Semiconductors*. `http://public.itrs.net`.
61. K. Jansen. Approximation Results for the Optimum Cost Chromatic Partition Problem. *Elsevier Journal of Algorithms*, 34(1):54–69, Jan. 2000.
62. M. S. Johnstone and P. R. Wilson. The Memory Fragmentation Problem: Solved? In *Proceedings of the 1st International Symposium on Memory Management (ISMM '98)*, pages 26–36. ACM Press, Oct. 1998.
63. M. Kamble and K. Ghosh. Analytical Energy Dissipation Models for Low Power Caches. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'97)*, Monterey, CA, USA, Aug. 1997.
64. M. Kandemir and A. Choudhary. Compiler-Directed Scratch Pad Memory Hierarchy Design and Management. In *Proceedings of Design Automation Conference (DAC'02)*, New Orleans, USA, Jun. 2002.
65. M. Kandemir, I. Kadayif, and U. Sezer. Exploiting Scratch-Pad Memory Using Presburger Formulas. In *Proceedings of the 14th Internation Symposium on System Synthesis (ISSS'01)*, Montreal, Canada, Sep. 2001.
66. M. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayaif, and A. Parikh. A Compiler-Based Approach for Dynamically Managing Scratchpad Memories in Embedded Systems. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems (TCAD)*, 23(2), Feb. 2004.
67. H. S. Kim, M. J. Irwin, N. Vijaykrishnan, and M. Kandemir. Effect of Compiler Optimizations on Memory Energy. In *Proceedings of IEEE Workshop on Signal Processing Systems (SIPS'00)*, pages 663–672, Lafayette, USA, Oct. 2000.
68. J. Kin, M. Gupta, and W. H. Mangione-Smith. The Filter Cache: An Energy Efficient Memory Structure. In *Proceedings of the 30th Annual International Symposium on Microarchitecture (MICRO'97)*, Research Triangle Park, North Carolina, USA, Dec. 1997.
69. D. E. Knuth. *The Art of Computer Programming: SemiNumerical Algorithms*, volume 2. Addison-Wesley Longman Publishing, Boston, MA, USA, 3 edition, 1973.
70. D. J. Kolson, A. Nicolau, N. Dutt, and K. Kennedy. Optimal Register Assignment to Loops for Embedded Code Generation. *ACM Transcations on Design Automation of Electronic Systems (TODAES)*, 1(2), Apr. 1996.
71. A. Krishnaswamy and R. Gupta. Profile Guided Selection of ARM and Thumb Instructions. In *Proceedings of the Joint Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'02) and Software and Compilers for Embedded Systems (SCOPES'02)*, Berlin, Germany, Jun. 2002.
72. C. Lee, J. K. Lee, T. Hwang, and S.-C. Tsai. Compiler Optimization on VLIW Instruction Scheduling for Low Power. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 8(2), Apr. 2003.
73. C. G. Lee. *University of Toronto Digital Signal Processing (UTDSP) Benchmark Suite*. `http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html`.
74. L. H. Lee, B. Moyer, and J. Arends. Instruction Fetch Energy Reduction Using Loop Caches for Embedded Applications with Small Tight Loops. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'97)*, San Diego, CA, USA, Aug. 1999.
75. S. Lee and T. Sakurai. Run-Time Voltage Hopping for Low-Power Real-Time Systems. In *Proceedings of Design Automation Conference (DAC'00)*, Los Angeles, CA, USA, Jun. 2000.
76. R. Leupers. *Code Optimization Techniques for Embedded Processors - Methods, Algorithms, and Tools*. Kluwer Academic Publishers, Norwell, MA, 2000.
77. Y. Li and J. Henkel. A Framework for Estimating and Minimizing Energy Dissipation of Embedded HW/SW Systems. In *Proceedings of Design Automation Conference (DAC'98)*, San Francisco, CA, USA, Jun. 1998.

78. M. Loghi, M. Poncino, and L. Benini. Cycle-Accurate Power Analysis for Multiprocessor Systems-on-a-Chip. In *Proceedings of the 14th ACM Great Lakes symposium on VLSI (GLSVLSI '04)*, New York, NY, USA, Apr. 2004.
79. M. Lorenz. *Performance- und energieeffiziente Compilierung für digitale SIMD-Signalprozessoren mittels genetischer Algorithmen*. PhD Thesis, University of Dortmund, Dortmund, Germany, 2003.
80. M. Lorenz and P. Marwedel. Phase Coupled Code Generation for DSPs Using a Genetic Algorithm. In *Proceedings of Design Automation and Test in Europe (DATE'04)*, Paris, France, Feb. 2004.
81. P. Machanick. Approaches to Addressing the Memory Wall. Technical report, School of IT and Electrical Engineering, University of Queensland, Nov. 2002.
82. A. Macii, L. Benini, and M. Poncino. *Memory Design Techniques for Low Energy Embedded Systems*. Kluwer Academic Publishers, Dordrecht, Boston, London, 2002.
83. S. Mamagkakis, C. Baloukas, D. Atienza, F. Catthoor, D. Soudris, J. M. Mendías, and A. Thanailakis. Reducing Memory Fragmentation with Performance-Optimized Dynamic Memory Allocators in Network Applications. In *Proceedings of Wired/Wireless Internet Communications (WWIC)*, Xanthi, Greece, May 2005.
84. T. Martin and D. P. Siewiorek. The Impact of Battery Capacity and Memory Bandwidth on CPU speed-setting: A Case Study. In *Proceedings of the International Symposium on Low Power Design (ISLPED'99)*, San Diego, California, USA, Aug. 1999.
85. P. Marwedel. *Embedded System Design*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1 edition, 2003.
86. P. Marwedel, L. Wehmeyer, M. Verma, S. Steinke, and U. Helmig. Fast, Predictable and Low Energy Memory References Through Architecture-Aware Compilation. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC'04)*, Yokohama, Japan, Jan. 2004.
87. Mediabench. *Benchmark Suite for Multimedia and Communication Systems*. `http://cares.icsl.ucla.edu/MediaBench/`.
88. H. Mehta, R. M. Owens, M. J. Irwin, R. Chen, and D. Ghosh. Techniques for Low Energy Software. In *Proceedings of the International Symposium on Low Power Design (ISLPED'97)*, Monterey, CA, USA, Aug. 1997.
89. MEMSIM. *University of Dortmund, Department of Computer Science XII*. `http://ls12-www.cs.uni-dortmund.de/~wehmeyer/LOW_POWER/memsim_doc`.
90. G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8), 1965.
91. G. E. Moore. No Exponential is Forever: but Forever can be Delayed! In *Proceedings of IEEE International Solid-State Circuits Conference (ISSCC'03)*, San Francisco, California, USA, Feb. 2003. ISSCC.
92. MOTOROLA. *Motorola Inc.* `http://e-www.motorola.com/files/shared/doc/selector_guide/SG1001.pdf`.
93. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, San Francisco, California, 1 edition, 1997.
94. Computer History Museum. *Timeline of Computers*. `http://www.computerhistory.org/`.
95. O. Ozturk and M. Kandemir. Integer Linear Programming based Energy Optimization for Banked DRAMs. In *Proceedings of ACM Great Lakes Symposium on VLSI (GLSVLSI'05)*, Chicago, Illinois, USA, Apr. 2005.
96. O. Ozturk, M. Kandemir, I. Demikiran, G. Chen, and M. J. Irwin. Data Compression for Improving SPM Behavior. In *Proceedings of Design Automation Conference (DAC'04)*, San Deigo, CA, USA, Jun. 2004.
97. P. R. Panda, N. Dutt, and A. Nicolau. *Memory Issues in Embedded Systems-On-Chip*. Kluwer Academic Publishers, Norwell, MA, 1999.

98. C. Park, J. Lim, K. Kwon, J. Lee, and S. L. Min. Compiler-Assisted Demand Paging for Embedded Systems with Flash Memory. In *Proceedings of International Conference on Embedded Software (EMSOFT'04)*, Pisa, Italy, Sep. 2004.
99. G. Peter, N. Dutt, and A. Nicolau. *Memory Architecture Exploration for Programmable Embedded Systems*. Kluwer Academic Publishers, Dordrecht, Boston, London, 2003.
100. J. L. Peterson and A. Silberschatz. *Operating System Concepts*. Addison Wesley, Massachusetts, USA, 1985.
101. P. Pettis and C. Hansen. Profile Guided Code Positioning. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI '90)*, White Plains, New York, USA, Jun. 1990.
102. J. M. Rabaey, A. Chandrakasan, and B. Nikolic. *Digital Integrated Circuits*. Pearson Education International, London u.a., 2 edition, 2003.
103. A. R. Rajiv, D. N. Pracheeti, S. D. Ganesh, D. M. Eric, M. S. Robert, A. M. Scott, and B. B. Richard. Compiler Managed Dynamic Instruction Placement in a Low-Power Code Cache. In *Proceedings of International Symposium on Code Generation and Optimization (CGO'05)*, San Jose, CA, USA, Mar. 2005.
104. RTEMS. *Real-Time Executive For Multiprocessor Systems*. `http://www.rtems.com`.
105. Y. Shin, K. Choi, and T. Sakurai. Power Optimization of Real-Time Embedded Systems on Variable Speed Processors. In *Proceedings of International Conference on Computer Aided Design (ICCAD'01)*, San Jose, CA, USA, Nov. 2001.
106. T. Simunic, L. Benini, and G. D. Micheli. Event Driven Power Management of Portable Systems. In *Proceedings of International Symposium on System Synthesis (ISSS'99)*, San Jose, CA, USA, Nov. 1999.
107. T. Simunic, L. Benini, G. D. Micheli, and M. Hans. Source Code Optimization and Profiling of Energy Consumption in Embedded Systems. In *Proceedings of the International Symposium of System Synthesis (ISSS'00)*, Madrid, Spain, Sep. 2000.
108. J. Sjödin, B. Fröderberg, and T. Lindgren. Allocation of Global Data Objects in On-Chip RAM. In *Proceedings of Workshop on Compiler and Architectural Support for Embedded Computer Systems*, Washington, USA, Dec. 1998.
109. M. D. Smith, N. Ramsey, and H. Glenn. A Generalized Algorithm for Graph-Coloring Register Allocation. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI'04)*, Washington, DC, USA, Jun. 2004.
110. M. B. Srivastava, A. P. Chandrakasan, and R. W. Broderson. Predictive Shutdown and Other Architectural Techniques for Energy Efficient Programmable Computation. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, 4(1):42–54, 1996.
111. ST. *STMicroelectronics Ltd.* `http://www.st.com`.
112. S. Steinke. *Untersuchung des Energieeinsparungspotenzials in eingebetteten Systemen durch energieoptimierende Compilertechnik*. PhD Thesis, University of Dortmund, Dortmund, Germany, 2003.
113. S. Steinke, N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan, and P. Marwedel. Reducing Energy Consumption by Dynamic Copying of Instructions onto Onchip Memory. In *Proceedings of the 15th International Symposium on System Synthesis (ISSS'02)*, Japan, Oct. 2002.
114. S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel. An Accurate and Fine Grain Instruction-Level Energy Model Supporting Software Optimizations. In *Proceedings of International Workshop on Power And Timing Modeling, Optimization and Simulation (PATMOS'01)*, Yverdon-Les-Bains, Switzerland, Sep. 2001.
115. S. Steinke, L. Wehmeyer, B. S. Lee, and P. Marwedel. Assigning Program and Data Objects to Scratchpad for Energy Reduction. In *Proceedings of Design Automation and Test in Europe (DATE'02)*, Paris France, Mar. 2002.

116. C. L. Su, C. Y. Tsui, and A. M. Despain. Saving Power in the Control Path of the Embedded Processors. *IEEE Design and Test*, 11(4), (Winter) 94.
117. The Economist. *Not just a flash in the pan*. `http://www.economist.com/displaystory.cfm?story_id=E1_VVSTVQQ`, 2006.
118. V. Tiwari, S. Malik, and A. Wolfe. Instruction Level Power Analysis and Optimization of Software. *Journal of VLSI Signal Processing Systems*, 13(3):223–238, Aug. 1996.
119. H. Tomiyama and H. Yasuura. Optimal Code Placement of Embedded Software for Instruction Caches. In *Proceedings of the 9th European Design and Test Conference (ED&TC'96)*, Paris, France, Mar. 1996.
120. UMC. *United Microelectronics Corporation*. `http://www.umc.com`.
121. F. Vahid. *Embedded System Design - A Unified Hardware/Software Introduction*. John Wiley & Sons, New York, USA, 2002.
122. M. Verma and P. Marwedel. Memory Optimization Techniques for Low-Power Embedded Processors. In *Proceedings of VIVA Workshop on Fundamentals and Methods for Low-Power Information Processing*, Bonn, Germany, Sep. 2005.
123. M. Verma and P. Marwedel. Advanced Memory Optimization Techniques for Low-Power Embedded Processors. In *Fundamentals and Methods for Low-Power Information Processing*. Springer, Dordrecht, The Netherlands, 2006.
124. M. Verma and P. Marwedel. Overlay of Scratchpad Memory for Low Power Embedded Processors. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, 14(8), Aug. 2006.
125. M. Verma, K. Petzold, L. Wehmeyer, and P. Marwedel. Memory Optimization Techniques for Low-Power Embedded Processors. In *Proceedings of IEEE 3rd Workshop on Embedded Real-Time Multimedia (ESTIMedia'05)*, Jersy City, New York, USA, Sep. 2005.
126. M. Verma, S. Steinke, and P. Marwedel. Data Partitioning for Maximal Scratchpad Usage. In *Proceedings of Asia South Pacific Design Automation Conference (ASPDAC'03)*, Kitakyushu, Japan, Jan. 2003.
127. M. Verma, L. Wehmeyer, and P. Marwedel. Cache-Aware Scratchpad Allocation Algorihm. In *Proceedings of Design Automation and Test in Europe (DATE'04)*, Paris, France, Feb. 2004.
128. M. Verma, L. Wehmeyer, and P. Marwedel. Dynamic Overlay of Scratchpad Memory for Energy Minimization. In *Proceedings of Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Stockholm, Sweden, Sep. 2004.
129. M. Verma, L. Wehmeyer, and P. Marwedel. Efficient Scratchpad Allocation Algorithms for Energy Constrained Embedded Systems. *Lecture Notes in Computer Science (LNCS)*, 3164(1): 41–56, 2004.
130. M. Verma, L. Wehmeyer, and P. Marwedel. Cache Aware Scratchpad Allocation Algorithms for Energy Constrained Embedded Systems. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 25(10):2035–2051, 2006.
131. M. Verma, L. Wehmeyer, R. Pyka, P. Marwedel, and L. Benini. Compilation and Simulation Tool Chain for Memory Aware Energy Optimizations. In *Proceedings of Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS VI)*, Samos, Greece, Jul. 2006.
132. L. Wang, W. Tembe, and S. Pande. A Framework for Loop Distribution on Limited On-Chip Memory Processors. In *Proceedings of the International Conference on Compiler Construction (CC'00)*, Berlin, Germany, Mar. 2000. CC.
133. L. Wehmeyer. *Fast, Efficient and Predictable Memory Accesses - Optimization Algorithms for Memory Architecture Aware Compilation*. Springer, Dordrecht, The Netherlands, 2005.
134. L. Wehmeyer, U. Helmig, and P. Marwedel. Compiler-optimized Usage of Partitioned Memories. In *Proceedings of the 3rd Workshop on Memory Performance Issues (WMPI'04)*, Munich, Germany, Jun. 2004.

135. L. Wehmeyer and P. Marwedel. Influence of Memory Hierarchies on Predictability for Time Constrained Embedded Software. In *Proceedings of Design Automation and Test in Europe (DATE'05)*, Munich, Germany, Mar. 2005.
136. E. W. Weisstein. *Function: MathWorld-A Wolfram Web Resource*. `http://mathworld.wolfram.com/Function.html`.
137. S. J. E. Wilton and N. P. Jouppi. CACTI: An Enhanced Cache Access and Cycle Time Model. *IEEE Journal of Solid-State Circuits*, 31(5), May 1996.
138. M. E. Wolf and M. S. Lam. A Loop Transformation Theory and an algorithm to maximise parallelism. In *Proceedings of The 3rd Workshop on Programming Languages and Compilers for Parallel Computing (PLCPC'90)*, Aug. 1990.
139. W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *ACM Computer Archtiecture News*, 23(1), Mar. 1995.
140. S. Wuytack, F. Catthoor, L. Nachtergaele, and H. D. Man. Power Exploration for Data Dominated Video Applications. In *Proceedings of the International Symposium of Low-Power Electronics and Design (ISLPED'96)*, Monterey, CA, USA, Aug. 1996. ACM.
141. C. Zhang, F. Vahid, J. Yang, and W. Najjar. A Way-Halting Cache for Low-Energy High-Performance Systems. In *International Symposium on Low-Power Electronics and Design (ISLPED'00)*, Newport Beach, CA, USA, Aug. 2000.