FAST, EFFICIENT AND PREDICTABLE MEMORY ACCESSES

Fast, Efficient and Predictable Memory Accesses

Optimization Algorithms for Memory Architecture Aware Compilation

by

LARS WEHMEYER
University of Dortmund, Germany

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-10 1-4020-4821-1 (HB)
ISBN-13 978-1-4020-4821-0 (HB)
ISBN-10 1-4020-4822-X (e-book)
ISBN-13 978-1-4020-4822-7 (e-book)

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

www.springer.com

Printed on acid-free paper

Printed in the Netherlands.

For Antje

Contents

Acknowledgements

This work would not have been possible without the help of several people. The support of our colleagues and the technical discussions with Heiko Falk, Markus Lorenz, Robert Pyka, Stefan Steinke, Jens Wagner and Manish Verma have always been a great help. Special thanks go to Heiko Falk for all the support that he provided.

Working with students has been a most rewarding part of our work during the past years. We thank in particular the master students Lars Hornbach, Sergej Schwenk, Urs Helmig, André Kernchen and Thorsten Wilmer for the good cooperation and for having contributed to this work.

Finally, we are deeply indebted to our families who have given us the love and support that enabled us to write this book.

Dortmund, February 2006

Lars Wehmeyer
Peter Marwedel

1

Abstract

The influence of embedded systems is constantly growing. Increasingly powerful and versatile devices are being developed and put on the market at a fast pace. The number of features is increasing, and so are the constraints on the systems concerning size, performance, energy dissipation and timing predictability. Since most systems today use a processor to execute an application program rather than using dedicated hardware, the requirements can not be fulfilled by hardware architects alone: Hardware and software have to work together in order to meet the tight constraints put on modern devices. This work presents approaches that target the software generation process using an energy and memory architecture aware C-compiler. The consideration of energy dissipation and of the memory architecture leads to a large optimization potential concerning performance and energy dissipation.

This work first presents an overview over the used timing, energy and simulation models for one processor architecture and for different memory architectures like caches, scratchpad memories and main memories in both SRAM, DRAM and Flash technology. Following an introduction to the used compilation framework, the compiler based exploitation of partitioned scratchpad memories is presented. A simple formalized Base model is presented that models the consequences of statically allocating instructions and data to several small scratchpad partitions, followed by a number of extensions that treat memory objects and their dependencies at a finer granularity. A method for allocating objects to separate scratchpad memories for instructions and data, as found in the most recent ARM designs, is also presented. Finally, a model that also considers the leakage power of memories is introduced. Results show that significant savings of up to 80% of the total energy can be achieved by using the presented scratchpad allocation algorithms. The flexibility and extensibility of the presented approaches is another benefit.

Many embedded systems have to respect timing constraints. Therefore, timing predictability is of increasing importance. Whenever guarantees concerning reaction times have to be given, worst case execution time (WCET) analysis techniques are being used during the design of the system in order

to provide a guaranteed upper bound on the WCET. The contribution of this work deals with the influence of scratchpad memories on timing predictability. It is shown that scratchpad memories, allocated using the algorithms mentioned above, are inherently predictable, since the positions of all objects in the different memories are fixed at compile time and no dynamic decisions have to be taken at runtime. The results show that the determined WCET values for systems with a scratchpad memory scale with the performance benefit observed during average case simulation, indicating that scratchpad memories lead to improvements both concerning average case and worst case. In particular when compared to caches, the WCET analysis for scratchpad based systems is simpler, yet allows the generation of tighter bounds. The effects of allocating instructions and data to the scratchpad using a dynamic allocation algorithm are shown in this work for the first time. This allocation technique both outperforms the cache and leads to better timing predictability, making scratchpad memories a natural choice for timing constrained embedded systems.

Advances in main memory technology include the availability of memory chips with integrated power management. The first optimization targeting main memories exploits these features by allocating memory objects to a scratchpad partition in order to allow the main memory to be put into power down mode whenever instructions and data are being accessed from the scratchpad memory. The allocation problem uses the standby energy of the main memory in SDRAM technology to allocate objects to the scratchpad memory so as to maximize the power down periods of the main memory. Total energy savings of up to 80% were achieved. In the second main memory optimization, suitable Flash memories are being used as instruction memories using eXecute-In-Place (XIP). By considering the tradeoff between the overhead required to copy instructions to the faster SDRAM and the benefits achieved due to the faster execution, the compiler determines an optimal allocation of instructions to Flash and SDRAM memories. The main benefit of this approach is significant savings in the required amount of instruction memory in SDRAM technology, one of the main cost factors for embedded systems.

Finally, the influence of the size of the register file on the quality of the generated code is studied. It is shown that if the register file is too small, then a lot of code overhead is generated due to the need to spill register values to memory. Beside presenting results for the spill code overhead, performance and energy dissipation of the generated code, a compiler guided method to choose an adequate size for the register file for a certain application is presented.

The work is concluded by a summary and an outlook on future work.

2 Introduction

Over the years, a change concerning the use and perception of computing systems can be observed. While in the old days of computing, computers filled whole living rooms and were only affordable by research institutes and governments, desktop computers have lead to the computer becoming a common object in everyday life for most people. In the past few years, embedded systems that are not perceived as computers by the user have become popular. Mobile phones are a prime example for this development: they are used by many people all over the world, yet in general, the user is not aware of the fact that his mobile phone contains a processor and executes a program.

In general, computers are moving away from the desktop and can now be found everywhere, which is why the terms “ubiquitous computing” or “pervasive computing” have become popular. They express the change in paradigms concerning computer systems in catchy buzzwords: the computer systems of the future are everywhere, they can either be carried around by the user, or they are found in the form of electronic entertainment devices, in cars, trains and planes and in many other objects that we encounter every day. The fact that computer systems are embedded in devices that offer a certain service makes it hard to perceive these system as being computers. Many of the services offered by embedded systems would not even be possible without them: an anti lock braking system in a car could hardly be implemented without an embedded system to control it. The availability of safety or comfort functions has stimulated the demand for new services and applications. The area of embedded systems is thus experiencing a rapid growth.

The following, incomplete list provides a couple of popular examples for embedded systems and some of their properties:

Handheld Devices: Small embedded systems with a mobile energy supply that can be carried everywhere are now very common: mobile phones, personal digital assistants or portable MP3 players are examples of such handheld devices. For some tasks like taking notes and scheduling appointments, PDAs have taken the role of paper organizers, at the same time providing a load of additional convenience features like automatic alarms, repeating appointments and games. The emergence of mobile phones has increased people's availability, influencing business as well as private life. The clear distinction between these different mobile devices is blurring: designers are working towards universal personal assistants. Smartphones that incorporate both PDA and mobile phone are one example, mobile phones with integrated digital cameras another. In the future, the user may only have to carry around a single highly personalized embedded system to meet all his needs.

Automotive/Avionics: The use of electronic control units is very common in the automotive industry today: Cars without anti lock brake system or engine management system could hardly be sold. The car used to be the domain of electrical and mechanical engineers. Now, embedded systems are more and more taking over control in the automotive area. Since changing the hardware is expensive once it is integrated into the product, processors executing software applications that can be modified using a Flash update are being used extensively. In modern cars like the BMW 7 series, there are more than 60 MB of software running on up to 70 microcontrollers and processors [Saa03]. Technologies like "drive-by-wire" which are still lacking acceptance in the automotive sector are considered state-of-the-art in avionics. Modern aircraft can not be flown by the pilot alone without the aid of computers, which is why these aircraft have been described as "computers with wings".

Telecommunication: Modern telecommunication switching centers used to be controlled by relays and lots of hardware. Today, embedded systems have taken over. These systems are expected to work for a long time in a reliable way. An interesting aspect is the fact that the maximum heat that can be tolerated during operation of these devices is strictly limited due to the high packaging density of the corresponding systems in the automatic telephone exchange. For the home user, telephone systems provide convenient services like in-house phone-to-phone calls or automatic activation of a fax machine. The fact that these systems are really computers show when firmware updates have to be installed, or when the system crashes and has to be restarted.

The previous examples give a very limited overview over the relevance of ubiquitous or pervasive computing: computers surround us everywhere, and a lot of our everyday lives are directly or indirectly influenced by this development. This broad invasion of embedded systems has become possible due to a number of developments. However, in order to further improve the performance and convenience of using embedded devices, there is still a lot of room for improvement.

All the embedded systems mentioned so far consist of some kind of processor that executes an application program in order to deliver a certain service or to control its environment. While in particular microcontrollers have been programmed in assembly language for a long time, the increasing complexity of the software executed on embedded devices makes the use of high level programming languages mandatory. Currently, using the C programming language is one of the most popular ways of developing software for embedded systems. One advantage of C is the higher level of abstraction that the language offers compared to assembly programming, yet the programmer has the chance to access "low level" functions of the processor using inline assembly code for performance-critical sections of code. The use of a high level language makes it necessary to provide efficient compilers that translate the application program into the used processor's machine code. Efficient optimizing compilers are available to translate the C programming language into the machine code of many popular processors used in embedded devices. For the popular ARM7 processor series that was used in the experiments in this work, several alternative compilers exist. Beside commercial vendors, ARM offers a complete compiler toolsuite which also contains assembler, linker and an instruction set simulator [ARM98a]. Furthermore, there is a version of the popular gcc [GCC05] that generates efficient code for the ARM processor. In general, the quality of the application code executed on the target system has a strong impact on the performance and the energy dissipation of the entire system. In fact, the requirements of future systems will only be met if both hardware and software work together.

This work uses a specific research compiler to generate code from an application program written in C. The special feature of the used compiler that distinguishes it from competitors is the fact that it is capable of generating code that is optimized for energy using an instruction level energy model. In addition, it takes into account several properties of the surrounding memory architecture in order to generate optimized code that effectively exploits the architectural features that are present in the system. The algorithms and optimizations described in this work target the memory hierarchy of the system under consideration in an attempt to make memory accesses fast, efficient and timing predictable.

The following section provides a general overview over the problems and issues encountered during the design of embedded system today, highlighting in particular issues related to the generation of optimized code. The contributions of this work are presented in the subsequent section. Finally, the last section of this introduction provides a short overview over the entire work.

2.1 Motivation

A number of properties have to be fulfilled by embedded devices in order to be accepted and attractive for the potential user. On one hand, the customer buys e.g. a portable device since he is interested in the services offered by the product. In contrast to e.g. desktop processors, there will hardly ever be a discussion about the processor used in a mobile phone: what really counts is that the system meets the user's requirements, how this is done is not relevant. In order to provide unobtrusive service, the device should have a sufficiently high performance to e.g. show good reaction times to the user's requests or to be able to playback a video without flickering. It is therefore one prime concern of designers to implement high-performance embedded systems. If the processors found in commonly used embedded systems today were not as powerful as they are and dedicated hardware was used instead of software applications, most of the devices that are widely used today would be unaffordable.

Considering the effect of performance alone will lead to powerful processors and abundant peripherals to be included used in the system. However, portable devices in general have to operate on battery power instead of using abundant power from wall sockets. In particular for those systems, the running time with a single charge of the battery is an important aspect. Users will hardly accept a mobile phone whose battery needs to be recharged every other day, or even risk missing calls due to empty batteries. Even if the performance of the system is satisfactory, the battery standby and operating times are another vital aspect. Therefore, the designer has to determine a tradeoff between the performance of the processor and the acceptable energy dissipation. In addition, lower currents flowing through the device's circuits will also lead to improved long-term reliability of the system.

Having considered performance and energy dissipation in the processor, other factors that influence the performance and the energy dissipation of embedded systems should be mentioned. In general, a development that has so far mainly affected desktop computers is currently becoming a problem for embedded systems: the increasing demands concerning the versatility of embedded devices inevitably leads not only to the use of faster processors, but also to the requirement for larger memory capacities. The development of processors is proceeding at a fast pace, and more powerful processors that consume less energy per operation are developed with great speed. The fact that processors today perform operations at a speed that was only reached by application specific circuits a couple of years ago is one of the main reasons for the widespread use of embedded systems. Recent developments indicate, however, that the increasing size of the memories used to store both the application's instructions and data will turn out to be the real bottlenecks. In fact, it is to be expected that in a couple of years, slow memories will dominate the performance not only of embedded systems, but of computers in general. This effect has been termed the "Memory Wall" by [Mac02, WM95]. Figure 2.1 shows that currently, processor performance is improved by around 50 to 100% per year, whereas memory speed is only increased by about 7% in the same time. This leads to an increasing gap between the speed of the processors and the memories. In fact, the gap doubles in size every two years. The slow development concerning the performance of main memories in DRAM technology is caused by the fact that that developments in this area have primarily been targeted at increased capacity rather than speed.

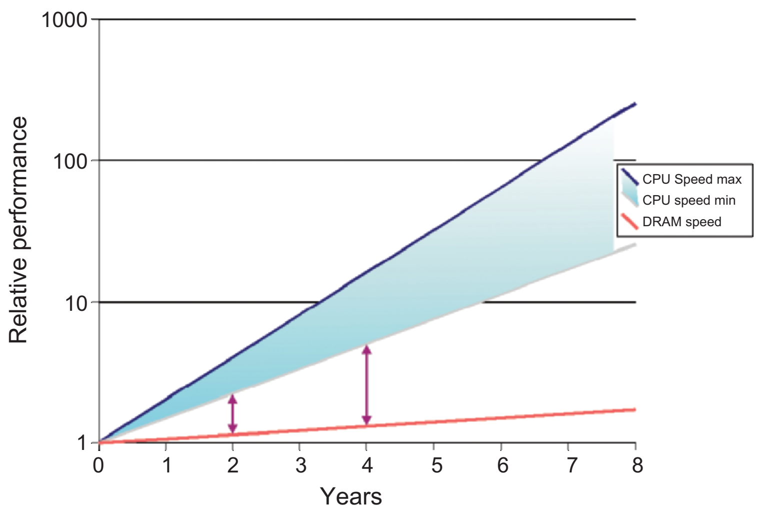


Fig. 2.1. Gap between CPU and memory performance [Mac02]

Publications about the memory wall argue that in a couple of years, highly efficient processors will spend most of their time waiting for the slow memories. In addition to dominating the performance of systems, memories are also responsible for a high percentage of the energy dissipated in a computing system: researchers have found that the energy dissipated in the memory subsystem can be higher than that consumed by the processor [KVIY00]. Thus, just like the memory wall has already been found to be a major threat for the future development of systems concerning performance, a similar effect can be expected to occur concerning energy dissipation as well.

For any given technology, access times as well as the energy required for one single memory access are a function of the memory size: The larger the memory, the larger the access times and the energy consumed per access. This relationship is shown in Figure 2.2. The fact that more and more memory capacity is required in order to fulfill the various tasks demanded of computing systems today increases the growing gap between the ever-faster processors and the slower main memory.

Several techniques can be used to take into account the effects of the memory subsystem on the design targets. In order to reduce the negative effect of the large main memories on the overall system performance, several small memories are usually found in modern systems in the form of a memory hierarchy. This is possible since most application programs exhibit a high degree of temporal and spatial locality: Temporal locality means that an item that has been accessed in the recent past is likely to be accessed again in the near future. Spatial locality means if an object has been accessed in the past, then it is probable that an element in the vicinity of that object will be accessed in the future. By placing small and efficient memories close to the processor that exploit the applications' locality, the long access times and

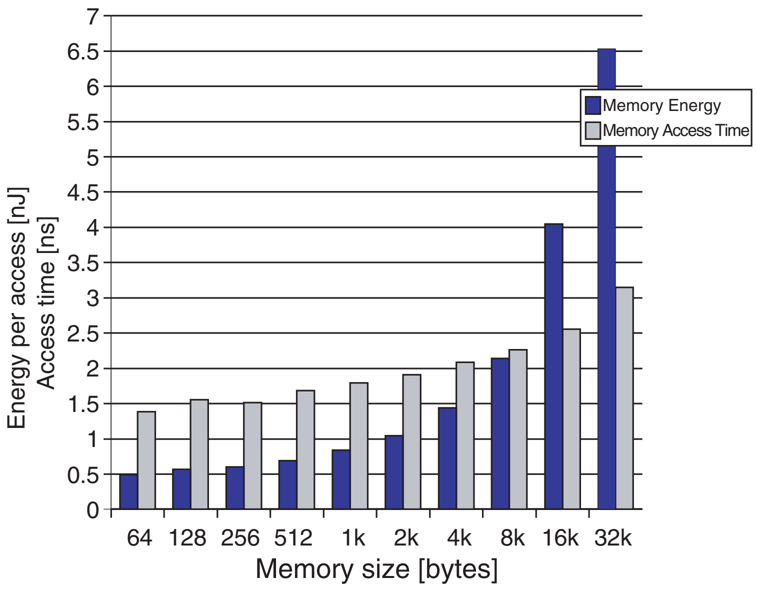


Fig. 2.2. Energy and time per access for increasing memory sizes

high energy consumption of larger background memories can be avoided. The storage locations that are closest to the processor are the registers that all operands are read from in RISC architectures. If all values can be kept in the registers instead of having to access main memory, maximum performance can usually be obtained from the system. However, if the main memory has to be accessed, the gap between the fast registers and the slow main memory is still very large. Thus, more intermediate levels are introduced into the memory hierarchy. The most popular instantiation of this concept is the use of caches: the cache automatically loads a value requested by the processor and keeps it for future reference. Since caches that are close to the processor are usually quite small, the performance of the system is not impaired if the requested values can frequently be fetched from the faster cache instead of the main memory.

In particular for desktop PCs, caches are the solution usually found in order to improve the performance of the memory subsystem. Caches are capable of automatically detecting whether a requested item is stored within their data memory (cache hit) or not (cache miss). If the item is not in the cache, the next level of the memory hierarchy is accessed in order to retrieve the requested data. The ability to keep track of their current contents is the main advantage of caches: since they do this automatically, a system can benefit from the presence of a cache without requiring any further change in hard- or software. The drawback of caches lies in their unpredictable behavior, which can lead to performance degradations instead of improvement if wrong cache parameters are chosen, and in the additional energy consumed by the architecturally complex address comparison circuitry. In fact, it has been determined in [KG97] that onchip caches can consume 25 to 45% of the total chip power.

Despite the fact that no software or compiler support is required to take advantage of a cache, knowledge about the cache architecture can help to generate code that exploits the cache more efficiently, e.g. by choosing addresses in such a way that no conflict misses occur, or by using optimizations like loop tiling which reorganizes the application program such that its iterations and reused data elements match the cache's organization to provide a minimal number of cache misses.

Scratchpad memories are another concept to build a hierarchy of several memories in order to improve performance and energy dissipation. A scratchpad memory is simply a small memory that is usually located close to the processor. In most cases, the scratchpad will be on the same chip as the processor to allow fast and efficient accesses. Consisting only of a small memory array, it is not capable of automatically keeping track of its contents like the cache is. The drawback of this property is that explicit software support, provided either by the programmer or by the used compiler, is required to exploit the presence of a scratchpad memory by e.g. allocating instructions and data to this memory region and generating code that accesses the objects from the small, fast and efficient scratchpad instead of main memory. The advantage, on the other hand, is that due to the missing comparison logic, scratchpad memories are both smaller and more energy efficient than caches [BSL$^+$02]. However, scratchpad memories are not as widely accepted and understood as caches. Despite their advantages, which will be covered in detail in the further course of this work, a complete and consistent toolchain for their utilization is still missing in industry.

There are basically two ways of allocating objects to an available scratchpad memory: the static case, where code and data is loaded in the scratchpad before the start of the program and all memory objects stay on the scratchpad throughout the execution, and the dynamic case, where objects may be swapped between main and scratchpad memory at runtime to react to changing working sets of the executed program. Even in the dynamic case, the instructions to copy certain memory objects to and from the scratchpad are already inserted into the application at compile time, meaning that no dynamic decisions have to be taken at runtime.

This property of fixing program behavior at compile time not only improves the obtainable performance and energy dissipation, but also the timing predictability of the system. Timing predictability is an important issue e.g. in automotive equipment. A popular example is the airbag system: if the airbag is not fully inflated within 40 milliseconds following a crash, the entire system is worthless. In such hard real time systems, it is not performance that plays the major role, but timing predictability. To design a predictable system, the designer needs to be able to specify at design time an upper bound on the execution time that is always guaranteed and never exceeded, even under the worst of circumstances. This upper bound is called the worst case execution time (WCET). If in the actual operating environment, a safety critical system does not meet the WCET specifications guaranteed by the designer, severe injuries may occur. Note that the aspect of building a timing predictable system is quite different from optimizing a system for average case performance. While e.g. a cache can help improve the observed average case execution time of a system, the integration of caches into WCET analysis techniques is not trivial: to determine a guaranteed upper bound on the execution time, either pessimistic assumptions concerning cache performance have to be made which lead to poor WCET results, or complex analysis techniques have to be employed to predict the dynamic runtime behavior of the cache in advance.

Main memories are in general large and slow, and they should only be accessed when no smaller memory that is located closer to the processor can supply the required data element. Due to their size and due to the long bus lines connecting main memory and processor, the energy dissipation for accessing the main memory is very high. To overcome these disadvantages, modern SDRAM memories provide power management features that allow e.g. parts of the circuit to be shut down if they are not being used.

Most embedded systems today feature another memory type that is sometimes only used at startup of the device: Memories in Flash technology are being used as non-volatile memories that store applications and data as well as configuration information even when the power supply of the system is switched off. Usually, the contents of the Flash memory are copied to the correct locations in the memory hierarchy at startup of the device and then executed or accessed from there. Not using the Flash memory during normal operation of the systems is actually a waste of resources: Flash memories in NOR technology may be used as so-called eXecute-In-Place (XIP) memories, which means that the processor fetches instructions directly from the Flash memory instead of first having to copy them to another memory region. The fact that Flash memories are relatively slow compared to e.g. SDRAM main memory results in a tradeoff of selectively allocating instructions either to the Flash or to SDRAM memory.

Finally, the register file, being the part of the memory hierarchy that is closest to the processor, must not be neglected. Due to its small size, it provides very fast access. Since in a RISC processor, the register file is generally used to hold all operand values for all performed operations, it is a place of extremely high switching activity and thus high energy dissipation. Based on the fact that in the energy efficient M*Core architecture, 16% of total processor power and as much as 42% of the data path power is consumed within the register file [SLAM98], it is mandatory to treat the register file with care when it comes to designing energy efficient embedded systems. The code generated by the compiler is also influenced by the size of the register file: if an insufficient number of registers is available to hold the required values, a strong overhead concerning code size, performance and energy is introduced since register values have to be saved to memory if insufficient space is available in the register file.

The discussed topics represent challenges both to designers of embedded systems and to compiler architects. The following section shows how this work contributes to solving several of the mentioned problems and issues.

2.2 Contributions of this Work

In order to observe and evaluate the performance of embedded systems, a timing model for the considered processor is required. An instruction level timing models for the ARM processor is thus presented. This model is used within the compiler environment to optimize and evaluate the performance of the generated code.

Instruction level processor energy models are required to observe the influence of generated code on the energy dissipation of a system. The used energy model for the ARM7 is based on actual physical measurement. It can be used in the compiler framework to generate code that considers the energy dissipation of the generated code sequences and thus leads to energy optimized code. In addition, the model allows the evaluation of the code quality. However, in RISC architectures the choice of available instructions to perform a certain operation are limited, so that the compiler hardly ever has a choice of generating code that is significantly more energy efficient than an alternative code sequence that performs the same operation. The results for energy optimized code are usually identical to the results for performance optimization, at least if the memory hierarchy is not considered.

In order to provide more potential for compiler optimizations, the used memory architecture should thus be included in the considerations. This work handles caches by providing a flexible memory hierarchy simulator that is capable of simulating arbitrary configurations of different memories, including different cache organizations. No explicit optimizations that directly target the use of caches are presented, since the concept of using scratchpad memories offers more promising optimization potential. Energy models for both caches and the more energy efficient scratchpad memories are presented in the work.

Unlike caches, scratchpad memories require explicit support from either the programmer or the compiler. In this work, the compiler is used to automatically allocate memory objects, i.e. instructions and data, to the fast and energy efficient scratchpad memories. Allocation can be performed in a static or in a dynamic way, as explained in the previous section. In both the static and the dynamic case, the compiler analyzes the code concerning loops and frequently accessed data elements which are promising candidates for allocation to the scratchpad. Allocating these candidates usually both improves performance and reduces the energy dissipation when the program is executed. This work presents detailed algorithms and analyses of static allocation algorithms that take advantage of partitioned scratchpad memories. Since smaller memories require less energy per access (cf. Figure 2.2), providing 4 kB of scratchpad memory as e.g. two memories of 2 kB each can be expected to be more energy efficient than using one single partition. This work first provides a formal representation of the basic concept of exploiting partitioned scratchpad memories in the compiler using static allocation techniques. This basic model is subsequently extended significantly by providing formalization and experimental data for three different approaches that differ in the treatment of memory objects at a fine granularity. In addition, generating code that supports the TCM architecture found in the most recent ARM designs [ARM04a] and taking into account the leakage energy dissipation of multiple scratchpad partitions are novel contributions.

Both static and dynamic allocation approaches fix all decisions concerning memory layout at compile time. This concept leads to an inherent predictability of systems that use a scratchpad memory and the proposed allocation algorithms. This work presents results that show the effect of both a cache and a statically allocated scratchpad on the achievable worst case execution time (WCET). The results in this work for the first time consider the allocation of basic blocks instead of only complete functions, which further improves the quality of the achieved results. It is shown that while a cache, despite requiring complex analysis techniques, does not improve the WCET of a system, the use of a scratchpad memory has a positive effect both on the average case performance and on the WCET. As a further novel contribution, this work for the first time considers the effect of a dynamic scratchpad allocation algorithm on the WCET of a system. Results show that even in the dynamic case, the scratchpad remains inherently predictable. As a side note, the dynamic allocation algorithm is capable of outperforming an architecturally more complex 4-way set associative cache.

Main memories are one of the main contributors to both runtime and energy dissipation in computing systems. Today, the large main memories are usually built using SDRAM technology. These dynamic memories are more difficult to handle in the compiler than the previously considered static memories, since their timing and energy dissipation depends on the access pattern. In addition, the standby power dissipation of SDRAM memories is very much higher than that of static RAM cells due to the necessity to periodically refresh the memory contents. This refresh power has to be considered when SDRAM memories are integrated into the system. SDRAM timing and energy models are introduced and their integration into the compiler framework is presented. An optimization that targets the power management features of modern main memories is formalized and integrated into the compiler. As a result, the compiler statically allocates instructions and data to an available scratchpad memory partition in order to maximize the time the main SDRAM memory can be kept in power down mode.

The exploitation of the presence of Flash memories in most embedded systems is also covered in this work. Instructions can either be executed directly from the slower Flash memory, or they can be copied to the faster main memory at startup of the device. The resulting tradeoff can be solved by the compiler, which thus determines a suitable allocation of objects to the Flash and the SDRAM memory. As a result of this proposed optimization, significant savings in the required SDRAM instruction memory are reported.

Finally, the effect of the register file size on the achievable code quality is studied. The results show that the register file size is an important factor for the efficient execution of application code on the processor, since an insufficient number of registers results in a large amount of spill code to be inserted into the generated code that copies register values into memory and reloads them when required. As a novel contribution of this work, the compiler can also analyze an application program in order to determine an optimal register file size. Since the compiler is aware of the register pressure at all times during the application's execution, it can find the minimal register file size that leads to an acceptable amount of spilling. While register files used to be fixed in size, the increased use of configurable cores also puts this parameter under the control of the system designer.

This concludes the short look at this work's contributions concerning memory architecture aware compilation. The following section provides a short overview over the structure of the remainder of this work.

2.3 Overview

Following this introduction and overview, Chapter 3 presents the models and tools that were used in the further course of the work in order to capture the properties of both the processor and the memories with respect to behavior, timing and energy dissipation. The timing and energy models are used within the compiler to generate optimized code according to the respective optimization, and also to evaluate the generated code concerning performance and energy dissipation. This evaluation is usually done by performing a simulation run of the generated code on the assumed hardware. To allow for maximum flexibility for simulation, several simulator models are presented that can be used depending on the performed compiler optimization. Section 3.6 presents the used encc compiler in-depth and also provides information concerning the development environment and the used workflow to perform memory architecture aware compilation.

Chapter 4 presents the optimizations that primarily target the scratchpad memory. Following a look at related work, a static allocation model of both instructions and data to a partitioned scratchpad memory is presented. This simple Base model is subsequently extended to treat memory objects at a finer granularity and to map objects to a Harvard-style TCM-architecture found in modern ARM designs. Finally, results for taking into account memory leakage power are presented.

In the second half of Chapter 4, the impact of scratchpad memories on the worst case execution time (WCET) of real time capable systems is examined.

The results of both using a static and a dynamic scratchpad allocation indicate that since scratchpad memories are inherently predictable, no additional analysis techniques are required to obtain good results concerning WCET.

The power management features of modern main memories in SDRAM technology are the target of Chapter 5. First, an energy model based on the standby energy of the main SDRAM memory is used to allocate memory objects to a scratchpad in such a way that the main memory can be kept in the energy-saving power down mode for a maximum amount of time. Following the results for this algorithm, an optimization that uses a Flash memory as instruction memory is presented. Since the entire application is assumed to be kept in the Flash memory when the power is turned off, significant savings in terms of SDRAM instruction memory requirements are observed.

Finally, the size of the register file and its impact on the quality of the generated code is considered in Chapter 6. Following the observations made by modifying the compiler and generating code for different register file sizes, a compiler guided method of determining an adequate size for the register file is presented.

The summary and an outlook on possible future work conclude this work.

3

Models and Tools

In order to perform architecture aware compilation, optimization and simulation, it is necessary to model the environment that is to be optimized with respect to the used cost function. In the context of memory architecture aware compilation, the entities that need to be considered are the processor and the memory subsystem. The tools used in the course of code optimization each require knowledge about certain aspects of the processor and the memory subsystem. This information is passed to the optimizing toolchain through the use of models. The entire system with its information flow is shown in Figure 3.1. Starting at the left hand side of the figure, processor and memory both have a certain timing and energy behavior captured in a corresponding model that can then be used during code generation and optimization. Simulation models are useful to determine the effect of an optimization and to evaluate its benefit. An instruction set model is additionally required for the processor.

The right hand of the figure comprises the tools generally used in code generation and optimization: the compiler with integrated optimization routines and a simulator to evaluate optimization results. To generate code of high quality, the compiler and the optimizer require detailed information concerning both the timing and the energy dissipation of instructions. In general, the entire instruction set architecture (ISA) of the used processor must be known in order to be aware of all code generation and optimization alternatives. Additional information can be used within the compiler to generate more efficient code using the optimizer. The link between compiler and optimizer is so close that these two tools are usually integrated, indicated in Figure 3.1 by the dotted box. If the code is to be optimized with respect to performance, then instruction timing is vital. If energy dissipation within the processor is an issue, then the processor energy model provides the information on how much energy is required to execute a certain instruction.

Within an architecture aware compiler and optimizer, memory timing also has to be considered. If the code is optimized for performance, then the access times to different memories are taken into account e.g. in the decision to either

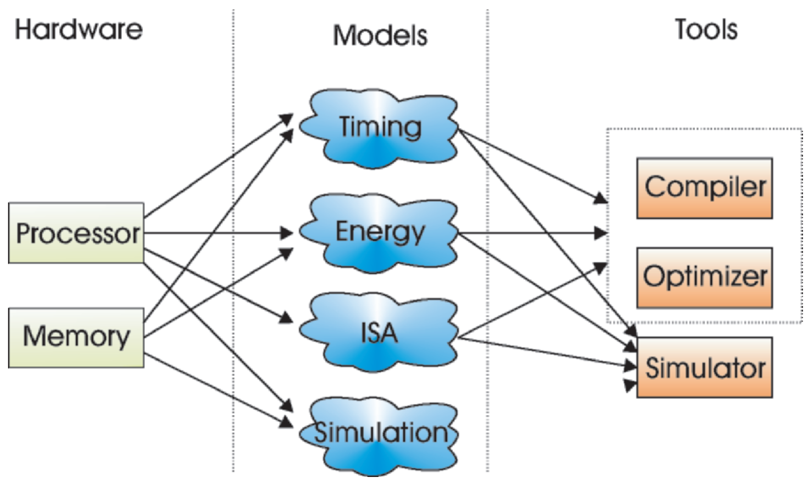


Fig. 3.1. Required models for memory architecture aware optimization

temporarily store a value in memory or to regenerate the result. Timing information may also include the ability of a memory to support burst transfers which make sequential accesses faster. Information concerning several memories with different properties and how they can be exploited will generally have an influence on optimization decisions and on the quality of the generated code. Energy dissipation information concerning the used memories can also be used by the compiler and optimizer to improve the quality of the generated code. Instructions and data elements may be distributed among several memory partitions depending on their energy behavior. Availability of power management functions allow the memories to be shut off memories whenever they are not used for a longer period of time. All this information is required in order to generate optimal code with respect to a used memory hierarchy.

After code generation and optimization, the resulting code should be evaluated. Despite the fact that the used cost function allows a first assessment of the expected improvements during code generation and optimization, not all factors may have been considered, since parts of the required information (e.g. cache behavior) is only available at run time. In order to validate the optimization and to actually quantify the benefit, simulating the entire system with the optimized code is helpful. This simulation run is accompanied by an evaluation step that analyzes in detail how the system behaves during execution of the program. In general, the behavior of the processor is encapsulated within an instruction set simulator. At least one homogeneous memory is supported by most simulators. More complex memory hierarchies may be modeled, but in order to gain maximum freedom in specifying the memory subsystem, a separate memory simulator may have to be used. For single processor systems with a single issue pipeline which are considered in this work, the times required for instruction execution on the processor and the times required to access instructions and data from memory can be appropriately combined in order to determine overall performance and energy consumption.

In the following sections, the models that make up the optimization and simulation environment will be discussed in detail. We start with the instruction set model of the used processor and the behavior of the considered memories, then proceed to the timing and energy models of both processor and memories. Finally, the simulation models provide the required information for the evaluation of the optimization results.

3.1 Instruction Set Architecture Model

The memory hierarchy aware compilation techniques in this work consider the ARM7TDMI [ARM01], which implements the ARM instruction set architecture version 4T. The ARM7 is widely used in embedded systems like portable audio players, wireless headsets and digital cameras. It is recommended in particular for ultra low energy applications, making it a suitable target processor for the proposed memory aware compilation optimizations.

The integer unit of the ARM7TDMI processor features 16 general purpose registers with 32 bits each as well as an ALU, a hardware multiplier and a dedicated barrel shifter. The three-stage instruction pipeline consists of the fetch, decode and execute stage. The processor is connected to its environment by 32 bit data and address buses.

Apart from the integer core, our ATMEL implementation of the ARM7 processor, the AT91M40400 microcontroller, also carries a 4 kB freely addressable so-called scratchpad memory implemented directly on the processor die. No caches are provided in this processor model.

In order to provide a trade off between reduced energy consumption and performance, the ARM7TDMI features two instruction sets: first, the full 32 bit "ARM" instruction set and second, the 16 bit "THUMB" instruction set. The THUMB instruction set offers less functionality compared to the 32 bit ARM instruction set. In particular, the number of directly addressable registers is reduced to only 8 (compared to 16 in ARM mode), since only 3 bits are available to code the register number in the 16 bit instruction word. The range of immediate values within an instruction is also restricted, making e.g. long conditional branches and direct loading of large constants more complex than in the ARM instruction set. In the 32 bit ARM instruction set mode, the barrel shifter can be used in parallel with the ALU, whereas in THUMB mode, shifting always requires a dedicated instruction.

Beside the reduction in number of usable registers, the strongest restriction is the fact that conditional execution of instructions is not possible: in ARM code, it is possible to execute or skip an instruction depending on the value of the processor status bits. In THUMB mode, explicit conditional branches are required, leading to longer code with complex control flow and also more potential pipeline stalls at runtime.

The advantage of using THUMB mode is the increased code density: according to [SCG95, KG02], an average increase in the range of 30% is achievable compared to ARM code, promising a better utilization of the available instruction memory. THUMB mode is also recommended for ultra low energy applications, since for each instruction, only 16 bits have to be fetched from memory. Since instruction fetches account for a large percentage of overall energy, the reduced bit width generally results in energy savings. Despite the reduced capabilities of the THUMB instruction set, execution times do not increase significantly compared to ARM mode. For some timing-critical sections of the code, ARM code may be preferred in order to meet deadlines. Switching between ARM and THUMB modes is done by executing a special branch instruction, so that 32 bit and 16 bit code can be freely mixed to benefit from both instruction sets' advantages. There have been several studies that consider using both instruction sets at varying granularity [KG02].

In summary, THUMB code is considered the first choice for energy aware embedded systems. Apart from the situation where features only supported by the ARM instruction set can be used effectively, twice the amount of instructions can be allocated to a memory of a certain size. This is of particular interest considering the fact that our implementation of the ARM7TDMI on the evaluation board provides an onchip scratchpad memory of only 4 kB capacity. Using THUMB mode, the restricted capacity of this highly energy and performance efficient memory can be used for holding instructions in a beneficial way.

In order to generate executable programs for the evaluation board, the ARM software development toolkit can be used. It contains a compiler for both ARM and THUMB mode, assembler and linker, provides communication software to transfer the generated program to the board and also supports the Angel debug monitor to allow remote debugging and tracing of the program on the processor. The toolkit also contains an instruction set simulator called ARMulator which can be used to easily validate the correctness of a generated program without having to execute it on the actual hardware. In our workflow, the compilers provided with the software development toolkit were mostly used to validate the correctness of our own *encc* compiler which will be described in section 3.6.

The instruction timing of the ARM7 processor, derived from the instruction set description in a straightforward way, will be described in detail in Section 3.3.1. The employed energy model was derived from the actual hardware using physical measurement [SKWM01] and will be summarized in Section 3.4.2.

3.2 Memory Models

Memories have in the past couple of years undergone a significant development that is comparable to that of processors: They are becoming smaller, yet have more capacity and shorter access times. Still, memories do not grow in size and speed quite as fast as processors do. The resulting speed gap has become an increasing concern to system designers, since the development has reached a point where the memory can severely limit the overall performance and dominate system power. It is therefore interesting to find a subtle balance between memories and processors such that e.g. the memory does not impair the performance of the entire system. To perform these studies, it is necessary to consider current processor as well as memory technologies. The processor model used in this work was described in the previous section. The memories used in embedded systems today can in general be split into three categories:

- Static Random Access Memory (SRAM)
- Dynamic Random Access Memory (DRAM)
- Flash Memory

Read-only memories like ROMs, EPROMs and EEPROMs were previously used in embedded systems to hold information that does not change over time. However, they are increasingly being replaced with Flash memories since updating a ROM requires additional efforts and architectural modifications (e.g. the use of UV lamps or additional voltage levels for erasing). This makes them less suitable for use in embedded systems where it is desirable to have an easy way of performing e.g. firmware updates. If Flash memories are used, this task can even be left to the user who simply needs to insert a CD-ROM containing the new firmware.

The following sections present an overview over these memory types along with a description of their functional behavior, special performance and power enhancing features as well as their integration into an embedded system. The timing behavior and energy dissipation of the different memories will then be covered in sections 3.3.2 and 3.4.3, respectively.

3.2.1 SRAM

The basic storage principle of a single static memory cell can be compared to that of a flipflop. Each memory cell, capable of storing a single bit, is made up of 6 transistors (cf. Figure 3.2). If the cell is accessed using the address lines, then the stored value (denoted as “0” and “1”, respectively in the figure) can be read from the opposite “*Data*” and “$\overline{Data}$” lines. If the data lines are forced to opposite values from the memory circuitry, this new value will be stored in the 6-transistor cell.

Despite the fact that SRAM memories occupy a relatively large area per bit, they are frequently implemented directly on the processor die as so-called onchip memories since the same production processes used for the processor transistors can also be used for the SRAM cells. This keeps the manufacturing process simple and helps reduce the cost. Typical onchip SRAM memories are fast: in most systems, it is possible to access such an onchip memory within

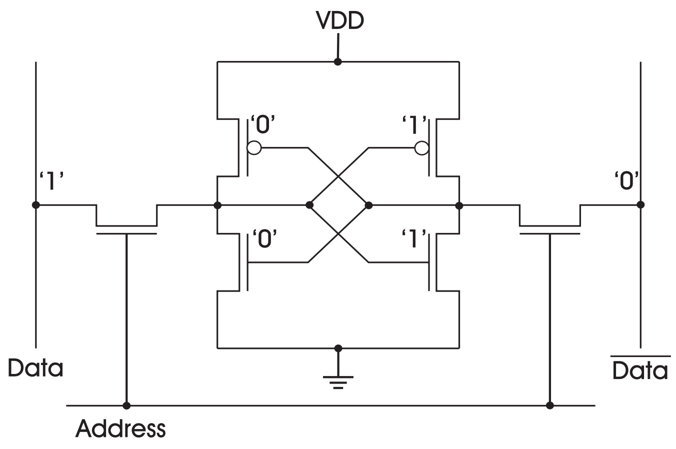


Fig. 3.2. Basic 6-transistor SRAM cell [BÖ2]

one processor cycle, a fact that is also owed to the fact that the memories need to be kept small since area is a scarce resource.

The asynchronous protocol used to access an SRAM memory is fairly simple: the memory's inputs comprise the address bus, chip select, output enable and write enable and the bidirectional data bus.

For a read access to an SRAM, chip enable and output enable both need to be active, i.e. '0', whereas write enable is '1'. After the address is applied and the specified read time has elapsed, the data will be available on the data output pins.

For a write access, write enable is set to '0' and the data and the address are both put on the corresponding bus. After the specified write cycle time, the data is latched into the memory on a positive edge of either write or chip enable.

Operated in the standard asynchronous read/write mode, the timing and energy behavior of a single SRAM memory can be described using a relatively simple model: for each access, the same amount of time as well as energy can be assumed. The generation of timing and energy models for SRAM will be covered in more detail in sections 3.3.2 and 3.4.3, respectively.

On the evaluation board used in this work, SRAM memories are used both as main memory and in the 4 kB scratchpad memory located directly on the processor chip. Since in contrast to caches, which will be described in Section 3.2.4, the scratchpad does not come with a hardware control logic, the utilization of this small, energy efficient memory is left to the programmer or the compiler. In industry, the use of scratchpad memories is now becoming more common (cf. the "Tightly Coupled Memory" in the newer ARM9 designs [ARM]). Yet, as of today, a complete toolchain to systematically exploit the advantages of this architectural feature is largely missing [MWV$^+$04]. This is illustrated by the fact that the software delivered with the evaluation board is not capable of providing adequate support for utilizing the scratchpad.

The latest development in current SRAM technology goes toward synchronous SRAMs (SSRAM), which have an increased throughput due to the use of internal pipelining. Despite the fact that onchip SRAM cells are usually capable of reading or writing one data item per processor cycle, SSRAM memory cells also offer a burst mode where the address is automatically incremented, thus relieving the address bus of the necessary address transfers [BÖ2]. The drawback of SRAM's large area requirements is currently being tackled by completely new fabrication techniques using thin film transistors, with an expected reduction of SRAM memory areas by up to 50%.

3.2.2 DRAM

Dynamic (DRAM) memory cells are architecturally simpler than static cells, since they are made up of only one single switching transistor and a storage capacitor per stored bit (cf. Figure 3.3). They thus require only about one sixth of the area per bit compared to an SRAM. In addition, DRAM offers the capability to store a lot of information at a cheap price, making it the first choice for large main memories where high access speed is not as crucial as for the higher levels of the memory hierarchy.

On the downside, the protocol used to access DRAM is more complex than the corresponding SRAM counterpart. Since the small capacitors used to store the bits lose their information after a short period of time, DRAMs require a more sophisticated surrounding logic to refresh the contents of the memory periodically.

To enhance DRAM performance, synchronous DRAMs (SDRAMs) are being used in current architectures. DRAM memories are already described by memory cell vendors (e.g. Micron Technology Inc.) as being obsolete. Therefore, only the currently used SDRAM will be considered in the following. SDRAMs use a synchronous protocol by latching control and address bit values in registers, then reading the actual memory array in the background. After a certain number of clock cycles, the information is made available to

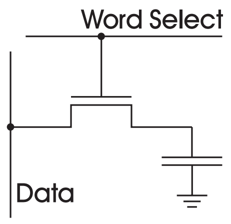


Fig. 3.3. Basic 1-transistor DRAM cell

the processor on the data bus. Additionally, pipelining is performed within the memory to improve throughput. By organizing DRAM in several banks and ensuring an interleaved access pattern, some delays stemming from refresh periods or addressing overhead can be hidden.

Another performance improvement can be achieved by using so-called double data rate SDRAM (DDR-SDRAM). These cells can achieve twice the throughput of regular SDRAMs by supplying data on both the rising and the falling clock edge. Since the internal memory cells operate at the same speed as conventional SDRAM cells, this is achieved by always accessing two cells in parallel, then buffering the results and providing results in an interleaved fashion. Furthermore, more complex synchronization logic is required in order to guarantee correct timing of DDR-SDRAM cells.

To achieve generality, we will describe the typical behavior of an SDRAM without considering the double data rate feature. The input and output signals of such an SDRAM component are shown in the block diagram in Figure 3.4.

As an illustration for the possible states and transitions during SDRAM operation, please refer to Figure 3.5.

The Clock enable signal (CKE) controls whether the clock signal (CLK) is propagated to the memory. If CKE is low, then the memory goes to the power-down state, shutting off parts of the circuitry to provide a low standby power consumption. In Figure 3.5, this corresponds to the PDN state. If CKE is high, then the clock input (CLK) can be used to control the timing of the memory, since all inputs are latched into the memory on a positive edge of the clock signal. Chip select (CS) must be active to enable the command decoder. Write Enable (WE), Column Address Strobe (CAS) and Row Address Strobe (RAS)

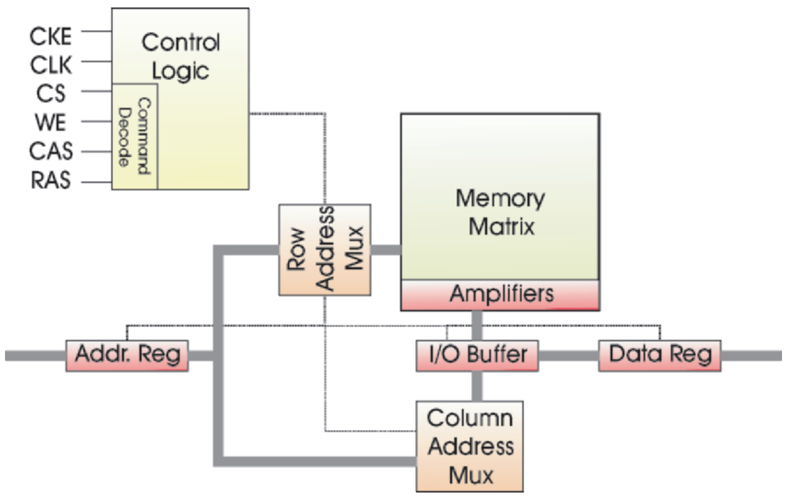


Fig. 3.4. Simplified DRAM block diagram

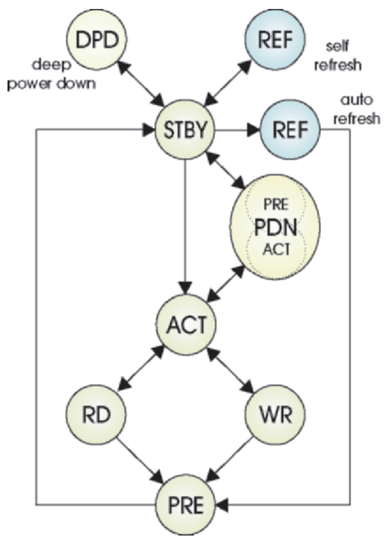


Fig. 3.5. Simplified state machine for SDRAM

which go directly into the command decoder of the control logic determine which action is to be taken by the memory. To start an access from the standby state (STBY), the memory first requires an Activate command, issued by setting RAS to high and CAS to low. The address bits applied at this time are latched as the row address, and the memory goes to the active state (ACT). One entire row of data is read from the memory array, amplified and stored in the I/O buffer. On the next clock edge, the column is read from the address inputs when RAS is low and CAS is high. With the Write Enable signal set to low, this corresponds to a Read command (state RD). In this case, the requested data element (column) is forwarded to the data register. Reading the line from the memory array destroys the values stored in the memory itself, since the capacitors are discharged in order to drive the bit lines. Therefore, the whole line has to be written back from the I/O buffer to the memory by issuing a precharge command (state PRE). A similar procedure is used for writing data (state WR): The line of data containing the word to be written is read, then the modified value is read from the data register and stored in the corresponding positions of the buffer. Finally, the entire line is written back to the memory array. This write-back process is also known as “precharge”, since it also precharges all bit lines within the memory array to ‘1’ in order to allow faster access for subsequent read operations.

Even when a certain line of data is not accessed, the capacitors within the DRAM cell lose their information within a short time. They therefore have to be rewritten periodically, which essentially means reading the entire memory to the buffer and subsequently writing back the read values.

The fact that a complete line is always addressed and put in the output buffer can be exploited to increase throughput of DRAM memories. The time required for a subsequent access to a data element within the same line is reduced since there is no need to address and read the same row again. Rather, only the column address needs to be changed, and the value can be read directly from the buffer. Current SDRAMs assume that more than one data element is to be read from a particular row. They therefore internally increment the column address to deliver values contained in the accessed row successively. Such fast accesses that do not require any re-addressing are generally known as "burst mode" accesses. Their timing is described e.g. as 5-1-1-1, meaning that the initial memory access (including row and column addressing) requires 5 cycles, but the four subsequent sequential reads only require one cycle each.

The performance enhancing burst feature makes SDRAM cells attractive in particular for instruction memories which show a high percentage of sequential memory accesses. At the same time, burst transfers make it more difficult to accurately model DRAM with respect to timing, since the time required for a single access may depend on the accesses in the past.

To account for this complex timing behavior during code generation, a couple of measures have to be taken in order to represent the memory's behavior in a timing model. The methods used in this work to model DRAM memories will be presented in section 3.3.2.

3.2.3 Flash Memory

In contrast to static and dynamic RAMs, Flash RAM cells are non-volatile memories which means they keep their information even when the supply voltage is switched off. For this reason, Flash memories are frequently used in embedded systems to permanently store e.g. the boot loader, operating system and applications that do not change frequently.

For read accesses, Flash memories require a low amount of energy per access, and their standby energy dissipation is negligible. This would make them an ideal read-only memory for embedded systems. However, Flash memories are a lot slower than SRAM and even DRAM, which can impair overall system performance. To overcome this performance penalty, the applications stored in an embedded system's Flash memory are usually first copied to SRAM or DRAM memory at startup of the device and then executed from the faster memory ("Store-And-Download" architecture). Either the complete application is loaded into working memory at startup (fully shadowed code) or accessed pages are loaded at runtime (demand paged code).

Flash memories can not be used as the only memory component due to their limited ability to execute write operations: only 10^5 to 10^6 store or erase operations are possible on current Flash memory technology before the chip becomes unusable [HP03]. Also, all write operations to a Flash memory have to be performed in a blockwise fashion, which consumes a considerable amount of energy. Concerning their organization, there are two different kinds of Flash memories: NOR and NAND Flash. They differ in the way the cells are organized internally, in the access speed and the granularity of write operations.

NAND flash is cheaper compared to NOR flash, and reaches higher densities due to its serial internal organization which requires less contact pins. Concerning write accesses, NAND flash memories offer smaller write and erase blocks than those built in NOR technology. Due to the strict organization in pages and the necessity to re-read the entire page if one element is accessed in a non-contiguous way, NAND flash memories are not suitable as random access memory in embedded systems. In file system based devices, however, sequential read operations are common, which is why NAND flash memories are mainly being used in such devices, e.g. in SmartMedia Cards or USB memory sticks. NAND Flash memories require a certain amount of error correction including ECC algorithms and bad block marking because of the frequently occurring read and write errors.

NOR flash, on the other hand, offers an acceptable random read access behavior. Due to its parallel structure, data within one page can be accessed in acceptable time using so-called intra-page accesses. The write behavior, however, is poor: only very large blocks of data can be erased and rewritten at a time (e.g. 128 kB for a Micron Q-Flash MT28F640J3 [Mic04b]). This, of course, does not impair the use of NOR flash as a read-only memory in an embedded system, making NOR Flash cells attractive for use as "execute in place" (XIP) memories. Code that is executed "in place" is fetched directly from the Flash memory into the processor. This avoids the overhead of copying less frequently used parts of the program and reduces the requirements concerning main memory capacity since parts of the application code can remain in the Flash memory, even at runtime. This approach and its application within a compiler will be discussed in-depth in chapter 5.3.

The communication protocol used by the mentioned Micron Flash memory is a simple asynchronous protocol similar to the one used by SRAM cells that only requires the processor to apply the requested address and subsequently enable the output pins of the memory. The Flash memory then provides information from its internal data array on the output pins if "write enable" is not activate. Since the block-oriented writing to Flash memories is not required in the scope of this work, this issue will not be discussed.

3.2.4 Caches

Caches are common in today's processors and computing systems. Despite the fact that a cache is not considered a memory technology, a description of the general behavior and the parameters is provided at this point since caches are commonly used in many computing systems today. In particular in Section 4.3 of this work, the influence of caches and scratchpad memories concerning the predictability of a system is compared.

The word cache comes from the French “cacher”, to hide. One aspect of caches is thus that they are in general hidden both from the user and also from the compiler: if a cache is present in the system, it will be used automatically without requiring any change in the workflow. Of course, optimizations that improve the performance of a cache-based system are still possible, but compared e.g. to a scratchpad memory, the presence of a cache can also be exploited without explicit support.

Caches are small memory arrays that hold frequently accessed data and/or instructions. Due to their small size, accesses can be performed in a fast and energy efficient way. However, to ensure that the current working set is always available in the cache in an automatic way, additional circuitry is required: for every read access, the cache first determines whether the requested information is currently in the cache memory. If it is, then it is forwarded to the processor (“cache hit”). If the element is not in the cache (“cache miss”), it is fetched from the next lower level of the memory hierarchy. To determine a hit or a miss, the cache uses the stored tag bits of the data elements’ addresses (cf. Figure 3.6):

The accessed address is split into tag, index and offset bits, the sizes of which depend on the parameters of the cache. The index bits are used to address the cache’s tag memory and to read the corresponding tag bits. The stored tag is then compared to the tag part of the original access address. If they are identical, then the requested element is stored in the data array of the cache, resulting in a cache hit. In this case, the offset bits are used to determine the requested word within the cache line. The valid bits that mark a cache line as being up-to-date are not shown in Figure 3.6 for simplicity.

If two data elements have the same index bits, then they will occupy the same cache line and thus cause each other to be evicted from the cache if a

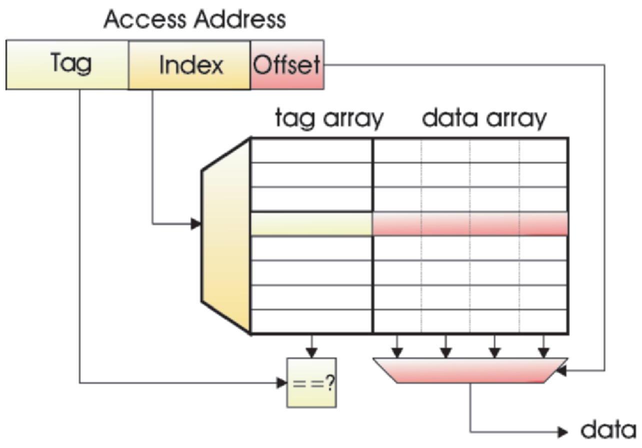


Fig. 3.6. Example cache architecture using a direct mapping

direct mapping as shown in Figure 3.6 is used. If this happens frequently (an effect known as "cache thrashing"), the performance of the cache will degrade. To overcome this problem a set associative mapping is used in many caches. The tag and data arrays are present two or more times (depending on the associativity), and the tag bits of all the sets are compared in parallel. If one set contains a matching tag, then the requested element is stored in the corresponding data array. If all comparisons fail, then the line needs to be fetched into the cache. In this case, a replacement strategy is required to determine from which set the cache line is to be evicted. Commonly used replacement strategies include "random" or "least-recently-used" (LRU), the latter being more efficient and providing better predictability, but also requiring more control logic.

The ARM7TDMI processor used on our evaluation board does not provide caches. Therefore, all experiments concerning caches in the ARM architecture were performed using the similar ARM710T architecture. Since the ARM710T was not physically available, the cache behavior was investigated using cache simulation. For this reason, it was possible to study a wide range of varying cache configurations.

To conclude this section, a short overview over possible design parameters of caches is given:

- Split or unified caches: In a split cache architecture, instructions and data do not share the same cache. Split caches in general show superior performance compared to their unified counterparts since accesses to both instructions and data usually show a high degree of locality. However, if sequential instruction fetches are interrupted by data accesses, this potential locality is lost and data and instructions may evict each other. On the other hand, including separate caches for instructions and data leads to a higher amount of required space, thus increasing the cost of the device.
- Number of cache levels: In common desktop processors, lacking the strict space and cost constraints of embedded systems, several levels of caching are now common, with the first level cache (L1-cache) usually situated on the processor chip. The caches are connected with the neighboring cache levels, meaning that an element not present in a certain cache level is accessed in the following level and so forth. Such multi-level cache hierarchies are thus generally considered to be fully inclusive, i.e. any data contained in one cache is also contained in all lower level caches. Multi-level caching strategies require a careful sizing of the individual cache levels to treat the cost-performance tradeoff adequately.
- Cache Size: The capacity of a cache is indicated in terms of number of bytes of data that it can accommodate, regardless of the fact that additional space is required for the tag memory. The larger the cache, the better its potential performance (since it can hold more data), but the higher the area and energy overhead.

- Associativity: Using a direct mapping scheme bears a high risk of cache thrashing. Therefore, direct mapped caches are usually only used as instruction caches since instructions accesses are in general more sequential than data accesses, which avoid excessive thrashing. For other purposes, at least two-way set associative caches are usually used, unless the given space constraints prevent the inclusion of the required additional comparison circuitry. Since the comparison operation of several sets are performed in parallel, the influence on performance can be expected to be minimal, but the energy will increase with the chosen associativity due to the increased switching activity.
- Write Strategy (Write Through vs. Write Back): When memory words are being written in the presence of a cache, two strategies are possible: either, the word is only updated in the cache and a "modified" flag is set. Since subsequent reads will results in a cache hit, the updated value will be read from the cache without accessing memory. If a modified cache line is replaced, then it needs to be written back to memory before being evicted ("write back"). On the other hand, it is also possible to write the modified data element immediately both in the cache and in the memory ("write through"). This may result in an increased number of write operations on the main memory, however, it saves the modified-bit in the cache lines. In order to increase performance, write buffers are usually introduced such that the memory write can be performed when the bus is available, without unnecessarily having to stall the processor.
- Line Allocation on Write Miss: If a data word is written and the access results in a cache miss, then the line containing the missed word can either be loaded into the cache ("allocate on write miss"), or the value can simply be written to memory ("no allocate"). The allocation of a line on write miss can make sense assuming that the written value, due to access locality, will soon be accessed for reading again.
- Cache Line Fill Strategy (Critical Word first, complete vs. incomplete line): On cache miss, the line that caused the miss is brought into the cache. Usually, the accessed word is read first so that the memory latency experienced by the processor is minimal. In instruction caches, streaming can help keep the miss penalty low: instructions that are being read to fill the cache line are also directly forwarded to the CPU. Once the end of the line is reached, the data before the element that initially caused the miss is usually fetched to fill the complete line and set the valid bit of that line to true. If, instead of only having one valid bit per cache line, one bit per word is provided, then there is no need to fetch all words of the line if they are not accessed.

Controlling all these parameters to design a cache such that it optimally fits a certain device and application is a difficult task, in particular because all parameters are connected and have a certain impact on each other. It should also be noted that the considered optimization goal is of importance: authors have shown in [SC99] that performance and energy optimization of a cache results in a different cache organization. The compiler can be used to extract some basic information concerning a beneficial cache organization.

3.3 Timing Models

Knowledge about the timing of considered components is of substantial importance in the course of architecture aware compilation. Timing models for the processor and the memory are required in order to understand and simulate the components' behavior. For code generation, the timing of instructions needs to be known in order to choose those instructions that lead to minimum execution times of the final application. Performance in terms of number of cycles is still the prime optimization goal pursued by the majority of designers and compilers today.

But even if reduced energy dissipation is the goal of the optimization, it is vital to be aware of the influence that execution time has on energy, since energy depends on both power and time (cf. Section 3.4.1).

During simulation and evaluation, the timing of the processor and the memories has to be known so that the simulated behavior matches that of the actual hardware. If wrong assumptions about the timing behavior of the underlying components are made, then the performance and energy evaluation results will certainly differ from the results obtained on the physical system.

3.3.1 Processor and Instruction Timing

Of the two subtasks of describing processor and memory timing, the processor is the easier one. Information about the timing of the processor is usually provided by the vendor. While timing is unambiguous for most instructions, there may well be instructions with operand dependent cycle counts (e.g. multiplication operations, depending on the hardware support provided).

The ARM7 architecture provides a three-stage pipeline consisting of the stages fetch, decode and execute. A single instruction therefore requires at least three cycles to pass through the pipeline. Assuming a filled pipeline and a standard ALU operation, one instruction can be executed per cycle. A complete overview over the instruction cycle timing of the ARM7TDMI processor can be obtained from the reference manual [ARM01]. Table 3.1 summarizes the timing of relevant instructions.

The number of cycles per instruction are not given in absolute processor cycles, but depend on the used memory. This is on one hand due to the fact that the ARM processors' pipeline is stalled whenever the CPU is waiting for a memory access, both data and instruction fetches. On the other hand, since the considered ARM7 does not use caches, it is directly dependent on the timing of the memory. For an ALU operation to be performed, the instruction timing is thus assumed to be the number of cycles it takes to fetch the subsequent

Instruction	Cycle Count
ALU op	S
Load	S+N+1
Store	2N
Load Multiple	nS+N+1
Store Multiple	(n-1)S+2N
Branch	2S+N
Multiplication	S+mI

Table 3.1. Instruction cycle timing summary for ARM7TDMI

instruction. This is denoted in table 3.1 as “S” cycles, denoting one sequential memory access. In case of a LOAD operation, the data element also has to be fetched from data memory. Since this may be an access to a different region of the memory, it is assumed to be a non-sequential access, “N”. The LOAD operation of the ARM always requires one additional processor cycle in order to determine the access address. The STORE operation is assumed to take two non-sequential accesses since a write access among read accesses is always assumed to be non-sequential, thus the following instruction fetch has to be considered as an “N” access. This reasoning can be extended to the LOAD-Multiple and STORE-Multiple instructions, which are assumed to access n different data items.

When a BRANCH instruction has reached the execute stage of the pipeline, it determines whether the branch is taken, and it will always fetch the instruction at the current PC from memory using a sequential access (cf. instruction “Inst_M” in Figure 3.7). The fetch can not be prevented, despite the fact that this instruction may not be executed. If the branch is taken, the next instruction is then fetched from the branch target address (“Inst_A”) in a non-sequential access. Since the pipeline is not yet filled with valid instructions, the instruction following the branch target is fetched using a sequential access. The branch target instruction then proceeds to the execute stage of the pipeline, finishing the execution of the branch instruction after “2S+N” cycles. This is pictured in Figure 3.7.

If the branch is not taken, then it behaves just like an ALU operation, meaning it requires only one “S” cycle to execute. In the ARM documentations, this information can be found in the description of the 32 bit ARM instruction set’s conditional execution [ARM01] as an instruction that is not taken due to the processor status bits.

The “Branch Link” instruction represents an exception in the THUMB instruction set: while all instructions have a length of 16 bits, a “BL” is encoded in two subsequent 16 bit values, with the second halfword containing the remaining bits of the target address that do not fit into the immediate fields of the first 16 bit instruction. This encoding is necessary to allow function calls across large distances within the memory address space: distances of up to 4 MB are possible using “BL”, compared to only 256 bytes for conditional

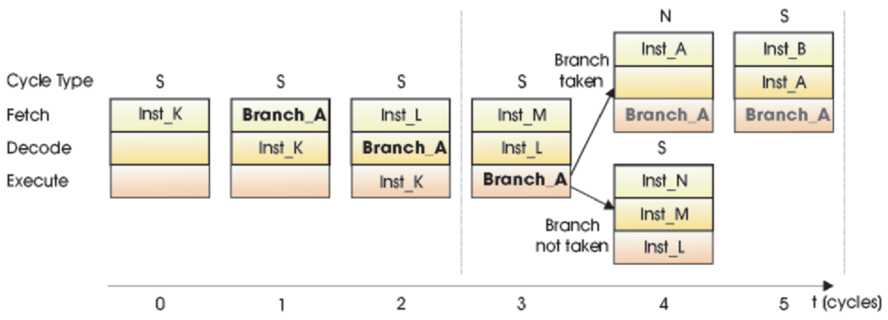


Fig. 3.7. Pipeline stall due to a branch for the ARM processor

and 2 kB for unconditional jumps. The timing of the “BL” instruction is similar to that of an unconditional jump: the first 16 bit instruction word is read and executed in a single cycle, while the second halfword has the same timing as any branch instruction.

The timing of the multiplication operation depends on the operand values: depending on the number of bits in the second operand that are set to zero, it takes more or less cycles, which is represented by the parameter “m”. The letter “I” represents a “memory idle” cycle, meaning that there is no activity on the memory bus during the calculation of the multiplication result. Since the operand values of a multiplication can generally not be determined within the compiler, an average-case execution time of the multiplication operation is assumed. Using a bit level data flow analysis as presented e.g. in [WL02b], some information concerning the values of operands could be gathered, however this complex analysis is only common for processors that support bit level operations.

In general, the instruction cycle counts were also validated through physical simulation and analysis using an oscilloscope. These validated results were then transferred to a database used within the compiler to enable the code generation process to be aware of the resulting execution times.

3.3.2 Memory Timing

If a complete system is to be modeled with respect to timing behavior, only considering the processor is not sufficient. The memory subsystem plays an important role for the overall timing of a device, in fact memory performance can be the bottleneck of a system. This becomes manifest in the integration of ever larger first- and second level caches into e.g. Intel’s and AMD’s processor designs. Since accesses to the large, slow main memory can easily consume the equivalent of several hundred instruction executions, caches are introduced in such a size that they are able to hold the current working set of the executed applications most of the time. Pessimistic observers fear that even the time required to initially fill the caches may become the limiting factor in memory performance [WM95].

With memory timing being a vital part of the behavior of a system, there is a need to model memory timing, and in particular to capture the used memories' timing in the workflow.

The main properties of three memory technologies (SRAM, DRAM and Flash) were already discussed in Section 3.2. In this section, the timing behavior of the different types of memory will be considered in-depth. Timing information for a certain memory can usually be acquired from data sheets provided by the memory cell vendors. Measurements are another possibility to obtain information on the memory timing of a system that is physically available to perform the required measurements.

SRAM Timing Model

For static RAM cells, the available data sheets can provide information concerning the used memory timing. Since regular SRAM memories like those used on our ARM7 evaluation board do not support any kind of burst mode, sequential and non-sequential accesses all take the same time to complete. This significantly simplifies the memory timing model, since no state-dependent information has to be considered. Data sheets usually provide the memory access times in nanoseconds. By transformation to a multiple of the processor's clock period, each type of access using SRAM memory takes a certain number of clock cycles. The scratchpad memory embedded directly on the processor die usually takes only one clock cycle per access, meaning that the processor does not have to insert any wait cycles to wait for data or instructions.

On our ARM evaluation board, SRAM cells are also being used as main memory. Since the effects of longer bus lines and larger capacitances have to be accounted for, the resulting overall memory timing was measured instead of solely relying on processor and memory data sheets. For the setup on the evaluation board, memory timing depends on the width of the memory access, since two memory chips each featuring an 8 bit bus are connected to the processor in parallel. This allows 16 bits to be transferred per access. If a 32 bit value is to be read or written, then two sequential accesses to both memories are required. The two 16 bit values are then assembled to form one 32 bit value which can finally be stored to a processor register (cf. figure 3.8).

This memory setup results in a 32 bit memory access requiring three wait-states in addition to the actual memory access cycles. 8 or 16 bit accesses only require one additional wait cycle. These additional wait cycles required within the processor are summarized in table 3.2. Since the actual access cycle is usually hidden by the pipelining within the processor, these values are regarded as the additional number of cycles required to access main memory. In the course of our measurements, we validated the wait states using an oscilloscope to monitor the address and data bus lines connecting processor and memory.

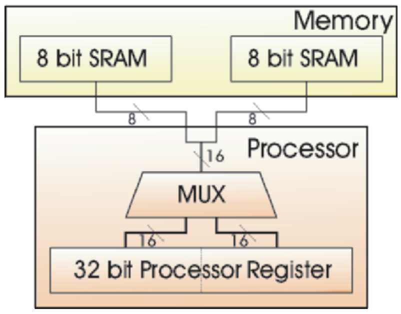


Fig. 3.8. Main SRAM memory setup on the ATMEL evaluation board

Access Width	Scratchpad Memory	Main Memory
1 byte (8 bit)	0 waitstates	1 waitstate
2 bytes (16 bit)	0 waitstates	1 waitstate
4 bytes (32 bit)	0 waitstates	3 waitstates

Table 3.2. Memory timing for the ARM7 SRAM memory

DRAM Timing Model

In order to understand and model the timing of a DRAM cell, some closer understanding of the internal structures and the operating modes (cf. Section 3.2) is required. We will concentrate on the timing behavior of current synchronous DRAM (SDRAM) since the sale of regular DRAM cells has been discontinued by most memory chip vendors.

To describe the timing behavior of SDRAMs, it is essential to be aware of the current state the memory is in, since timing can vary significantly depending on whether an initial random access or a burst access is taking place. Figure 3.9 shows the situation for the case of a random access: the corresponding line within the data array first has to be activated. The required time "active to read/write delay" is designated as t_{RCD} in the data sheet and describes the duration of RAS-to-CAS delay, i.e. the time it takes for the row address to be considered valid. Following the row activation, the column address is put on the address bus. In case of a read access, the internal "CAS-latency" T_{CAS} delay is required to transfer data to the output register. In case of a single random read access, a precharge command is required to terminate the access. A delay of "precharge command period" t_{RP} needs to pass before the memory can once again be activated. Altogether, a single random read access, assuming a 16 bit wide memory, thus takes

$$T_{SDRAM\_RND16\_RD} = \left\lceil \frac{t_{RCD}}{t_{CLK}} \right\rceil + T_{CAS} + T_{DOUT} + \left\lceil \frac{t_{RP}}{t_{CLK}} \right\rceil \qquad (3.1)$$

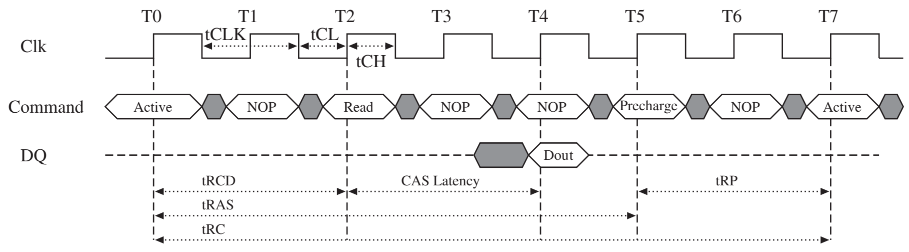


Fig. 3.9. Access timing for a random read access to an SDRAM

if we assume that an additional T_{DOUT} cycles are required to actually drive the output data. t_{CLK}, the duration of one clock cycle is used to scale the absolute time values t_{RCD} and t_{RP} to the used clock. The "ceiling" operators are required to account for the fact that data is only latched on the rising clock edge of the memory. If a low operating frequency is used, the slack time before the next clock edge can be substantial. It is therefore vital to tune the frequency to the timing characteristics of the used memory in order to achieve maximum performance. T_{CAS} and T_{DOUT} are specified in multiples of the cycle time in the data sheet.

Assuming a write access, the "write recovery time" t_{WR} which describes the duration from valid input data to the next possible precharge command has to be added to the equation, whereas the time to transfer the data element to the output register T_{CAS} and to drive the output data $T_{DOUT}$is omitted:

$$T_{SDRAM\_RND16\_WR} = \left\lceil \frac{t_{RCD}}{t_{CLK}} \right\rceil + \left\lceil \frac{t_{WR}}{t_{CLK}} \right\rceil + \left\lceil \frac{t_{RP}}{t_{CLK}} \right\rceil \tag{3.2}$$

In order to initiate a burst read access, the first access is considered to be a random read access. Reading from contiguous addresses following the initial read address is then possible in the faster burst mode. In case of a 16 bit wide memory, this applies in particular for reading 32 bit words: the first access is in the general case considered to be a random access, whereas the second half of the 32 bit word can be assumed to be read in burst mode. The speedup caused by a burst mode access is considerable: for the subsequent memory accesses, only the data output time T_{DOUT} is required. The time to read a single 16 bit value in burst mode is thus

$$T_{SDRAM\_SEQ16\_RD} = T_{DOUT} \tag{3.3}$$

and the time required to read a 32 bit value from a random position in the memory amounts to

$$T_{SDRAM\_RND32\_RD} = T_{SDRAM\_RND16\_RD} + T_{SDRAM\_SEQ16\_RD} \tag{3.4}$$

If T_{DOUT} is one cycle (as is frequently the case), this means that after the initial access, one word per cycle can be read from the memory. Write bursts can also be performed by passing one new data value to the data port of the memory every T_{DIN} cycles:

$$T_{SDRAM\_SEQ16\_WR} = T_{DIN} \tag{3.5}$$

In the SDRAM configuration register, the maximum burst length defines how many accesses can be performed in the burst mode. The maximum value is limited by the organization of the SDRAM: no burst access is possible across rows, since the row activation has to be performed if a new row is accessed. Shorter bursts than the value defined in the configuration register are possible by using the "burst terminate" command. New SDRAM cells even allow a burst to be interrupted, freezing the current state of the memory, and to continue the burst access at a later point in time.

This concludes the timing behavior of a typical SDRAM memory. For double data rate SDRAMs, the speedup of being able to deliver data on both the rising and the falling edge of the clock signal in a burst access has to be accounted for. Apart from that, the timing is similar to that of SDRAM.

In a compiler, not all aspects of SDRAM timing can be considered. For example, it is not known during code generation which addresses will actually be contiguous in the final executable. Therefore, the compiler has to assume approximate values e.g. for the relative number of possible burst accesses. The precision within the compiler can be improved by performing one simulation of the application before considering memory optimization and keeping track of the different access types. The collected information can then be used in the optimization process to allow more precise energy models to be used. Details of this approach will be presented in Section 3.4.3.

Flash Memory Timing Model

There are three different modes of accessing Flash memories for reading:

- asynchronous random access
- intrapage access
- burst access (in synchronous mode)

Random read accesses are the standard way of accessing a Flash, and any first access to a certain page of the Flash memory is always a random access. It requires the "address to output delay" t_{AA} to complete. Assuming a memory bit width of 16 bits, the time required for one read access is given as

$$T_{FLASH\_RND16\_RD} = \left\lceil \frac{t_{AA}}{t_{CLK}} \right\rceil + T_{DOUT} \tag{3.6}$$

again assuming T_{DOUT} cycles to drive the output values.

If the next access goes to the same page, then this initial delay is not required, since the page is already open. Rather, only the shorter "page address delay" t_{APA} is required to determine the next address. Since Flash memories use an asynchronous protocol, the access times taken from the data sheet can be directly used to model the timing behavior of the memory. The time for a random 32 bit access to a 16 bit Flash memory is

$$T_{FLASH\_RND32\_RD} = T_{FLASH\_RND16\_RD} + \left(\left\lceil \frac{t_{APA}}{t_{CLK}} \right\rceil + T_{DOUT} \right) \qquad (3.7)$$

where the term in parentheses can be interpreted as $T_{FLASH\_SEQ16\_RD}$.

Writing to Flash memories always requires entire blocks to be erased and written. Since in embedded systems this is usually only done in order to perform e.g. firmware updates and not during the standard operation of the device, write accesses to Flash memory are not considered in this work.

A new development for Flash memories is the use of a synchronous interface, which allows the use of burst mode accesses in a similar way as for the SDRAM memories described above. Synchronous memories will not be considered in this work.

Cache Timing Model

The timing for caches has to be divided into timing for cache hits and misses, since hits are usually serviced much faster than a cache miss, which also includes several accesses to the next lower level of the memory hierarchy.

The time required for a cache hit can be derived by only looking at the used cache model. We have used the CACTI model [WJ94, WJ96] to determine characteristic properties of a certain cache architecture. Beside the overall access time, CACTI generates detailed data on the timing of the individual components within the cache, including the comparators and sense amplifiers. For our cache models, knowing the time for accessing a data item on a cache hit is sufficient. Another simplification is possible in our model since most of the considered caches can be accessed in a single processor clock cycle. In these cases, accessing the cache can therefore be assumed to take one cycle in case of cache hit. In case of a miss, the time taken to service the miss is made up of the time it takes to establish the miss (i.e. one cycle in general) plus the time it takes to fetch the entire cache line from the next level of memory in the hierarchy. This level may be another cache or an SRAM/DRAM/Flash memory whose timing was described above. In this way, the cache timing behavior is sufficiently described.

3.4 Energy Models

Energy is becoming one of the most important optimization factors in embedded system design. It is now common knowledge that it is vital to consider energy dissipation at an early stage in the design process to avoid lengthy redesigns if the requirements are not met. In order to model energy at an early time, energy models are required that cover all parts of the system that are under the control of the designer and that have an impact on energy dissipation.

Beside the pure architectural hardware view on system design, the software side also has to be considered when energy is an issue, since only optimizations in hardware and software together will be able to meet the increasing demands on the energy consumption of embedded devices. Energy models for use within the compiler can help generate energy aware code, and providing the compiler with information concerning the energy consumption of e.g. different memories that are present in the system will allow code generation to be performed in such a way as to minimize the overall energy consumption of the software.

It is an important requirement for a complete energy model of a system to cover all relevant components to avoid the situation where e.g. optimizing for memory energy leads to additional overhead within the processor which might negate any benefit obtained through the initial optimization. Another important aspect of a model is its granularity: if software is to be optimized using an energy model within a compiler, then the model should be able to distinguish between the energy contribution of different instructions, otherwise, instruction selection will not have any impact on the energy when using this model.

It can often be observed that optimizing for high performance, i.e. short execution times, will in general also yield acceptable results concerning energy consumption. However, there are exceptions to this rule when the memory subsystem is also considered. In this case, code may take longer to execute, but still achieve an overall reduction in energy consumption. An example for this kind of behavior is the register pipelining optimization presented in [SSWM01], where energy is decreased by 17% whereas execution time increases by nearly 9%.

In the design of caches, it has also been found that the optimal configuration with respect to performance can be quite different from the energy optimal solution [SC99]. The paper shows that accepting a higher number of cache misses (and thus, reduced performance) can yield a benefit for energy, since a simple, smaller and thus more energy efficient cache can be used.

These examples show the necessity to consider energy as a distinct cost function in addition to performance considerations if energy optimal results are to be obtained.

In the following section, a short introduction to the terms and units used in energy optimization is given, followed by the processor energy model used in this work. Finally, the energy contribution of the memory system is captured using memory energy models.

3.4.1 Sources of Energy Dissipation

In the literature, power and energy are often used synonymously. The terms "low power" and "energy aware" are both being used to describe efforts to enhance the battery lifetime of portable applications, despite the fact that power and energy are not identical.

Electrical power is defined as the product of Voltage V and Current I and is measured in the unit "Watt":

$$P = V \cdot I = V_{dd} \cdot I \tag{3.8}$$

Power is an important factor when considering reliability and the overall lifetime of electronic devices: high currents in the connecting buses can lead to electromigration, where atoms of the connecting metal are torn out by the flowing electrons, in particular when the wiring shows sharp bends. The missing material can eventually lead to failure of the entire device. These effects are of particular interest for systems that are required and expected to have a long lifetime, such as the electronics used in telephone switching systems.

In electronic devices, the operating voltage is usually assumed to be constant. Thus, it is sufficient to measure the flowing current in order to determine how the power changes over time. Optimizing a device's average power dissipation would then mean to avoid high currents. Since the use of slower instructions that incur a lower current in the circuits is encouraged, this can lead to an increase in the running time of the program.

Energy is described by the integral of power over time and is measured in the unit Watt-second or "Joule":

$$E = \int P \; dt = \int V \cdot I \; dt \tag{3.9}$$

Assuming that the measured current does not show a high degree of variation over a certain period of time t (or an average value can be determined for the current during the interval t), and, as above, considering voltage to be constant, the equation can be simplified to

$$E \approx V \cdot I \cdot t \tag{3.10}$$

As a simplification, current and voltage in the used compiler toolchain are modeled as being constant during the execution of one single instruction or one single memory access.

Since practically all computers and embedded devices are built using CMOS technology, we now take a quick look at the reasons for energy dissipation at the transistor level. In a typical CMOS circuit, there are two complementary transistors: one nMOS and one pMOS transistor. The circuit shown in Figure 3.10 inverts the input signal In by switching the nMOS transistor when the input signal is high, thereby connecting the output to ground. In the opposite case, only the pMOS transistor is conductive, thus producing a high logic level on the output. The reasons for energy consumption in this example circuit are threefold: Since the transistors are never perfect insulators, even when they do not switch, there is always a small leakage current that flows

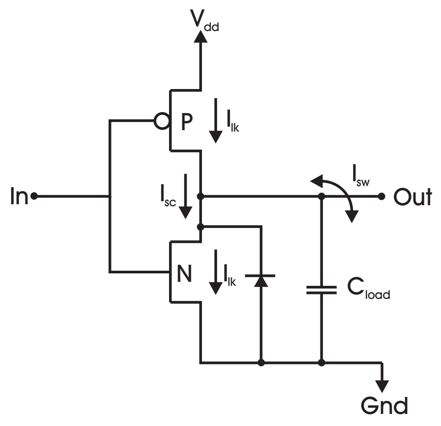


Fig. 3.10. Example CMOS inverter cell (taken from [Syn96])

from the supply voltage V_{dd} to *Gnd*. This current is depicted in Figure 3.10 as I_{lk}. The second current only flows at the moment when the inverter changes its state, i.e. when the input logic level changes. Due to the different switching times of nMOS and pMOS transistors, there is a short time when both transistors are conductive, leading to the short circuit current I_{sc} to flow between the supply voltage V_{dd} and ground *Gnd*. Finally, the capacitance on the output, shown as C_{load} in the figure, has to be driven whenever the inverter output changes its value. This switching current I_{sw} is the largest of the three currents that make up the energy consumption of the CMOS inverter cell, accounting for 70 to 90% of the total energy in active (i.e. switching) circuits. Short circuit power can consume up to 30% of the total energy budget if the circuit is active and if the transition times of the transistors are long [Syn96]. In active circuits, leakage current only contributes up to about 1% to total energy, but it is quite relevant for idle circuits. Considering e.g. a mobile phone's active and standby times, it becomes obvious that the idle energy is of growing importance for a prolonged battery lifetime. In the near future, the importance of considering leakage energy will increase strongly due to the smaller feature sizes of the underlying technology.

Reducing the overall energy dissipation has in the recent past become one of the prime concerns for the design of embedded systems. The amount of energy used by a device has a direct influence on the standby or operating time of portable, battery operated devices like e.g. mobile phones, PDAs or portable MP3 players. The batteries in these devices can only store a limited amount of energy, and since battery technology is unable to keep up with the ever increasing energy requirements caused by faster processors and large memories required to hold increasing amounts of application data, the energy dissipation of the used components must be controlled in a strict way to meet the consumers' demand for long uptimes of battery operated systems.

Due to this high relevance of energy dissipation in the design of embedded systems, the following sections will concentrate on the modeling and measurement of energy dissipation in those components of embedded systems that contribute a high percentage to the overall energy budget: the processor and the used memories.

3.4.2 Processor Energy

The need to model processor energy is a natural consequence of optimizing an embedded system for power: the processor, being the central part of any device should receive a fair amount of attention, since it is responsible for a high percentage of overall energy dissipation of a system. If a compiler is to be used to reduce energy consumption, then it needs information on all relevant parts of the embedded system. Since the compiler generates the processor instructions, it has to be aware of the consequences of choosing a particular sequence of instructions.

Before a description of the processor energy model used in this work is presented, we first give an overview over the possible approaches that were investigated to model processor energy. After that, the energy model for the ARM7 processor is presented.

Related Work

There has been a substantial amount of work on the field of processor energy models.

One concept for a very simple processor energy model assumes a certain average current to flow whenever the processor is busy. In this model, the choice of instruction sequences does not have any impact on the energy consumption. This is unsatisfactory, since it is well known that some instructions require less power than others, and should thus be preferred in the code selection process. This oversimplification can be overcome by using a number of methods to model the processor energy consumption in a more precise way. Three different approaches can generally be distinguished: energy models based on component data sheets are relatively easy to generate since the required information is provided by the hardware vendor. The granularity is one issue, since hardware vendors usually do not supply very low-level or fine granularity information in order to protect their IP. Simulation-based models are somewhat more elaborate, since processor model simulations may be quite time consuming. This is true in particular for low-level simulations required to achieve an improved accuracy of the obtained results. As the third main concept, physical measurements may be performed on the actual hardware if it exists and is suitable for this purpose. The advantage of physical measurements is that actual energy values can be derived, potentially taking into account all effects that take place in the processor.

We now take a look at several publications that have used these principles to build energy models.

The prime advantage of the data sheet based energy models is the readily available power information provided by hardware vendors. Using the energy information, it is usually possible to construct a convincing energy model of a system. The problems lie in the granularity of the model and in the possibly differing assumptions made by the vendors concerning typical scenarios.

A publication by Brooks et al. [BTM00] uses parameterizable power models of common structures found in modern microprocessors. In this way, a model of the processor can be assembled and its energy consumption can be estimated. One main advantage of this approach is the high speed at which energy can be evaluated.

Simunic et al. [SBM99a] propose an energy model based on available component data sheets. Each of these components is assumed to be in one of two states at any time: *active* or *idle*, depending on the processor state at that particular time. By providing energy values for components both in active and idle states, the authors build up an energy model by adding the active and idle energies of all modeled components for each cycle. For long sequences of instructions, the model achieves an acceptably high precision of within 5% compared to hardware measurements. For short sequences or even individual instructions, the results may show a large margin of error. Since only cumulative effects are accurately captured, this model is not suitable for use within a compiler to e.g. guide decisions during instruction selection. The model was used in [SBM99b] to analyze and optimize the energy dissipation of an ARM-based embedded system.

Simulation based methods are in general more elaborate since the energy behavior of all components has to be captured in a suitable way. A large number of simulation runs may be necessary to model the energy contributions of certain processor states and components. The question of granularity can be addressed by appropriately choosing the level of simulation. The available choices are, with increasing complexity:

- behavioral simulation
- register transfer level simulation
- gate level simulation
- transistor level simulation
- layout simulation

Behavioral simulation only ensures that the model shows a correct outside behavior. RTL simulation covers components and their interaction to achieve the observed behavior. Gate level simulation considers the internal structure of the components and the way they provide their functionality using simple logic gates. These gates are in turn made up of transistors, which have to be layed out in a specific way. These last two simulation levels are usually too complex and time consuming to be really useful for the generation of an energy model.

Behavioral simulation can be used to measure the switching activity e.g. on bus lines that are known to have a high capacitance and therefore consume a relevant amount of energy when their value changes. The measured switching activity can then be used as a measure for the dissipated energy. However, the quantitative impact of bus toggling on overall energy consumption can vary significantly. Generally, such simulation results require validation against real hardware to show their applicability and the margin of error.

Assuming that the simulation is performed using a VHDL model of the processor to be measured, then commonly available VHDL simulators can be used to generate information concerning the switching activity at the different levels sketched above. At the behavioral level, switching activity on all pins can be used as a measure for energy consumption. If the behavioral model is synthesized down to a gate level description using synthesis libraries characterized for power, then energy values can be derived on this level as well with the advantage of taking into account glitching, i.e. very fast, unwanted transitions in an otherwise stable signal which do not occur in behavioral simulation. Using this method, it is possible to obtain a good estimate on processor energy dissipation. In practice, however, it is difficult to control all possible parameters, and the used tools are known to have limited accuracy. For a gate level simulation assuming a zero delay model, the accuracy of simulation versus a low-level SPICE simulation is within 10 - 25% according to the Synopsys Power Tools reference manual [Syn96].

A simulation based approach to determine an energy model for the M3-DSP was e.g. presented by the authors of [LLM+01] who used gate level VHDL simulation in conjunction with the Synopsys Power Tools to generate an energy model for the M3-DSP. An energy model for the LEON processor [Gai] based on VHDL simulation is presented in [Sch03]. The SyCHOSys system [KHZA00] compiles a processor description into a performance and energy simulator. Using capacitance annotations from the circuit layout, an energy evaluation of a data path circuit showed a precision of within 7% compared to SPICE simulation [KHZA00].

Physical measurement of e.g. the processor or memory current to determine energy values is another method. The advantage of measured values is their direct connection to the hardware and their high credibility. In spite of possible measuring errors, the qualitative energy consumption can be determined by measuring the actually flowing currents, which guarantees that all effects within the actual hardware are being taken into account. One of the problems with physical measurement is its applicability: for a processor like the ARM7, measuring the current that flows through the processor is feasible since the used evaluation board features pins for this purpose. If, however, more components are used in the processor, like e.g. a cache, then it becomes increasingly difficult to extract energy values solely for the processor, without any interference from the cache. Complex pipeline structures or branch prediction units make it increasingly difficult to find test patterns that put the processor through all possible states and transitions in order to provide a sufficiently complete energy model which enables the compiler to estimate the effects of the decisions it takes.

One of the first measurement based energy models was presented by Vivek Tiwari, who published detailed information on the energy consumption of an Intel 486DX2 processor, a RISC and an embedded processor [TL98, TMW94a, TMW94b, TMW96]. Later measurements using his energy model were also taken for an ARM processor by Sinevriotis [SS99]. Tiwari's energy model is based on "instruction base costs" and "inter-instruction costs". While the processor executes a long sequence of one single instruction, the average processor current is measured. This value is used as the "base cost" for the instruction under consideration. Having determined base costs for all instructions, long sequences of pairs of two different instructions were measured. Additional costs caused by circuit state changes were observed in this case. This difference between the average currents of the involved instructions is called the "inter-instruction effect". The authors found that measuring pairs of instructions is sufficient to capture inter-instruction effects. In fact, the additional overhead incurred by executing different instructions was usually around 5%. To simplify the energy cost model, a fixed amount is added to the base cost to account for any inter-instruction effect.

The authors of [SBT00] also adopted the energy model to speed up power estimation. They have refined the consideration of inter-instruction costs and have added the value of instruction operands to the scope.

The main drawback of Tiwari's model is the fact that only the processor itself is being modeled, thus neglecting the memory subsystem. This can lead to a mismatch between the predicted energy consumption of the processor and the energy dissipation of the complete system, including the memory. Therefore, Tiwari's model was extended by Steinke et al. [SKWM01] to also account for the energy consumed within the memories. Since their energy model is also used for the ARM7 in this work, it will be described in detail in the following section.

An energy model for a far more complex architecture, the VLIW M3-DSP, is presented in [LLM$^+$01]. It is also based on Tiwari's work, though the model was first developed using VHDL simulation and only later validated using the actual produced silicon of the processor. It was found that the approximations were better when the notion of inter-instruction effect was completely dropped from the model. Instead of Tiwari's "base cost" for each instruction plus "inter-instruction cost" between instructions, the model always considers pairs of instructions. The M3's four stage pipeline is initially filled with the instructions under observation, then four cycles are executed and traced. In this way, one pair of instructions is observed as it moves through the entire pipeline. The resulting energy model is used in such a way that a sequence of instructions is broken down into pairs of instructions, for which the energy contribution is added. Some care has to be taken to treat start and end instructions correctly for short instruction sequences. The model is used in [LWD02] to show the effectiveness of compiler optimizations using a genetic code generator. Comparing the values predicted by the energy model and VHDL simulation results for a specific example shows that the differences are below 0.5%. Compared to measurements on real hardware, differences of less than 2% are reported.

Another measurement approach with specialized equipment was performed by Chang et al. [CKL00]. They produced the required test data using an FPGA vector generator and measured the flowing current for every clock cycle and every pipeline stage. To construct an energy model, the hamming distance and the number of ones in subsequent instruction was used.

To generate an energy model for a design implemented on an FPGA, the authors of [LNC03] set up complex measurement devices using the "switched capacitor method" to capture the power required by the device at every clock cycle. These contributions were summed up to form the overall energy dissipation.

ARM7 Energy Model

The used energy model for the ARM7 processor has been described in some depth e.g. in [Ste03] and in [SKWM01].

It is based on the results of a master's thesis [The00] where the ARM7 evaluation board was measured according to the energy model proposed by Tiwari et al. [TMW94b]. In this model, as in Tiwari's work, the overall average inter-instruction effect contributes less than 5% energy dissipation for nearly all instructions. Therefore, the inter-instruction effect was taken into account by adding a constant offset to the base costs of the instructions.

In [SKWM01], this energy model is extended to also cover the used memories. The system's energy consuming components are modeled using a simple and generic block diagram which also includes data and instruction memories. The energy costs are divided into the CPU costs (instruction and data dependent) and memory costs (instruction dependent for instruction memory, data dependent for data memory). For each of these costs, the model considers the number of bits that are set to one ("number of ones") on the connecting bus lines as well as the hamming distance between the current and the subsequent word on the bus, since these two factors were found to have a strong impact on energy consumption. The "number of ones" and the hamming distance are weighted with parameters α and β, whose size is determined using a defined measuring procedure. In this way, the proposed energy model can be adapted to other processors as well, without requiring detailed knowledge of the internal architecture of the processor.

The measured values are used to describe the energy consumed by a specific sequence of instructions using Tiwari's energy model [TMW94b]. Validation of the used model shows that the average deviation of the predicted values from the measured results was only 1.7%, making this energy model sufficiently precise for use within a compiler.

In some cases, the energy consumption of the processor and of the memory show different trends when e.g. the hamming distance of subsequent instructions is increased. It is therefore obligatory to consider the memory present in the system in order to get the full picture concerning energy behavior. The used energy models for different classes of memory architectures will be described in the following section.

3.4.3 Memory Energy

The energy dissipation of the processor has for some time been the main target of research, since its contribution to the energy consumption of a system is obvious. Several studies have shown, however, that the memory's contribution to overall energy dissipation is increasing. Onchip caches were found to consume the majority of a processor's energy [KG97], and the ever increasing requirements for data storage in modern embedded systems make the integration of more and larger caches and main memories necessary. All of these memories consume a large amount of energy, and just like the "memory wall" [WM95] has already been found to be a major threat for the future development of systems concerning performance, a similar effect can be expected to occur concerning energy dissipation as well.

For these reasons, it is vital to have working and valid models for the energy consumption of all sorts of memories that are to be used in a design to be able to estimate the overall energy dissipation during an early phase of development. Failure to assure that restrictions concerning energy are met at an early phase may lead to costly redesigns.

In the following section, we first take a look at related work concerning energy models for different kinds of architectures. We then present the energy models used in our environment for SRAM memories that are mainly being used as onchip scratchpad memories, but are also part of the memory arrays in caches. Then, we consider DRAM memory modules, a technology that is being used to implement larger main memory. Since nearly all embedded systems today include a non-volatile Flash memory region, an energy model for Flash memories is also presented. The Flash memory is either used to store configuration data or to safely hold the program and data when the device is switched off. Alternatively, Flash memories can also be used as instruction memory using "execute-in-place" (XIP), which is described in Section 5.3.

Related Work

Many researchers have considered the issue of modeling memory energy dissipation. The work is motivated by the observation that only optimizing the processor without also considering the used memory hierarchy does not always lead to satisfactory results with respect to energy consumption of the entire system. This is due to the high percentage of overall system energy being spent in the memory subsystem. Onchip caches using SRAM were found to consume 25 to 45% of the total chip power [KG97]. In the StrongArm 110 processor, a later version of the ARM7 used in this work, the cache consumes 43% of total processor energy [Mos01]. An entire system including both processor and memory was considered in [KVIY00], where energy consumption is reduced using well known compiler optimizations. The results indicate that more energy is consumed within the memory subsystem than in the processor core itself, at least if unoptimized code is being used.

In our work, memory energy models are used in two different stages of the system design and software generation process: on one hand, the energy model is used within the compiler to e.g. evaluate the benefit of storing a certain memory object in a specific partition of the memory. On the other hand, once the program has been compiled, it can be executed using e.g. an instruction set simulator. The information generated during this simulation can then be combined with the memory energy model to provide information on the energy dissipated within the memory by this particular application. For some memories that have an access-dependent timing and energy behavior, like e.g. DRAM cells, only this evaluation of a complete access trace can provide accurate results, since within the compiler, information concerning dynamic run-time decisions concerning branches is not available, and therefore the precise address sequence is not known.

Despte the fact that some publications provide a more fine grained enumeration, there are basically three different general approaches to develop an energy model for a memory.

- Data Sheet Models: If data sheets are available for the used memory cell, then the given information can in general be used to provide an insight into the energy dissipation of this memory in a system. Depending on the kind of memory, different parameters will be specified in the data sheets. This information can be used to generate a sufficiently accurate energy model. For SRAM memories, there is usually an "energy per access" value, since all accesses to these memories have the same impact on energy. For DRAMs, however, there will usually be a typical current for different states during memory accesses. The model must then also include all known information about access frequency and address traces in order to determine how long the memory is considered to be in a particular state. Once a model has been developed and integrated into the workflow, a different memory can be integrated by simply exchanging the values from the data sheet.

  The DRAM energy model published by Micron Technology Inc. [Mic01] is an example for a data sheet based memory energy model that is being widely used, e.g. in [MCB$^+$03]. It also forms the basis of Section 3.4.3 in this work. The work of Simunic et al. [SBM99a] also relies on data sheets to estimate the energy dissipation of memories, assuming that a memory can either be active or idle.

- Measurement based Models: If a certain memory is physically available and integrated into a system context, then it may be possible to measure the current flowing through the memory during different accesses. In this work, measurement was only done for SRAM memories, since each access can be assumed to consume the same amount of memory. For DRAM models, different access modes (e.g. burst access) also have to be considered in order to build a valid energy database, making the model and also the measurement more complicated. In general, physically measuring the energy consumption of a memory is not an easy task, since designers usually do not provide a means for measuring. Thus, the supply pins of the memory, if accessible, will have to be cut in order to connect an ammeter. Despite the fact that the measurements performed on our evaluation board were successful, it is not probable that all memories, in particular newer, faster cells, will still work with an ammeter (or any other measurement instrument) in place. All measurement-based energy models can be used only for the memory that was used to measure the initial energy values.

  Measurement based memory energy models have been described e.g. in [SKWM01], where a high-level representation of the entire system, including data and instruction memory, is used to formulate a model which is then connected to actual energy values using physical measurement. The authors of [SJC$^+$03] describe a complex measuring methodology in conjunction with a state machine to characterize the energy dissipation of SDRAM memories, including the memory buses.
- Analytical Models: These energy models are probably the most versatile models, since neither a physical memory nor data sheet information is required. Analytical models are based on observations that are true for all memories of a particular kind. They would e.g. assume a certain amount of energy for a full access to a DRAM cell, and a lower energy amount for a sequential burst access. Some analytical models go down to the transistor and layout level in order to justify their assumptions and values. Still, it can be difficult to provide a realistic margin of error for these models. The big advantage of these models is their versatility, meaning they can be adjusted to cover many forms of memory organization. If all underlying assumptions are understood, then such a model can be ported to new technologies as well, at the risk of losing precision if some of the dependencies are not fully understood.

  The reason that an analytical models are at all possible and common for memories is that memories, in contrast to e.g. processors, have a highly regular structure and only few states that need to be covered.

  A well known example for an analytical memory model is the CACTI tool [WJ94, WJ96], which, based on information collected during HSPICE simulations, determined an analytical description of the dependencies between the organizational parameters of a cache and its timing and energy dissipation.

In the following sections, we will present memory energy models for those kinds of memories commonly found in embedded systems. We start with SRAM memory, which is frequently used as onchip memory, e.g. in caches or scratchpad memories. In some systems, SRAM can also be found as main memory, though DRAM is more common for this purpose. Following the data sheet based DRAM energy model, we finally present a way of modeling the energy consumption of Flash memories.

SRAM Energy Model

Concerning timing, SRAM memories can be modeled quite easily, as shown in Section 3.3.2. Similar accesses are assumed to take an equal amount of time, since standard SRAM cells do not support the notion of burst transfers or improved sequential accesses. With respect to energy, one way to determine the impact of a memory access on the energy dissipation is to perform a physical measurement on the used memory. Alternatively, analytical models can be used to determine the energy consumption by reasoning about the internal structure of a memory. Finally, data sheets may be used to determine the energy dissipation during an access in a certain state of operation.

- Data Sheet based model: Using data sheets to model memory energy is quite simple and straightforward for SRAM cells: the cost per read or write access can be directly determined from the data sheet and used in the model. This energy value is then included in the energy databases used throughout the workflow. This method was not adopted for SRAM in the considered workflow, since measurement based values were available.
- Measurement based method: Since SRAM cells show similar behavior for each access of a certain type, it is relatively easy and straightforward to physically measure the consumed energy per access. In order to measure the energy consumed by the SRAM used on our ARM7 evaluation board as main memory, the supply pins of the memory chip were cut in order to connect the ammeter. We found that for the main memory located on the evaluation board, the current (and therefore the energy per access) differs depending on the width of the access. This is mainly due to the fact that two memory chips have to be addressed in case of a 32 bit access. Also, different values were recorded for reading and writing. Therefore, our test programs repeated a number of e.g. read operations for a certain bit width, and the average current was determined and included in the energy model. Using the current, the supply voltage and the timing of the accesses (determined as described in section 3.3.2), energy values for the different accesses were determined and stored in the energy data base. Table 3.3 gives an overview over the determined values for different access widths and for read and write operations.
  The reason for the strong dependence of the used energy on the access width is due to the way the main memory is connected to the processor

Access Width	Read Energy	Write Energy
1 byte	154.8 μJ	149.8 μJ
2 bytes	240.0 μJ	298.8 μJ
4 bytes	493.2 μJ	411.0 μJ

Table 3.3. Measured energy values for accessing main memory

(cf. Figure 3.8 on Page 33): two memory chips with an 8 bit data interface each are being used. Thus, to access a halfword, both memory chips are accessed, resulting in a significantly higher energy dissipation. If a 4 byte word is accessed, then both memories are accessed over a duration of two cycles, roughly doubling the previous energy amount.

For the scratchpad memory located directly on the processor chip, it was not possible to determine the current in the way described above, since the current flowing through this part of the processor can't be measured in isolation from the other processor components. The effect of using the scratchpad memory can thus only be measured in combination with the processor energy. Since the scratchpad memory is accessed whenever the address of an instruction or of a data item lies within the scratchpad's address range, test programs were generated that kept their memory objects either on the scratchpad or in the main memory. In this way, it was possible to determine one average overhead value caused within the processor when the scratchpad memory was being accessed (cf. [The00] for details). This value is used in the energy model as the "energy per access" value of the scratchad memory. It was found to be an order of magnitude lower than that for a main memory access. Since the scratchpad memory can not be switched off completely on our evaluation board, the standby base cost of the scratchpad is always consumed, a fact which is also reflected in the energy model. Newer processor designs would probably allow the user to switch off the scratchpad memory, enabling further energy savings e.g. by compiler controlled energy management.

In general, physical measurements are assumed to be the most trustworthy possibility of obtaining energy values, since they respect all aspects of the used memory and the way it is integrated into the system context.

- Analytical model: One drawback of measurement based energy models is the fact that energy can only be determined for the exact configuration that was used during the measurement experiments. To allow the evaluation of the impact of different sizes of scratchpad memories in our system, an analytical approach was adopted. In [BSL$^+$02], the authors describe how an analytical energy model for caches can also be used to model the energy consumption of a scratchpad memory. The idea behind this approach is that a cache consists of a data memory array which is organized just like a scratchpad memory. A cache in addition requires a tag memory and comparators in order to perform automatic hit/miss detection.

By simply eliminating the energy contribution of those components not present in a scratchpad, the cache is stripped down to its data memory array, which is very similar to a scratchpad. [BSL$^+$02] have used the well-known CACTI cache model [WJ94, WJ96] to also determine the energy dissipation for scratchpad memories of different sizes. This analytical cache model gives detailed information concerning the timing and energy contribution of several architectural parts of the cache memory. This made it possible to extract those values relevant for a plain memory and thus determine the "per access" energy dissipation of a scratchpad memory. CACTI is used to generate values for a direct mapped cache that has the same data memory size as the scratchpad memory to be modeled. Direct mapping is used since the data memory organization for this associativity is closest to a plain memory array. From CACTI's output, only those components that are also present in a scratchpad memory are considered. This means that all values belonging to the comparison logic part of the cache are omitted. In Figure 3.11, those components that were not considered for the energy dissipation of the scratchpad memory are shown within a grey box.
Despite the fact that the CACTI model is known not to be 100% accurate (estimates are within 6% of HSpice results according to [WJ96]), it is being

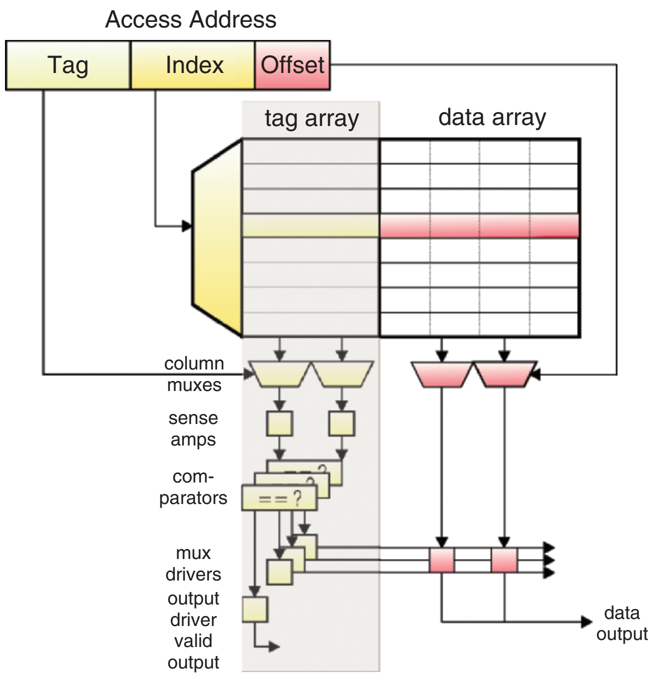


Fig. 3.11. Block diagram of the CACTI cache model

widely used by many research groups. It delivers an easy way of generating energy models for different caches and memories in a fast and efficient way. Since we do not have access to the SPICE simulation values used by the developers of CACTI to validate their model, it is difficult to estimate the actually attained accuracy of the model. Due to the consideration of all underlying assumptions as discussed in [BSL$^+$02], we believe that the model stays very close to the precision of the original CACTI model.

This completes the considered possibilities of generating an energy model for SRAM based memories. It should be mentioned that the advances in technology may make it necessary to consider different, advanced techniques in the future, since SRAM memories are now being tuned for better performance as well as energy consumption. One way of increasing access speed is the use of overlapping read and write operations. So-called synchronous SRAM (SSRAM) cells operate their address and data registers at the processor's speed and use pipelining to increase throughput. In addition, some SSRAM cells offer a burst mode, where the address is incremented automatically, thus simplifying the access since no new address needs to be generated and transferred to the memory, but on the other hand complicating the energy model since the accesses thus become state dependent. Currently, there is a strong trend towards low power SRAM cells with decreased operating voltages. A voltage of 1.8 V is being pursued, with further reductions planned in the future.

DRAM Energy Model

The energy dissipation of DRAM memories is somewhat more elaborate to determine than that of SRAM memories. The point that makes the consideration of DRAM memories more complex than SRAM cells is the fact that their timing and energy behavior strongly depends on previous accesses. A precise energy model therefore has to consider a sequence of accesses in order to determine the energy contribution of a single access. Additionally, DRAM memories have to be refreshed in order to keep their contents. The energy required to refresh (i.e. read and re-write) the values in the memory is always consumed, independent of accesses. This standby or background energy has to be considered in the energy model to provide realistic values. Due to these reasons, there is no easy way of performing physical measurements to capture a DRAM's energy dissipation. The test programs would have to consider the state the memory is in and guarantee that the assumptions about previous states and accesses are always met. Since this is a difficult task, the energy model presented here represents a mixture between a data sheet and an analytical approach. The values in vendor specific data sheets are used to determine the average currents flowing through the memory when it is in a certain state. The analytical part of the model then connects these partial power figures to one final energy value, using assumptions about number and nature of state transitions as well as the time spent in the different states. The general idea of this model has been described in [Mic01], a detailed description can be found in [Ker05].

The energy model itself does not treat each memory access individually: by monitoring the memory access patterns during simulation, it is possible to determine the amount of time the memory spends in the different states (e.g. power down, active, precharged etc.). These durations are then used to determine the overall energy dissipation of the memory. This idea, which makes the final energy model easier to handle (since otherwise, each state transition would have to be considered) was also presented in [Mic01] and is being used by many other energy models as well.

Note that the maximum precision of this energy model can only be obtained when a complete application is being simulated and monitored, since only then is it possible to determine the sequence of memory accesses in order to determine the overall energy dissipation. The energy model used within the compiler has to be simpler, since during program generation, the complete memory access sequence is not yet known. To improve the precision of the model used within the compiler, one profiling run of the application can be performed in order to obtain a general idea of the number and kind of accesses to the main memory, e.g. the percentage of potential burst accesses. These statistics may be updated within the compiler when optimization algorithms change the memory layout or program behavior.

Due to the fact that regular DRAM memories are no longer considered state of the art, the currently used technology of synchronous DRAM (SDRAM) will be considered in this section.

Depending on the current state of the memory, it can consume different amounts of energy. The values provided in the vendors' data sheets together with an adequate energy model enable the user to estimate the total energy dissipation of an SDRAM memory. The values provided in the data sheets are more or less identical, even among different vendors, which makes it possible to choose arbitrary memory cells and use the values provided in their data sheets. In some cases, it may be necessary to consider different naming conventions. Also, the supplied parameters may be different depending on the precise organization of the used memory. In our examples, we use the values provided by Micron for both SDRAM and a double-data rate (DDR) SDRAM. Using these two memory architectures, we show how the given values can be used to generate an energy model for both architectures.

The overall energy dissipated within the memory is made up of the integral over time of the current flowing through the memory, multiplied with the operating voltage (which, in our model, is assumed to be constant):

$$E_{SDRAM} = \int I_{SDRAM} \cdot V_{DD} \; dt = \int P_{SDRAM} \; dt \tag{3.11}$$

In order to simplify this equation, an average current is determined for each of the different operating states of the memory, and this average is assumed throughout the time the memory is in this state. The overall power of the memory is thus made up of the power consumed in each of the possible states, which comprise power down, standby, active, read, write, driving data and refresh:

$$\begin{aligned} P_{SDRAM} = {} & P_{SDRAM\_PDN} + P_{SDRAM\_STBY} \\ & + P_{SDRAM\_ACT} + P_{SDRAM\_RD} \\ & + P_{SDRAM\_WR} + P_{SDRAM\_DQ} \\ & + P_{SDRAM\_REFRESH} \end{aligned} \tag{3.12}$$

Using these average power figures as the base, we can now determine the overall energy by simply multiplying with the time the complete application requires to execute on the CPU:

$$E_{SDRAM} = P_{SDRAM} \cdot t_{CPU} \tag{3.13}$$

In the following paragraphs, the above-mentioned power values for all states of the DRAM memory will be determined from the values provided in the memory's data sheet. Once all individual power figures have been calculated, the overall energy dissipation can be determined using Equations 3.12 and 3.13.

Typically, SDRAM data sheets provide values for the current that flows through the memory in a certain state. Table 3.4 shows which current values for both an SDRAM and a double data rate DDR-SDRAM are provided by a typical data sheet, using the naming conventions of Micron Technology Inc. When the table shows an 'X', the corresponding current value for that component will be provided in the data sheet. With these currents, the known supply voltage and considering the time the memory spends in a certain state, it is possible to determine the overall energy dissipation of the memory cell with sufficient accuracy. Despite the fact that different values are given for the two memory architectures, it is possible to determine the required power figures for both SDRAM and DDR-SDRAM using our model.

There are four major different kinds of contributions to the overall power of an SDRAM: state-related power, activation related power, access related power and refresh related power. We will cover the contributions in this order. In order to simplify the presentation, we will first only consider the SDRAM memory.

- State related power for SDRAM:
  The overall operation mode of the memory is controlled by the control signal "Clock enable". If it carries a low signal, then all buffers and the internal clock are disabled, leading to a reduced energy dissipation in the powerdown state (PDN). On a positive edge on this signal, the memory is activated and changes to the standby state (STBY). These are the two main states the memory can be in when no access is taking place.

Value name	symbol	SDRAM	DDR-SDRAM
active precharge current	I_{DD0}	-	X
active operating current	I_{DD1}	X	-
precharge power-down standby current	I_{DD2P}	-	X
idle standby current	I_{DD2F}	-	X
power-down standby current	I_{DD2}	X	-
active power-down standby current	I_{DD3P}	-	X
active standby current	I_{DD3N}	-	X
active standby current	I_{DD3}	X	-
read current	I_{DD4R}	-	X
write current	I_{DD4W}	-	X
operating current	I_{DD4}	X	-
auto refresh current	I_{DD5}	X	X

Table 3.4. Typical data sheet information

If the memory changes from active to the power down state, the current I_{DD2} can be assumed to flow for a regular SDRAM. The power consumed while the memory is in the power down state can thus be determined as

$$P_{SDRAM\_ACT_{PDN}} = I_{DD2} \cdot V_{DD} \tag{3.14}$$

In order to perform an operation, one row of the memory has to be opened by issuing an "activate" command, putting the memory in the active state (ACT). Since no access is taking place at this time, the current that flows when the memory is active is given as I_{DD3} (active standby current). Thus, active standby power is determined as

$$P_{SDRAM\_ACT_{STBY}} = I_{DD3} \cdot V_{DD} \tag{3.15}$$

Following an "activate" command, a "precharge" command is always required to finish any access. When this happens, the memory is in the precharged state (PRE). Depending on the clock enable signal, it can either be in the precharged power down or in the precharged standby mode. For SDRAMs, the same currents as for the active state (ACT) are used for the precharged state, therefore the model can be simplified by assuming identical values for precharged and active memories:

$$P_{SDRAM\_PDN} = P_{SDRAM\_PRE_{PDN}} = P_{SDRAM\_ACT_{PDN}} \tag{3.16}$$

$$P_{SDRAM\_STBY} = P_{SDRAM\_PRE_{STBY}} = P_{SDRAM\_ACT_{STBY}} \tag{3.17}$$

Besides ACT and PRE, the memory can also be in the deep power down or sleep mode. If the memory goes to this mode (and loses all its contents due to the suspended refresh), a very small current called I_{ZZ} is specified in the data sheet.

$$P_{SDRAM\_DPD} = I_{ZZ} \cdot V_{DD} \tag{3.18}$$

- Activation related power for SDRAM:
  In order to determine the power caused by the activation of the memory by an "activate-precharge" pair of commands without considering any power for the actual access to an SDRAM memory, we first need to consider the data sheet value for the "active operating current" I_{DD1}, which represents the average current for two successive read accesses within the same row. By subtracting the current of the two read operations, the power for an "activate-precharge" pair activation can be determined for an SDRAM memory. The "operating current" I_{DD4} of an SDRAM is the average current required to perform either a read or a write access. To only account for the additional overhead incurred by the read operation, the "active standby current" I_{DD3} needs to be subtracted. The actual read accesses are assumed to take two clock cycles for our memory. The distance between two activations of the memory is assumed to be t_{RC}. This information leads to the following equation for the activation current I_{DD0}:

$$I_{DD0} = I_{DD1} - \frac{(I_{DD4} - I_{DD3}) \cdot 2 \cdot t_{Clk}}{t_{RC}} \tag{3.19}$$

  It is only necessary to determine this activation current value for SDRAM memories, since for DDR-SDRAM, the value is directly specified in the data sheet. The net activation power can now be determined by subtracting the constant current value I_{DD3}:

$$P_{SDRAM\_ACT} = (I_{DD0} - I_{DD3}) \cdot V_{DD} \tag{3.20}$$

- Access related power for SDRAM:
  Having opened a single row, the actual read or write access can take place. To model its power contribution, the average current I_{DD4} is introduced for both read and write accesses. In some cases, two values may be provided, but they are usually very close to each other. Since any read or write operation can only take place between two activate commands, the activation current I_{DD3} must be subtracted from the read/write value supplied in the data sheet in order to determine the contribution of read/write operations in isolation:

$$P_{SDRAM\_RD} = P_{SDRAM\_WR} = (I_{DD4} - I_{DD3}) \cdot V_{DD} \tag{3.21}$$

  One contribution that needs to be considered for a read access in addition to the previous values is the fact that when a memory is being read, the output buffers need to drive the values from the output buffers onto the bus. The capacitative load is represented by the value C_{LOAD}, and the net load depends on the number of data bit lines DQ and the number of output control bits (data strobe) DQS.

Depending on the operating frequency CLK of the memory, the driving power of the memory amounts to

$$P_{SDRAM\_DQ} = \frac{1}{2} \cdot C_{LOAD} \cdot (V_{DD})^2 \cdot CLK \cdot (DQ + DQS) \tag{3.22}$$

- Refresh related power for SDRAM:
  The only operating phase that has not been covered so far is the refresh cycle of DRAM memories. The data sheet value I_{DD5} (auto refresh current) represents the average refresh current in power down state. Thus, in order to determine the current for the continuously occurring auto refresh during normal operation, we need to subtract the constant power down current:
  $$P_{SDRAM\_REFRESH} = (I_{DD5} - I_{DD2}) \cdot V_{DD} \tag{3.23}$$

This concludes the considerations for a regular SDRAM memory. For a DDR-SDRAM, different values are usually provided as shown in Table 3.4. In the following, we will therefore quickly cover the determination of the required power values for a DDR-SDRAM. In most cases, the reasoning behind the equations is similar to the SDRAM case, and only the differences will be explained:

- State related power for DDR-SDRAM:
  I_{DD3P} denotes the active power down current of a DDR-SDRAM, thus
  $$P_{DDR\_ACT_{PDN}} = I_{DD3P} \cdot V_{DD} \tag{3.24}$$
  For the active state (ACT), I_{DD3N} is provided:
  $$P_{DDR\_ACT_{STBY}} = I_{DD3N} \cdot V_{DD} \tag{3.25}$$
  In the precharged state (PRE), the current flowing in a DDR-SDRAM is represented by I_{DD2P} if buffers and the clock input are deactivated and I_{DD2F} if they are active:
  $$P_{DDR\_PRE_{PDN}} = I_{DD2P} \cdot V_{DD} \tag{3.26}$$
  $$P_{DDR\_PRE_{STBY}} = I_{DD2F} \cdot V_{DD} \tag{3.27}$$
  Note that for a DDR-SDRAM, active and precharged states are not considered to be identical, as was the case for SDRAM.
  The equation for deep power down is identical to the SDRAM case and is thus omitted for brevity.
- Activation related power for DDR-SDRAM:
  In contrast to a regular SDRAM, the active precharge current I_{DD0} is directly specified for a DDR-SDRAM and does not need to be computed. The activation power can thus directly be expressed as:
  $$P_{DDR\_ACT} = (I_{DD0} - I_{DD3N}) \cdot V_{DD} \tag{3.28}$$

- Access related power for DDR-SDRAM:
  This contribution is identical to the SDRAM case. Assuming distinct values for read and write accesses, the read and write power is:

$$P_{DDR\_RD} = (I_{DD4R} - I_{DD3N}) \cdot V_{DD} \tag{3.29}$$

$$P_{DDR\_WR} = (I_{DD4W} - I_{DD3N}) \cdot V_{DD} \tag{3.30}$$

  The power required to drive the data bit lines is identical to the SDRAM case.
- Refresh related power for DDR-SDRAM:
  Refresh power is determined by subtracting the precharge power down standby power from the auto refresh current:

$$P_{DDR\_REFRESH} = (I_{DD5} - I_{DD2P}) \cdot V_{DD} \tag{3.31}$$

To conclude this section, we now describe how to determine the overall energy dissipation of a regular SDRAM memory cell using the values determined above on one hand and information about an application's memory access pattern on the other. To determine the latter information, the application is simulated and the memory accesses are recorded. For each cycle, the corresponding memory state is determined and a counter is incremented. In the end, a distribution of the overall runtime of the application to the different memory states is obtained. Other models have proposed the use of profiles (e.g. "moderate usage", "high stress") to account for different usage scenarios, however this is usually associated with some loss of precision. In particular, the following values are required:

- BNK_{ACT}: fraction of program cycles at least one of the memory banks is activated, putting the memory in the active state
- BNK_{PRE}: fraction of program cycles the memory is in the precharged state. Deep power down is not considered in this context, since it causes the memory to lose all its information. Therefore, we can assume $BNK_{PRE} = (1 - BNK_{ACT})$
- CKE_{LO_PRE}: fraction of BNK_{PRE} during which CKE is '0'
- CKE_{LO_ACT}: fraction of $(1 - BNK_{PRE})$ during which CKE is '0'
- RD/WR: fraction of program cycles during which data is being read or written, respectively

Furthermore, during actual operation of a memory, it cannot be assumed that all activation commands will occur in direct succession. Rather, a certain average time will be spent until the next activation takes place. This time (in cycles) is introduced as n_{ACT} and can also be determined from the application simulation. It is used to scale the activation power determined above:

$$\begin{aligned} P_{SDRAM\_ACT} &= (I_{DD0} - I_{DD3}) \cdot \frac{t_{RC}}{n_{ACT} \cdot t_{CK}} \cdot V_{DD} \\ &= (I_{DD0} - I_{DD3}) \cdot \frac{t_{RC}}{t_{ACT}} \cdot V_{DD} \end{aligned} \tag{3.32}$$

Not only the activation frequency of a memory has to be scaled according to the actual situation: the same is also true for the operating frequency and the used voltage. The power equations above assume the memory being run with the maximum specified frequency f_{spec}, while a memory that is actually installed in a system may use a different operating frequency f_{use}. All power values that depend on the operating frequency thus need to be scaled with the factor

$$\text{scale}_f = \frac{f_{use}}{f_{spec}} \tag{3.33}$$

The only power values that do not need to be scaled, since they do not depend on the frequency, are

- P_{SDRAM_PDN} (or $P_{SDRAM\_PRE_{PDN}}$ and $P_{SDRAM\_ACT_{PDN}}$, if considered separately), since the memory does not receive a clock signal in this state
- P_{SDRAM_ACT}, since this power value solely depends on the time between activations t_{ACT}
- $P_{SDRAM\_REFRESH}$, because the refresh power is determined using an absolute interval which is independent from the frequency

If the supply voltage is changed from the maximum voltage specified in the data sheet, then all voltage values need to be scaled with the factor

$$\text{scale}_{V_{DD}} = \frac{(V_{DD_{use}})^2}{(V_{DD_{spec}})^2} \tag{3.34}$$

The scaling factor contains the squared voltage values since when voltage is reduced, the operating currents also decrease. Therefore, e.g. a 5% reduction in the supply voltage also reduces the current by about 5%, resulting in a 9.8% reduction in power if $\text{scale}_{V_{DD}}$ is used accordingly:

$$P_{use} = P_{spec} * \text{scale}_{V_{DD}} = P_{spec} * \frac{V^2_{DD_{use}}}{V^2_{DD_{spec}}} \tag{3.35}$$

The data driving power is not scaled, however, since the voltage used in P_{DQ} is not necessarily identical to the memory supply voltage.

The power consumed in a certain state of the SDRAM memory is denoted as P_{SDRAM_state}, as before. The power that also includes the percentage the memory is in the considered state will be written as $\mathbf{P}_{SDRAM}$.

For SDRAM memories, the values for precharge power down and active power down are identical (cf. Equation 3.16). Therefore, CKE_{LO_PRE} and CKE_{LO_ACT} can be summarized to CKE_{LO}, which corresponds to the fraction of total program execution cycles during which the memory is in the power down state. The same is true for the active standby and precharged standby states (cf. Equation 3.17):

$$\mathbf{P}_{SDRAM\_PDN} = P_{SDRAM\_PDN} \cdot CKE_{LO} \cdot \text{scale}_{V_{DD}} \tag{3.36}$$

$$\mathbf{P}_{SDRAM\_STBY} = P_{SDRAM\_STBY} \cdot (1 - CKE_{LO}) \cdot \text{scale}_{V_{DD}} \cdot \text{scale}_f$$

Access related power can be determined using the corresponding fraction of access cycles:

$$\mathbf{P}_{SDRAM\_RD} = P_{SDRAM\_RD} \cdot RD \cdot \text{scale}_{V_{DD}} \cdot \text{scale}_f \tag{3.37}$$

$$\mathbf{P}_{SDRAM\_WR} = P_{SDRAM\_WR} \cdot WR \cdot \text{scale}_{V_{DD}} \cdot \text{scale}_f \tag{3.38}$$

$$\mathbf{P}_{SDRAM\_DQ} = P_{SDRAM\_DQ} \cdot RD \cdot \text{scale}_f \tag{3.39}$$

Finally, the activation and refresh components still need to be considered. Since the activation power has already been weighted with t_{ACT} and refresh is assumed to occur throughout the execution time, only voltage scaling needs to be done for these values:

$$\mathbf{P}_{SDRAM\_ACT} = P_{SDRAM\_ACT} \cdot \text{scale}_{V_{DD}} \tag{3.40}$$

$$\mathbf{P}_{SDRAM\_REFRESH} = P_{SDRAM\_REFRESH} \cdot \text{scale}_{V_{DD}} \tag{3.41}$$

This concludes the calculation of all the power components of an SDRAM memory. To determine the overall energy, the partial power contributions need to be added and multiplied with the overall execution time of the application (cf. Equations 3.12 and 3.13).

Flash Memory Energy Model

To determine the energy dissipation of a Flash memory, we again resorted to a data sheet based approach similar to the one employed for SDRAM memory. Flash memories integrated in embedded systems are generally used only for reading data, since the number of write cycles is limited. Therefore, we only consider read accesses.

As already described in Section 3.2, a Flash memory is first accessed by an asynchronous random access, possibly followed by an intrapage access, if the subsequent read goes to the same page. Finally, synchronous Flash memories may support a burst mode, a feature that is not covered in this work.

The characteristics of a Flash memory both concerning timing and energy dissipation can easily be extracted from the corresponding data sheet. Again, the names chosen by different vendors are usually very similar. Table 3.5 shows the designations as used by Micron Technology Inc. Since both power and timing values are required to determine the energy dissipation, both characteristics are given in the table.

In contrast to DRAMs, Flash memories do not require a thorough modeling of all possible states, since only the two access types mentioned above need to be distinguished. Additionally, the background power of Flash memories when they are not being accessed is very low and thus negligible. For these reasons, the Flash energy model can be kept simpler than the DRAM model, similar to the per-access-model used for SRAMs.

Designation	Unit	Symbol
Supply Voltage	V	V_{DD}
Read async. access time	ns	t_{AA}
Read intrapage access time	ns	t_{APA}
Read burst access time	ns	t_{CLK}
Read async. access current	mA	I_{DD1}
Read intrapage access current	mA	I_{DD2}
Read burst access current	mA	I_{DD3}

Table 3.5. Flash memory characteristics

The first read access, assuming a 16 bit wide memory, is always performed as a random access, requiring the corresponding time and current from Table 3.5:

$$\begin{aligned} P_{FLASH\_RND16\_RD} &= V_{DD} \cdot I_{DD1} \\ E_{FLASH\_RND16\_RD} &= P_{FLASH\_RND16\_RD} \cdot t_{AA} + E_{FLASH\_DQ} \\ T_{FLASH\_RND16\_RD} &= T_{DOUT} + \left\lceil \frac{t_{AA}}{t_{CLK}} \right\rceil \end{aligned} \tag{3.42}$$

E_{FLASH_DQ}, similar to the SDRAM case, is the energy required to drive the data outputs of the Flash memory. T_{DOUT} is the time it takes to drive the output values (cf. Section 3.3.2).

Accessing a 32 bit word requires one initial random access, as above, plus one faster and more energy efficient intrapage accesses:

$$\begin{aligned} P_{FLASH\_SEQ16\_RD} &= V_{DD} \cdot I_{DD2} \\ E_{FLASH\_SEQ16\_RD} &= P_{FLASH\_SEQ16\_RD} \cdot t_{APA} + E_{FLASH\_DQ} \\ E_{FLASH\_RND32\_RD} &= E_{FLASH\_RND16\_RD} + E_{FLASH\_SEQ16\_RD} \end{aligned} \tag{3.43}$$

To determine the energy required to access the Flash memory when it is operated in synchronous burst mode, the same equation as for intrapage access can be used, if the corresponding current and timing values are inserted accordingly.

Cache Energy Model

The CACTI model was already mentioned in Section 3.4.3 where a subset of the energy values was used to estimate the energy dissipation of an onchip SRAM array according to the method described in [BSL$^+$02]. Using CACTI, the energy per access of a cache can be determined by specifying the parameters of the considered cache. The analytical CACTI model then determines the energy consumed when this kind of cache is being accessed. The energy results are separately given for different parts of the cache, like the tag and the data parts of the cache. To model the energy dissipation of the cache

Access Type	Cache Accesses
Read Hit	1
Read Miss	1 + N
Write Hit	1
Write Miss	1

Table 3.6. Number of accesses to a cache

appropriately, Table 3.6 provides information on how many accesses to the cache are necessary for read and write accesses that may either result in hits or misses:

In case of a read hit, the cache is only accessed once to determine the hit and the corresponding value is forwarded to the CPU. In case of a miss, one cache access is required to read and compare the corresponding tag elements. If no match is found, then the entire cache line of size "N" is fetched from the next level of memory in the considered cache architecture. For write accesses, only one single access is assumed in our model, since the energy required to update the cache entry is negligible compared to the initial tag comparison. Assuming a no-allocate-on-write miss strategy, a write miss also comprises only one access to establish the miss. The given table only specifies the number of accesses, not the cycles it takes e.g. to fill the cache line. The timing of a cache line fill is usually dominated by the accesses to the next lower level of memory in the hierarchy, such that the actual timing has to be determined considering the access timing of that level.

3.5 Simulation Models

Following the structure of the previous sections, this section first presents an overview over possible simulation models required to derive information about the behavior of the processor. Following previous and related work, a description of the ARM7 simulator used in the course of this work is given. In the subsequent section, the simulation of arbitrary memory hierarchies using MEMSIM is presented in-depth, again putting previous work on the topic into perspective.

3.5.1 Processor Simulation Model

Processor simulation is widely used in both in hardware and software development. Developers and researchers resort to the simulation of processors for a number of reasons, among others:

- Unavailability of a physical processor: the situation that an actual processor is not available can be found in particular in hardware design, when the processor under development has not been manufactured as a prototype.

The designers still need to perform simulation runs early in the design phase in order to prevent costly re-designs after production of the device.

- Cost: Due to the high hardware costs, software engineers whose programs need to support a variety of different processors may not be able to physically acquire a specimen of all supported processor architectures only to ensure correct software behavior.
- Complexity: Using a physical processor to perform measurements or to validate the correctness of software may be appropriate for one single desktop processor, but for embedded processors, the situation is quite different: a large number of different processors is used in embedded systems, all of which require a certain environment in order to function correctly. An external memory is generally required, just like a circuitry that generates the processor clock signal. If peripheral devices are to be tested, the interfaces have to be supplied and either the devices themselves or analytical tools have to be connected. In addition, each processor and each evaluation board used during the early testing phase may require different sets of tools e.g. to transfer the software to be simulated to the processor. Setting up suitable environments for a number of different processors may thus be a complex task.

Processor simulation offers a way out of the situations described above: by using a processor simulator, a simple software toolchain is set up in order to simulate a program on a processor. If a cross-compiler is used, then the compiler and the simulator may even be executed on the same machine, further facilitating the software development process. Generally, in addition to simulation, final tests will have to be performed on at least one instance of the actual hardware for validation.

There are a number of possibilities of performing processor simulation at different levels of abstraction. The following section gives a short overview and highlights a small number of available solutions.

Related Work

To describe the behavior of new systems, hardware architects today use hardware description languages like VHDL [HPH$^+$00], Verilog [Pal03] or SystemC [GLMS02]. Beside generating a synthesized description which can be used to actually manufacture the processor, simulation at a higher, behavioral level is also possible to ensure that the device under test behaves according to the specification. A number of simulators for the various HDLs are available, including Symphony EDA's free version of VHDL Simili [Sym], Synopsys VCS [Syn] or the popular ModelSim [Men] from Mentor Graphics.

After successful behavioral simulation, the design entity is usually synthesized using a hardware manufacturer specific synthesis library (cf. Figure 3.12) which maps the behavioral descriptions of the VHDL model (e.g. "+" operator) to the available building blocks within the library (e.g. full adder).

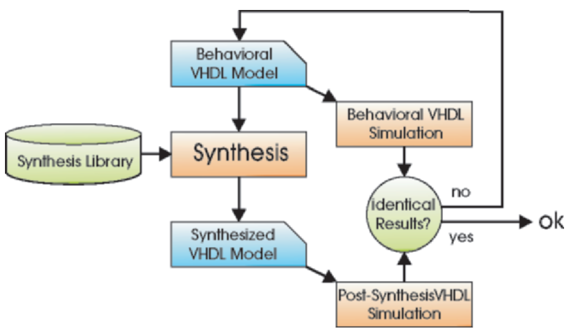


Fig. 3.12. Workflow of synthesis and post-synthesis simulation

Post-synthesis simulation is required to ensure that synthesis did not change the behavior of the entity. One disadvantage of VHDL simulation is speed: for complex models, even high-level behavioral VHDL simulation can take a long time due to the fine granularity of VHDL simulation, which is generally not required to achieve high-level estimation results.

In particular for software developers, instruction set simulators are an attractive alternative, since even a high-level VHDL description of a processor contains more details than a programmer generally needs to be aware of. Instruction set simulators in general do not model the actual building blocks of the system, but are capable of reproducing system behavior at the instruction level. For a certain input application, the behavior of the system is thus simulated in terms of executed machine instructions and a sequence of processor states (e.g. register contents), abstracting from low level details found on the register transfer level. The use of low level assembly instead of high-level programming languages is necessary since only machine instructions have a defined impact on the target hardware.

In order to allow an efficient estimation of the performance and the energy dissipation of a program executed on a certain target hardware, information about the number and kind of executed instructions, number of cycles as well as accesses to different stages of the memory hierarchy should be provided by the instruction set simulator. The analysis of e.g. energy consumption or memory accesses may either be directly integrated into the simulator or developed as a separate tool. The advantage of a tight integration is that only one single tool is required, whereas a separate analysis step can be developed in such a way that it can be used to evaluate energy models of several processors, if a uniform interface is used.

There are two main approaches to implement high-level instruction set simulators: compiled and interpreted simulation. They differ in the way the application program to be simulated is integrated into the simulator:

In a compiled simulation, one single executable containing the application to be simulated as well as the simulator is generated. In [WL02a], the authors describe how assembly level processor instructions can be transformed to C++ code and thus directly included into the simulator. Some additional data structures to represent e.g. the processor's status bit registers or the memory are required in order to represent an instruction's effect on the processor state. One advantage of this method is the fact that a pseudo-debugger for the target architecture can easily be generated. Using a debugger interface, the debugger's output can be controlled in such a way that only the original assembly instructions and e.g. register contents are displayed. The second advantage of using compiled simulation is the faster simulation speed: the executable contains all required information, can be compiled using an optimizing compiler for the host machine and is then executed.

In a simulator using interpreted simulation, the simulator itself is an executable, but the application program whose execution is to be simulated on the target hardware is not. The simulator may use operating system calls to read the application from a disk file, parse its content, instruction by instruction, and then simulate the instructions accordingly. This process is slower than a compiled simulation, since the program has to be read, analyzed and simulated at runtime. However, changing the simulated application program only requires modifying the assembly code and restarting the simulator, whereas a compiled simulation additionally requires the simulator, along with the application, to be recompiled.

Some work has been carried out concerning the automatic generation of simulators from an architecture description of a processor. In the generic low-level intermediate representation (GeLIR) [Lor03, LMD$^+$04] used in a genetic algorithm based compiler, this is realized by specifying the behavior of the machine instructions of the target machine in C++. This specification is used with the application to perform a compiled simulation. By providing an interface to describe the effect of machine instructions, it is possible to generate low-level simulators for different machines with little overhead.

The process of generating a compiled simulator from a processor described in an architecture description language (ADL) is further automated in the LISA framework [PHM00]. An HDL description of the processor itself (in the form of a SystemC-model) and common software tools (compiler, debugger) are automatically generated from the processor description in the LISA ADL. The simulators generated by LISA also use the principle of compiled simulation.

After this short overview over different simulation techniques, the next section discusses the ARM instruction set simulator provided with the ARM software development toolkit.

ARM7 Simulation Model

To support software development for the different models of available ARM processors with their respective features, ARM Ltd. provides one universal configurable cycle-true instruction set simulator called ARMulator. Depending on the configuration settings, it is able to simulate

- different ARM processor cores, ranging from ARM2 to ARM9
- one level of caches with configurable size and organization
- THUMB and ARM instruction sets
- external memories with configurable parameters

Since the simulator is only distributed as an executable, no information concerning the used internal data structures is available. The simulator provides an interface to e.g. user-specific memory models that may be attached to the simulated processor core. However, the possibilities of this interface are limited, so that the development of a more flexible memory hierarchy simulator was necessary (cf. Section 3.5.2). Also, the possible cache configurations only allow one single level of caching to be used, which is insufficient to perform cache design space exploration. ARMulator is capable of providing a number of values during simulation of an application program, it is e.g. possible to set breakpoints to halt the processor and examine register and memory contents. If instruction logging is activated, every simulated instruction as well as memory access information is written to a tracefile. In our workflow, the generated tracefiles are later analyzed using the enprofiler tool (cf. Section 3.6.2). A typical ARMulator tracefile excerpt for an architecture without caches is shown in Figure 3.13.

The lines beginning with the letter "M" denote memory access lines, with the following letters specifying sequential or non-sequential accesses, read or write access, opcode fetch and access width. Lines beginning with "IT" denote taken instructions, whereas "MI" stands for memory idle. For details please refer to [ARM98a].

```
MNR2O__ 00500060 B500
MSR2O__ 00500062 480A
MSR2O__ 00500064 210A
IT 00500060 b500 PUSH     {r14}
MNW4___ 006FFFF4 0040000D
MNR2O__ 00500066 F7FF
IT 00500062 480a LDR      r0,0x50008c
MNR4___ 0050008C 00500090
MI
MSR2O__ 00500068 FFCB
IT 00500064 210a MOV      r1,#0xa
MSR2O__ 0050006A 2101
IT 00500066 f7ff (1st instr of BL pair)
MSR2O__ 0050006C 1E48
IT 00500068 ffcb BL       0x500000
```

Fig. 3.13. ARMulator tracefile

3.5.2 Memory Simulation Model

Since in embedded systems, a majority of the energy is consumed within the memory hierarchy, the memory costs also need to be considered, including multiple levels of caches, scratchpad memories, loop buffers and main memories.

In order to overcome the limitations imposed by ARMulator's memory model (in particular the restriction to only one level of caching), a new flexible memory hierarchy simulator called MEMSIM was developed. It is capable of simulating arbitrary memory hierarchies specified by the user. These hierarchies may contain caches at different levels of the hierarchy. Different architectural cache parameters, cacheable and non-cacheable regions, loop caches and scratchpad memories are also supported. The input for MEMSIM consists of the description of the memory hierarchy and an instruction and memory access trace of a program. Using MEMSIM, it is possible to evaluate the effect of changes in the memory hierarchy, such as increasing associativity in a cache or integrating a loop cache into the design.

MEMSIM was developed in order to overcome limitations both of the used ARM software development toolkit and of available cache simulation frameworks. It was our intention to be able to evaluate simulation results for a variety of memory hierarchies on a single processor platform. The obtained results should include the number of cycles required to execute an application and the energy consumption of all components of the memory hierarchy. Since caches are notorious for their unpredictable behavior, simulation is one convenient way to study the impact of different memory layouts on the performance. In this design space exploration, the designer should not be restricted by limitations of the used simulation tool, which should thus offer a high degree of flexibility and configurability. Much of the previously published simulators that cover caches do not fulfill this requirement. The properties and limitations of existing cache simulators will be discussed in the following section. After that, some design parameters of the proposed memory hierarchy simulator will be illustrated.

MEMSIM has been used in a number of publications [VWM04a, VWM04c] to produce results for a system with both scratchpad memory and a cache and also to compare these results with the commonly used loop cache architecture.

Related Work

Cache simulation at different levels of abstraction has been widely studied as an approach to provide performance estimates for caches.

One of the most well known and cited cache simulators is Dinero IV [EH]. The main limitation of this simulator is the fact that it supports no notion of time. This makes it difficult to accurately model memories with different access times during the simulation. The only information provided by Dinero is the number of cache accesses, separated into cache hits and cache misses.

Additional efforts would have to be taken in order to determine the timing behavior of the complete memory hierarchy. Also, Dinero is not a functional simulator, meaning that no contents of caches are considered. While this is not a fundamental drawback, examining cache contents can sometimes help to understand the behavior of a cache. Dinero only supports the simulation of caches, other architectures like e.g. loop caches or scratchpad memories are not explicitly covered.

The same author also developed the Tycho cache simulator [HS89]. Beside being somewhat outdated, the available configuration options of the simulated caches are restricted. Tycho accepts as input a trace file consisting of memory accesses. From this memory trace, it evaluates several alternative single processor cache architectures in one simulation run. It is mainly useful to obtain first performance estimates and thus to reduce the size of the search space while the exact evaluation of particular cache configurations is not supported.

The simulator described in [SSR01] serves a somewhat different purpose: it features a model that can be used to determine the performance of an application, taking into account information about the used memory hierarchy. It estimates the number of accesses to the different levels of the memory hierarchy and thus calculates the number of cycles using a Latency-of-Data-Access model. The simulator also supports the modeling of multiprocessor systems, including e.g. an implementation of the MESI protocol. The main drawback of this work is that only data accesses are considered - the instruction memory hierarchy is completely ignored. Also, all data accesses within the used application program have to be instrumented with additional function calls in order to enable counting of cache events, which in general is not acceptable.

The widely used SimpleScalar simulator [Aus, ZKSI03] is a complete system simulator infrastructure that can be used for detailed microarchitecture modeling and hardware-software co-verification. Available extensions allow e.g. power estimation, but this framework is too complex for our projected task of evaluating the impact of the memory hierarchy. Although a cache simulator is available, it can only be configured to simulate a maximum of two levels of instruction- and data caches. Also, only a flat main memory model is supported, meaning that no memory regions with different access characteristics as required for scratchpads can be specified.

The Valgrind memory debugging tool [SNF04] features a cache analysis tool which determines the number of hits and misses within a cache hierarchy consisting of split L1 caches and a unified L2 cache. The configurability of the hierarchy is thus restricted to using just this setup. Also, Valgrind is only capable of analyzing executables for x86-Linux. No notion of memory partitions with different access properties is supported. Valgrind's main purpose is to serve as a debugger for memory related problems in application programs.

RSIM (e.g. used in [PRASA99]) is another popular project that allows simulation of highly complex multiprocessors using instruction level parallelism and out-of-order scheduling. The memory hierarchy, however, is restricted to using a two level cache hierarchy, which does not meet our demands.

An overview over computer architecture simulators, including, but not limited to cache and memory hierarchy simulators, can be found at the Computer Architecture Page at the University of Wisconsin-Madison [XMBH].

We can thus conclude that no simulators that meet the requirements mentioned for our investigations were found. For this reason, MEMSIM was conceived and developed.

Technical Requirements

The following issues were taken into account during the design and implementation phase of MEMSIM. They describe our requirements for the complete system. According to its prime purpose of providing performance and energy values for complex memory hierarchies, MEMSIM has to be:

- Cycle true: Since performance and energy consumption of the memory hierarchy under investigation are the vital characteristics we want to extract using MEMSIM, the simulator has to be cycle-true. If execution times are not modeled precisely, energy and performance will not be determined in a correct way. We chose a cycle-based simulation that is capable of accounting for the delays incurred by the used components.
- Configurable: Since the simulator is to be used to explore the design space of possible memory architectures and hierarchies, there is a need for an easy way to configure and modify the used memory components as well as the connections among them. The parts of the memory hierarchy can be chosen and configured using a graphical user interface which allows to place and connect the components in building-block style. Attributes for the components, like delays or energy consumption can also be set. In order to accelerate the simulation speed and reduce the host's memory requirements, the feature of actually considering the transmitted data is configurable and can be deactivated.
- Capable of handling complex memory hierarchies: Most available simulators only support modeling simple one- or two-level cache hierarchies. Features that are more advanced, e.g. multi-level caches, loop caches and non-cacheable regions are usually not supported, making those tools unsuitable for detailed studies concerning the memory hierarchy. MEMSIM, on the other hand, is capable of simulating a wide range of configurations of the memory hierarchy, given that new components are modeled according to the uniform interface specification.
- Adaptable to different processors: The memory hierarchy simulator should be usable not only for one given processor architecture since the processor plays a secondary role during memory hierarchy design space exploration. Therefore, the simulation of the processor itself is not considered to be part of MEMSIM. Rather, a memory access trace file is used as input, making the memory simulation independent of the underlying processor.

The memory access trace consists of a list of memory accesses performed by the processor during the execution of an actual application benchmark. The processor delays which are implicitly contained in the trace file can then be appropriately overlayed with the delays configured within the components of the memory hierarchy, leading to overall performance results. The energy consumed within the memory hierarchy can also be determined using these timing values. Note that the support of multiprocessor systems is not in the scope of MEMSIM.

Workflow

Due to the integration of MEMSIM into our toolchain, we can evaluate the effect of compiler optimizations on the runtime and the energy consumption of an application assuming a more complex memory hierarchy than was previously possible. Thus, the encc compiler which will be presented in Section 3.6 optimizes the application program with respect to the used memory architecture and MEMSIM subsequently provides the actual performance and energy values when the program is executed on the target hardware. The general workflow is shown in Figure 3.14.

The first step consists of compiling the application program using the encc compiler. Encc also requires some information on the used memories such that the application can be optimized with respect to e.g. a scratchpad present in the system. The resulting executable is simulated using ARMulator as described above, assuming a flat memory model such that all timing delays are caused by the processor, whereas the memory is modeled to require only one cycle per access without additional wait cycles. The tracefile generated

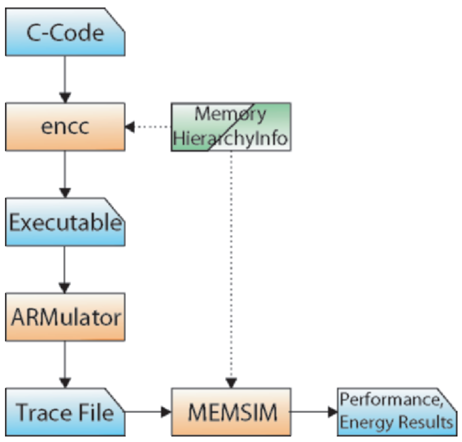


Fig. 3.14. General workflow of encc used in conjunction with MEMSIM

during this ARMulator simulation contains all executed instructions as well as the accesses to the flat main memory. This trace file is then passed on to MEMSIM, which analyzes all memory accesses found in the trace file and processes them according to the provided memory hierarchy information.

The additional delays caused within the memory hierarchy, e.g. by slow main memory accesses or cache misses, are accounted for by the components within MEMSIM. Since the processor's delays are already captured within the input trace file, the memory access times can simply be overlayed with the processor execution time to generate correct values for the overall execution time of an application. The energy models incorporated into the components of the memory hierarchy are used to determine the energy contribution for each of these objects, resulting in an overall energy value for the considered memory architecture. Note that modeling processor energy was not considered part of MEMSIM, since it should stay focused on memory considerations. However, an approximated energy value for the processor energy can easily be determined by appropriately scaling the original CPU energy value determined during processor simulation with ARMulator. Taking into account the configured memory hierarchy, a different cycle count for the execution of the application is determined using MEMSIM. The original CPU energy value can then be scaled with the ratio of these two execution cycle counts. In this way, the average power dissipation of the CPU is assumed to remain constant, while the time during which this power is consumed is varied. This reflects the fact that the memory configuration only changes the timing, but not the functional behavior of the CPU.

The diagonal line in the "Memory Hierarchy Info" box in Figure 3.14 indicates that the number of components supported by encc does not necessarily have to match the number of available components that can be simulated using MEMSIM: it is e.g. possible to optimize the code within the compiler for only one scratchpad memory and then show the effect of adding a cache during simulation. This was done in [VWM04a], where the authors show that a cache does indeed change the situation to an extent that renders the originally beneficial scratchpad optimization useless when a cache is introduced into the system. The mentioned publication also uses MEMSIM to integrate loop buffers into the complete framework and evaluate their benefit compared to caches or a scratchpad.

Simulation Kernel

In order to be able to simulate the parallel activity inherently present in hardware, the simulation kernel distributes a global clock to all components. In every clock cycle, all memory components that have scheduled an activity for that cycle are first identified. In a second step, these components are activated. This two step approach is repeated until there is no more activity in this cycle. This scheme allows for an arbitrary number of actions within one cycle, similar to the δ-cycle concept used e.g. in VHDL [HPH^{+}00, Mar93].

If there is no more activity in a clock cycle, then the global clock cycle counter is incremented and distributed to all components of the memory hierarchy. For the CPU in particular, a new clock cycle and no pending memory access requests means it has to read the next line of the trace file and schedule an appropriate activity if the line represents a memory access.

Except for the global clock, no global communication among the components is used during simulation. Each component solely communicates with its direct neighbors. This allows even complex hierarchies to be simulated without requiring modifications to the simulation kernel itself.

The performance and energy values are also taken care of by the individual components. The energy and delay information is thus kept local to the used components, which helps improve configurability and scalability.

Components

This section describes the concept and implementation of the components that make up MEMSIM's memory hierarchy.

- Components are objects: Each instance of a component of the memory hierarchy is implemented as an object derived from a class describing the properties of that particular kind of component. This guarantees simple instantiation of components as well as uniform interface functions for all components of the same kind.
- Internal state: Each component has to keep track of the state it is currently in using an internal state machine. The component can thus react to a stimulus (e.g. request for data) coming from another component by changing to a "processing" state. In general, it will only be free to accept new requests after the previous stimulus has been properly processed. A cache can e.g. receive a request for data. Upon checking the contents of the tag array, a cache miss is detected. The cache thus changes to the "cache miss" state and issues a request for the missing data to be fetched from the next lower level of the hierarchy. Once the data has arrived, the cache services the initial data request and returns to the "ready" state. This organization together with the two-step approach within the central simulation instance enables MEMSIM to accurately model parallel activity.
- Flow of information: The components communicate with each other directly and do not require the central simulation instance to act as a message passing and routing system. Messages are sent using function calls to the neighboring components. Information is transferred in the parameters of the function call. In order to keep the communication simple, all components only communicate with a neighboring "hub" component which serves mainly as connecting entity. A hub may, however, decide which recipient a function call will go to. The hubs' functionality will be described in the following section.

- Energy and Time: The energy spent by a component as well as the time it takes to service a certain request for data is computed within the component itself. When a component, e.g. a memory is instantiated, the memory's latency and its energy consumption thus has to be set up in order for the memory to show the desired behavior.
- Simple interfaces: All components have similar interfaces which are kept as small and simple as possible. The abstract "component" class provides the minimal interface that all components have to implement in order to be valid memory hierarchy components. Apart from this minimal interface which includes the functionality to accept requests and react accordingly, individual components are free to implement additional functions that they require. As an example, caches may provide functions to query the number of cache hits and misses. This concept allows an easy extension of MEMSIM's capabilities by adding new components.

  Each component only needs to know and exchange information with its direct neighbors. The direction of data requests in general goes to the successor nodes, whereas the actual data is delivered back to the requesting predecessor. Thus, the CPU component usually has no predecessor, whereas memories in general do not have a successor. Usually, data requests contain the address of the requested item so that the queried component can react accordingly. Once the information is available (e.g. after the memory access time of a memory has passed), the successor component notifies the querying component, at the same time transmitting the data element.
- Notion of time: To model time, the global clock has to be passed from the simulation kernel to the components. Since each component is aware of its own delay Δt in servicing a memory request arriving at time t_{now}, it can schedule the reaction to this request for cycle $t_{now} + \Delta t$. Once the internal clock counter has reached this number, it will react and e.g. send the data to the requesting component.

  To ensure a proper coordination, all components are queried for activity in every clock cycle. If a component is not idle, then it is inserted into a list of active components. Once all components have been queried for activity, the list is processed and all components that scheduled an activity in this cycle are activated, allowing the component to e.g. transfer a retrieved data element to its predecessor. This process of querying and activating is repeated until there is no more activity in this cycle. This notion is necessary for those components that have a zero-cycle-delay in their reaction, e.g. components that model buses. These so-called hub components that connect the individual components to form a memory hierarchy may send information on to the recipient in the same cycle, since a transfer along a bus does not necessarily require one full CPU cycle.

Due to the uniform interface, all components used within MEMSIM have a similar basic behavior. The differences in the internal organization of the already implemented components will be described in the following paragraphs.

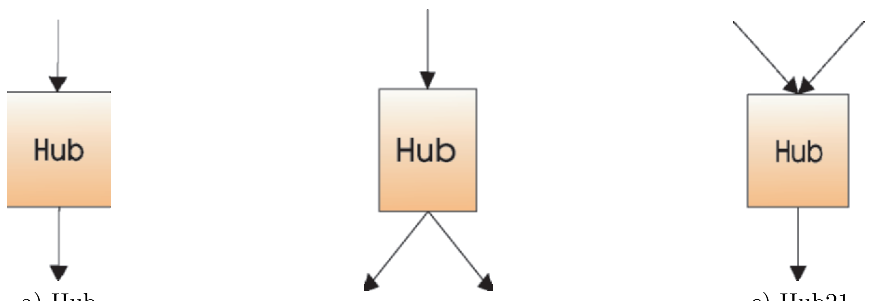
- CPU: MEMSIM currently only supports single processor systems. In each of the simulation cycles, the CPU component reads the next line of the trace file. If it finds a memory access in that line, it schedules an activity. The memory access request is sent to the CPU's successor. The CPU then waits for the request to be processed and remains idle during this time. Only when the successor delivers the requested data does the CPU read the next trace line. The CPU can additionally keep track of the number of executed instructions and the number of issued memory accesses. Since MEMSIM was initially conceived as a memory simulator, we have refrained from also integrating a detailed analysis of the CPU energy dissipation in order to keep the design centered on memory hierarchy evaluation.

  For more sophisticated processor architectures, some modifications may have to be considered. If parallel memory accesses are to be modeled, e.g. assuming completely separate instruction- and data-memories, then the CPU component is required to have two memory bus interfaces instead of just one.
- Cache: Caches are internally made up of several objects, which is due to their origin from another simulation project. To reuse them, it was only necessary to add an appropriate interface for use with MEMSIM. Any communication from the interface is then passed to the actual cache object. The cache is simulated along with the tag bits corresponding to the stored data elements. If a requested element is found to be in the cache, the element is returned, otherwise, a cache miss occurs. This miss is passed to the successor components via the cache interface, which also handles the reception of new elements to fill up the cache. The cache itself counts the cache hits and misses and keeps track of the time and energy spent during cache accesses.
- Memory: In our initial implementation of memory components, the data that is actually being stored is not modeled. As mentioned above, this can be useful in order to reduce the host computer's memory requirements and to increase performance. If simulation of actual memory data is required, e.g. to allow debugging of memory contents, the data within the memory can easily be integrated into the system. The values for access times and energies are always required for a memory. These may be different depending on the bit width of the access, so it is mandatory that the bit width is also taken into account. For each memory access that occurs, the memory accumulates the number of cycles spent in fetching the data and the energy consumed by the access.
- Hubs: Hubs were introduced into MEMSIM to model the energy and delay of connecting buses, which can be significant e.g. for a wide off chip bus connecting the system to a large background memory. In this case, it is possible to actually attribute e.g. the energy consumption to the bus

instead of the memory, which helps to keep the components independent from the way they are connected. Also, hubs help to keep the components' interface simple. Any component only needs a means to communicate to its neighboring hub, regardless of what is connected to it. Finally, hubs also act as address decoders and routers within the MEMSIM environment. If a memory access from the CPU goes to the scratchpad memory, then the hub acts as a decoder and, depending on the address, sends it to the appropriate successor node.

Three different kinds of hubs are required: the simplest hub only has one predecessor and one successor (cf. Figure 3.15 a)). All requests from the predecessor are passed on to the successor. Once the successor delivers the requested information, it is sent back to the predecessor. The next form of hub has two successors (cf. Figure 3.15 b)). It accepts requests from the one predecessor and decides which of the two successors is addressed by a request depending on the accessed address. This facility makes it straightforward to implement a scratchpad or a non-cacheable area in MEMSIM by configuring the hubs accordingly. Finally, the third kind of hub has one successor and two predecessors. While the hub described above can be considered as a split-node, this hub serves as the merge or join node (cf. Figure 3.15 c)) to e.g. model cacheable and non-cacheable areas within one memory.

Using the three types of hubs shown in Figure 3.15, it is possible to construct arbitrary memory hierarchies. If required, the hubs can be arranged in a cascading manner to allow one node to have more than two successors or predecessors. A complex example memory hierarchy using all three hubs is shown in Figure 3.16. The uppermost hub is used to differentiate between cached and uncached regions. Accesses to the cached region go to the right hand side of the figure, all other accesses are passed to the next hub on the left hand side. Depending on the access address, this hub either routes the access to the scratchpad memory, or passes it to the main memory via the lower hub.



a) Hub b) Hub12 c) Hub21

Fig. 3.15. Three different kinds of hubs

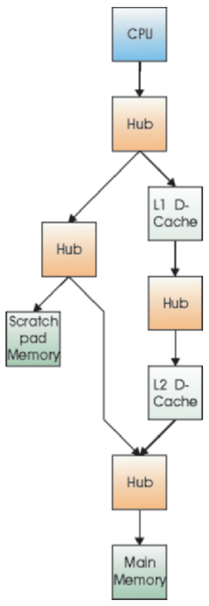


Fig. 3.16. Example memory hierarchy using hubs

Configuration

The memory hierarchy to be used by MEMSIM can be configured using a graphical setup tool that allows the detailed description of components such as the processor, memories, caches and hubs along with their respective attributes like delays and energy consumption. The components can be interactively placed and connected on the screen. The tool allows the user to automatically check for mistakes in the setup (e.g. a hub that is only connected to one other component). If the setup passes this check, it can be saved to a file in XML format. This file includes all information that MEMSIM requires in order to perform a simulation of this hierarchy. The file contents are read using the XERCES library [Apa04] and consequently used to generate the internal data structures that represent the memory hierarchy. This is done in a two-step approach: first, the constructors of all memory hierarchy components are called, followed by the setting of their attributes. Finally, the connections between the components are set up so as to reflect the structure that was graphically entered by the user. When the hierarchy is thus generated, the actual simulation can begin.

3.6 The encc Compiler Framework

All compiler optimizations presented in this work are integrated into the energy aware C compiler encc. The encc compiler framework was developed to generate code for the 16 bit THUMB instruction set of the ARM7 processor. encc is an energy-aware compiler, meaning that beside performance optimizations, it can also take into account the energy dissipation of instructions during code selection by using an energy model provided for the used processor (cf. Section 3.4.2). The initial motivation for the development of the encc compiler was to investigate the energy savings achievable during the instruction selection phase. However, since the ARM processor uses a RISC type instruction set, the choice of instructions or addressing modes in order to perform a certain operation is usually very much restricted. The code generated using energy as cost function was mostly identical to the performance optimization (optimizing for number of cycles) due to the close relationship between energy and time ($E \approx P \cdot t$, cf. Equation 3.10 in Section 3.4.1). By extending the focus of the work to also include the available memory hierarchy on the evaluation board, using energy dissipation as a minimizing cost function produced different results compared to the classical performance optimization: the standard compiler optimization register pipelining [SSWM01] uses a number of free registers to avoid repeated accesses to the main memory. Using this optimization, the energy dissipation of applications was reduced by 17% on average, while the execution time increased by more than 8%.

The main memory was modeled as consisting of static RAM cells because the used evaluation board only contained memories in SRAM technology. Dynamic RAM cells (DRAM, cf. Section 3.2.2), commonly used as large main memory today, have now also been integrated into the encc compiler framework. The power saving features of currently available DRAMs can be used to further improve the energy behavior using compiler supported main memory optimizations. These will be presented in Chapter 5.

The consideration of Flash memories completes the encc compiler framework with respect to the memory technologies used as main memories in current systems. Using the Flash memory commonly found in embedded devices to enable execute-in-place (XIP, cf. Section 5.3) is a promising technique to reduce energy dissipation and, more important, also bring down the production cost due to the reduced main memory requirements.

Apart from main memory optimizations, it is very profitable with respect to both performance and energy dissipation to also consider the compiler supported utilization of other memories available in the memory hierarchy. Since the ARM7 processor used in this work features a 4 kB scratchpad memory (cf. Section 3.1), techniques to utilize this freely addressable memory integrated into the ARM core were developed. First, a static approach that considered moving both instruction and data to the energy efficient scratchpad using the well-known knapsack problem formulation was published in [SWLM02]. To further improve results, in particular compared to a highly dynamic and adaptable cache, a dynamic approach that only copied instructions to and from the scratchpad memory followed [SGW$^+$02]. To complete the picture, a recent publication [VWM04b] uses the encc compiler framework to distribute both instructions and data to the scratchpad memory in a dynamic way.

The effect of using not one, but a number of scratchpad memories for static scratchpad allocation will be presented in Section 4.2, followed by a look at the integration of a DMA unit to efficiently copy data and instructions among the memories.

In addition, the encc compiler framework was used in conjunction with the worst case execution time (WCET) analysis tool aiT from the German company AbsInt GmbH [Abs04b] to study the effect of scratchpad memories on timing prediction and analysis. Results in Section 4.3 show that scratchpad memories are indeed effective in reducing the WCET, with a reduced analysis effort compared to a cache capable of delivering similar average-case performance.

3.6.1 Workflow

Figure 3.17 shows the general workflow employed when using the encc compiler to generate an executable.

The application, written in ANSI C [ANS], is scanned and parsed using the LANCE2 frontend [Leu00, Leu01]. After lexical, syntactical and semantic analysis, standard optimizations like constant folding, constant propagation, common subexpression elimination and dead code elimination are performed on the LANCE2 intermediate representation (IR). The IR can alternatively be written to a file as a low-level three-address-C-code, or it can be represented as a data structure modeling a forest of data flow trees (cf. figure 3.18 a). This latter format is used as the interface between LANCE2 and the encc compiler backend.

Instruction selection within the compiler's backend is performed by a code generator implementing a tree based pattern matching algorithm. The code selector, generated using the olive tool [FHP00], uses a grammar file to transform each of the LANCE2 data flow trees into a sequence of machine

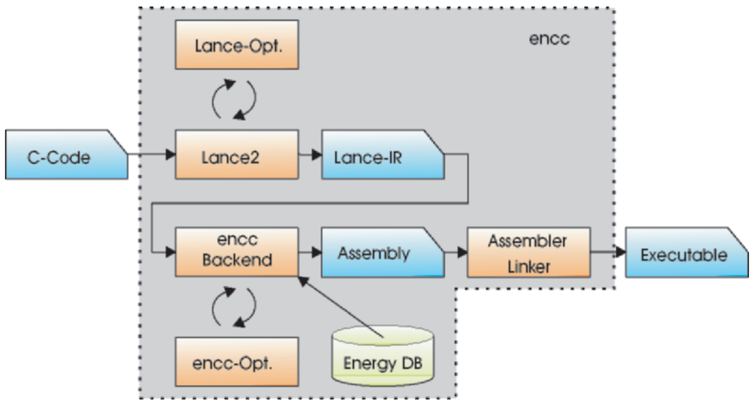
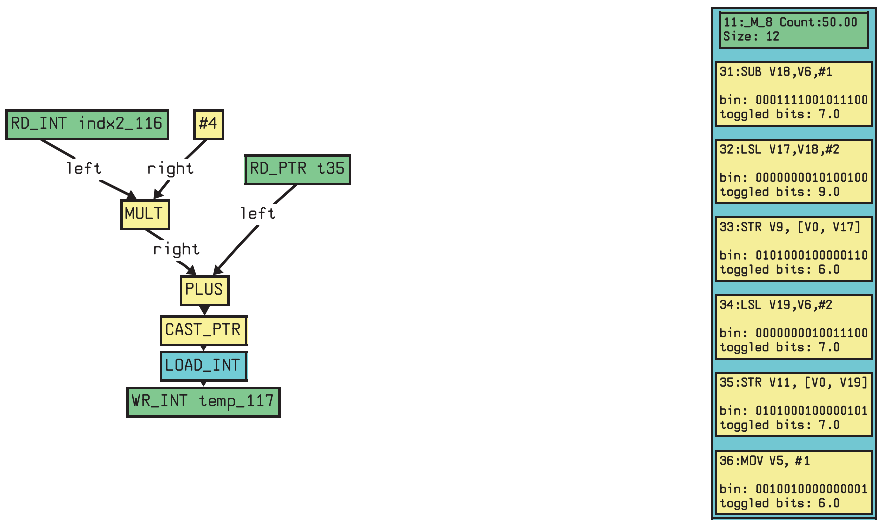


Fig. 3.17. General Workflow using the *encc* compiler



```
11:_M_8 Count:50.00
Size: 12

31:SUB V18,V6,#1

bin: 0001111001011100
toggled bits: 7.0

32:LSL V17,V18,#2

bin: 0000000010100100
toggled bits: 9.0

33:STR V9, [V0, V17]

bin: 0101000100000110
toggled bits: 6.0

34:LSL V19,V6,#2

bin: 0000000010011100
toggled bits: 7.0

35:STR V11, [V0, V19]

bin: 0101000100000101
toggled bits: 7.0

36:MOV V5, #1

bin: 0010010000000001
toggled bits: 6.0
```

Fig. 3.18. a) Example Data Flow Tree, b) example CFG basic block (both figures generated using AbsInt's aisee graph visualization tool)

instructions. The choice of an optimal cover for a particular data flow tree is controlled using a user-supplied flexible cost function, which in encc defaults to the energy dissipation according to the instruction level energy models described in Section 3.4.2. The compiler thus generates energy optimal code sequences. The energy models are made available to the compiler using an energy database containing information about the power consumption of instructions as well as memory accesses.

In this way, the LANCE2 intermediate representation is transformed into a control flow graph (CFG) containing 16 bit THUMB mode assembly instructions. This CFG is the main internal data structure used throughout the encc backend. An example representation of a basic block is shown in figure 3.18 b).

Optimizations concerning to the instructions themselves (e.g. instruction scheduling [Muc97]) as well as optimizations related to the used memory hierarchy are conducted after instruction selection, since most optimizations require knowledge at the assembly level in order to evaluate their pros and cons. Using register pipelining [SSWM01] as an example, the length of the register pipeline has a strong impact on the quality of the generated code: if too many registers are used (which usually improves results), then the necessity to introduce spill code can actually negate any benefits from the optimization. Similarly, most memory related optimizations need to be aware of the exact sequence of memory accesses, which is only true on the assembly level. In the C source code, loading the starting address of an array may not be perceived as a memory load operation. For these reasons, all optimizations described in the following operate on the assembly-level CFG data structure within the encc backend.

Following the assembly code generation process, the tools from the ARM software development toolkit SDT [ARM98a] are used to assemble and link the code into one executable file. The SDT supports a so-called “scatter-loading” mechanism that allows program and data to be distributed to different memory regions. Assuming the executable program is first uploaded to the target device’s Flash memory, a routine is inserted that copies individual program parts to different memory regions upon startup. The specification of target addresses is contained in a scatter-loading file provided to the linker. Scatter-loading is used in all optimizations that take advantage of the different memories’ properties in the system.

The resulting executable can then be simulated e.g. using the ARM instruction set simulator described in Section 3.5.1. The process of simulation and performance and energy evaluation is shown in Figure 3.19. For simplicity, the encc frontend and backend with all optimizations and intermediate steps have been summarized in the node labeled “encc” (cf. the grey box shown in Figure 3.17).

The ARMulator is set up to generate a trace file containing information concerning all executed instruction and all memory accesses, including accessed address and access width. If a memory hierarchy that is not supported by ARMulator is to be simulated, then the more flexible MEMSIM simulator can be used following the initial instruction set simulation as described in Section 3.5.2.

All information supplied in the ARMulator (or MEMSIM) tracefile is then analyzed by the energy and performance profiling tool “enprofiler”, which will be described in-depth in the following section. Enprofiler uses the same energy database also utilized during code generation to determine the overall performance and energy dissipation by summing up the individual contribution of each instruction and also considering memory accesses.

In order to perform memory related optimizations, it is often necessary to obtain knowledge about “hot spots” within the application, consisting of basic blocks within innermost loops which are executed frequently during execution of the application. Optimizations that concentrate on these hot spots have a high chance of substantially improving the code quality of the generated program. Determining these innermost loops can in a first step be obtained by static analysis of the control flow graph: by e.g. assuming each back edge in the CFG to be a loop edge and by assuming a fixed number of iterations for each loop, the innermost loops can be determined in a straightforward way.

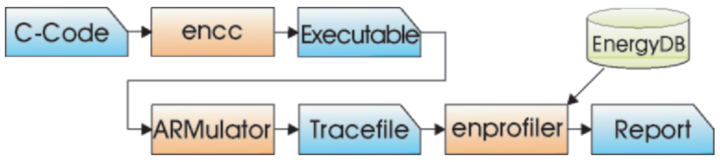


Fig. 3.19. encc-Workflow including enprofiler

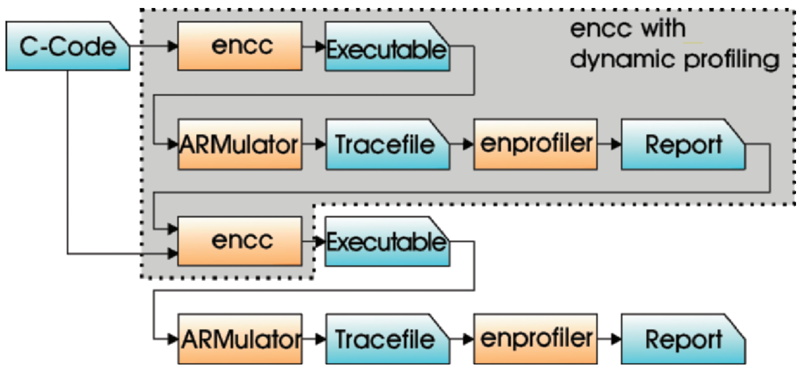


Fig. 3.20. encc-Workflow with dynamic profiling

If two individual loops are at the same level, however, this simple static analysis alone can not decide which of them is executed more often, unless data flow analysis is used to determine the loop bounds. This has been frequently studied, but is out of the scope of this work. Instead, a simulation and profiling run is also capable of providing an exact execution count of each basic block, thus delivering precise profiling data which can be exploited during the optimization phase. This notion of simulating a complete application and back-annotating information from this simulation run into the compiler is known as "dynamic profiling", shown in Figure 3.20: The application is compiled once with only standard optimizations applied, then simulated and analyzed, in general assuming a flat memory model. All analysis results obtained from the enprofiler report file are then fed back into the encc compiler, which uses the information to annotate the control flow graph accordingly. The optimizations within the compiler can use the annotated values to provide better estimates concerning the effects of different possible solutions. The grey box in Figure 3.20 again denotes the encc compiler including dynamic profiling, which will be denoted as the "encc" box in the following figures for simplicity.

Despite being a fairly simple and popular technique, dynamic profiling is not without problems. On one hand, the time required to generate the final executable is increased not only by having to compile twice, but also by the possibly long simulation times. However, for each application, it is only necessary to perform dynamic profiling once and reuse the obtained results in later compiler runs. Care also has to be taken concerning the data values provided for the simulation run of the application: since an application may show a different behavior depending on the input data, a certain dependency between the input data and the analysis and optimization results may be introduced by dynamic profiling. In order to overcome this effect, typical input data should be used which has been proven to generate a realistic behavior for the application. Also, several input data sets can be used and the average determined from these profiling runs may be used in the actual optimization. This of course further increases compilation times. If guarantees concerning execution times have to be provided, worst case execution time analysis techniques (cf. Section 4.3.1 for an overview) should be used instead of simulation.

3.6.2 enprofiler

In order to perform dynamic profiling (as described in Section 3.6.1 above) and also to evaluate the effects of the optimizations that will be presented in the following sections, a means of measuring an application's overall energy consumption at runtime is required. Since the original physical measurement of the energy consumption of instructions and memory accesses is very time consuming, it should not have to be repeated for every application. Therefore, the energy models described in Sections 3.4 are not only being used during code generation to guide the compiler towards energy-saving optimizations, but the energy model is also used to validate the effect of energy optimizations at execution time.

This is done by simulating the executable generated by the encc compiler using the ARMulator instruction set simulator according to the workflow shown in Figures 3.19 or Figure 3.20 in the previous section. ARMulator generates a tracefile which includes the sequence of all executed instructions as well as all memory accesses. A short sequence of an example trace file is shown in Figure 3.13 in Section 3.5.1 on Page 65.

The trace file contains information on all taken and skipped instructions, as well as all memory accesses, including information concerning the bit width, read or write access and whether an opcode fetch was performed. This trace is then fed into the enprofiler tool, which accumulates the time and energy contributions of all instructions and all memory accesses using the timing and energy models presented in Sections 3.3 and 3.4, respectively. The analysis is performed by first considering an instruction fetch line, which is always the first occurrence of a potentially executed instruction in the ARMulator trace. Execution cycles and possible memory accesses of this instruction are connected with this instruction and summed up to determine this instruction's contribution to the overall cycle time and energy dissipation. If DRAM memories are used, enprofiler first has to determine the fraction of time spent in the different operating states (cf. Section 3.4.3), which allows it to determine a value not only for the access-related energy, but also for the standby power. In this way, the performance (in terms of number of cycles) and energy consumption of an entire application can be determined. Since the tracefile also contains information about the addresses of memory accesses, it can distinguish e.g. between an access to the main memory and accesses to the energy efficient scratchpad memory. By using different energy models within enprofiler, it is possible to also consider the effect of different memory sizes or even completely different technologies, as described in section 3.4.3. Beside the total number of cycles and the overall energy consumption, enprofiler provides a large amount of information collected during the execution of the application, mainly:

- used memory areas in the corresponding memory partitions
- number of executions, processor and memory energy for each function and for each basic block
- execution schemes for all basic block combinations (e.g. how often BB2 was executed following BB1)
- number of accesses to variables and related energy for each basic block
- total energy dissipated in accessing variables
- number of executed instructions
- number of CPU cycles
- energy consumed by the CPU, memory and total
- hamming distances and additional ones-cost

Some of this information is of direct interest to the programmer or the designer of an embedded device, like the energy dissipated or the number of executed instructions. Other information is mainly collected for use in dynamic profiling. In this case, information is back-annotated from a first simulation run to the internal compiler data structures in order to provide additional information. Knowledge about the number of times that certain basic blocks are executed in direct succession can e.g. be used to add a weight to the edges of the control flow graph. These edge weights express how close the relationship between two basic blocks is. Another relationship, namely which variable is accessed from within which basic block, will be exploited in Section 5.2.

Enprofiler also supports the analysis of caches. In this case, the cache energy and the time it takes to e.g. service a cache miss is also stored along with each instruction. In the end, hit/miss ratios as well as the energy dissipation within the caches are also reported.

To improve the configurability of enprofiler, the entire analyzer uses an inherently object-oriented design. This makes it easier to determine which part of the program is responsible for handling a specific input trace line. Also, the concept of a “TraceConverter” was adopted which reads the specific ARMulator trace file format and stored the information in a data structure that is subsequently used by enprofiler. This decoupling of the trace file format from the information is vital to obtain flexibility concerning new trace file formats: if a new format is to be supported, only the TraceConverter class needs to be adjusted, and if the same amount of information is provided by the new trace file format, then the analysis of enprofiler does not have to be modified to adapt to the new input format.

3.6.3 Memory Architecture Aware Compilation

Optimizing for minimal execution time of the generated program is one of the most popular and intuitive optimization criteria. Apart from the recognized and widely accepted necessity to generate performance efficient code, minimizing the memory requirements for storing the executable is also a common optimization goal. In particular for embedded portable systems, the available memory is restricted, making it necessary not to waste any space. The notion of optimizing code to meet other requirements than the two mentioned criteria is not as common. One such potential optimization goal that is increasingly being studied is the energy dissipation of an application, which can be influence to some degree by the compiler. In particular for the simple and efficient RISC processor architectures often found in embedded devices, research results have shown that code selection alone does not provide sufficient potential to save energy [CKI$^+$01]. Since the memory hierarchy has been found to consume a large percentage of a device's overall energy [KG97, KVIY00], extending the compiler to also consider optimizations with respect to the memory architecture is a logical step. Since the compiler directly determines the number and the kind of memory accesses in the final program, it does have a high degree of control over how the available memory is used. By e.g. modifying the data layout, the code generator directly influences the memory access behavior of the application. By providing information to the compiler concerning the memory architecture, optimizations can be integrated into the code generation process that help reduce the energy dissipation and thus increase the standby times of the battery for handheld portable devices.

Beside exploiting a given memory hierarchy by passing information to the compiler and generating code accordingly, the code generation process can also be used to perform a design space exploration of one or more memory hierarchy parameters. The compiler is given an initial set of memory parameters and generates code taking this information into account. The code is then simulated, assuming the same memory setup that was also used to generate the code. The performance results of this simulation run are recorded. Then, the parameter under observation is varied, and the process of code generation, simulation and evaluation is repeated. In this way, it is possible to traverse parts of the possible design space and determine a suitable memory setup for the given application.

If accurate models of the design space to be explored are available within the compiler, the effects of varying parameters may even be estimated within the compiler's cost functions without actually having to generate and simulate code. In practice, however, combinations of several parameters may produce unpredictable results, making accurate models that capture the system behavior in an appropriate way difficult to obtain. Caches with their dynamic run time behavior are one example for the difficulties with this approach. In other cases, however, the compiler may be able to suggest design parameters due to its detailed knowledge during code generation: by e.g. analyzing the maximum register pressure observed during execution of a program, the compiler can without additional effort determine the number of required registers, an approach that will be described in Section 6.6.

As discussed above, the encc compiler framework is capable of using energy as a cost function as well as considering the used memory architecture to generate code that optimally exploits the given memories. Before presenting a selection of optimizations and obtained results in the further course of this work, this section first provides an overview over a superset of possible memory design parameters and how they can be exploited by the compiler. Knowledge about these parameters may analyzed by the compiler in order to generate optimized code for the considered memory architecture parameter settings. On the other hand, the presented parameters may also be actively modified by the compiler, depending on either the results of simulation runs or static analysis results. The following list of memory hierarchy parameters mainly covers those aspects that the compiler can have an influence on in either of these two ways:

- Memories:
  - Memory Size: Minimizing memory requirements is one of the traditional optimizations integrated into compilers, usually called "optimization for code size". Additionally, the compiler can analyze the used data structures and their access patterns and thus determine the minimum required memory size during execution of the application if only statically objects are used. In addition to the overall size, it can also determine the most frequently accessed elements and can thus propose an adequate size for the scratchpad memory size that should be present in the system in order to provide a tradeoff between performance, energy and area.
  - Memory Bit Width (Bus Width):
    The bit width of the memories used in the system can make the use of data types with a certain number of bits preferable to others. If e.g. an access to main memory can fetch a maximum of 16 bits within one cycle, then 32 bit data types should be avoided to improve application performance. On the other hand, the bit width is usually fixed in the application, since knowledge about the possible value ranges is necessary to the number of bits for a variable. Using data-flow analyses or even bit-level analysis [WL02b], the compiler can give hints on variables that only use a fraction of their possible values and could thus be reduced in length.
  - Memory Banks: Due to their increasing size, modern memories are increasingly subdivided into banks, which allow the compiler to optimize several aspects. On one hand, switching among the different banks during subsequent accesses can help to hide latencies that occur during addressing: the next desired address can already be decoded in another bank while the previous bank is still being accessed. On the other hand, modern memories support power down modes, usually at the granularity of banks. If the data and instructions can be reorganized in such a way that one bank of the memory is not accessed for a longer

period of time, then this bank can be switched off to reduce its standby energy dissipation. For the frequently used DRAM memories, this will be discussed in Chapter 5.

- Burst accesses: In particular instruction memories are often accessed in a sequential manner, i.e. the addresses only need to be incremented to access the next instruction. Consequently, memories offer a burst access mode: in a burst access, the first element is fetched in a random access. The following data can be read at a higher speed, since an internal counter can increment the address (cf. Section 3.3.2 for details). The compiler can exploit burst mode accesses e.g. by arranging instructions in the order they will most probably be accessed in. This can be achieved by using so-called traces [TY96], consisting of maximum straight-line sequences of instructions without any interfering control flow.
- Memory Timing: The timing of the used memories is a crucial element for the behavior of the entire system. If the processor frequently has to wait for instructions or data from the memory, the memory subsystem becomes the bottleneck of the entire system. If faster and slower memories are available, then the compiler should allocate as much data and instructions as possible to the fastest memory. Since fast memories are usually restricted in their capacity, the allocation can be formulated as an instance of the well-known knapsack problem [Sed98].
- Dedicated Memory Regions (memory mapped I/O, hardware registers, cacheable vs. non-cacheable memory): These memory regions require special treatment by the compiler and thus have to be known for proper code generation. In the case of memory-mapped I/O, the used variables should be declared as "volatile" in the C source code, since their value may change at anytime if they are used to implement input routines. For memory-mapped hardware registers, it may be necessary to execute certain instruction sequences before the results are valid. Finally, the compiler should know which parts of the memory regions are cached and which are not in order to distribute memory objects accordingly.
- Energy Dissipation: As previously mentioned, the energy dissipation within the memory subsystem makes up a large fraction of the total system energy [KG97, KVIY00]. If memory regions with different energy consumption are available, then the compiler should choose to allocate as many memory objects as possible to the most energy efficient memory. Since efficient memories are usually small, an extension of the knapsack problem can be used to solve this class of problems. The corresponding algorithms and results have been presented in [SWLM02] and are summarized in Section 4.1.
- Partitioned Memories: The fact that the access times and energy dissipation per access increase with the size of the used memory (cf. Figure 2.2 in the Introduction) leads to the idea of partitioning one memory into several smaller ones to improve both access times and

  energy dissipation. Care must be taken, however, to also consider the overhead associated with additional memories integrated into a system: data and address buses have to be inserted, accesses to the memories have to be controlled and distributed by a decoder. Additionally, every memory requires a certain amount of energy even when it is not accessed. These issues, as well as the distribution of memory objects to a set of partitioned memories, will be presented in Section 4.2.
  - Background Power: Beside the “energy per access”, a compiler should also be aware of the background power that is always required if a certain memory is present in the system. By not using a memory and shutting it down completely, energy dissipation may be reduced significantly. The compiler should thus also be aware of the background power of the used memories, as already explained for DRAM memories in Section 3.4.3.
  - Read-Only Memories: Usually, memories that can not be written to are only used in embedded devices to store that part of the application or the operating system that is not bound to change, e.g. the bootloader. For all other parts of the software, Flash memories are now being used, which only provide a limited number of write accesses. This makes them unsuitable for use as data memories. However, instructions are generally only fetched and never written. Reading a Flash memory is relatively fast and energy efficient. Therefore, Flash memories can be used in an embedded system to store instructions. This execute-in-place (XIP) concept will be presented in Section 5.3.
- Caches:
  - Cache Size: By analyzing data and instructions within the application program, the compiler can determine the size of the maximum working set and thus provide information concerning the required cache size.
  - Line Size: The number of words in one line of the cache influences the behavior of the cache by changing the granularity of considered address ranges: if long long cache lines are being used, then a cache hit can be sustained for more elements compared to a shorter line size. On the other hand, allocating such a long line to the cache causes additional overhead and is only beneficial if all elements of the line are actually accessed by the application. In case of unfavorable decisions concerning data eviction, long cache lines aggravate the situation.
    The influence of both cache and line size on the performance has been investigated e.g. in [PDN97, PDN99a] where the authors propose an iterative algorithm to find the cache size that optimizes performance.
  - Associativity: The compiler can exploit knowledge about the associativity of a cache obtained during the linking phase: data or instructions that are part of the same working set should be mapped to addresses that do not evict each other from the cache. If this can not be achieved, the compiler should propose a higher associativity for the cache.

  - Write Strategy (Write Through vs. Write Back): Assuming a write-back strategy, code can be generated in such a way that a maximum number of write operations is performed on one cache line before it is evicted and written back to memory. If the program is found to contain a large number of sequential write operations, then the compiler can propose to use a write-back instead of write through strategy.
  - Line Allocation on Write Miss: The allocation of a new cache line following a a write miss is only useful if the line is subsequently read, which can be determined by the compiler.
  - Cache Line Fill Strategy: Fetching the critical word first is usually a good practice, since the processor is stalled until this element arrives. Whether or not the entire line is fetched can be decided depending on the access pattern of the application.
  - Dynamically Configurable Caches: From the compiler's point of view, a dynamically reconfigurable cache represents a good choice: the compiler can control the cache behavior by adding the corresponding commands to reconfigure the cache to the generated program. Reconfigurable caches have been a target of research, with configurable cache size, associativity and line size. Dynamic way-management in caches is a popular approach [Alb99] to trade off performance against energy. Dynamically configurable caches are also available commercially: the Motorola M*CORE M340 processor can be configured concerning memory writes, way management and store buffers [MMC00]. However, while dynamic reconfigurability is capable of improving performance, the impact on energy has to be considered carefully, since the configurability requires additional circuitry in the caches, which can potentially overcompensate the performance gains.

  Note that for some of the mentioned cache optimizations, the final addresses of instructions and data elements generally need to be known in order to determine if data elements will be mapped to the same cache line or not. Dynamic profiling is capable of transferring this information from the linker back to the compiler. Care must be taken however, since code modifications can change the program address layout to a degree where no previous information can be reused.

  The parameters with the highest impact that will be worthwhile considering are the cache size, the cache line size and the associativity.

- Registers:
  - Total Number of Registers: The number and kind of registers is an important input data for the register allocation phase within the compiler. During register allocation, the virtual registers used during instruction selection and much of the remaining code generation process need to be mapped to the actually available physical registers of the processors. Since the number of virtual registers may be very large, depending on the application, the mapping process is an NP-hard task usually solved by graph coloring algorithms. The register allocation algorithm

is crucial for the quality of the entire code since additional spill instructions have to be inserted if the registers are not used efficiently. Looking at compiler guided architectural exploration, reducing the size of the register file size according to the register requirements of the application can help decrease the energy dissipation of the register file, which is a large percentage of the entire data path [SLAM98]. The results of this approach will be presented in Chapter 6.

– Bit Width of Registers: The bit width of the registers, which usually also defines the bit width of the data path, is a parameter affecting the entire processor and also the memory subsystem. As already mentioned under "Memory Band Width", if many bits are not used during actual computations, then a lot of energy may be lost. In general, the compiler is aware of the given bit width of the data path and generates code accordingly. It may additionally perform analyses to determine possible value ranges for registers in order to minimize the used bits. One approach to solve the problem of "wasted bits" is the Valen-C approach developed by Yasuura et al. [YTIE97]. In their work, the programmer annotates information concerning the bit width of variables to the source code of the application. The generation of an efficient architecture and code is then done by the Valen-C compiler.

4 Scratchpad Memory Optimizations

It is a well-known fact that one of the main contributors to execution time as well as energy consumption, beside the data path, is the memory used to store instructions and data [CFW+94, KVIY00, KG97]. Furthermore, the performance gap between processors on one side and the memory subsystem on the other side has been widening significantly. Since the 1980s, processor speed has improved by 50-100% per year, whereas DRAM speed has only increased by about 7% annually [HP03]. Figure 4.1 shows that the gap between CPU and memory performance is widening in an exponential way: assuming the above-mentioned values, the gap is doubled every two years. This will eventually lead to what the authors of [WM95] call the "memory wall": the performance of a system will only depend on the memory system, since the memories will be unable to provide data and instructions at the pace required by the processor.

In order to narrow the gap between current CPU and memory speeds, memory systems today generally form a hierarchy so that accesses to bigger, slower memories can be avoided if the data is available in a smaller, faster memory. This approach also helps to improve energy dissipation. Small memories and in particular caches are therefore being placed close to the processor. Caches, however, have a number of drawbacks in embedded systems. Onchip caches require a large amount of chip area since beside the actual memory, they also need a tag memory and the comparison logic [HP03]. Additionally, the authors of [WM95] claim that even the relatively few compulsory misses in caches will lead to performance problems. Additionally, caches are not suitable for real-time capable embedded systems, an issue that will be discussed in Chapter 4.3.

As an alternative to caches, the use of scratchpad memories has been proposed. Scratchpads are small onchip memories that are freely addressable and can in general be used to store data and/or instructions. They require less space than a cache since only the actual memory array is required [BSL+02]. Since no automatic hardware controlled allocation of memory objects to the scratchpad is done at runtime, it is left to the user or the compiler to find a

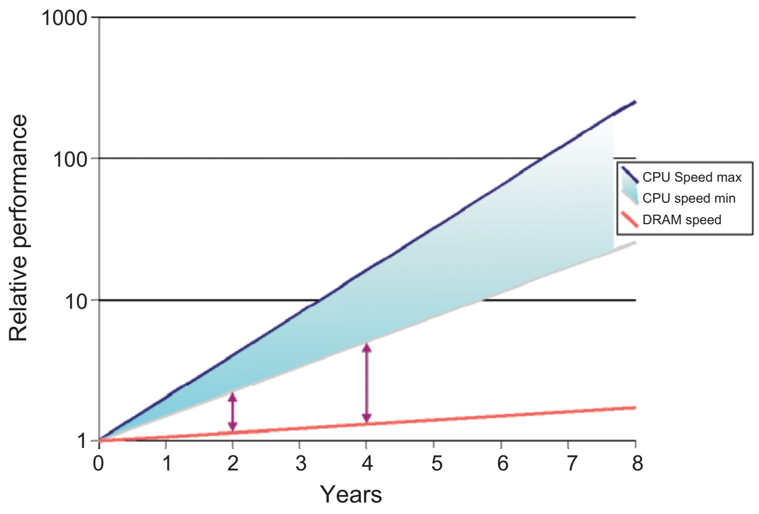


Fig. 4.1. Increasing gap between CPU and DRAM performance (semi-logarithmic scale) [Mac02]

suitable mapping of memory objects to the scratchpad space. Using its knowledge of iteration spaces of loops and memory access patterns, the compiler can decide which elements are most frequently used and should consequently be allocated to the scratchpad memory. Due to their small size, scratchpad memories can be accessed very quickly (usually within one processor cycle) and since they are close to the processor, they are also very energy efficient as long bus lines are avoided. For a detailed description of scratchpad memories' architectural features, please refer to the sections on SRAM in Chapter 3.

This chapter first takes a look at previous and related work concerning the use of scratchpad memories. The static scratchpad allocation algorithm that considers both instructions and data [SWLM02] is described in detail since it forms the basis of the multi memory allocation for partitioned scratchpad memories presented in Section 4.2. The initial and rather simple "Base" model is refined in the subsequent sections to consider dependencies between memory objects as well as the current ARM TCM architecture and the standby leakage current of scratchpad memories to obtain a compiler guided memory configuration. The results obtained using these models are presented in Section 4.2.9. Finally, in Section 4.3, the influence of using a scratchpad memory on the worst case execution time (WCET) is presented.

4.1 Related Work

Despite the fact that scratchpad memories are generally accepted as efficient and beneficial alternatives to the prevailing cache architectures commonly found in desktop processors, a full-featured toolchain for the efficient utilization of scratchpad memories is still missing in industry today. There has been a lot of research work, which can in general be classified according to the following three dimensions:

- the kind of memory objects that are allocated to the scratchpad memory: either instructions, or data, or both instructions and data
- the allocation strategy: static or dynamic allocation. Using a static allocation, the contents of the scratchpad do not change at runtime, while a dynamic allocation in general inserts instructions to swap scratchpad contents at runtime
- fixed memory architecture: some approaches are based on modifications of the used hardware

Most of the previous work on the allocation of scratchpad memories deals with assigning data elements to the scratchpad. The most obvious candidates are frequently accessed arrays, e.g. in innermost loops of an application. Panda, Dutt and Nicolau [PDN99b] consider an architecture with both a cache and a scratchpad memory. In this way, elements that are too large to fit in the scratchpad are accessed via the cache to avoid the performance penalty of the slow main memory. In [VSM03], the authors propose a different approach for large arrays by splitting them at an optimal splitting point such that at least the most frequently accessed fraction of the large array can be allocated to the scratchpad.

Alternatively, the scratchpad memory has also been used as a storing location for spilled register contents: in [CH98], the scratchpad used for this purpose is called “compiler controlled memory”. To reduce the required scratchpad space, values with non-overlapping lifetimes are assigned the same address.

The authors of [KKS01] use Presburger formulas to determine which set of data should be kept in the scratchpad memory. In contrast to the aforementioned work, they not only consider a static allocation of elements to the different levels of the memory hierarchy, but also consider copying data elements from e.g. main memory to the scratchpad at runtime.

Several views are put into perspective in the publication [PCD$^+$01]: beside source-to-source optimizations, the application specific synthesis of suitable memory hierarchies is also considered. Despite the fact that scratchpad memories and caches are mentioned, no real methodology for the exploitation of scratchpad memories is presented.

The first approach to assign both data and instructions to the scratchpad memory in a static way was presented in [SWLM02]. It will be described in depth later in this section since it forms the basis for the multi memory allocation presented in Section 4.2. The same authors also considered only copying instructions dynamically at runtime [SGW$^+$02]. This lead to improvements in particular for small scratchpad sizes which are not able to hold all hot spots of an application at the same time. Despite the fact that the allocation and deallocation of instructions to the scratchpad memory is done at runtime, all decisions are taken and fixed at compile time, resulting in an inherent predictability even of the dynamic scratchpad algorithm, an aspect that will be discussed in Chapter 4.3.

Until recently, no work considered the dynamic allocation of both instructions and data to a scratchpad memory. A relatively new paper [VWM04b] for the first time made this most flexible and versatile allocation possible. It uses a technique based on a register allocation algorithm for CISC architectures using flow equations to allocate both instructions and data to the scratchpad in a dynamic way. Considerable savings of up to 38% concerning energy consumption compared to a static approach are reported.

The multi memory allocation technique presented in the following section is an extension of the static single scratchpad allocation algorithm presented in [SWLM02]. The allocation is performed in a static way, meaning that the contents of the scratchpad never change during the runtime of the application. The objects which are to be allocated to the scratchpad memory may consist of functions, basic blocks and global variables. Since local variables are usually kept in registers or on the stack, they are not considered in the basic approach. The allocation algorithm uses the encc compiler described in Section 3.6 to utilize the scratchpad found in the ARM7 processor. It uses an integer linear programming (ILP) representation to formulate a variant of the well-known knapsack problem [Sed98] in order to find an optimal allocation of memory objects to the scratchpad memory.

To model the energy dissipation of a global data object v, the number of accesses $\#acc(v)$ to this object as well as the energy required to access one element of data $E_{data}(mem)$ is required, where mem specifies which memory the access goes to. The data energy can thus be expressed as

$$E(v, mem) := \#acc(v) \cdot E_{data}(mem) \tag{4.1}$$

For instructions, the situation is somewhat more difficult since functions and basic blocks are not independent. Rather, functions consist of basic blocks. Functions executed on the ARM processor are called using the branch link “BL” instruction, execution starts at their beginning and terminates with a return instruction. Therefore, functions can be handled as self-contained memory objects which do not require any further modification when allocated to the scratchpad memory. The energy dissipation of a function f is determined using the number of executions $\#exec(k)$ of each instruction k and the energy consumption of a single instruction fetch E_{ifetch} from memory mem as follows:

$$E(f, mem) := \sum_k \#exec(k) \cdot E_{ifetch}(mem) \tag{4.2}$$

The number of executions of each instruction of the function can be determined at compile time either using static analysis or dynamic profiling, as described in Section 3.6.1.

Functions are composed of several basic blocks. Considering these basic blocks for scratchpad allocation incurs an additional overhead: If basic blocks that are executed sequentially are allocated to different memories, then an additional jump has to be inserted to jump to the other memory, shown in Figure 4.2 by the bold control flow edges. This energy $E_{longjump}(mem)$ caused by the longjump instruction has to be considered in the ILP formulation as an overhead.

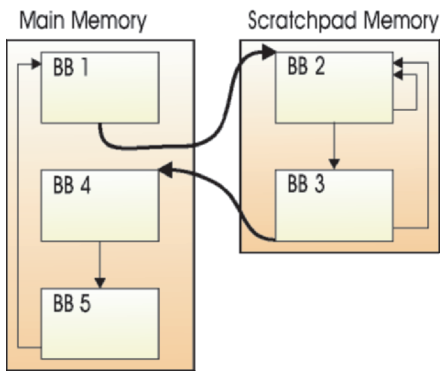


Fig. 4.2. Allocation of basic blocks to main and scratchpad memory

The energy dissipation of a basic block *bb* is given by the number of instructions $n(bb)$ within this basic block, the number of executions $\#exec(bb)$, the energy required for a single instruction fetch $E_{ifetch}(mem)$ and $l(bb)$ jumps between the memories:

$$E(bb, mem) := \#exec(bb) \cdot n(bb) \cdot E_{ifetch}(mem) + l(bb) \cdot E_{longjump}(mem) \tag{4.3}$$

With the energy dissipation of instructions and data thus formulated, the implications of storing these memory objects in different memories can be expressed by forming the difference between the energy required to read an object from either the main memory or the scratchpad memory. For an arbitrary memory object x, this value EB represents the resulting energy benefit if the considered element is allocated to the scratchpad instead of main memory:

$$EB(x) := E(x, MainMemory) - E(x, Scratchpad) \tag{4.4}$$

The binary decision variables required for the ILP formulation are defined as

$$m(x) := \begin{cases} 1, & \text{if } x \text{ is moved to the scratchpad} \\ 0, & \text{otherwise} \end{cases} \tag{4.5}$$

Using the sets of all variables V, all functions F and the set of all basic blocks BB, the optimization problem can be formulated as follows:

$$\begin{aligned} \text{Maximize} \quad & \sum_{f \in F} m(f) \cdot EB(f) + \\ & \sum_{bb \in BB} m(bb) \cdot EB(bb) + \\ & \sum_{v \in V} m(v) \cdot EB(v) \end{aligned} \tag{4.6}$$

In general, it will be profitable to place as many objects as possible onto the efficient scratchpad memory. However, due to its small size, space constraints have to be respected in the ILP problem's constraints. $Size(x)$ therefore denotes the size of memory object x in bytes, and the optimization has to be performed subject to the size constraint:

$$\begin{aligned} & \sum_{f \in F} m(f) \cdot Size(f) + \\ & \sum_{bb \in BB} m(bb) \cdot Size(bb) + \\ & \sum_{v \in V} m(v) \cdot Size(v) \quad \leq \quad Scratchpadsize \end{aligned} \tag{4.7}$$

In addition, it has to be ensured that either a function or its comprising basic blocks are allocated to the scratchpad. This can be achieved using the following constraint:

$$\forall bb \in BB, bb \text{ is contained in } f : m(bb) + m(f) \leq 1 \tag{4.8}$$

This version of the ILP formulation does not yet consider the benefit from moving contiguous basic blocks to the scratchpad: if two subsequent blocks are moved, there is no need to jump between the memories, since execution can simply continue in the scratchpad memory.

In order to model the advantage obtained from moving contiguous basic blocks together, [SWLM02] explicitly enumerates all possible combinations of sequential basic blocks and includes them in the equations as the set of so-called multi basic blocks. The Bottom-Up model in Section 4.2.6 will present a more elegant way to handle multi basic blocks.

Feeding the linear equation system into an ILP solver results in a solution that specifies the set of memory objects that should be allocated to the scratchpad memory. This information is then used to modify the program representation within the compiler accordingly, e.g. address modifications and inserting all required jumps to maintain correct control flow among the basic blocks.

Results for this method indicate that the overall energy dissipation of the considered system can be reduced by 12% to 43% with an average of 23% compared to a unified 4-way set associative cache of the same capacity as the scratchpad [SWLM02]. Compared to a system comprising only a main memory, savings of up to 75% are reported when a scratchpad is utilized according to the above approach [Ste03].

In the following section, this static scratchpad allocation technique will be extended to allocate memory objects to multiple scratchpad partitions. The motivation behind considering partitioned memories is the following: since larger memories require more energy and longer access times (cf. Figure 4.3), it is beneficial to use several smaller memories, each of which is fast and requires less energy per access. Partitioning large memories to exploit the efficiency of smaller memories has been considered by a number of researchers:

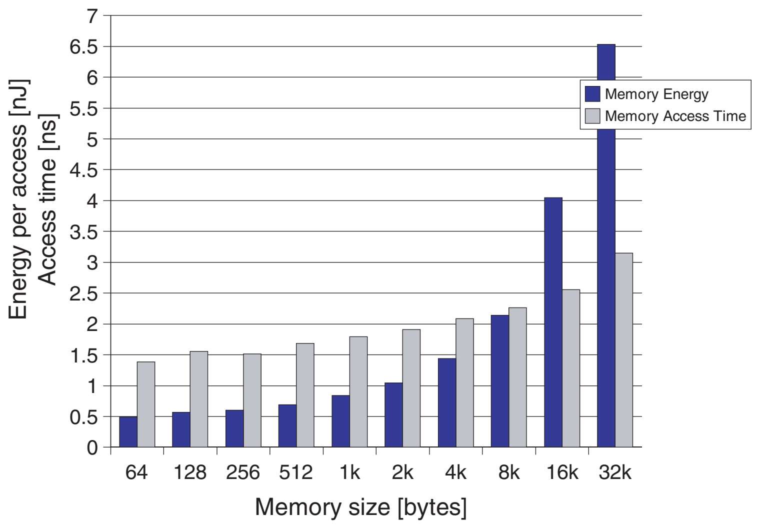


Fig. 4.3. Energy and time per access for increasing memory sizes

Nachtergaele et al. [NCB$^+$95] propose a memory partitioning that is closely fitted to a particular application, a medical image analysis program. They found that by partitioning the image data and allocating it to two different memory partitions, the memory energy dissipation can be reduced significantly, however at the cost of increased complexity in the data access scheme.

Partitioning a data cache and only using the active part of the cache at a time is discussed in [GAV95]. The authors decide which data to place in which partition of the cache by splitting the data according to spatial or temporal locality. One problem with this approach involves the decision to appropriately activate and deactivate cache partitions.

The authors of [PMP04] deal with multi-processor SOCs and consider partitioned memories as an energy efficient alternative to the multi port memories commonly found in multi-processor systems. The address space of the application is partitioned and mapped to a multi-bank memory architecture. Energy savings of about 56% and slight performance penalties are reported compared to multi port memories.

Address clustering is used in [MMP03] to increase the locality of memory access profiles and thus to improve the efficiency of partitioning. This clustering leads to average improvements of 25% compared to a partitioned memory architecture synthesized without address clustering.

The authors of [BMP00] propose the partitioning of onchip SRAM memories into smaller independent banks that are used to hold data objects. The memory is synthesized and allocated according to the results of a dynamic execution profile, leading to energy savings of 42%. A similar approach is presented in [ABC03]. It uses dynamic programming and achieves polynomial execution time. Since partitioning overhead is considered, the user does not have to provide an upper bound for the number of partitions.

A recent PhD thesis [His05] presents algorithms for partitioned memory hierarchies in embedded systems that allow the designer to study the impact of particular memories on the system without having to simulate the design. Compiler algorithms take care of memory object allocation to the partitions using a greedy algorithm. However, as in most of the previous work, only data is considered.

[UNS02] solves a problem that is closely related to the multi memory allocation taking into account memory leakage energy, which will be presented in Section 4.2.8. Given a collection of available memories including energy and capacity data, the algorithm decides which subset of memories should be used and how data elements should be distributed in order to minimize overall memory energy dissipation. Our approach extends the considerations to not only allocate data, but also instructions to the partitioned memories.

The idea of allocating both instructions and data to partitioned memories using a compiler was first described in [Hel04], and a description of the “Bottom-Up” model was subsequently published in [WHM04]. That work is extended here by providing a comparison among the different allocation approaches, presenting measures for the size of the generated ILP problems and by two extensions of the general allocation model.

In the following section, the general idea and approach of multi memory optimization is described, including the definition of required sets and variables to formulate an ILP problem to solve the allocation problem. An initial ILP formulation is then provided in form of the Base model, which solves the multi memory allocation problem by allocating global variables and individual basic blocks to partitioned scratchpad memories. However, it does not take into account the relationship between functions and their comprising basic blocks, which in general leads to suboptimal results. The Top-Down model, which is described in the subsequent section, allows the allocation of either a complete function or an arbitrary number of basic blocks contained in that function to any scratchpad partition. This model improves the achievable results by considering the advantage of allocating complete functions instead of separate basic blocks. Finally, the Bottom-Up model is described, which in addition to considering complete functions also takes into account the benefit of moving contiguous basic blocks to a particular memory. Following the description of the mentioned approaches, extensions to also allocate memory objects to the Harvard-style ARM TCM architecture and the notion of using the compiler to determine a suitable memory architecture configuration are presented. Results for all described models are presented and compared. Also, the size of the corresponding ILP systems in terms of number of constraints and number of decision variables is given.

4.2 Multi Memory Optimization

The scratchpad allocation algorithm [SWLM02] presented in the previous section can be extended to support not only one available scratchpad memory, but to partition the memory space into a number of smaller parts in order to enable further energy savings. Since smaller memories generally require less energy per access (cf. Figure 4.3), reducing the used memories' size and finding a way to allocate instructions and data to them will result in extra savings. The fact that smaller memories also require less time for an access has not been considered, since scratchpad memories are typically capable of delivering values within one single processor clock cycle. If larger scratchpad memories are assumed to introduce additional wait states during which the processor has to be stalled, then the benefit of partitioning memories becomes even more obvious due to the additionally achievable performance gains. In industry, a trend towards smaller partitioned memories can also be found: The ARM9E features two "tightly coupled memories" (TCMs), each of which is dedicated to either storing data or instructions [ARM00]. The more recent ARM11 architecture provides two regions of SRAM memory, separated into one data and one instruction partition, that can be individually configured either as TCMs or as SmartCaches. In the latter case, the SRAM mirrors a contiguous region in the background memory. This allows a simplified tag comparison in order to save energy [ARM04a]. If configured as TCMs, the SRAMs in the ARM11 architecture may be filled and written back to main memory using DMA transfers.

In contrast to ARM's current architecture model that uses a Harvard architecture, a more general approach is pursued in this work: we assume a multi-purpose scratchpad memory that is partitioned into several small contiguous regions, each of which is allowed to hold instructions as well as data, as shown in Figure 4.4. The scratchpad memory partitions are connected to

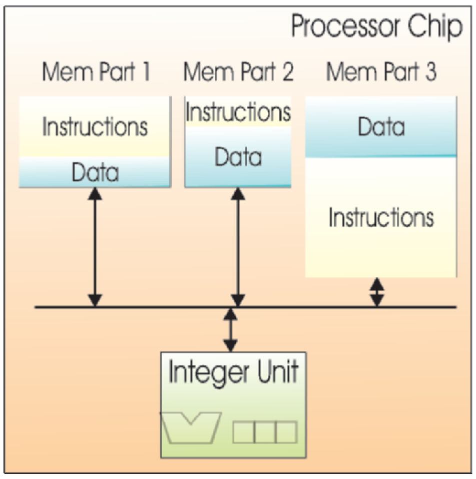


Fig. 4.4. Processor core with partitioned scratchpad memories

the processor by one single bus. The presented general approach can be modified in a straightforward way to match ARM's TCM architecture, which will be shown in Section 4.2.7.

The multi memory algorithm can either perform static program analysis or use the dynamic profiling workflow presented in Section 3.6.1 to obtain precise information concerning instruction executions and data object accesses. Furthermore, an architectural description of the currently used memory partitioning is required. The latter also includes the "energy per access" values of the considered memory partitions. Using this information and the energy model presented in Section 3.4, the optimization problem is formulated as an integer programming problem. A commercial ILP solver [ILO05] is then used to find a mapping of memory objects to memory partitions which minimizes energy dissipation.

An ILP representation to model the allocation problem was chosen for a couple of reasons. First, experiments show that the solving times to find solutions for the given system of equations are acceptable. Despite the fact that ILP solution algorithms can generally exhibit a poor runtime behavior in particular for increasing problem sizes, this does not hold for the class of problems described in this work. This is due to the fact that all allocation problems described here are variations of packing problems, which are easier to solve than general ILP problems. In addition, it is known that pseudo-linear or even linear approximation algorithms exist for this class of problems. Since the overall solution times were acceptable, and in particular since good solutions are usually found very soon in the solving process, the use of approximation algorithms is not considered in this work. Another reason to use ILP is its simplicity: once a system of ILP equations has been designed carefully, generating a text based ILP problem file and using a solver to find the optimal solution is much less error prone than describing different allocation algorithms in an arbitrary programming language. Finally, one of the main reasons that lead to the choice of ILP representations is their flexibility: using a generic ILP generation framework makes modifications of existing ILP problems very easy. Without having to go through the tedious implementation of different algorithms in a programming language, the known data values (consisting e.g. of object sizes and energy dissipation) are written to a file in the form of ILP equations and the ILP solver subsequently finds the desired optimal solution which is then read and processed within the compiler accordingly. As will be shown in the subsequent sections, it is straightforward to extend a simple basic model by integrating more and more information into the ILP problem. Note that the general workflow within the compiler always stays the same: profiling data is collected, an ILP problem is generated and solved, and the solution values are used during the final code generation process.

4.2.1 Memory Objects

In order to model the optimization problem, variables, functions and basic blocks need to be considered for allocation to the available memory partitions.

- Variables: The set of variables is denoted as V. The allocation of a variable $v \in V$ to the scratchpad memory is performed within the linker by changing the base address of the corresponding data element. This is straightforward for global variables kept in a data region of the application. Since local variables within a function are usually kept in registers or on the stack, these data objects do not have an address that can be treated accordingly. Moving the stack to the scratchpad memory is supported by the encc compiler, but the lack of detailed analyses makes this approach unsafe if the space required for the stack can not be determined in a sufficiently precise way. For this reason, local variables are not considered in this approach. The approach to allocate the entire stack to the scratchpad memory proposed in [Ste03] could however be integrated into the proposed model, allowing some of the local variables to be accessed from a scratchpad partition.

  Every global data element is considered as a single allocatable unit. Splitting larger data structures like structs or arrays into smaller parts and the resulting finer granularity may lead to improved results, however there is a risk of increases in code size due to the introduced addressing overhead. The array splitting approach proposed in [VSM03] can be integrated into the method without any changes in the workflow since the arrays are split on the source code level.
- Functions: Just like global variables, functions $f \in F$ can be moved to different memory partitions by changing their starting address. Since functions are self-contained memory objects which, assuming a structured program, are always executed starting at the first instruction and ending with a return statement, they can be assigned to an arbitrary address without modification.

- Basic Blocks: Basic blocks $bb \in BB$ require special treatment when they are allocated to a different memory partition, as already described in the previous section for the single-scratchpad memory allocation. The additional branch (or longjump) instructions have to be considered in the energy cost function as well as in the space constraints, since they consume energy when executed as well as additional space. The allocation of basic blocks to different memory partitions can be seen in Figure 4.5 which shows the additionally inserted long jumps using bold lines as well as edges between basic blocks allocated to the same partition. In order to improve the granularity of allocatable basic blocks and thus achieve better optimization results, [Hel04] proposed to split basic blocks into smaller units. This approach, albeit producing slightly improved results, leads to an increase in the number of considered memory objects and is not considered in this work.

  In contrast to the single-scratchpad case, the enumeration of all blocks of contiguous basic blocks (so-called "multi basic blocks") considering all possible allocations to different memories is not feasible due to the strong increase in the number of combinations which would lead to unacceptably long ILP solving times. As a solution, two alternative approaches ("Top-Down" and "Bottom-Up") will be presented to model the relationship between functions and basic blocks.

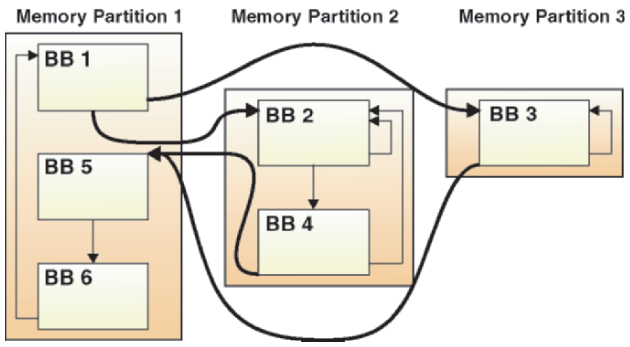


Fig. 4.5. Allocation of basic blocks to multiple memory partitions

4.2.2 Prerequisites

Before proceeding to generate an ILP model, some auxiliary variables and definitions are required. The set of all m available memory partitions, which also includes the main memory, is defined as

$$MP := \{mp_1, mp_2, \dots mp_m\} \tag{4.9}$$

Each of the memory partitions has certain attributes like their starting address or the size. Since the latter is required in the formulation of the ILP problem, function $Size(mp_j)$ is defined as the size in bytes of memory partition mp_j.

To make the ILP notation more legible, the set O contains all variables $v \in V$, functions $f \in F$ and all basic blocks $bb \in BB$ as follows:

$$O \subseteq V \cup F \cup BB := \{v_1, \dots v_p, f_1, \dots f_q, bb_1, \dots bb_r\} \tag{4.10}$$

There is a total number of n memory objects: $n = p + q + r$. Like for memory partitions, there is a function $Size(o_i)$ which returns the size of an arbitrary memory object $o_i \in O$. If o_i happens to be a basic block, then the maximum number of required long jumps is always included in the size to make sure the scratchpad capacity is not exceeded.

4.2.3 Energy Functions

To model the optimization as an ILP problem, a cost function that optimizes the energy dissipation within the memory subsystem is used. Similar to the model presented in the previous section, the energy attributed to a global variable v in the multi memory model corresponds to the sum of all accesses to this variable throughout the application execution $\#acc(v)$, taking into account the per-access energy of the memory partition mp_j used to store the global variable:

$$E(v, mp_j) := \#acc(v) \cdot E_{data}(mp_j) \tag{4.11}$$

A function's energy contribution is defined by its instructions k, by the number of executions of each of the instructions $\#exec(k)$ and by the instruction fetch energy from partition mp_j holding function f:

$$E(f, mp_j) := \sum_k \#exec(k) \cdot E_{ifetch}(mp_j) \tag{4.12}$$

For basic blocks, energy is defined by the number of executions $\#exec(bb)$, by the number of instructions $n(bb)$ in the basic block and by the instruction fetch energy for memory partition mp_j. To consider the potentially inserted additional long jumps for each basic block, the number of successors $\#succ(bb)$ of each basic block is determined from the control flow graph of the program: The maximum number of jumps required is equal to the number of successors of the basic block, since each successor may have to be reached using a long jump to a different memory partition.

This conservative assumption of every basic block requiring the maximum number of jump instructions will later be corrected in the ILP model.

$$\begin{aligned} E(bb, mp_j) := \#exec(bb) \cdot n(bb) \cdot E_{ifetch}(mp_j) + \\ \#exec(bb) \cdot \#succ(bb) \cdot E_{longjump}(mp_j) \end{aligned} \tag{4.13}$$

Note that during code generation, when the basic blocks are actually distributed to the different memory partitions, an automatic analysis is performed by the compiler whether the currently considered basic block actually requires the additional jump instruction assumed in the ILP model. If the longjump is not required (since the corresponding successor node is allocated to the same memory), then no additional jump will be added so as not to waste any memory space. This is a postpass optimization that takes place in the compiler during the actual code generation and may improve the performance and energy dissipation of the application when not all relationships of basic blocks are accurately modeled in the ILP formulation, which applies to the Base and the Top-Down model.

4.2.4 The Base model

This first basic model simplifies the formulation of the ILP problem by not taking the relationship of functions and their comprising basic blocks into account. In the Base model, functions are excluded from the set of considered memory objects, so that only individual and independent basic blocks and global variables are considered. In order to simplify the notation, the set of functions F is assumed to be the empty set and $q = 0$ for the Base model.

The decision variables in the ILP formulation denote whether a certain memory object $o_i \in O$ is to be allocated to a memory partition $mp_j \in MP$ or not. Since there are n memory objects and m memory partitions, an $n \times m$ matrix of binary decision variables results:

$$\tilde{O} := \begin{pmatrix} \tilde{o}_{1,1} & \cdots & \tilde{o}_{1,m} \\ \vdots & \ddots & \vdots \\ \tilde{o}_{n,1} & \cdots & \tilde{o}_{n,m} \end{pmatrix} \tag{4.14}$$

The value of matrix element $\tilde{o}_{i,j}$ is defined as

$$\tilde{o}_{i,j} := \begin{cases} 1, & \text{if } o_i \in O \text{ is allocated to } mp_j \in MP \\ 0, & \text{otherwise} \end{cases} \tag{4.15}$$

The ILP solver thus has to determine values for all binary decision variables $\tilde{o}_{i,j}$ in order to find an optimal solution to the optimization problem. With the main optimization goal being energy dissipation in the memory subsystem, the objective function consists of the energy functions for the different memory objects. Since multiple memory partitions are considered, the memory partition the object is allocated to also has to be specified in the objective function:

$$E(o_i, mp_j) := \begin{cases} E(v, mp_j), & \text{if } o_i \in V, o_i = v \text{ is allocated to } mp_j \\ E(bb, mp_j), & \text{if } o_i \in BB, o_i = bb \text{ is allocated to } mp_j \end{cases} \tag{4.16}$$

The objective of the allocation algorithm is to allocate all memory objects to specific memory partitions in such a way that the overall energy dissipation is minimized. This objective function has to be minimized since it expresses the energy dissipation.

$$\text{Minimize } \sum_{i=1}^{n} \sum_{j=1}^{m} E(o_i, mp_j) \cdot \tilde{o}_{i,j} \tag{4.17}$$

To generate a valid solution, a number of constraints have to be considered: Each memory object must be allocated to exactly one memory partition, so the optimization is performed subject to

$$\forall i : 1 \leq i \leq n : \sum_{j=1}^{m} \tilde{o}_{i,j} = 1 \tag{4.18}$$

In addition, the size constraints of all the memory partitions have to be respected, since no partition can hold more bytes than its capacity:

$$\forall j : 1 \leq j \leq m : \sum_{i=1}^{n} [Size(o_i) \cdot \tilde{o}_{i,j}] \ \leq \ Size(mp_j) \tag{4.19}$$

The energy and size functions of basic blocks in this case both include the additional energy required to perform the potential long jump between memories. Since some of these jumps may be removed during code generation when the compiler finds that two basic blocks are actually allocated to the same partition, the results obtained by the Base approach after simulation may actually be better than what could be expected from the ILP model alone. The Base model uses a pessimistic assumption about basic blocks requiring additional jumps since this is the easiest way to model the optimization problem.

Up to this point, the optimization problem thus consists of the objective function and one constraint per memory object to make sure that each object is allocated to one memory partition. In addition, one constraint for each of the m memory partitions is required to ensure the partitions' capacities are not exceeded. Since the relationship between functions and basic blocks is not modeled in this simple approach, ignoring the function objects (i.e. $F = \emptyset$) and only considering the p variables and r basic blocks as memory objects will lead to valid, albeit not necessarily optimal, results. This leads to the following model complexity in terms of number of constraints:

$$\#Constraints(\text{Base}) = m + p + r = m + n \tag{4.20}$$

Another metric to describe the complexity of an ILP problem is the number of decision variables. The dimensions of the matrix $\tilde{O}$ specify that $m{\cdot}n$ decision variables are required to specify the Base model.

The Base model's objective function and constraints are summarized in Figure 4.6. To further improve this basic model, the benefit of moving complete functions or blocks of contiguous basic blocks will be introduced using two different approaches in the following sections.

Set of memory objects:

$$O \subseteq V \cup BB := \{v_1, \ldots, v_p, bb_1, \ldots, bb_r\}; \quad n = p + r$$

Decision variables:

$$\tilde{O} := \begin{pmatrix} \tilde{o}_{1,1} & \cdots & \tilde{o}_{1,m} \\ \vdots & \ddots & \vdots \\ \tilde{o}_{n,1} & \cdots & \tilde{o}_{n,m} \end{pmatrix}$$

Objective function:

$$\text{Minimize} \sum_{i=1}^{n} \sum_{j=1}^{m} E(o_i, mp_j) \cdot \tilde{o}_{i,j}$$

Subject to space restrictions:

$$\forall j : 1 \leq j \leq m : \sum_{i=1}^{n} [Size(o_i) \cdot \tilde{o}_{i,j}] \ \leq \ Size(mp_j)$$

Select each object once:

$$\forall i : 1 \leq i \leq n : \sum_{j=1}^{m} \tilde{o}_{i,j} \ = \ 1$$

Number of constraints/decision variables:

$$\#Constraints(\text{Base}) = m + p + r \ = \ m + n$$
$$\#DecisionVariables(\text{Base}) = m \cdot (p + r) = m \cdot n$$

Fig. 4.6. Complete ILP formulation of the Base model

4.2.5 The Top-Down Model

The Top-Down model extends the Base model by specifying the relationship between functions and their contained basic blocks. As the name suggests, the Top-Down model starts its considerations at the function level and consequently models the contained basic blocks as dependent objects. The motivation to consider these connections among memory objects is the fact that when basic blocks are allocated to different memory partitions, additional jump instructions may have to be inserted which consume additional space, time and energy (cf. Figure 4.5 on page 100). It is therefore beneficial to allocate an entire function instead of separate basic blocks to a particular memory partition since a function does not require any additional jumps to sustain correct control flow. However, the allocation of a complete function to an energy-efficient memory partition implies that all basic blocks within that function are allocated to this partition, even including those basic blocks that are e.g. only executed once. It is the ILP solver's task to use the additional information provided by the extended model to trade off the advantage of allocating complete functions against the more flexible allocation of single basic blocks.

One additional advantage of the Top-Down approach is the fact that in the worst case, the Base model may choose an allocation that distributes the basic blocks of a function across all available partitions. Using the Top-Down approach, this diversion of control flow can be avoided once a complete function fits into one memory partition, resulting in more stable results of the allocation process.

The Top-Down model only considers either a complete function or an arbitrary subset of its contained basic blocks. The additional advantage of allocating a sequence of contiguous basic blocks is not modeled using the Top-Down technique since this would strongly increase the size of the model, making it practically unusable. An elegant solution to model contiguous basic blocks will be presented in the following section using the Bottom-Up approach.

The Top-Down model was presented for the first time in [Hel04]. However, no results were presented for this allocation technique since the proposed representation lead to a complex ILP representation resulting in very long runtimes and high memory requirements of the ILP solver, making the model inefficient for practical use. In this work, a novel representation is presented for the first time which effectively reduces the complexity of the ILP representation and allows a faster generation of results even for larger benchmarks. The general idea that leads to the better performance of this improved Top-Down model is the use of additional decision variables to capture the relationship of functions and basic blocks instead of using a large number of constraints. Despite the fact that the relationships modeled by the old and the new Top-Down formulation are identical, the new representation leads to results in significantly reduced ILP solver execution times.

To model the relationship between functions and their comprising basic blocks using the Top-Down model, the application's functions need to be considered in the set of considered memory objects (in contrast to the Base model, which ignored all function objects). To achieve a consistent model, either an entire function or its comprising basic blocks can be allocated to a memory partition. If a function is completely allocated, then its comprising basic blocks must not be considered for individual allocation anymore, and their binary decision variables should be assigned the value 0 for all memory partitions. Thus, the constraining Equation 4.18 needs to be modified to allow each object to be assigned to at most one partition (instead of exactly one):

$$\forall i : 1 \le i \le n : \sum_{j=1}^{m} \tilde{o}_{i,j} \ \le \ 1 \tag{4.21}$$

This is the set of constraints that was used by the Top-Down model presented in [Hel04]. As mentioned above, it is beneficial for the efficient formulation of the model to introduce new decision variables. These variables carry information about which memory objects are allocated to a certain memory partition and which objects aren't. The following modified set of constraints includes one new binary decision variable $h_i \in \{0, 1\}$ for each function object $f \in F$, but is otherwise identical to Equation 4.21 above:

$$\forall i : 1 \le i \le n : \begin{cases} \sum_{j=1}^{m} \tilde{o}_{i,j} - h_i = 0 \text{ , if } o_i \in F \\ \\ \sum_{j=1}^{m} \tilde{o}_{i,j} \le 1 \qquad\qquad \text{otherwise} \end{cases} \tag{4.22}$$

Note that the two representations 4.21 and 4.22 are equivalent, except for the fact that in the second notation, decision variable h_i is set to the value one if function o_i is allocated to a memory partition as one whole memory object, and to zero otherwise. The fact that both constraints 4.21 and 4.22 allow memory objects not to be allocated to a partition bears the problem that a trivial solution now becomes valid: By setting all decision variables to the value 0, all constraints can be satisfied, at the same time minimizing the energy dissipation. However, not allocating any object to any partition is not a desirable solution. One way to avoid this problem is to modify the used energy function accordingly by turning it into a maximization instead of a minimization problem. In this way, the trivial solution of not allocating any objects to a partition still represents a valid solution, but it is not attractive due to the fact that the modified objective function is to be maximized.

In order to transform the objective function, the energy values $E(o_i, mp_j)$ are negated to $-E(o_i, mp_j)$ and an offset value is added to the energy that is large enough to make all energy values positive. This is always possible since the energy values appearing within a program are bounded by some maximum value E_{max}:

$$E'(o_i, mp_j) := E_{max} - E(o_i, mp_j) \tag{4.23}$$

Consider the example illustrated in Table 4.1: basic blocks A and B have an energy dissipation of three and four units, respectively, when they are allocated to a particular memory partition. If they are not allocated, the resulting cost is zero. In the initial minimization problem with energy function E, the solver will choose not to allocate any object to any partition, since this results in minimal costs. The desired solution would be to first allocate basic block A (since it has a lower cost value), then basic block B, given there is sufficient space on the considered memory partition.

Assuming that the maximum occurring energy E_{max} is six, using the new energy function E' results in basic block A and B having energy values of three and two units, respectively. Using a maximizing objective function, basic block A will be allocated first, since it leads to a higher objective function value. If sufficient space is available, then basic block B will also be allocated to the partition. Note that the trivial solution of not allocating A or B is still valid, but will not be chosen since it does not maximize the objective function.

Object Name	E	E'
Basic Block A	3	3
Basic Block B	4	2

Table 4.1. Negating the energy function to avoid the trivial solution

Care must be taken, however, that the ratio of energy dissipation between basic blocks and functions is maintained during this transformation. Since the maximum energy value E_{max} is added to the negated energy value of every single basic block, precisely the same amount also has to be added to the complete function in order to make it equally attractive in the maximization objective function. In other words: functions require that the value $E_{max} \cdot BBC(f)$ be added to them, where $BBC(f)$ is the number of basic blocks contained in Function f. To formalize this, two new functions are introduced to determine the number of basic blocks contained in a function. For both functions, let $f \in O$ and $bb \in O$:

$$\sigma(f, bb) := \begin{cases} 1, \text{if } f \in F, bb \in BB \text{ and } bb \text{ is contained in } f \\ 0, \text{ otherwise} \end{cases} \tag{4.24}$$

$$BBC(f) := \sum_{i=1}^{n} \sigma(f, o_i) \tag{4.25}$$

Function $\sigma(f, bb)$ is defined for all objects $f, bb \in O$, but only returns the value 1 if f is a function and bb is a basic block and if bb is one of the basic blocks of function f. Function $BBC(f)$ returns the number of basic blocks contained in function f.

Assume that in the original representation, the sum of the basic blocks' energy coefficients was greater than the coefficient of the function f (as in the first equation below) due to additionally required jumps. If each basic block is negated and the maximum energy E_{max} added to it, then the function f on the right hand side requires an addition of E_{max} multiplied with $BBC(f)$, the number of basic blocks in that function, in order to maintain the correct relationship between function and basic blocks:

$$\begin{aligned}
E(bb_1) + \ldots + E(bb_{BBC(f)}) &> E(f) \\
&\qquad | \cdot (-1) \\
-E(bb_1) - \ldots - E(bb_{BBC(f)}) &< -E(f) \\
&\qquad | + BBC(f) \cdot E_{max} \\
(E_{max} - E(bb_1)) + \ldots + (E_{max} - E(bb_{BBC(f)})) &< (BBC(f) \cdot E_{max}) - E(f)
\end{aligned}$$

To illustrate this, consider the example given in Table 4.2: function F, comprising two basic blocks A and B, has an energy dissipation of five energy units, whereas the individual basic blocks consume three and four units, respectively. If either the function or the two basic blocks are to be allocated to a memory partition using the original minimization objective function, the function would be selected since its energy dissipation is less than the sum of the basic blocks'.

Object Name	E	E'
Basic Block A	3	3
Basic Block B	4	2
Function F	5	7

Table 4.2. Modified energy function for the Top-Down approach

Setting the new energy value of function F to $(BBC(f) \cdot E_{max}) - E(F)$ and assuming that $E_{max} = 6$ results in the values in the third column of the table. The energy of the function is thus $(2 * 6) - 5 = 7$, which generates an ILP maximization problem which is equivalent to the original problem formulation.

Equation 4.23 thus has to be refined to the following representation:

$$\forall o_i \in BB, V: \tag{4.26}$$
$$E'(o_i, mp_j) := E_{max} - E(o_i, mp_j)$$
$$\forall o_i \in F: \tag{4.27}$$
$$E'(o_i, mp_j) := (BBC(f) \cdot E_{max}) - E(o_i, mp_j)$$

Using this definition of energy functions E', the new objective function is given as

$$\text{Maximize} \sum_{i=1}^{n} \sum_{j=1}^{m} E'(o_i, mp_j) \cdot \tilde{o}_{i,j} \tag{4.28}$$

Due to this maximization formulation, the solver will now try to set as many decision variables as possible to the value 1 without violating any of the constraints. This effectively prevents the solver from accepting the trivial solution.

Having adjusted the objective function appropriately, it is still necessary to express the connections between functions and their comprising basic blocks. First, the set of basic blocks contained in a function f is defined as

$$\begin{aligned} BBs(f) &:= \{bb \in BB : \sigma(f, bb) = 1\} \\ &= \{bb_{f,1}, \ldots, bb_{f,BBC(f)}\} \end{aligned} \tag{4.29}$$

The additional constraints have to prevent the solver from assigning both a complete function and any of the basic blocks contained in that function to a memory partition. This is modeled using another new set of integer (not binary) decision variables h'_i which count the number of individually allocated basic blocks within each function o_i:

$$\forall o_i \in F, o_x \in BBs(o_i) : \sum_{x=1}^{BBC(o_i)} \sum_{j=1}^{m} \tilde{o}_{x,j} - h'_i = 0 \tag{4.30}$$

In this way, h'_i for each function contains the number of basic blocks that are allocated to an arbitrary memory partition. If the complete function is allocated, then the number of individually allocated basic blocks must be 0. This is achieved by the following, final set of constraints:

$$\forall o_i \in F : BBC(o_i) \cdot h_i + h'_i \leq BBC(o_i) \tag{4.31}$$

For each function o_i, the constraint ensures that if the entire function is allocated to a memory partition, implying $h_i = 1$, then no individual basic block within this function may be individually allocated, thus h'_i must be set to zero in order to satisfy the constraint. If, on the other hand, the complete function is not allocated to a partition as a whole memory object (and thus $h_i = 0$), then all of the individual basic blocks have to be allocated to some arbitrary memory partition. In this case, $h'_i = BBC(o_i)$ is achieved due to the maximization of the objective function. The presented set of constraints thus successfully prevents the allocation of both a function and its comprising basic blocks to any memory partition.

To guarantee the generation of valid solutions, the space constraints from the Base model also have to be included for each of the memory partitions which concludes the formulation of the Top-Down model.

Note that due to the negation of the objective function and the subsequent addition of E_{max}, a potential problem is introduced during the solution of the ILP problem: those energy values that were initially very small are now very close to the maximum occurring energy value. This implies a certain loss of precision: while floating point numbers have a high precision for small absolute values, the minimal distance between two distinguishable floating point numbers grows as their absolute size grows. This can cause some of the originally small, but distinct energy values to be projected onto the same floating point value. In most of the considered benchmarks, the maximum energy coefficients in the ILP formulation are in the range of 10^8, so that the described effect can actually occur for the performed experiments. However, the problem mainly concerns memory objects that originally had a small energy value associated to them, meaning that the most relevant objects will still be treated correctly.

Set of memory objects:

$$O \subseteq V \cup F \cup BB := \{v_1, \dots v_p, f_1, \dots f_q, bb_1, \dots bb_r\}; \quad n = p + q + r$$

Decision variables:

$$\tilde{O} := \begin{pmatrix} \tilde{o}_{1,1} & \cdots & \tilde{o}_{1,m} \\ \vdots & \ddots & \vdots \\ \tilde{o}_{n,1} & \cdots & \tilde{o}_{n,m} \end{pmatrix} \qquad \begin{array}{c} \forall i : p+1 \leq i \leq q, \\ o_i \in F : \\ h_i \in \{0,1\}, \quad h'_i \in \mathbb{N} \end{array}$$

Objective function:

$$\text{Maximize} \sum_{i=1}^{n} \sum_{j=1}^{m} E'(o_i, mp_j) \cdot \tilde{o}_{i,j}$$

Subject to space restrictions:

$$\forall j : 1 \leq j \leq m : \sum_{i=1}^{n} [Size(o_i) \cdot \tilde{o}_{i,j}] \ \leq \ Size(mp_j)$$

Select each object at most once:

$$\forall i : 1 \leq i \leq n : \begin{cases} \sum_{j=1}^{m} \tilde{o}_{i,j} - h_i = 0 & \text{if } o_i \in F \\ \sum_{j=1}^{m} \tilde{o}_{i,j} \leq 1 & \text{otherwise} \end{cases}$$

allocate either complete function or individual basic blocks:

$$\forall o_i \in F, o_x \in BBs(o_i) : \sum_{x=1}^{BBC(o_i)} \sum_{j=1}^{m} \tilde{o}_{x,j} - h'_i = 0$$

$$\forall o_i \in F : BBC(o_i) \cdot h_i + h'_i \leq BBC(o_i)$$

Number of constraints/decision variables:

$$\#Constraints(\text{Top-Down}) = m + n + 2q$$
$$\#DecisionVariables(\text{Top-Down}) = m \cdot n + 2q$$

Fig. 4.7. Complete ILP formulation of the Top-Down model

The complexity of the Top-Down model as it was originally described in[Hel04] was very high in particular with respect to the number of considered memory partitions. In fact, the number of constraints can be determined to be $m + n + m \cdot q^2$ and the number of decision variables was $m \cdot n$ for the original approach. This high complexity makes the Top-Down model as proposed in [Hel04] unusable for practical benchmarks. The memory requirements of the ILP solver [ILO05] using the original Top-Down model were so big that our compute server (a dual-processor Sun Fire V240 server featuring two UltraSPARC IIIi processors running at 1.3 GHz with 4 GB of main memory) was incapable of finding a solution for any of the larger benchmarks.

By introducing the new decision variables h_i and h'_i for each function object o_i, the complexity of the model could be reduced significantly, since a large number of constraints can be expressed using the value of the new decision variables that were not present in the original representation of the Top-Down model. In addition to the m memory size constraints, n constraints are required to model each memory object being allocated to at most one memory partition, at the same time setting the binary decision variables h_i to correct values. An additional q constraints (q representing the number of functions in the application) are required to determine the values h'_i for every function. Finally, another q constraints guarantee that either functions or their comprising basic blocks are allocated to memory partitions. The number of additional decision variables is determined as $2q$, one h_i and one h'_i for each function object. All in all, the number of constraints and decision variables required for the novel Top-Down model presented in this work is

$$\#Constraints(\text{Top-Down}) = m + n + 2q \qquad (4.32)$$

$$\#DecisionVariables(\text{Top-Down}) = m \cdot n + 2q \qquad (4.33)$$

This concludes the description of the Top-Down model. Figure 4.7 shows the entire ILP formulation of the Top-Down approach. Results generated using the Top-Down model will be presented in Section 4.2.9.

4.2.6 The Bottom-Up Model

In contrast to the Top-Down Model, the Bottom-Up Model perceives functions only as a number of contiguous basic blocks - hence the name “Bottom-Up”. Thus, only global variables and basic blocks are considered in the model and allocated to the different memory partitions. The achievement of this model is that it accurately models the effect of moving single basic blocks as well as arbitrarily long sequences of contiguous basic blocks, implicitly including functions. When only one basic block is allocated to a different memory partition, then long jump instructions are required to jump to this basic block and back again. If, however, two basic blocks connected by a control flow edge are allocated to the same memory partition, then no modification of this edge is necessary: either, the original jump condition is used to pass from one basic block to the next, or they are allocated to contiguous memory addresses and are just executed sequentially by incrementing the program counter. To avoid the increasing complexity in describing the number of combinations of contiguous basic blocks allocated to a number of partitions, the edges of the control flow graph are used in the constraints of the ILP problem to model the fact that it is beneficial to move two connected basic blocks to the same partition. By using the edges of the control glow graph, this more fine-grained approach results in higher energy savings and at the same time reduces the size of the ILP problem.

The edges of the control flow graph are represented in the ILP formulation by the set C which is defined as

$$C := \{c_1, \cdots, c_s\} \tag{4.34}$$

To express that two basic blocks are connected by an edge, an additional function is required. Function $V(x, y)$ returns the index number k of the connecting edge if both o_x and o_y are basic blocks, and if they are connected by a control flow edge. If o_x and o_y are not basic blocks, or if they are not connected by a control flow edge, the function returns -1.

$$V(x,y) := \begin{cases} k, \text{if } o_x, o_y \in BB \text{ and edge } c_k = (o_x, o_y) \in C \\ -1, \text{ otherwise} \end{cases} \tag{4.35}$$

The set C and function $V(x, y)$ are used to add another matrix of decision variables $\tilde{C}$ to the model. This matrix contains one binary decision variable $\tilde{c}_{k,j}$ for each edge k and every memory partition j which models the connection of basic blocks by control flow edges. A decision variable $\tilde{c}_{k,j}$ is assigned the value 1 if memory partition j is not left when control flows along edge k. In other words: if two basic blocks connected by a control flow edge are allocated to the same memory partition, then the corresponding decision variable is set to 1. With this additional matrix, a modified objective function can be formulated. The potential additional jump instructions are already included in the original problem formulation (cf. Equation 4.14 on page 102). To account for the benefit of staying within one memory partition, the Bottom-Up model subtracts this additional overhead if no jump between memories is required. Thus, if a control flow edge does not leave memory partition mp_j, an energy benefit $E(longjump, mp_j)$ is considered in the objective function.

Assuming the decision variables $\tilde{c}_{k,j}$ are set correctly, the objective function is formulated as

$$\begin{aligned} \text{Minimize } & \sum_{i=1}^{n} \sum_{j=1}^{m} [E(o_i, mp_j) \cdot \tilde{o}_{i,j}] \\ - & \sum_{k=1}^{s} \sum_{j=1}^{m} [E(longjump, mp_j) \cdot \tilde{c}_{k,j}] \end{aligned} \tag{4.36}$$

Note that the original form of the energy function from the Base model can be used here: Since functions are not considered as memory objects, each object can again be assumed to be allocated to exactly one partition (Equation 4.18). One other constraint does require modification, however: since the connecting edges, the allocation of objects to memory partitions and thus the required long jumps between basic blocks are now known within the ILP model, this knowledge also has to be considered in the size constraints: similar to the objective function, the additional overhead assumed for additional long jumps is subtracted if control flow along this edge does not leave the memory partition, i.e. if the corresponding decision variable $\tilde{c}_{k,j}$ is set:

$$\begin{aligned} \forall j : 1 \leq j \leq m : & \sum_{i=1}^{n} \left[Size(o_i) \cdot \tilde{o}_{i,j} \right] \\ & - \sum_{k=1}^{s} \left[Size(longjump) \cdot \tilde{c}_{k,j} \right] \ \leq \ Size(mp_j) \end{aligned} \tag{4.37}$$

Up to this point, the additional decision variables $\tilde{c}_{k,j}$ were considered to correctly reflect the control flow among the memory partitions. The following set of constraints ensures that the variables are set correctly:

$$(k = V(x, y), \forall k \neq -1, 1 \leq x \leq n, 1 \leq y \leq n), (\forall j, 1 \leq j \leq m):$$

$$\tilde{o}_{x,j} + \tilde{o}_{y,j} - 2 \cdot \tilde{c}_{k,j} \geq 0 \tag{4.38}$$

This constraint implies that a decision variable $\tilde{c}_{k,j}$ must be set to 0 if the two basic blocks connected by edge k are not allocated to the same memory partition j. Setting all possible decision variables $\tilde{c}_{k,j}$ to 1 is not explicitly modeled in this constraint, however: this is being taken care of automatically by the ILP solver, since setting the $\tilde{c}_{k,j}$ to 1 where this is possible without violating constraints helps to minimize the objective function.

During experiments, the Bottom-Up model was found to show a sufficiently fast ILP solution time, despite the fact that it accurately models the relationship between functions and basic blocks. Beside the objective function, m constraints are required to make sure that the memory capacities of the memory partitions are not exceeded. Each of the n objects is again allocated to exactly one memory partition, leading to n constraints. Modeling the control flow requires every edge of the control flow graph to be considered for each of the memory partitions, resulting in $m \cdot |E|$ constraints, where $|E|$ is the number of edges in the control flow graph. The total number of constraints for the Bottom-Up model is thus

$$\#Constraints(\text{Bottom-Up}) = m + n + m \cdot |E| \tag{4.39}$$

Since every basic block can have a maximum of two successors, the relationship $|E| \leq 2n$ holds for all regular control flow graphs. In the experiments

Set of memory objects:

$$O \subseteq V \cup BB := \{v_1, \ldots, v_p, bb_1, \ldots, bb_r\}; \quad n = p + r$$

Decision variables:

$$\tilde{O} := \begin{pmatrix} \tilde{o}_{1,1} & \cdots & \tilde{o}_{1,m} \\ \vdots & \ddots & \vdots \\ \tilde{o}_{n,1} & \cdots & \tilde{o}_{n,m} \end{pmatrix} \qquad \tilde{C} := \begin{pmatrix} \tilde{c}_{1,1} & \cdots & \tilde{c}_{1,m} \\ \vdots & \ddots & \vdots \\ \tilde{c}_{s,1} & \cdots & \tilde{c}_{s,m} \end{pmatrix}$$

Objective function:

$$\text{Minimize} \sum_{i=1}^{n} \sum_{j=1}^{m} [E(o_i, mp_j) \cdot \tilde{o}_{i,j}] - \sum_{k=1}^{s} \sum_{j=1}^{m} [E(longjump, mp_j) \cdot \tilde{c}_{k,j}]$$

Subject to space restrictions:

$\forall j : 1 \leq j \leq m$:

$$\sum_{i=1}^{n} [Size(o_i) \cdot \tilde{o}_{i,j}] - \sum_{k=1}^{s} [Size(longjump) \cdot \tilde{c}_{k,j}] \ \leq \ Size(mp_j)$$

Select each object exactly once:

$$\forall i : 1 \leq i \leq n : \sum_{j=1}^{m} \tilde{o}_{i,j} = 1$$

Restrict decision variables $\tilde{c}_{k,j}$ to correct values:

$$(k = V(x,y), \forall k \neq -1, 1 \leq x \leq n, 1 \leq y \leq n), (\forall j, 1 \leq j \leq m):$$

$$\tilde{o}_{x,j} + \tilde{o}_{y,j} - 2 \cdot \tilde{c}_{k,j} \geq 0$$

Number of constraints/decision variables:

$$\#Constraints(\text{Bottom-Up}) = m + n + m \cdot |E|$$
$$\#DecisionVariables(\text{Bottom-Up}) = m \cdot n + m \cdot |E|$$

Fig. 4.8. Complete ILP formulation of the Bottom-Up model

we found that the matrix $\tilde{C}$ containing $m \cdot |E|$ additional decision variables does not seem to have a negative impact on the runtime of the ILP solver. The same is true for the additional $m \cdot |E|$ decision variables used in the Bottom-Up model. All presented results were generated within an acceptable time.

This concludes the presentation of the Bottom-Up model, which is summarized in Figure 4.8. Results generated using all of the approaches presented above will be shown in Section 4.2.9.

4.2.7 The ARM TCM Model

In the currently available ARM9 and ARM11 processors, the partitioned scratchpad memories (or TCMs) can not be exploited using the equations supplied above, since one of the memories can only hold instructions while the other can only hold data, as shown in Figure 4.9.

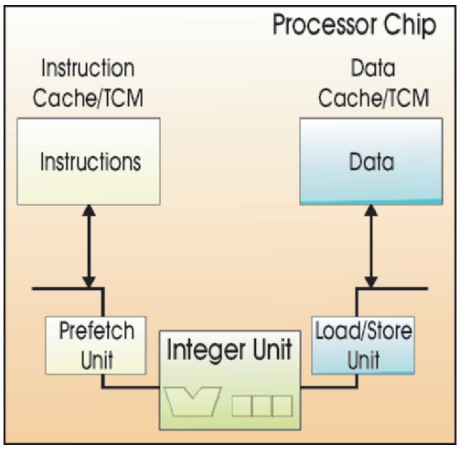


Fig. 4.9. ARM TCM architecture using separate instruction- and data buses

This restriction is due to the Harvard architecture with separate buses for instructions and data found in the recent ARM designs [ARM00, ARM04a]. If one of the scratchpad memories is to be used as a SmartCache, then the original model for only one scratchpad [SWLM02] can be used if the considered memory objects are restricted to either only variables or only functions and basic blocks, depending on which of the two scratchpad memories is being used as a TCM. If both scratchpad memories are to be used in addition to the main memory, the Top-Down or Bottom-Up version of the multi memory approach may be used with a slight modification to allocate memory objects to the TCM partitions accordingly. Assuming that in the most general case a number of memory partitions can only hold instructions, others can only hold data, and the remaining partitions can hold both instructions and data, the original multi memory allocation model can be adapted to the new situation in the following way:

The set of available memory partitions $MP := \{mp_1, \dots mp_m\}$ is partitioned into three disjoint sets at positions g and h such that $1 \leq g \leq h \leq m$:

$$MP_I := \{mp_1, \dots, mp_g\} \text{ only instructions} \tag{4.40}$$

$$MP_D := \{mp_{g+1}, \dots, mp_h\} \text{ only data} \tag{4.41}$$

$$MP_{ID} := \{mp_{h+1}, \dots, mp_m\} \text{ both instructions and data} \tag{4.42}$$

A correct allocation of memory objects to memory partitions can then be performed by assuring that the corresponding memory objects are only allocated to suitable memory partitions. The set of memory objects was defined as

$$O \subseteq V \cup F \cup BB := \{v_1, \dots v_p, f_1, \dots f_q, bb_1, \dots bb_r\} \tag{4.43}$$

with v being global variables, f being functions and bb basic blocks. By adding the following constraints to the optimization problem, it is ensured that no data object is allocated to an instruction partition and vice versa:

$$\forall j : 1 \leq j \leq g : \sum_{i=1}^{p} \tilde{o}_{i,j} = 0 \tag{4.44}$$

$$\forall j : (g+1) \leq j \leq h : \sum_{i=p+1}^{n} \tilde{o}_{i,j} = 0 \tag{4.45}$$

The straightforward addition of these two constraints to the optimization problem shows the advantages of choosing an ILP representation for the allocation problem: Modifications like supporting a Harvard architecture can be easily incorporated into the model without requiring a tedious re-implementation of the entire allocation algorithm.

The complexity of the underlying model is actually reduced by this modification, since the setting of decision variables to the fixed value 0 effectively removes them from the ILP equation system and they do not need to be considered by the solver anymore. With $|I|$ being the number of instruction memories and $|D|$ the number of data memories, the number of equations is thus increased by $|I| + |S|$ compared to the Bottom-Up model, leading to a total of constraints. However, the solver will be able to reduce the number of constraints and decision variables in the first iteration of its algorithm by the amounts given below:

$$\#Constraints(\text{Harvard}) = m + n + m \cdot |E| + |I| + |D| \tag{4.46}$$

- $|D| \cdot |E|$ edge constraints can be removed, since there are no edges between basic blocks in any of the data partitions
- $|D| \cdot |E|$ decision variables for edge constraints can be removed, since the constraints were removed
- $|D| \cdot (n - p)$ decision variables that handle instructions allocated to data partitions can be removed, since their value is 0
- $|I| \cdot p$ decision variables that handle data elements allocated to instruction partitions can be removed, since their value is 0

- $|I| + |D|$ constraints can be removed, since by removing all of the above elements, it is already ensured that no instructions are allocated to data partitions and vice versa

This leads to a reduction of $|D|(|E| + (n - p)) + |I| \cdot p$ decision variables compared to the Bottom-Up model. If we assume both the number of data and instruction partitions in the system to be one, which is the situation found in current ARM designs, the overall number of decision variables for the Harvard allocation can be determined as $(m - 1) \cdot (n + |E|)$. This means that the ILP formulation using separate data and instructions partitions has the same complexity as the general Bottom-Up model with the number of memory partitions reduced by one.

Thus, a reduction in the complexity of the model has actually been achieved. Since experiments have shown that the ILP solver is capable of efficiently eliminating redundant constraints and decision variables, we have refrained from actually generating the more efficient ILP representation with the mentioned constraints and decision variables removed. The reduced complexity is also reflected in the solving times, which were significantly reduced compared to the Bottom-Up approach for most of the considered benchmarks.

4.2.8 Leakage-Energy Aware Memory Configuration

The idea of partitioning scratchpad memories in order to improve the energy dissipation is based on the fact that larger memories require more energy per access than smaller memories. Taken to an extreme, this partitioning approach in theory would lead to a large number of very small memories, each barely large enough to hold a single memory object. In practice, however, introducing a large number of very small memories is not feasible – ARM as the market leader in embedded processor cores merely provides two scratchpad memories in their designs. This is on one hand due to the fact that the wiring required to connect a large number of memories would require a high amount of the valuable and scarce onchip space, making the resulting chip more expensive. Additional address decoders will also be required in the system to decide which memory partition is to be accessed. On the other hand, the mere presence of a memory will cause a certain amount of leakage energy to be dissipated, independent of its utilization or access frequency. Leakage currents will always flow within the transistors of the memory, causing a noticeable energy dissipation. The fact that leakage is becoming more important and will in fact dominate overall energy dissipation in particular for future submicron technologies [Bor99] emphasizes that the consideration of leakage energy is vital in particular during the design of embedded systems.

In the multi memory allocation schemes presented in the previous sections, the partitioning of the memory was assumed to be given in a fixed form. If the target system and its memory structure are yet to be designed, the designer has to determine suitable memory partition parameters, including number and size of partitions, for a particular application. This section presents a solution where the information available within the compiler can be used to help the designer to only utilize the most beneficial memory partitions from a set of available memories. The compiler needs to be aware of the cost of integrating extra scratchpad memories (in terms of leakage energy) so that it can trade off the benefit due to reduced energy per access against the overhead caused by a large number of small partitions. The energy per access cost is a well-known metric and has been used for all previously presented optimizations. The overhead of an additional scratchpad partition is modeled by assuming each partition has a certain power dissipation when it is present in the system. This power is spent throughout the execution time of the application and represents the leakage energy of the memory. In order to keep the used model simple, only this leakage energy value is added to the set of ILP equations. If the additional overhead required for the wiring and necessary decoders is to be modeled, it is straightforward to introduce additional variables to model these restrictions appropriately. In the presented model, however, we refrain from representing each of these factors by a separate variable and summarize the overhead caused by including a scratchpad memory partition in the abstract leakage energy value. By scaling this value as required, the designer can control the result of the tradeoff and thus achieve less memories to be used when the leakage for partitions is set to a high value. The compiler can thus be used to help the designer decide whether the benefit from allocating objects to an extra partition is outweighed by this partition's overhead or not.

This task can be accomplished by modifying the multi memory allocation technique. The representation of the problem in the form of ILP equations can again be exploited, since the notion of a leakage current can be integrated into the allocation model in a straightforward way. The leakage energy $E_{leakage}(mp_j)$ is introduced as an additional property for each of the memory partitions. It expresses the amount of energy that is always constantly dissipated by memory partition mp_j, even when it is not accessed. The total energy of this memory partition consists of the leakage energy and the energy caused by the accesses to this partition.

An additional set of decision variables needs to be integrated into the multi memory allocation problem to reflect whether a certain memory partition is being used, i.e. whether at least one memory object is allocated to it. This set of decision variables is defined as

$$\widetilde{MP} := \{\widetilde{mp_1}, \cdots \widetilde{mp_m}\} \tag{4.47}$$

where each decision variable $\widetilde{mp_j}$ has the value 1 if the corresponding memory partition mp_j is being used. Assuming these decision variables are set correctly, the objective function can be modified to also consider the leakage energy of the used memory partitions:

$$\text{Minimize} \sum_{i=1}^{n}\sum_{j=1}^{m}[E(o_i, mp_j) \cdot \tilde{o}_{i,j}] + \sum_{j=1}^{m}[E_{leakage}(mp_j) \cdot \widetilde{mp_j}] \tag{4.48}$$

In order to set the new decision variables mp_j to correct values, additional constraints are required to guarantee that a decision variable will be set to the value 1 if and only if this partition is being used to hold at least one memory object. This can be achieved with the following set of constraints:

$$\forall j, 1 \leq j \leq m : \sum_{i=1}^{n} \tilde{o}_{i,j} - n \cdot \widetilde{mp_j} \leq 0 \tag{4.49}$$

These constraints will guarantee that the decision variable mp_j is set to the value 1 if any of the $\tilde{o}_{i,j}$ are 1, since otherwise the equation would take a positive value and thus be violated. All remaining decisions variables mp_j will automatically be set to 0 since this helps to minimize the objective function.

If the designer provides a value for a certain memory's leakage energy, then the compiler is able to determine which memory partitions should be used for maximum benefit, leading to a compiler guided multi memory configuration. No memory objects will be allocated to a partition if its benefit is outweighed by the overhead caused by its leakage energy dissipation. If the memory architecture design is fixed, the unused memory partitions can be turned off, assuming the presence of modern memories' power management features, in order to eliminate their leakage energy.

If the memory architecture is not given and design space exploration is being performed, the designer can determine an appropriate number of memory partitions and their corresponding size by offering the compiler a large set of available memories and using the compiler's knowledge about the application behavior and memory requirements to decide which ones are most profitable from the compiler's point of view. Results generated using this method will be presented at the end of the following section.

The complexity of the multi memory allocation problem is only increased marginally by also considering the standby energies of memories: one additional constraint is required per memory partition in order to determine whether it is being used or not, leading to m additional constraints. Assuming the leakage consideration is added to the Bottom-Up model, the total number of constraints is

$$\#Constraints(\text{Leakage}) = m + n + m \cdot |E| + m \tag{4.50}$$

whereas the number of decision variables is increased by m, one variable per memory partition, to a total of

$$\#DecisionVariables(\text{Leakage}) = m \cdot (n + |E| + 1) \tag{4.51}$$

4.2.9 Results for Multi Memory Optimization

This section presents experimental results obtained using the different versions of the multi memory optimization approach.

For the scratchpad memories used in this section, the per access energy values shown in Table 4.3 were used to generate results. These values were determined using a subset of the CACTI cache model [WJ96] as described in Section 3.4.3 and in [BSL+02], assuming a technology feature size of $0.5 \mu m$.

Memory Size [*bytes*]	Per access energy [*nJ*]
64	0.50
128	0.57
256	0.60
512	0.69
1024	0.84
2048	1.05
4096	1.44
8192	2.14
16384	4.05
32768	6.53

Table 4.3. Per access energy values for scratchpad memories

The main memory energy values were determined using measurements on our ARM evaluation board. In contrast to the onchip scratchpad memory model, main memory accesses require a different amount of time and energy depending on the type of memory access (read or write access) and on the bit width of the access. The reasons for this fact are explained in detail in Section 3.4.3 on page 33. The values for the main memory are given in Table 4.4.

Access Width	Read Energy	Write Energy	Waitstates
1 byte	154.8 μJ	149.8 μJ	1
2 bytes	240.0 μJ	298.8 μJ	1
4 bytes	493.2 μJ	411.0 μJ	3

Table 4.4. Measured per access energy values for main memory

An important aspect that has to be considered for the experimental work concerning multi memory partitions is which set of partitions should be used. The number of combinations of different memory partitions is very large if more than three partitions are included in the model. Therefore, a systematic approach is necessary to restrict the number of possibilities to an acceptable level. The total scratchpad memory capacity is assumed to vary between 64 bytes and 32 kB, which is a useful range for the considered benchmarks. The partitioning of the scratchpad is performed according to the following rules: One memory partition that holds the entire capacity is divided into two parts of equal size. In an iterative process, the smallest partition is then repeatedly divided in two until the smallest memories reach the minimal size of 64 bytes. This procedure automatically takes care of a problem encountered during the first experiments: whenever there are two or more identical memories, the solution of the ILP problem takes a substantially longer time, since the solver is not aware of the fact that a couple of memories are alike and that accordingly, several equivalent allocation patterns exist in this case. Early pruning of the search space is thus not possible, and the memory requirements and runtimes are increased considerably. Using our proposed approach, only two memories of equal size are ever present in a setup, which helps avoid the problem. As an additional rule, the total number of memories was also restricted to a maximum of eight, which makes sense considering e.g. the area and wiring overhead required to provide a large number of scratchpad memories. Table 4.5 shows the considered memory partitions using the example of a scratchpad memory with a total capacity of 4 kB.

For all of the performed experiments, a time limit of 15 minutes was set for the ILP solution algorithm in order to generate results within a reasonable amount of time. Due to the existence of equivalent solutions when two identical memories are present, it could often be observed that a near-optimal solution was found very soon during the solving process, however the entire search space (i.e. all of the equivalent solutions) still had to be explored in order to ascertain that no better solution exists.

Total Size	Number of	number of partitions of size:						
[bytes]	partitions	4K	2K	1K	512	256	128	64
4096	1	1	0	0	0	0	0	0
	2	0	2	0	0	0	0	0
	3	0	1	2	0	0	0	0
	4	0	1	1	2	0	0	0
	5	0	1	1	1	2	0	0
	6	0	1	1	1	1	2	0
	7	0	1	1	1	1	1	2

Table 4.5. Example choice of memory partitions for 4 kB total capacity

The benchmarks that were chosen to demonstrate the effectiveness of the multi memory allocation approach are shown in Table 4.6. They consist of typical applications frequently found in embedded devices. Sorting integer elements is a task that frequently has to be performed by any computing device. The G.721 benchmark performs an encoding and decoding of sample data according to the "Adaptive Differential Pulse Code Modulation" (ADPCM) standard G.721 proposed by the International Telecommunication Union. Fourier transformations and discrete cosine transformations are frequently required operations in the domain of digital signal processing often performed on embedded devices. The reference inverse discrete cosine transformation operates on "double" data types, whereas the Fast_IDCT implementation uses integer representations.

Benchmark	Code Size [*bytes*]	Data Size [*bytes*]	Description
Multi_Sort	716	1204	Sorting benchmark (combining several sorting algorithms)
G.721	2784	2424	Encoding and decoding according to G.721 using "Adaptive differential Pulse Code Modulation"
FFT	480	15364	Integer implementation of Fast Fourier Transform (FFT)
Ref_IDCT	588	2564	Reference (float) implementation of inverse discrete cosine transform (IDCT)
Fast_IDCT	1428	6552	Integer implementation of inverse discrete cosine transform (IDCT)

Table 4.6. Selected benchmarks to evaluate the multi memory allocation approach

Comparison of Base, Top-Down and Bottom-Up

Using the energy values and memory partitions described above, results to compare the three primary approaches, i.e. the Base model, the Top-Down refinement and finally the Bottom-Up method, are provided. To clearly and concisely show the differences between the three approaches, only one single application, the Multi_Sort benchmark, was chosen for the comparison. For both the Top-Down and the Bottom-Up models, values considering all of the benchmarks are provided later in this section.

First, we show the possible improvement in memory energy when the three allocation approaches are used. Since the proposed models control the allocation of objects to the memory partitions, the results presented in this section are given in terms of energy saved within the memory subsystem. These results are given as an improvement relative to a system with only SRAM main memory (using the same access parameters as assumed in our experiments). This minimalistic point of reference was chosen since if e.g. a single scratchpad was present in the reference system, then the allocation technique used to fill this scratchpad would interfere with the comparison of the Base, Top-Down and Bottom-Up approach.

- The Base model: Results for the Multi_Sort application allocated to the partitioned scratchpad memory using the Base approach are shown in Figure 4.10. For a single scratchpad partition, the improvement of energy dissipation within the memory subsystem compared to a system with no scratchpad rises steeply up to a level of about 94% for a scratchpad capacity of 4096 bytes. Increasing the size of the single scratchpad partition does not improve the energy dissipation any further, since larger partitions require more energy per access. If a single scratchpad partition of 32 kB is used, the energy savings are reduced down to 75%. Increasing the number of partitions however leads to a better energy behavior for larger capacities: if the scratchpad is partitioned into two halves (i.e. two partitions of 16 kB each), then the relative energy savings are increased to about 85%. Further partitioning of the scratchpad leads to even higher savings, demonstrating that the maximum savings achievable for one scratchpad can be sustained or in some cases even improved. This shows that if a processor with a given amount of scratchpad memory is used, it can be beneficial to provide the total capacity not as one big scratchpad, but rather as several smaller partitions, provided the compiler is able to exploit this kind of memory architecture. Energy savings within the memory subsystem of up to 97% were obtained using the multi memory allocation technique, while the average solution time of the ILP equations was below one second.

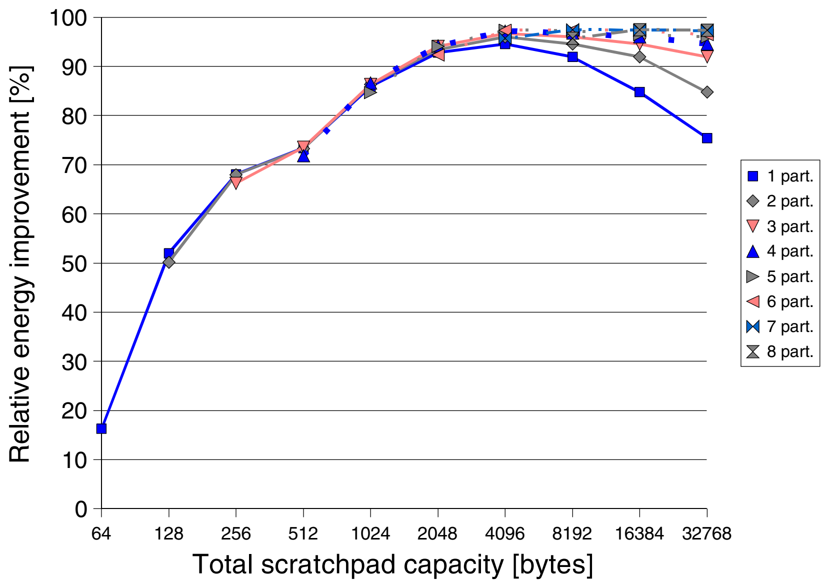


Fig. 4.10. Energy savings for the Base model compared to a system with no scratchpad

Further improvements, in particular for smaller scratchpad sizes, are to be expected if the allocation algorithm is aware of the relationship between basic blocks and functions.

- Results for the Top-Down model: The Base model does not consider the relationship of functions and basic blocks at all during the optimization of the ILP problem and thus pessimistically assumes that additional jump instructions are always necessary when a basic block is moved to any memory partition. The Top-Down model considers the allocation of complete functions in addition to separate basic blocks.

  The results obtained for the Multi_Sort application using the Top-Down approach are shown in Figure 4.11. It can be observed that for the small memory partitions, the Top-Down allocation algorithm generates the same results as the Base approach. This is due to the fact that if no complete function can be allocated because of size constraints, the same individual basic blocks as in the Base case are chosen. Once a function fits entirely into one memory partition, it is selected and allocated accordingly, leading to a reduced energy dissipation compared to the Base approach.

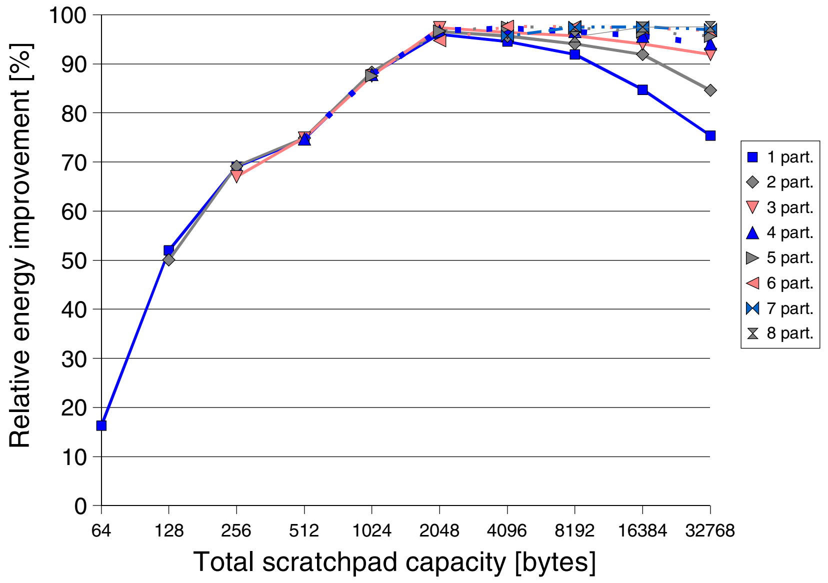


Fig. 4.11. Relative energy for the Top-Down model

The smallest scratchpad size where this becomes visible is 512 bytes: while for the Base model, the splitting of functions into separate basic blocks leads to energy savings of 71.82% for 4 memory partitions, the Top-Down model can allocate an entire function and achieve savings of 74.72%, thus outperforming the Base approach by about 3% percentage points.

For a total capacity of 1024 bytes, the improvement for the Top-Down model can be observed for all data points: while the Base model achieves savings of about 86%, the Top-Down model reaches 88% for two partitions. While this is only an improvement of 2 percentage points, considering the absolute energy values of $704.14\mu J$ and $828.22\mu J$ for Base and Top-Down model, respectively, the Top-Down model outperforms the Base model by a total of 15% with respect to absolute energy dissipation.

In summary, the Top-Down model on average generates similar, in some cases better results than the Base approach, as expected. Achievable gains of up to 15% in energy dissipation were observed compared to the Base model.

This work has shown that the new implementation of the Top-Down model, in contrast to the original formulation presented in [Hel04], is capable of generating valid results without any of the previously encountered runtime and memory requirement problems during the solution of the ILP problem. Since the Top-Down model is more complex than the Base model, the ILP solving times are increased, but the presented results can still be generated in an acceptable time, in particular due to the effect of equivalent solutions mentioned above, which results in good approximations being found early in the optimization process.

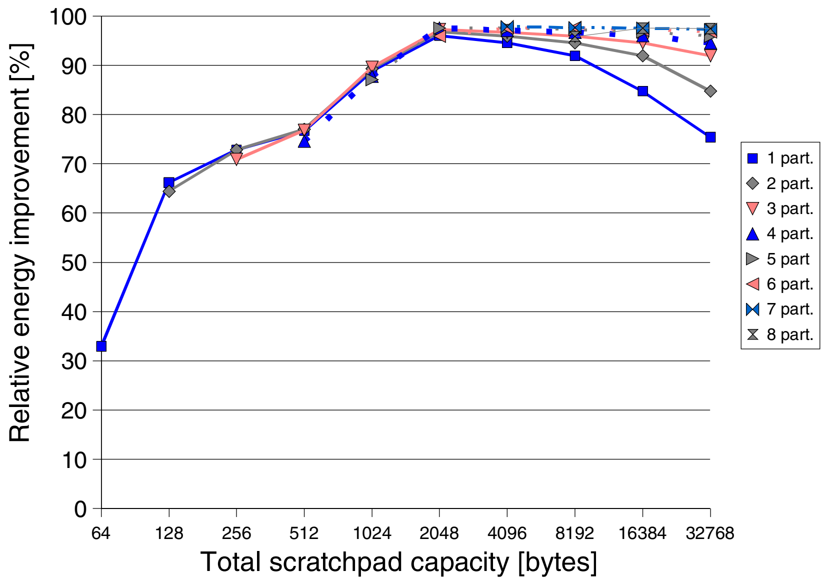


Fig. 4.12. Relative energy for the Bottom-Up model

- The Bottom-Up model: The results of the Bottom-Up approach compared to a scenario without scratchpad are shown in Figure 4.12. In particular for small scratchpad sizes, the Bottom-Up approach is able to save more energy than the Base and Top-Down approaches since it is aware of the fact that moving contiguous basic blocks to the small scratchpad is better than moving non-contiguous blocks. For the smallest single-partition setup using a scratchpad of 64 bytes, the Bottom-Up model saves 33% of energy compared to only 16% for Base and Top-Down approach. This general advantage becomes obvious for all scenarios where smaller scratchpad partitions are present in the system. This shows that for portable embedded systems where energy efficiency is a prime concern, the Bottom-Up allocation technique should be used since it is capable of exploiting even the small scratchpad memories found in these devices.

The Bottom-Up approach is thus the most beneficial allocation technique to be used in practice. The solving times usually stayed well below the 15 minute time limit, showing that the Bottom-Up description, albeit capturing the relationship among functions and basic blocks in the most accurate way, does not create an additional overhead that would prevent the efficient generation of memory layouts.

Finally, the results of the three approaches are directly compared in Figure 4.13. It shows the relative benefits of the Top-Down and the Bottom-Up models compared to the Base approach using the Multi_Sort application.

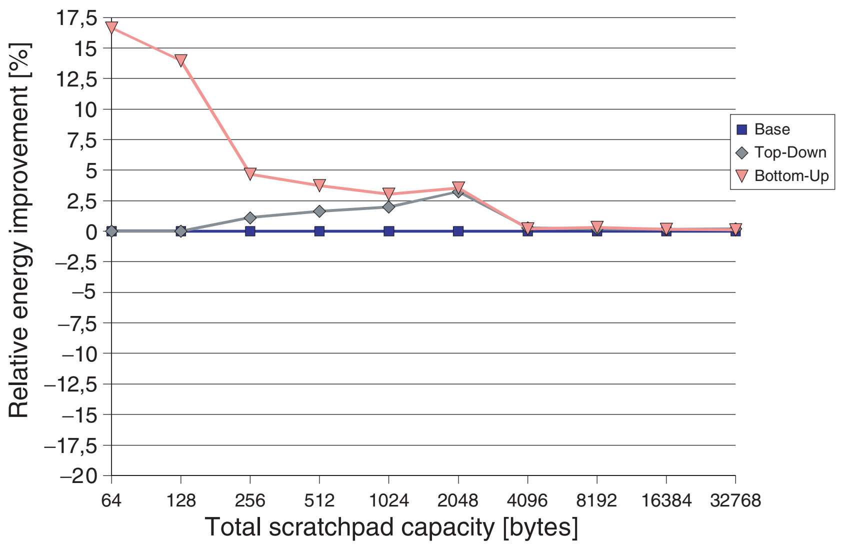


Fig. 4.13. Comparison of Top-Down and Bottom-Up relative to the Base approach

In the figure, the maximum relative energy savings (in percent) were chosen for each total scratchpad capacity and the maximum value for the Base case was subtracted, putting the Base approach at a constant "0" for comparison.

The graph thus shows savings in terms of percentage points, not absolute energy savings.

It can be seen that the Base approach forms the baseline: since it is only capable of allocating single basic blocks to memories, it can not take advantage of the connections and dependencies that are present in the application's structure. The Top-Down approach improves the situation for scratchpad capacities of more than 256 bytes, where it is for the first time possible to allocate a complete function to one scratchpad partition. For smaller scratchpad partitions, the Top-Down approach chooses the same allocation as the Base case. For the allocations between 256 bytes and 2 kB, the Top-Down approach is always marginally better than the Base case. This is due to the fact that it allocates complete functions whenever a partition is large enough to do so, which is more efficient than only allocating single basic blocks.

For the Bottom-Up allocation, the results are clearly superior to the Base case and to the Top-Down case, in particular for small scratchpad partitions, where being able to allocate contiguous basic blocks is a big advantage. Further improvements of up to 16% (in terms of percentage points) compared to the Base allocation's improvement can be observed.

The scratchpad capacity of 2 kB marks a special case for this application, since both the Bottom-Up and the Top-Down allocation perform equally well for this capacity. This is caused by the fact that for this scratchpad size and a partitioning into two 1 kB memories, two complete functions can be allocated to one scratchpad memory. This is of course beneficial compared to the Base algorithm's allocation, which randomly distributes the basic blocks among the memories.

The presented savings in terms of percentage points were obtained by comparing the relative percentual improvements of the different approaches. If the minimum absolute energy values are used as the basis for comparison, the achieved relative gains are higher. Consider the 2 kB scratchpad capacity, where the reduction of the energy by 94% for the Base case and by 97% for Bottom-Up results in a gain of 3 percentage points. If the corresponding absolute energy values of $356\mu J$ and $142\mu J$, respectively, are used for comparison, then this corresponds to a significant improvement of 60% for the Bottom-Up approach.

For the remaining scratchpad capacities, the disadvantages of the Base approach are hardly visible any more. All memory objects fit into one scratchpad partition and therefore, no more additional jumps are actually inserted into the code. The small advantage of the Bottom-Up approach stems from the possibility to use e.g. a scratchpad partitioned into two instead of one single scratchpad partition without additional overhead.

To summarize, the allocation using the Bottom-Up approach yields the best results for all scratchpad configurations. The most benefit from this approach can be obtained for small scratchpad capacities due to its capability of efficiently exploiting small partitions by allocating contiguous basic blocks. The Top-Down approach does not offer as much benefit compared to the Base approach, but it can be used if small functions are to be allocated to scratchpad partitions.

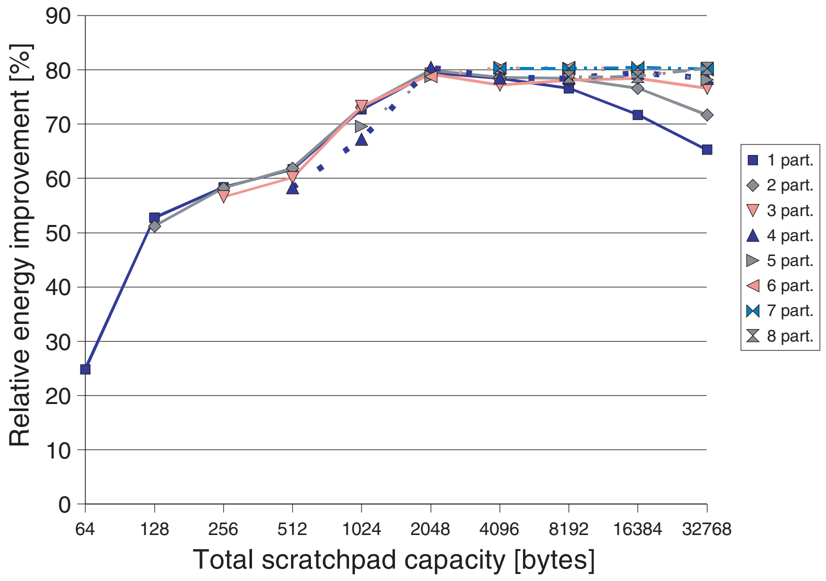


Fig. 4.14. Multi_Sort: total energy savings (CPU and memory) using the Bottom-Up model

Since our simulation framework also allows the evaluation of overall system energy, Figure 4.14 shows the overall energy savings, including CPU and memory subsystem, obtained by the Bottom-Up model compared to a system without scratchpad. Since scratchpad memories have a significantly shorter access time compared to the main memory (cf. Table 4.4), the energy dissipated within the CPU is also reduced, albeit not at the same rate as the energy savings in the memory system alone. Still, overall energy savings of up to 80% can be achieved by using partitioned scratchpad memories.

Figures 4.15 and 4.16 show the maximum energy improvements obtained using the Top-Down and the Bottom-Up approach, respectively, for all considered benchmarks. For each total scratchpad capacity, that number of partitions was chosen which resulted in the highest energy savings. The results are again compared against a system without scratchpad. Both figures show that the trends for the Multi_Sort application are confirmed for the other applications: the energy dissipated in the memory subsystem is significantly reduced by the use of partitioned scratchpad memories, and the savings can be sustained for large capacities since a number of smaller scratchpad partitions with their reduced energy dissipation can be used to obtain the total capacity.

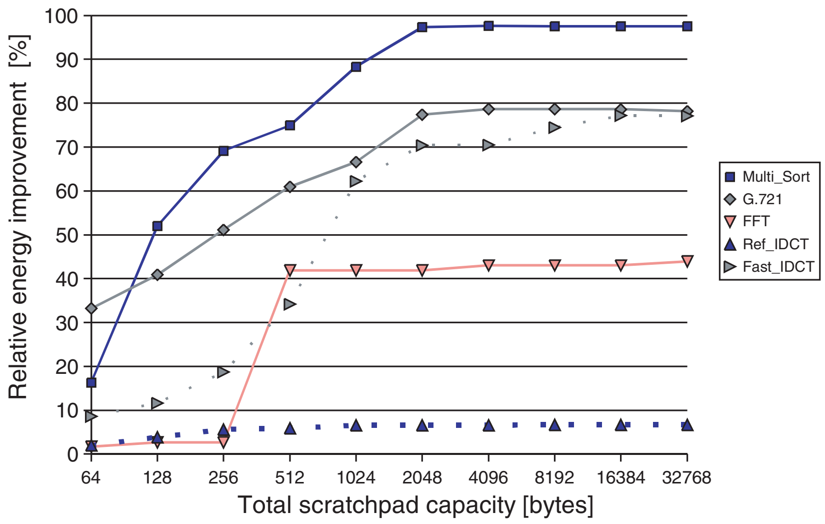


Fig. 4.15. Maximum energy savings using the Top-Down model for all benchmarks

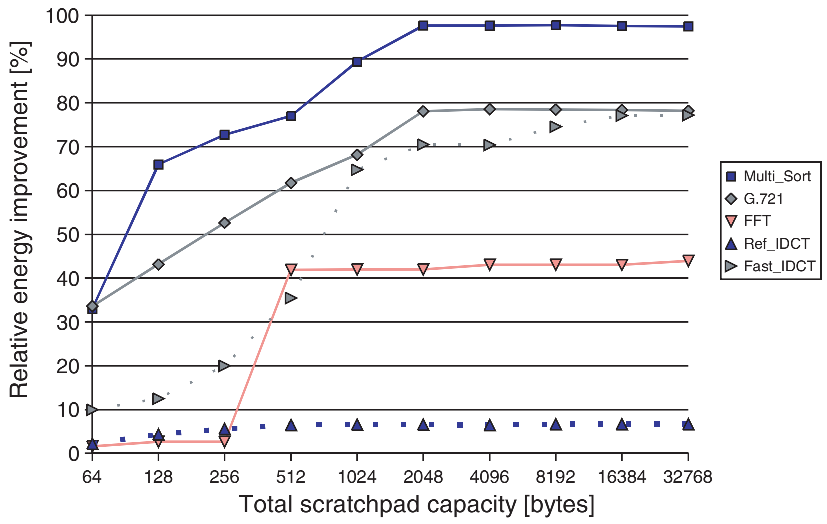


Fig. 4.16. Maximum energy savings using the Bottom-Up model for all benchmarks

Comparing the two figures, it becomes clear that the Top-Down approach performs as well as the Bottom-Up approach for the considered benchmarks, the only exception being Multi_Sort for small scratchpad capacities. This is due to the fact that all other benchmarks contain a larger number of smaller functions that can be allocated even to small scratchpad partitions. For most considered total capacities and number of partitions, the Bottom-Up allocation slightly outperforms the Top-Down model.

Looking at more detail at the curves for the individual applications in Figure 4.16, the results obtained for the "Ref_IDCT" benchmark are surprisingly low compared to the other benchmarks, meaning that only little energy could be saved. This is due to the fact that floating point values are used in this application. Lacking a floating point unit, the ARM7 processor has to perform the floating point calculations by calling library functions. These libraries are linked to the executable only after the compiler has finished its analysis. Since nothing is known about the used library functions, they can not be allocated to scratchpad partitions, despite the fact that a high percentage of the time is spent in these routines. The results for this benchmark show one limitation of the presented approach: precompiled libraries are not suitable for multi scratchpad allocation. If library calls are required in an application, the source code of the libraries should be provided in order to allow all functions to be mapped to the appropriate memory partition.

For the "FFT" application, the results show up to 40% of energy savings in the memory subsystem compared to a system without scratchpad. The fact that less energy could be saved compared to the other benchmarks is due to the data access style used in this program: the Fourier transform is highly data dominated (cf. Table 4.6), and in order to provide efficient access to the large arrays, one of the arrays is solely accessed using pointer arithmetic and pointer dereferencing. Generally, a complex pointer alias analysis is required in a compiler to determine which array is being accessed when pointers are used. Since encc does not include a pointer alias analysis, it can not determine the array that is being accessed by the pointer in the application and thus finds no accesses to one of the arrays at all. The ILP solver consequently decides not to allocate the array to the scratchpad memory, since this would not improve the value of the objective function. One of the frequently accessed arrays thus stays in the main memory, and the high amount of energy per access to this array is responsible for the reduced gains for this application. This example shows that the extent of achievable savings not only depends on the exact formulation and solution of the allocation problem, but also on the information provided by the compiler, and by the programming style found in the application program.

Finally, Figure 4.17 shows the maximum improvement that could be obtained over the previous work, namely the static scratchpad allocation technique using only one single partition as presented in [SWLM02]. As in Figure 4.16, that partitioning was chosen which resulted in maximum energy savings. Using partitioned memories instead of a single scratchpad yields energy improvements in the memory system of up to 22%.

If the comparison is restricted to only that single scratchpad partition size that yields maximum savings and the corresponding partitioned capacity, then the obtainable savings are reduced to around 7% which is mainly caused by the limited size of the used benchmarks. Considering the fact that ARM based

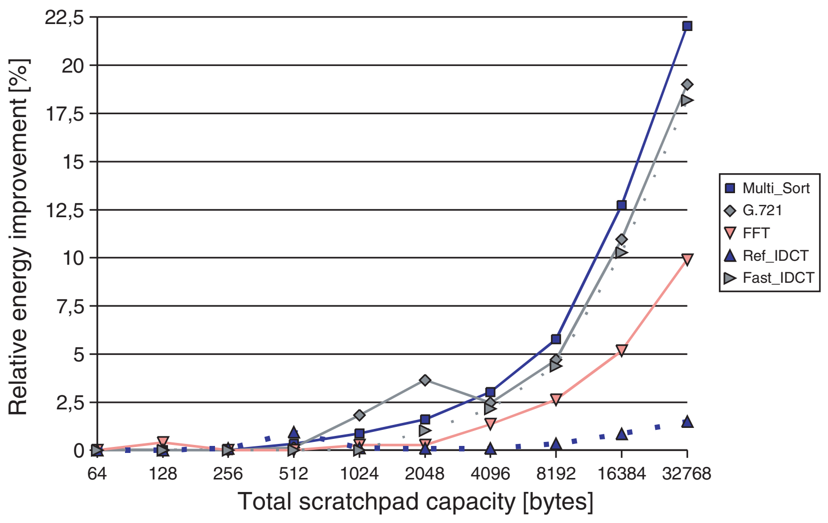


Fig. 4.17. Maximum energy savings using the Bottom-Up model compared to a single partition for all benchmarks

systems today are shipped with 32 kB of scratchpad memory (partitioned into 16 kB data and 16 kB instruction TCMs) [LSI04], we can conclude from the presented results that if a system has a certain scratchpad size available, it is advantageous to provide the total capacity in multiple partitions instead of one single memory.

The following sections consider the extensions supporting the ARM Harvard Architecture during multi partition allocation and the compiler guided memory configuration based on the leakage energy of memories.

Results for the ARM TCM Architecture

To compare the possibilities provided by ARM's Harvard style architecture concerning memory allocation with the results from the previous section, all benchmarks are once again evaluated, this time assuming the Harvard style memory setup found on the ARM9 and ARM11 processors (cf. Figure 4.9). The two available memory partitions have the same storage capacity and are capable of only holding either instructions or data. The Bottom-Up approach is used as the reference allocation technique since it generated the best achievable results so far. The comparison shows that restricting the free choice of allocating instructions and data to the available memories incurs an overhead in terms of energy dissipation since the memories' storage capacity can not be exploited in an optimal way: frequently executed instructions that should be stored on the scratchpad memory may not fit into the instruction partition, whereas the data partition still has sufficient capacity.

The absolute energy values for the Multi_Sort benchmark are shown in Figure 4.18. The reference allocation technique, the Bottom-Up approach, clearly outperforms the Harvard style allocation for small memory sizes. Once both instruction and data memories are large enough to hold all the hot spots of the program, the two allocation techniques result in identical energy values.

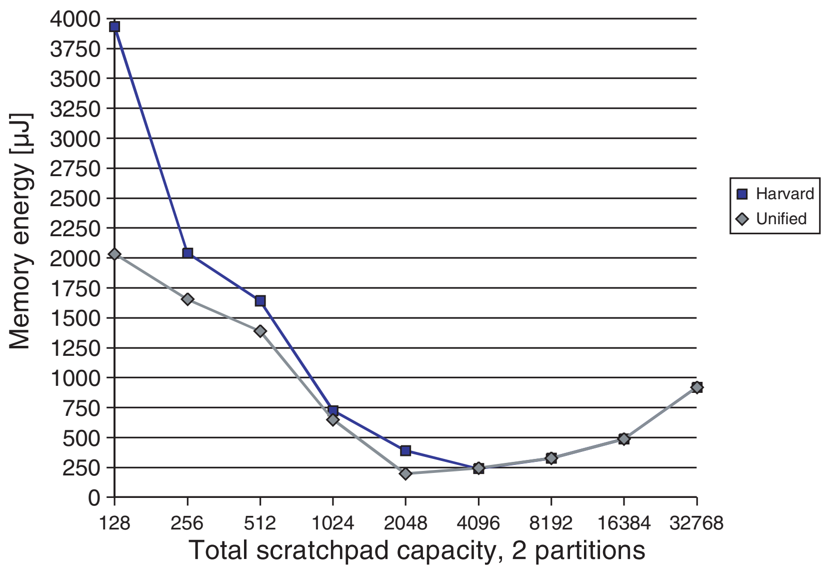


Fig. 4.18. Bottom-Up vs. Harvard allocation for Multi_Sort benchmark

Figure 4.19 provides a comparison of the Harvard style allocation with the results obtained by the Bottom-Up approach for all considered benchmarks. The energy values obtained using the Bottom-Up allocation technique are normalized to the baseline "0". Only memory configurations with two equally sized memory partitions were selected to allow a direct transfer of the obtained results to the TCM architecture found in current ARM designs.

It can be observed that the energy dissipation can suffer severely by using separate instruction and data memories. For the Multi_Sort benchmark, an overhead of up to 97% can be observed for two 1 kB scratchpad partitions. Most of the other applications show a significantly increased energy dissipation compared to the Bottom-Up approach. For two benchmarks, namely "Ref_IDCT" and "FFT", the results using the TCM model are nearly identical to those achieved with the Bottom-Up model. For "Ref_IDCT", this is due to the reasons already mentioned in the previous section: it uses function calls to unallocatable library functions to handle float data types, which has a negative effect both for the Bottom-Up and for the TCM model. For the "FFT" application, the used data arrays are so large that they do not fit into the

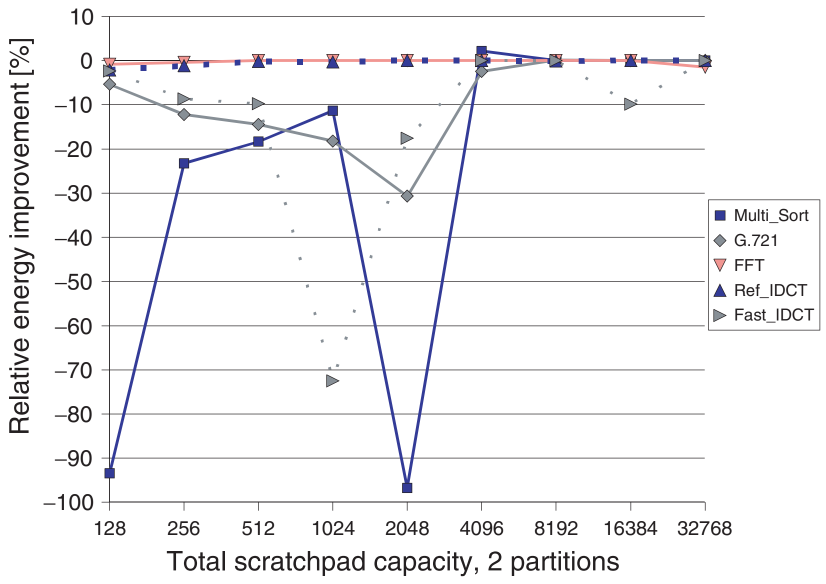


Fig. 4.19. Bottom-Up (normalized to “0”) vs. Harvard allocation

smaller scratchpad memory partitions no matter which allocation technique is being used. With the scratchpad partition sizes used in the experiments, the number of arrays that fit into the scratchpad are identical for both Bottom-Up and TCM allocation, leading to similar results. This picture may change in favor of the more general Bottom-Up allocation model if a setup is chosen in which both scratchpad memory partitions are large enough to accommodate a data array. The TCM model could only allocate one array to the scratchpad data partition, whereas the Bottom-Up allocator would allocate both data arrays to the two available scratchpad partitions.

In general, it is advantageous if the compiler can use the available scratchpad partitions freely, without any restriction concerning their ability to hold only instructions or data. A free allocation of memory objects to the scratchpad partitions results in an optimal exploitation of the available resources.

Results for Compiler Guided Memory Configuration

By increasing the amount of leakage energy assumed for the memory partitions, the total number of partitions that are chosen for allocation by the compiler is expected to decrease when the leakage energy value is so high that it overcompensates the benefit of reduced energy per access. Figure 4.20 shows the number of partitions utilized by the compiler for increasing leakage energy values using the Multi_Sort benchmark as an example.

The compiler was given a wide choice of memories: two identical memories of each capacity from 64 bytes up to 1 kB were provided as input. The first

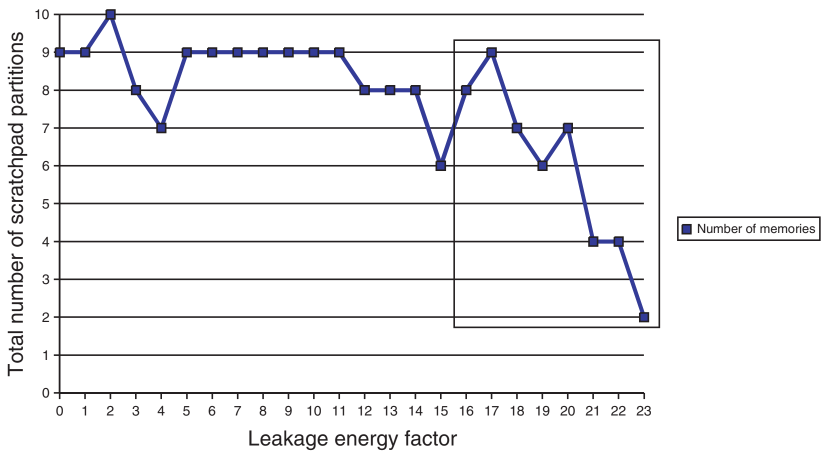


Fig. 4.20. Number of partitions for increasing leakage energy

data point labeled "0", represents a leakage current of $2^0 = 1$ time the access energy for each of the memories. The following entries increase the leakage energy in an exponential way moving from left to right: every subsequent data point assumes twice the leakage value of the previous one. In addition, since the access energies only increase at a very slow rate for small memories (cf. Table 4.3), an additional factor was included to ensure that larger memories have a higher standby power dissipation. This effectively puts the compiler into a dilemma: on one hand, a minimal number of scratchpad memories is to be selected since only those partitions that are being used will contribute to the leakage energy. If a large number of objects should be allocated to the few memories, however, memories with large capacities have to be chosen, which in turn consume more standby energy.

For a large region of leakage energy values, the number of memories nearly remains constant at around nine. At a leakage energy factor of 2^{15} times the energy per access, only six memories are chosen. Since the Multi_Sort benchmark performs a total of 227154 memory accesses (which is below 2^{18}), the leakage energy in this region approaches the total access energy of the small scratchpad memories. The gain achievable by using an additional scratchpad memory is thus reduced up to the point where it makes sense to also utilize the large and energy hungry main memory. Since the main memory is assumed to be always required as background memory in an embedded system, it was considered to have no additional leakage energy, which is why it becomes increasingly attractive for higher scratchpad leakage values. The main memory is being utilized starting from a leakage energy factor of 2^{16} (shown as an unfilled box in Figure 4.20). The use of the main memory with its high energy per access changes the overall situation for the compiler in such a way that more scratchpad memories are again being used, but only up to a leakage energy factor of 2^{17}. From that point on, the number of memories decreases up to the point where only two memories are being used: the main memory and one additional scratchpad memory.

Figure 4.21 shows the obtained results in some more detail: As in the previous figure, the leakage energy factor increases exponentially on the x-axis, but the y-axis uses stacked bar elements to show which memory partitions were being chosen for a particular leakage energy value.

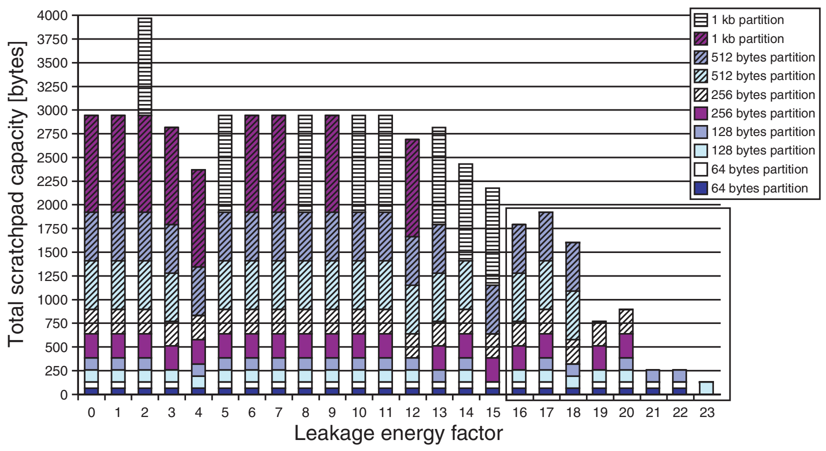


Fig. 4.21. Selected scratchpad memory partitions for increasing leakage energy values

The height of the stacked bar elements representing the different memory partitions corresponds to the size of the memory partitions. The total memory capacity corresponding to a certain leakage energy factor can thus be determined by checking the bar's total height on the y-axis. The use of main memory is again indicated as an unfilled box around the rightmost stacked bars.

For small leakage energy factors, the large scratchpad memory partitions are preferred by the algorithm, since they offer sufficient space for a large number of memory objects. As the leakage factor is increased, however, the largest memories receive the highest penalty concerning their contribution to leakage. Once the main memory with its relatively high per-access costs, but without any leakage energy dissipation, is sufficiently attractive to be used, only scratchpad partitions smaller than 1 kB are being used in the system. As the leakage further increases, smaller and fewer scratchpad memories are chosen. The rightmost data point only considers using a single scratchpad of 128 bytes capacity, whereas the two previous points additionally use two 64 bytes partitions.

It was thus shown that it is straightforward to extend the ILP based formulation of the multi memory allocation problem to account for additional parameters and to more accurately model the behavior of the entire system. Using the leakage factor of the scratchpad memories as an input parameter, the designer can have the compiler choose an appropriate memory architecture design and have objects allocated to the proposed memories at the same time. This is of particular interest with respect to the increasing popularity of fixed processor cores which can be modified according to the customer's requests in order to meet the goals for his individual design.

4.3 Impact of Scratchpad Allocation Techniques on WCET

In addition to the energy and performance issues discussed so far, another notion is gaining more importance in the design of embedded systems. In particular safety-critical embedded systems, like the ones used in avionics and automotive environments, are expected to be extremely dependable. This applies in particular to the timing requirements that can frequently be found in these areas. Timing predictability is therefore becoming one major concern in devices that have to meet real-time constraints. One well-known and widely used example for a hard real-time system is the airbag found in nearly every car today: if the airbag controller fails to trigger the airbag such that the airbag is fully inflated within about 40 milliseconds following a crash, then the whole system is worthless. On the contrary, inflating the airbag at a later point in time may actually pose a threat to the emergency staff or to possibly wounded passengers. For hard real-time systems it is therefore critical that their timing conditions are never violated. In general, the guaranteed timing behavior of the system has to be determined and fixed at design time. This guarantee is usually given by supplying an upper bound for the maximum reaction time of the system, called the Worst Case Execution Time (WCET). This upper bound must never be exceeded since lives may depend on this particular timing property. Beside hard real-time, there are also systems that have to obey so-called soft real-time constraints, where the violation of timing constraints, despite being undesirable, does not cause a lot of damage. An example for this is an MP3 player: if it cannot decompress the audio samples in time, the listener will notice an interruption in the audio stream. If this happens too frequently, the device will lose market shares due to unsatisfied customers.

In order to avoid costly re-design cycles, it is vital that the timing properties of a system can be determined at design time. The important metric for the predictability of a device is its worst case execution time, i.e. the maximum time that the system will take to perform a certain task under worst case conditions. Several ways of specifying the WCET are possible: as in the airbag example, it may be the time from an event to the completion of the system's action. In other scenarios, the time between two events within the system must not be exceeded. Since reactive systems are not in the scope of this work, the notion of WCET considered here is the maximum execution time of an application. The results of WCET analysis are thus provided in the form of an upper bound for the execution time of the application under consideration. This upper bound is guaranteed to be safe by the WCET analysis, meaning that it represents an upper bound that will never be violated under any circumstance. While it is mandatory at least for hard real-time systems that the upper bound is guaranteed to be safe, it is also desirable to provide an upper bound that is as tight as possible, i.e. a bound that is as close as possible to the actually occurring worst case execution time. The advantage of a tight upper bound is thus that the system can be implemented using the minimal resources required to meet the WCET deadline. If the WCET is not tightly bounded, the designer may decide the system has to be more powerful. This results in an oversized system, which is an expensive and unnecessary safety precaution.

Since real-time aspects are of growing significance in different sectors, this section first takes a look at a couple of architectural features that have an influence on the predictability of a system. Many processors today use components to enhance performance, e.g. pipelines, memory hierarchies, branch prediction units and others. While in the general, average case, these architectural enhancements improve the performance of computing systems, they are not necessarily helpful in providing timing predictability:

- Pipelining: Nearly all processors designed and built today include some kind of pipeline. Pipelines increase the throughput of the processing units by performing several tasks in parallel. The standard five stage pipeline frequently found in many different processors consists of the stages Fetch, Decode, Execute, Memory Access and Write Back. Every instruction has to pass through these stages during its execution. By performing these steps in an interleaved way and not waiting for one instruction to go through the entire processing, the throughput can be improved. If each stage takes one cycle, then one instruction can be completed in every cycle once the pipeline has been filled. Of course, structural, control and data dependency hazards which interrupt this ideal pipelining have to be taken into account [HP03]. These hazards are one issue that make pipelines difficult to predict concerning their timing. The time it takes for one single instruction to pass through all stages of the pipeline depends not only on the instruction itself, but also on all instructions that are in the pipeline with it. If the instruction itself is processed without any additional delay, it may enter the first stage of the pipeline in cycle i and be completed (i.e. leave the fifth stage of the pipeline) at cycle $i + 5$. However, if any other instruction in the pipeline causes a hazard or a pipeline stall, then the considered instruction is also stalled. The frequency and effect of hazards is difficult to predict at design time, since e.g. control flow hazards depend on conditions that are evaluated at execution time, like the value contained in a register at a certain point in time. The outcome of these decisions is difficult, if not impossible, to predict at the design time of the system. One simple way to provide a safe upper bound for the maximum execution time of an instruction sequence or an entire application is therefore to assume that the maximum number of possible pipeline hazards will actually occur. This is obviously a very pessimistic and conservative approach, but it is the only simple technique that can guarantee that no timing deadline will ever be violated. Since such a high overestimation is usually not acceptable, complex pipeline analyses have to be used in order to determine sufficiently tight upper bounds for the WCET. The advantage of using a pipeline in the average case thus turns into a disadvantage concerning predictability.

- Branch Prediction: Since branch predictors take their decisions in a dynamic way at runtime, the reasoning concerning pipelines also applies to branch prediction: in the average case, the branch predictor may have a probability of more than 50% in successfully predicting the outcome of a branch decision, i.e. whether a branch is taken or not. In case it successfully predicted the branch, some cycles can be saved since the predicted branch target instruction can immediately be fetched, avoiding a pipeline stall. If maximum execution times have to be guaranteed, however, the designer will have to assume (in a simple approach without additional complex analyses) that the branch prediction unit will mispredict every single branch to be on the safe side and to guarantee that the deadline will always be met. Always mispredicting a branch has a higher negative impact than using no branch prediction at all. This shows that while branch prediction in general has a positive effect on the average case runtime, it results in worse, less tight bounds on the guaranteed worst case execution time.
- Caches: It has been mentioned that most modern computer systems utilize a memory hierarchy in order to overcome the increasing speed difference between processor and memory. The most frequently used architectural feature to bridge this gap is the cache. Whenever an instruction or a data element is accessed, the cache first checks whether it contains the requested element. If it does, then the item is directly supplied from the cache, saving an access to the large, slow and energy consuming main memory. If the element is not in the cache, then the main memory has to be accessed and the element is both passed to the processor and saved in the cache for future accesses. As for pipelines and branch prediction, the cache takes its decisions concerning cache hits or misses at runtime, and it is hardly possible to determine the current working set contained in the cache at design time, making it difficult to determine which memory access will result in a cache hit. Once again, in order to guarantee that deadlines are never violated, and without resorting to complex cache analysis techniques, the designer has to assume that all cache accesses will result in a cache miss

and consequently in an access to the main memory. Since several elements are usually transferred to the cache in case of a miss, this results in more memory accesses and bus traffic than if no cache was used, showing that caches generally help to improve performance in the average case, but do not help concerning the predictability of a system.

- Scratchpad Memories: This alternative to using caches has already been presented in-depth in the previous section of this work. Scratchpad memories are small memories that, in contrast to caches, do not take any decisions dynamically at runtime. They are controlled by the programmer or the compiler at design time. It is thus always known which elements are allocated to the scratchpad memory. Even in the case of dynamic allocation where the contents of the scratchpad memory can change during the execution time of the application [SGW$^+$02, VWM04b], the compiler controls the insertion of code that swaps instructions and data between the scratchpad and the main memory, which again results in full control over the location of elements within the memory hierarchy and over time. Since no dynamic decisions are taken, there can never be a hit or a miss when the scratchpad is accessed. Rather, each memory access either goes to the main memory or to the scratchpad memory. If the timing properties of the used memories are known at design time, then an accurate and tight upper bound for the worst case execution time can be determined. Scratchpad memories are thus useful both for the average case and for WCET analysis. As shown in the previous sections and in numerous publications, scratchpad memories achieve significant savings concerning both execution time and energy [SWLM02, VWM04b, WHM04]. But in contrast to the other features described in this section so far, they do not impair worst case execution time analysis. On the contrary, they are helpful in providing tight upper bounds on the WCET: if a program is found to execute faster due to the use of a fast and energy efficient scratchpad memory, then this performance improvement can be expected to translate directly into an improved upper bound for the WCET analysis. This will be investigated and shown using experimental results in the further course of this section.

The remainder of this section is structured as follows: after taking a look at related work in the field of real-time embedded systems and worst case execution time analysis, the tools and the workflow used to generate results will be presented. The impact of a statically allocated scratchpad memory on the worst case execution time is then investigated, followed by a direct comparison of the scratchpad memory with a cache. Finally, the dynamic allocation algorithm presented in [VWM04b] is used to show the effect of this allocation strategy on the predictability of an embedded system.

4.3.1 Related Work

Due to the ever increasing need for improved computing power, modern systems have to become faster and more powerful in order to meet performance expectations. Most performance-enhancing features target the average case performance, which is the dominating factor for marketing a system, since the average case performance is the speed generally observed by a user. Processors and computing systems are becoming increasingly complex by including different kinds of performance-enhancing features that mostly rely on dynamic decisions taken at execution time of the system. Some examples like branch predictors and caches were mentioned in the previous section.

In contrast to standard PCs and high-performance computer systems, systems that have to meet real-time requirements do not focus on the average case performance, but should rather provide a high level of timing predictability. This means that even at design time, tight upper bounds for the worst case execution time of the device can be provided. These guaranteed upper bounds will never be violated during the actual execution of applications on the system. Timing predictable systems in general only contain those performance enhancing features that will still allow the specification of a tight upper bound on the execution time. For this reason, many of the features found in modern processors are not included in such real-time capable, timing predictable systems. One of the obvious design decisions is not to use a cache in such systems, since in general, caches do not improve the worst case performance. If caches or other features that take decisions dynamically at runtime are present, the required worst case execution time analysis techniques become increasingly difficult [HLTW03].

In particular when caches or other features that target the average case performance are present in a system, simulation must not be used to validate the timing of safety-critical systems. Simulation can provide a general idea about the expected average case performance, but there is no guarantee that the given deadlines will always be met, even if the deadlines are never violated during simulation. The use of different input data sets may lead to a completely different picture, in particular in the presence of data caches. The task of finding a worst-case input set is in general not feasible for arbitrary applications: no guarantee can be given that no other input set can ever be found that will cause a longer execution time.

Since soft real-time is of growing significance, the idea of making WCET analysis easier for such systems is becoming popular. For systems that do not have to guarantee hard real-time constraints, simulation with an assumed worst-case input set is being used as a cheaper alternative to an actually guaranteed WCET [Pus99]. The results generated by this method are sometimes called “observed worst case execution time”. While WCET analysis becomes a lot easier when the upper bound is not required to be absolutely safe, all WCET analysis results presented in this work are guaranteed to be safe, since this is the notion of WCET used in the tools chosen to generate results. Since simulation is not a viable method to determine safe upper bounds, advanced and complex analysis techniques have to be used to be able to guarantee correct timing even in worst-case scenarios. This analysis is performed in WCET analysis tools.

A general overview over available WCET analysis techniques for the architectural features mentioned above can be found in [PB00]. Of particular interest is the work by Li et al. [LMW95], who consider the presence of a direct mapped instruction cache in the WCET analysis of embedded systems. A cache conflict graph is used to approximate the behavior of the cache and to determine the total number of hits and misses. A follow-up paper [LMW96] extends the work to also cover set associative instruction caches as well as data and unified caches. One solution to the problem of considering data caches presented in [Lun02] is the introduction of predictable data structures, which should be used by the programmer for timing critical code. Tan et al. [TM04] extended the consideration of caches to also cover the case of multi tasking systems. In this case, the preemption of tasks can lead to additional cache miss overhead which has to be considered and evaluated, further complicating cache analysis.

A concept for separating program path analysis and microarchitectural analysis into two steps in order to reduce the complexity of WCET analysis is presented in [TFW00]. Results are reported to be comparable to combined analysis techniques. This approach is also used in aiT [Abs04b], a commercial WCET analysis tool that is available for several processor and cache architectures. aiT is actively used in industry, e.g. by Airbus Industries in order to determine upper bounds for the execution times of critical avionics software. As input, the tool takes an executable for a specific platform along with user supplied annotation data concerning e.g. loop bounds and access addresses as well as architectural information concerning the memory layout. The tool then generates a safe upper bound for the expected WCET. Using aiT, the elaborate (if at all feasible) task of finding input sets for which a simulation run yields the maximum execution time is no longer required. The commercially available version of aiT for the ARM7 processor is currently not equipped with a cache analysis. AbsInt GmbH provided us with a simple experimental cache analysis for the ARM7 cache that uses only a subset of the analysis techniques available with commercial versions of aiT [Fer97]. One of the difficulties with caches integrated into ARM processor cores is the fact that they use a random replacement policy, making precise estimates for cache behavior difficult. For caches that use an LRU replacement, WCET analysis can generally determine tighter bounds.

Summarizing the cache-related previous work, it becomes clear that a lot of work has gone to the integration and analysis of caches concerning their impact on a system's WCET. Despite the existence of complex and often time-consuming analyses, designers of hard real-time constrained systems in general still refrain from using caches in their designs. In this context, it does make sense to consider the use of small, fast scratchpad memories instead of caches in time constrained systems: the benefits concerning performance are comparable to those of caches, and in addition scratchpad memories are more energy efficient than caches: in [SWLM02], 23% of energy savings are reported when a scratchpad is used in the system instead of a cache. At the same time, no further analysis techniques are required to determine the WCET of a system when only scratchpad memories are being used, since no uncertainty concerning memory access timing is introduced into the system. A scratchpad memory can simply be introduced as a new, distinct memory region with certain properties, e.g. a fast access timing.

Scratchpad memories are becoming more popular and are now widely available in processors for embedded systems, e.g. as Tightly Coupled Memories (TCM) in the ARM9 processor series [ARM00]. Some methodologies of actively exploiting scratchpad memories by having the compiler allocate memory objects to the fast and energy efficient memory regions has already been presented in this work. All of the approaches fix their decisions at compile time and thus do not depend on unpredictable dynamic decisions to be taken at runtime.

Both static allocation techniques [PDN99b, SWLM02] and their dynamic counterparts [KKS01, SGW^{+}02, VWM04b] keep the contents of the scratchpad memory under full control of the compiler or the programmer, making these methods inherently predictable. The integration of scratchpad memories instead of caches is a viable and promising alternative approach which allows the system to benefit from a performance gain comparable to that of caches while at the same time maintaining predictability. The fact that caches, despite requiring complex techniques, can have a negative impact on the predicted WCET is shown in the generated results. The estimated WCET for scratchpad memories on the other hand scales with the achieved performance gain at no extra analysis cost.

4.3.2 Tools and Workflow

The experimental work presented in the following sections is based on the idea of allocating instructions and global data to the scratchpad memory and evaluating its impact both on the average case execution time, using simulation with typical input data, and on the worst case execution time, using the aiT analysis tool. Results for both a static and a dynamic allocation of objects to the scratchpads are presented. For the static allocation, the Bottom-Up allocation technique presented in Section 4.2.6 was used to allocate instructions and data to the scratchpad. To reduce the complexity of the presented results, the approach was restricted to only consider a single scratchpad partition. Using this configuration, the multi memory allocation algorithm generates similar memory layouts as the single scratchpad memory allocation algorithm described in [SWLM02]. Using multiple scratchpad partitions will not change the general statement of the results, since for the WCET analysis, only the timing and not the energy dissipation of the scratchpad partitions is being considered. Since all scratchpad partitions were assumed to be accessible within one processor cycle, several partitions can be merged to form one larger partition. Except for the additionally required longjumps between the memories, the timing determined both by the simulated program and by the WCET analysis will not change.

Note that the results presented in this work for the first time cover the allocation of individual basic blocks combined with WCET analysis: both in [WM04] and in [WM05], only complete functions were allocated to the scratchpad memory using the allocation algorithm described in [SWLM02], since this reduced the number of annotations concerning longjumps between the memories. In this work, the more fine-grained allocation of basic blocks is used for the first time, leading to better utilization in particular of the smaller scratchpad memories.

In the second part of the experimental evaluation, the approach presented in [VWM04b] was used to allocate memory objects to the scratchpad memory in a dynamic way. The generated programs are again both simulated and analyzed using aiT, as described in the following section.

In order to analyze the generated programs using a WCET analysis tool, detailed information about the used memory architecture, including main memory and scratchpad memory timing and address information, has to be provided to the tool. Apart from this, no further information or analysis is required compared to a system that only uses main memory.

To compare the impact of the proposed scratchpad allocation algorithms on WCET with caches, an executable generated without using the scratchpad optimization in the compiler is simulated using unified, four-way set associative caches of different sizes. WCET analysis is performed using a cache analysis tool for the ARM7 processor.

An overview over the workflow used to compare the impact of scratchpads and caches on WCET analysis results is shown in Figure 4.22. The benchmark programs, written in the C programming language, are compiled using the encc compiler (cf. Section 3.6) into binary executable files.

The left branch of Figure 4.22 shows the scratchpad setup: The compiler reads the size and the energy per access of the scratchpad to be used. This information is used to generate the ILP representation of the “Bottom-Up” allocation problem (cf. Section 4.2.6). After solving the optimization problem, basic blocks, functions and global data elements are statically allocated to the single scratchpad memory. For the second set of experiments involving a scratchpad memory, the dynamic allocation approach presented in [VWM04b] was used within encc to distribute the memory objects to the scratchpad memory at runtime.

Note that just as in Section 4.2, energy dissipation is the only metric that is used in the compiler’s cost function to determine the allocation of memory objects to the scratchpad and main memories. No notion of worst case execution time is included in the compiler at this time.

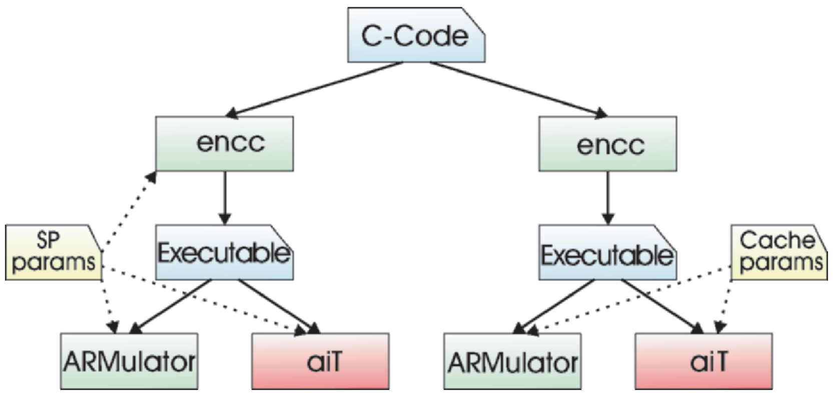


Fig. 4.22. Workflow for evaluation of scratchpad memories on WCET

The generated executable is then simulated using the workflow presented in Section 3.6.1. The ARM instruction set simulator ARMulator is used to determine the number of cycles required to execute the benchmark using a typical example input data set, taking into account the reduced access latencies of the scratchpad memory compared to main memory. The result of this step is a simulated average case execution time.

The influence of using a scratchpad on WCET was determined using the aiT worst case execution time analysis tool developed by AbsInt Angewandte Informatik GmbH [Abs04b]. aiT takes the executable of the application as input. To analyze the compiled binary file, it extracts information about the program structure, most of which is done automatically. The remaining information like loop bounds that could not be determined automatically or information that is not included in the application itself, e.g. concerning memory regions, has to be provided by the user. The required annotation information will be presented in the following sections. Using its analysis techniques, aiT then determines the worst case execution time and provides a guaranteed, yet tight upper bound for the execution time of the application. The aiT tool includes a detailed pipeline-analysis that is capable of predicting the behavior of the ARM7 processor's pipeline, thus enabling tight upper bounds despite the presence of a pipeline. Information about the pipeline state can be generated from aiT for different spots in the application program.

Since changing the capacity of the used scratchpad memory results in different memory layouts and thus different executables, code generation, simulation as well as the annotation and analysis of the executable using aiT has to be repeated for each of the considered scratchpad sizes.

The workflow used to evaluate the impact of a cache in the processor is shown in the right hand side of Figure 4.22. Since using a cache in the architecture is considered to be transparent for the compiler, the cache is not considered during the code generation step. Information concerning the cache architecture is on one hand required to perform a simulation of the application program assuming a cache is present in the system, again generating an average-case execution time using the same typical input data that was also used for the scratchpad simulation. On the other hand, information concerning the cache architecture is also required for the WCET analysis. In contrast to using a scratchpad memory, a complex cache analysis is required for this purpose. While AbsInt provides commercial cache analysis tools using sophisticated algorithms [Fer97] for different processors, no thoroughly evaluated and certified cache analysis currently exists for the ARM7 processor. On our request, however, AbsInt provided an experimental cache analysis for the ARM7 which was used to generate the results presented in this section. While this cache analysis does not contain all the features available in commercial versions of aiT, it still allows an evaluation of how caches affect the worst case execution time. Of course, the simplified cache analysis that does not consider all possible effects within the cache can potentially overestimate the cache's contribution to WCET. Using a full-featured cache analysis may improve the results obtained for the cache, but it is not very probable that the good results concerning WCET obtained for a scratchpad memory can be obtained when a cache is assumed in the system. This is partly caused by the fact that the caches in the ARM architecture use a random replacement policy which makes it hard to determine the current working set contained in the cache, but even for direct mapped caches with their improved predictability, the results presented in [WM05] coincide with the findings presented here.

Except for the task of allocating memory objects to the scratchpad memory at compile time, only some additional annotations concerning the memory regions and their corresponding access times are required. Specifying the timing of memory regions is required even if only the main memory is being used. The annotation of memory regions as well as other required information that has to be passed to aiT is discussed in the following section.

4.3.3 Required Annotation Information

Much of the information required to perform WCET analysis is automatically extracted from the executable of the application when the executable is read by aiT. Information not contained in the application, like the configuration of the memory hierarchy, and information that can not be extracted by aiT automatically, including the number of times that some of the more complex loops are executed, has to be provided by the user. It is vital for the WCET analysis that this information is known for all loops present in the application. Other information, like the address ranges accessed by load and store operations, help to generate tighter bounds on the WCET and thus improve the quality of WCET analysis results. Finally, annotation information is required for all longjump instructions inserted into the code to jump from main memory to the scratchpad and vice versa. Since the only jump that can cover the distance between the distinct memory regions is the "BL" instruction, it is necessary to annotate that these longjumps are not to be interpreted as function calls. The required annotation files and formats for all of these aspects will be presented in this section. For further details concerning annotation information, please refer to the aiT reference manual [Abs04a].

- Memory Configuration: The memory configuration considered in this section takes the ARM7 evaluation board as the basis. The main memory access times on the evaluation board depend on the bit-width of the access: for 8 and 16 bit accesses, two cycles are required, whereas a 32 bit access takes 4 cycles. This is due to the fact that two 8-bit memory chips are connected to the processor using a multiplexer to retrieve one 16 bit value, as shown in Figure 3.8 on Page 33. Since aiT does not currently support the specification of memory access times depending on the accesses' bit width, the different memory regions within the executable of the application have to be enumerated with their corresponding access times. This information is included in the aiT annotation files.

```
# Code:
MEMORY_AREA: 0x400000 0x400f0f 1:1 2 READ-ONLY CODE-ONLY
# Literal Pool:
MEMORY_AREA: 0x400f10 0x400f1f 1:1 4 READ-ONLY DATA-ONLY
# Code:
MEMORY_AREA: 0x400f20 0x408fff 1:1 2 READ-ONLY CODE-ONLY
# Data, 32 bit:
MEMORY_AREA: 0x409000 0x4090ff 1:1 4 READ&WRITE DATA-ONLY
# Data, 16 bit:
MEMORY_AREA: 0x409100 0x40910f 1:1 2 READ&WRITE DATA-ONLY
# Scratchpad memory region:
MEMORY_AREA: 0x409110 0x410000 1:1 1 READ&WRITE CODE&DATA
```

Fig. 4.23. Example memory region annotation file for aiT using a scratchpad memory

Figure 4.23 shows an example annotation file that describes the memory setup. Since the encc compiler generates 16 bit THUMB code, fetching a single instruction from the code region in the main memory requires a total of two cycles (one cycle for the access itself plus one wait state). Within the code section, it is often necessary to store large constants that contain e.g. the starting address of an array. ARM calls these tables of constant values "Literal Pools". An access to a constant stored in a literal pool generally requires four cycles in total, since it is performed as a 32 bit access. Following the literal pool, the 16 bit code region is continued in the example.

The data region can hold arrays and other global data structures. Depending on the bitwidth of the data types stored in the corresponding region, each access takes either two or four cycles.

Up to this point, only accesses to the main memory have been considered in the example. This part of the annotation always has to be performed, regardless of whether or not a scratchpad is being used. The last region in the example describes the scratchpad region: whatever is stored in the scratchpad requires one cycle per access, resulting in one single annotation entry for the entire scratchpad. The specification of "`CODE&DATA`" tells aiT that both instructions and data may be stored in the scratchpad region.

The starting addresses of functions, basic blocks and global data elements change when the application is recompiled using a different memory configuration since a new allocation of objects to the scratchpad memory is performed. Consequently, a unique annotation file has to be generated for every considered scratchpad size. The overhead caused by this annotation is limited since the extraction of required information from the executable is largely automated.

If a cache and the experimental cache analysis are to be used during WCET analysis, the annotation file has to be adjusted to reflect the different access mode implied by a cache. An example for the annotation format if a cache is present in the system is shown in Figure 4.24. Whenever a cache hit occurs, the requested value is available within one clock cycle, making the cache as fast as the scratchpad. If a miss occurs, however, one entire line consisting of four 32 bit words is fetched from the main memory. One cache line fill was determined to requires twelve cycles to complete.

```
CACHE_CONFIG: 0
CACHE_SIZE: 1024
CACHE_LINESIZE: 16
CACHE_ASSOC: 4
START_CACHE: empty
PERSISTENCE: no
MAY_ANALYSIS: no

MEMORY_AREA:
0x000000 0xffffff 1:1 1-12,1-3,1-3,1-3 CACHED READ&WRITE CODE&DATA
```

Fig. 4.24. Example memory region annotation file for aiT using a cache

The cache configuration and timing are annotated for aiT as shown in the example: the cache size, bytes per cache line and the associativity are required for cache analysis. Persistence analysis is only supported in the commercial cache analysis versions of aiT. The same is also true for the "May"-analysis [Fer97]. The annotation example specifies that the entire memory region is cached, implying a unified cache, and that a cache hit takes one, a cache miss 12 cycles. The remaining entries concerning cache timing are ignored in the currently used experimental cache analysis version. Current embedded systems assume up to 100 processor cycles for a cache miss. If longer main memory latencies are assumed, the advantage of using a scratchpad memory instead of a cache for timing-aware systems will become even clearer. This again emphasizes the fact that the comparison of scratchpad and cache based systems is fair, despite only using an experimental cache analysis. The mentioned cache configuration was used for all caches in the performed experiments, both for simulation and WCET analysis. The only parameter that was varied from one experiment to the next was the cache size.

- Loop bounds: If the maximum loop execution count can be determined from the executable using static analysis, aiT automatically extracts the maximum number of times a loop is executed. In some cases, e.g. when the number of loop iterations depends on input data or when the automatic analysis of loop bounds fails, the information has to be provided by the user. Since the number of loop iterations has an immediate impact on the WCET, it has to be specified for every loop within the application in order for aiT to be able to generate results.

  An example specification of loop bounds that were not automatically determined by aiT is shown in Figure 4.25. Three loops whose iteration counts were not analyzed automatically are specified with their address and the maximum number of executions. The address of a loop is the starting address of the basic block that represents the loop head. The keyword "`end`" states that the check for the loop bounds is performed at the end of the loop.

```
# LL79_2 (0x400024)
loop 0x400024 end max 6;
# LL15_0 (0x400056)
loop 0x400056 end max 99;
# LL41_2 (0x400124)
loop 0x400124 end max 49;
```

Fig. 4.25. Example loop annotation for aiT

  The more intuitive annotation of loop bound information directly within the application program is currently only supported for the Texas Instruments TMS470 C compiler.

- Annotation of Longjump instructions: During the analysis of the executable, aiT extracts information concerning the control flow of the application such that a complete control flow graph is obtained. When a scratchpad memory is being used during compilation, longjump instructions are inserted into the code to jump from the main memory to the scratchpad and vice versa. The only instructions hat are capable of bridging the gap between the main memory and scratchpad regions are "Branch and Link" instructions. Since "BL" instructions are usually used to perform a function call, aiT requires the additional information that the added BLs are really jumps and not function calls. If the BLs are not annotated accordingly, aiT complains about recursive function calls whenever a loop is partially assigned to the scratchpad memory. To avoid this situation and to achieve a correct control flow graph, longjumps have to be annotated as shown in Figure 4.26.

```
# BL target: _M_29
instruction 0x4001ee is a branch and never returns;
instruction 0x4001ee branches to thumb::0x400000;
# BL target: LL79_2
instruction 0x400252 is a branch and never returns;
instruction 0x400252 branches to thumb::0x400024;
```

Fig. 4.26. Example longjump (BL) annotation for aiT

The annotation tells aiT that the specified longjump instructions are not function calls, but are simply to be considered as branches. Additionally, the target address is specified. This annotation is fully automated in the workflow: all required information can be extracted from the application executable. With some knowledge about the code generation process, only those BL operations that are inserted due to the scratchpad memory allocation can be isolated and annotated accordingly.

- Start address and initial stack pointer value: In order to determine the WCET from the start to the end of an application, the entry point for the binary image has to be supplied in the form of a start address. The first instruction of the actual application should be specified here. In conventional C programs, this corresponds to the starting address of the "main" routine which is the first function of a program that is executed after some initial setup routines from the operating system.
  To allow aiT to derive correct timing information for stack related accesses, the initial value of the stack pointer should also be supplied. In this way, the access parameters of the stack memory region are known and aiT can determine tight bounds for the timing of e.g. "PUSH" and "POP" operations.
- Address regions accessed by Load and Store instructions: Once the memory regions have been specified, loop bounds provided, longjump instructions annotated accordingly and the starting point for the application is known,

aiT is capable of generating upper bounds for the WCET. Additional annotation information can help to improve the quality of the results, i.e. to make the upper bound tighter. In general, it is not possible to statically analyze which address is being accessed by a particular load or store operation, since usually, register-offset addressing is being used. A complex data flow and range analysis would have to be performed to be able to determine the accessed address range for at least some of the data accesses. To generate safe results, aiT always assumes the maximum memory access time found in the memory specification for all accesses with undetermined target address. Allocating data to the scratchpad memory thus has no impact on the generated WCET, unless all references to the scratchpad memory are annotated. During cache analysis, the possible range of addresses covered by a specific load or store operation is also relevant to determine the worst case impact on the information stored in the cache: if no information concerning possible addresses is provided, aiT has to assume that any cache line may have been replaced by a load operation. By restricting the range of addresses that are potentially accessed, the number of lines in the cache affected by this operation can be reduced.

```
# SymbolName: my_array3
instruction 0x00400038 accesses FROM 0x4010e8 TO 0x401276;
# SymbolName: my_array
instruction 0x0040027e accesses FROM 0x401408 TO 0x40159e;
# SymbolName: my_array
instruction 0x00400282 accesses FROM 0x401408 TO 0x40159e;
# SymbolName: my_array2
instruction 0x004002a2 accesses FROM 0x401278 TO 0x401406;
```

Fig. 4.27. Example load/store annotation for aiT

With some knowledge about the used benchmarks, e.g. by ensuring that no operation ever tries to access past array bounds, it is safe to assume one certain load or store operation in the generated executable will always access one particular array or scalar variable. Instead of using complex analysis techniques, it is thus possible to determine the array accessed by a certain load or store operation from the simulation trace file and annotate the instructions accordingly. Figure 4.27 shows an example annotation for a couple of load and store instructions occurring in an application.

Following the annotation, aiT determines an upper bound for the worst case execution time of the application. The used benchmarks and the configuration of the memory hierarchy are described in the following section, followed by the generated results for different scratchpad and cache configurations.

4.3.4 Benchmarks and Memory Hierarchy Configuration

The benchmarks used to generate results are presented with their code and data sizes in Table 4.7. They comprise applications from different areas of embedded systems: a speech codec according to the G.721 standard, a mix of several sorting algorithms, an ADPCM en- and decoding algorithm and finally an integer implementation of an inverse discrete cosine transform.

Benchmark	Code Size [*bytes*]	Data Size [*bytes*]	Description
G.721	2784	2424	Encoding and decoding according to G.721 using "Adaptive differential Pulse Code Modulation"
Multi_Sort	716	1204	Sorting benchmark (combining several sorting algorithms)
ADPCM	724	6928	Encoder and decoder using Adaptive Differential Pulse Code Modulation
Fast_IDCT	1428	6552	Integer implementation of inverse discrete cosine transform (IDCT)

Table 4.7. Selected benchmarks to evaluate the effect scratchpad memories and caches on WCET

The scratchpad memory used in the system can be accessed within a single cycle, which is the only relevant parameter for the results presented in this section, since energy is not being considered except in the compiler to guide the allocation process. The energy values used during this optimization both for the scratchpad and the main memory are the same as those given in the section describing the results of the Bottom-Up allocation (Table 4.3, Page 120). More important for the results presented here is the timing of the main memory: according to the measurements performed using the ARM evaluation board, a 32 bit main memory access requires 4 cycles to complete, i.e. one cycle for the access itself plus three additional waitstates. For 16 and 8 bit access, the number of waitstates is reduced to only one.

Note that in contrast to the previous publications [WM04, WM05], the experimental setup in this work does not use the scatter-loading mechanism to distribute memory objects from the Flash memory (used to permanently store the application) to their corresponding physical address ranges. The copying of memory regions from the Flash memory to the main memory or to the scratchpad partition is not fully analyzable by aiT and thus leads to confusing analysis results for some benchmarks. Without scatter-loading, aiT is only aware of addresses within the Flash memory region (starting with $0x4....$). The main memory as well as the scratchpad region are thus mapped to regions within the Flash memory. Due to the complete timing annotation for all used memory addresses, the results remain equivalent and comparable to the ones presented in the previously published work.

The caches, being the final element of the memory hierarchy considered in this section, are organized as unified caches, i.e. there is only one cache for both instructions and data. The cache size is varied from 64 bytes up to a total capacity of 8 kB. The cache is organized as a 4-way set associative cache with 16 bytes or 4 words per cacheline, which is a common configuration found in ARM processor cores [ARM98b]. A cache access resulting in a hit takes one cycles to complete, whereas accessing the main memory to fill an entire line takes 12 cycles. As in most ARM designs, the cache uses a random replacement strategy. This makes the precise analysis of the cache's contribution to the WCET more difficult, since most of the cache analysis tools assume an LRU replacement strategy, which is more predictable. However, in the previous work [WM05], a direct mapped cache was used where the cache only has one single way and thus no replacement strategy is required. Even for this cache configuration, the results were similar to the ones presented in this section, which shows that using more predictable replacement policies only has a limited influence on the quality of the generated WCET, at least considering the experimental cache analysis for the ARM7. The 4-way set associative cache configuration was chosen in this work in order to present results that closely reflect the currently used ARM architecture.

4.3.5 WCET Results for Static Allocation

The impact of using a scratchpad memory on the energy dissipation of embedded systems has already been shown in previous sections of this work. In this section, results concerning the effects of a scratchpad memory on worst case execution time analysis are presented. The scratchpad memory that is assumed to be present in the system is statically allocated using the Bottom-Up allocation technique described in Section 4.2.6. The obtained results are then compared to the effect of the commonly used cache on WCET.

The first extensive set of figures is shown for the G.721 benchmark since it shows the typical behavior also observed for the other benchmarks in the set. Figure 4.28 shows the number of simulated processor cycles when the application is executed using ARMulator with a typical input data set and varying scratchpad capacities, as well as the corresponding worst case execution time, determined using aiT's analysis techniques. The impact of using a scratchpad memory on both the average case execution time and on the WCET can thus be seen in the figure.

Looking at the darker bars that represent the simulation values, the figure once again underlines the benefit of utilizing a scratchpad memory in embedded systems: from the initial performance of more than 2 million cycles when only main memory is utilized, the performance is improved by 39% when a scratchpad memory of 2 kB is present in the system. In this configuration, the G.721 benchmark only requires about 1.3 million cycles to complete with the used typical input data set. Even for the smallest considered scratchpad

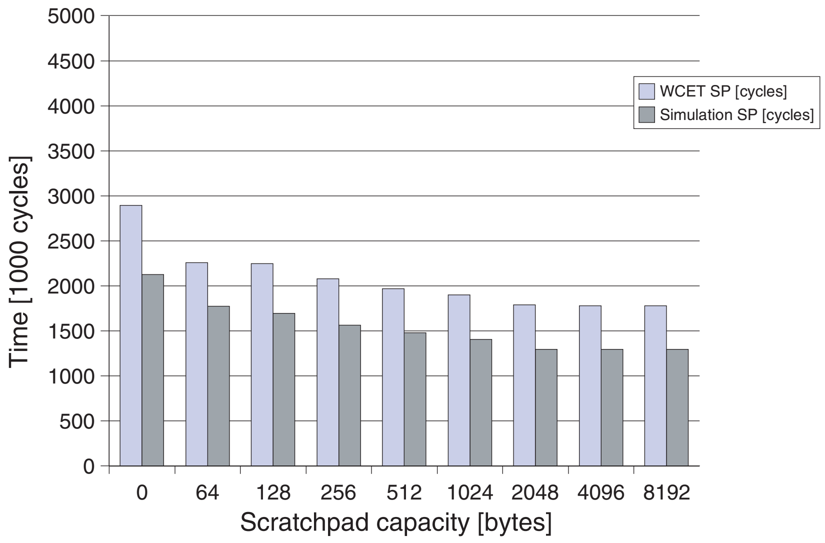


Fig. 4.28. Simulation and WCET for the G.721 benchmark using a scratchpad

memory of 64 bytes capacity, the gains of nearly 17% in terms of executed cycles are remarkable. Note that this strong improvement for a small scratchpad memory is only possible if the allocation algorithm distributes memory objects at a fine granularity. In the previously published results [WM05] which only considered functions but not basic blocks as instruction memory objects, the smallest scratchpad size that improved the performance of the G.721 benchmark was 512 bytes, since this was the first time that an entire function could be allocated.

The second, lighter set of bars in Figure 4.28 represents the worst case execution time determined by the WCET analysis using aiT. The executable, with parts of the instructions and the data allocated to the scratchpad memory according to the Bottom-Up model, is passed to aiT along with the required annotation information as described in the previous section. aiT uses all information, in particular the specified access times for different memory regions, to determine a guaranteed upper bound for the WCET. It is remarkable that despite a certain offset, the decrease in the WCET values follows the shape of the observed performance using simulation. This means that when a scratchpad memory is present in the system, then the benefit of this scratchpad memory does not only apply to the typical average case execution, but it also has a positive effect on the WCET of the system. For designers of real-time systems, this means that the benefits achieved by integrating a scratchpad memory and the corresponding allocation algorithms immediately improves the WCET of the system as well, without requiring any additional analysis effort in the WCET tool. This is a clear advantage of the scratchpad memory over the other performance-enhancing architectural features mentioned in the previous section.

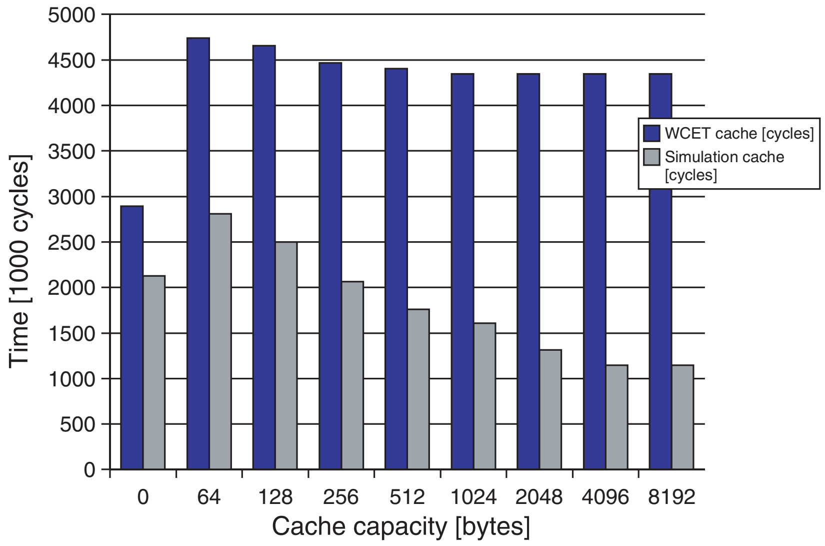


Fig. 4.29. Simulation and WCET for the G.721 benchmark using a cache

In contrast, Figure 4.29 shows the situation when a cache of different sizes is present in the system instead of a scratchpad memory. For the performance values obtained during simulation, the curve shows a significant increase in the number of cycles for very small cache capacities. This is caused by the high number of cache misses for these small cache sizes: during the execution of the application, not all memory objects of the current working set can be kept in the cache. This leads to the so-called “cache thrashing”, with memory objects repeatedly evicting each other from the cache without ever resulting in a cache hit. This makes it clear that more care must be taken when choosing the dimensions of the cache, since in contrast to a scratchpad memory, a cache that is too small will actually result in a reduced performance of the system.

For larger cache sizes, the number of cycles are reduced down to the level that is attainable using a scratchpad, and even below that. Depending on the application, the cache may be more beneficial compared to a statically allocated scratchpad since it can react to changes in the current working set of the application. To compensate this effect, the next section will consider using a scratchpad memory that is allocated using a dynamic scheme, i.e. allowing the scratchpad contents to change at runtime so as to further improve the scratchpad memory’s performance.

Looking at the WCET analysis results when a cache is present in the system, the strong increase in the number of cycles determined as the upper bound is apparent. For a small cache size, it is difficult to determine a guaranteed number of cache hits, in particular since the used cache analysis algoxsrithm does not use all known analysis techniques. For larger caches, the analysis determines that more cache hits will take place which can be seen in the decreasing WCET values for larger caches. In contrast to the scratchpad case, however, the WCET values do not decrease at the same rate as the observed simulation times. Of course, for a worst case input data set, the actually observed execution time may be longer than what was simulated using typical input data, but the figure clearly shows that the G.721 benchmark benefits from increasing cache sizes of up to 4 kB, whereas the WCET values do not change after a cache capacity of 1 kB. In addition, the differences between the observed and the worst case execution time are very much higher for all cache sizes than for the corresponding scratchpad memory capacities. This is underlined in Figure 4.30, which shows the determined worst case execution time values for both a scratchpad memory and a cache in one figure.

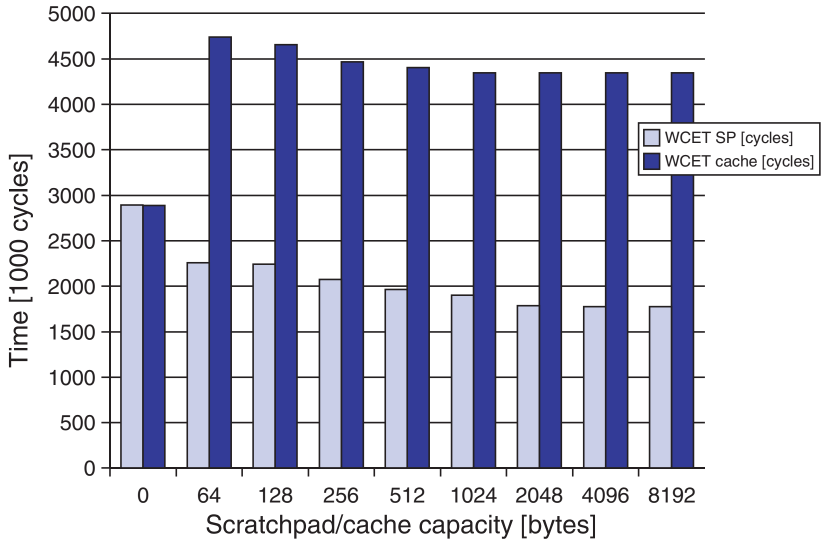


Fig. 4.30. Comparison of WCET values for scratchpad and cache for the G.721 benchmark

In this direct comparison, the overhead in the determined WCET values for a cache becomes evident: while using a scratchpad keeps the WCET values low, and shows a steady decrease for growing scratchpad sizes, the cache WCET values are very much higher, and they do not scale well with the higher performance gains of the cache. In fact, it should be noted that while the scratchpad memory keeps the ratio between simulated performance and guaranteed WCET nearly constant for all considered scratchpad capacities, the WCET for the cache steadily increases compared to the actual execution times, in particular for the larger cache sizes.

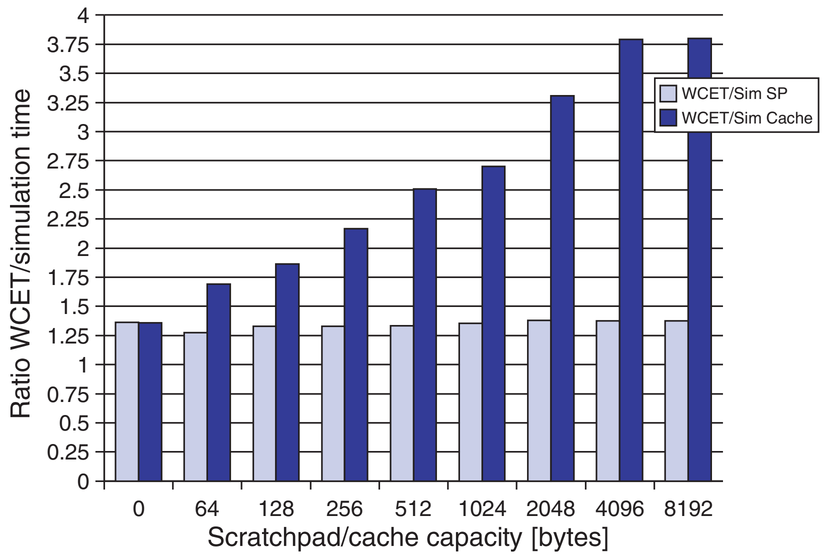


Fig. 4.31. Ratio of overestimation for scratchpad and cache for the G.721 benchmark

The ratio between the simulated performance and the determined WCET values makes it possible to consider the overestimation that occurs both when using a scratchpad and when using a cache in one single graph, which is shown in Figure 4.31. For the scratchpad setup, the ratio for the overestimation of the WCET compared to the observed simulation performance always stays below the value of 1.4 which is a reasonable value. More important than the value itself is the fact that the ratio stays nearly constant over the entire range of scratchpad sizes. The overestimation ratios for larger caches are significantly higher: while for a small cache of 64 bytes, it is only a factor of about 1.7, the overestimation for the 4 kB cache amounts to a ratio of 3.8, meaning that the WCET is nearly four times higher than the actually observed execution time. The consequence of this effect is that when a large cache is used in a system, then the upper bound on the WCET will become less tight. This clearly shows that while the presence of a scratchpad has a positive effect both on the observed average case performance and on the WCET analysis results, the cache's performance benefit in the average case does not translate to an improved WCET, due to the inherent unpredictability of a cache.

For the other considered benchmark applications, similar findings as those presented for the G.721 program were found. In the following paragraphs, only the differences and additional observations will be discussed.

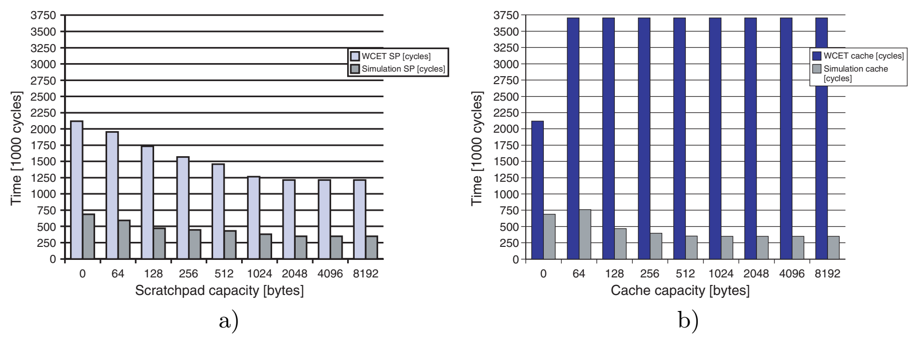


Fig. 4.32. Simulation and WCET analysis using scratchpad and cache for Multi_Sort

For the Multi_Sort benchmark shown in Figure 4.32 a), using a scratchpad memory results in a maximum of nearly 50% reduction in number of cycles for the simulation, and in a reduction of more than 40% concerning the WCET. Note that for this application, the determined WCET is very much higher than the observed simulation performance. This is due to the fact that in contrast to the G.721 benchmark, Multi_Sort is less data-dominated and includes a large number of control-flow instructions. These create different paths through the executable, of which the longest path always has to be considered for the WCET, whereas much shorter paths may actually be taken during simulation (if e.g. a part of an array that is to be sorted is already in the desired order). For a true worst case input set, the overestimation will become negligibly small, so the large difference between simulation and WCET is actually only caused by the used typical input data and not by imprecisions in the methodology (this was validated using the Bubble_Sort sorting algorithm for which the worst case input set, an array sorted in the reverse order, can be easily determined). Despite the high overestimation, the scratchpad memory's average case improvement directly translates to improved WCET analysis results: for increasing scratchpad sizes, the WCET decreases at about the same rate as the observed simulation time.

For the cache, shown in Figure 4.32 b), the initial high WCET value for a cache size of 64 bytes does not change at all when larger caches are being used in the system, despite the fact that the simulated performance is improved by nearly 50% for the larger caches. aiT is not able to guarantee a significant number of cache hits, which is due to the mixed accesses to large data arrays and to instructions. Since a unified cache is used in the considered setup, the cache analysis fails in determining accesses that have to go to different sets or lines of the cache. Thus, the large number of assumed cache misses is sustained throughout the range of considered cache sizes.

Summarizing the effects, Figure 4.33 shows the ratio of overestimation for the two memory configurations. While the scratchpad setup always stays at a level of well below 4, the WCET analysis overestimates the execution by

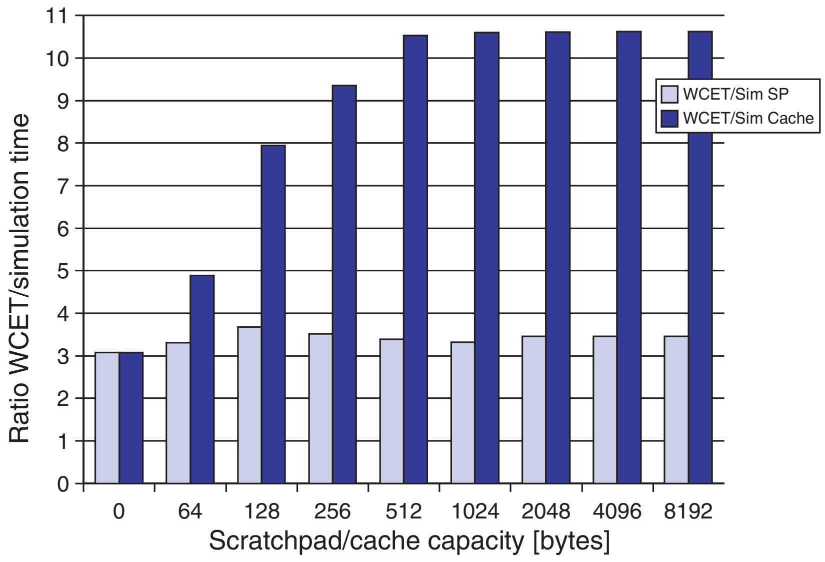


Fig. 4.33. Ratio of overestimation for scratchpad and cache for Multi_Sort

more than a factor of 10. The constant overestimation ratios for the scratchpad show that the WCET scales well with the actually observed improvements in execution time. The relatively high absolute overestimation ratio of between 3 to 4 is caused by the used input data set which, in this case, is far from the worst case set. The fact that the ratio does not change significantly throughout the range of scratchpad sizes still shows that WCET benefits directly from the use of this memory architecture, whereas the cache has a negative influence for this benchmark.

Figure 4.34 shows the results obtained for the ADPCM benchmark. The findings are again similar to the previous benchmarks and underline the fact that using a scratchpad memory is beneficial for real-time embedded systems. The overestimation caused by the used input data set is quite small for this application, and the reduction of simulated as well as WCET cycles can be seen clearly in part a) of the figure. The right hand side again shows that for small cache sizes, the performance is degraded, whereas large caches lead to reductions that are similar to those obtained from using a statically allocated scratchpad. While the observed performances of scratchpad and cache based systems are comparable, the WCET using a cache is once again very much higher than that determined for the scratchpad. The WCET value does not change even if the cache size is increased from 64 bytes to 8 kB.

This can also be seen in Figure 4.35 which shows similar ratios of overestimation for scratchpad or cache sizes of up to 128 bytes. From that point on, the cache values rise up to a ratio of 3.5, while the scratchpad ratios stay well below 1.5.

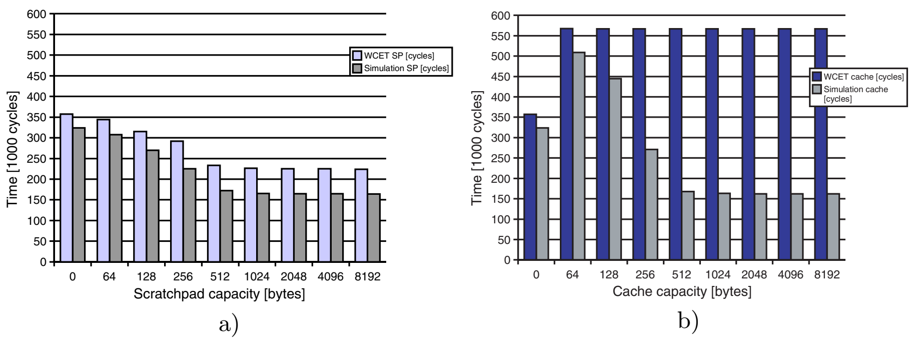


Fig. 4.34. Simulation and WCET analysis using scratchpad and cache for ADPCM

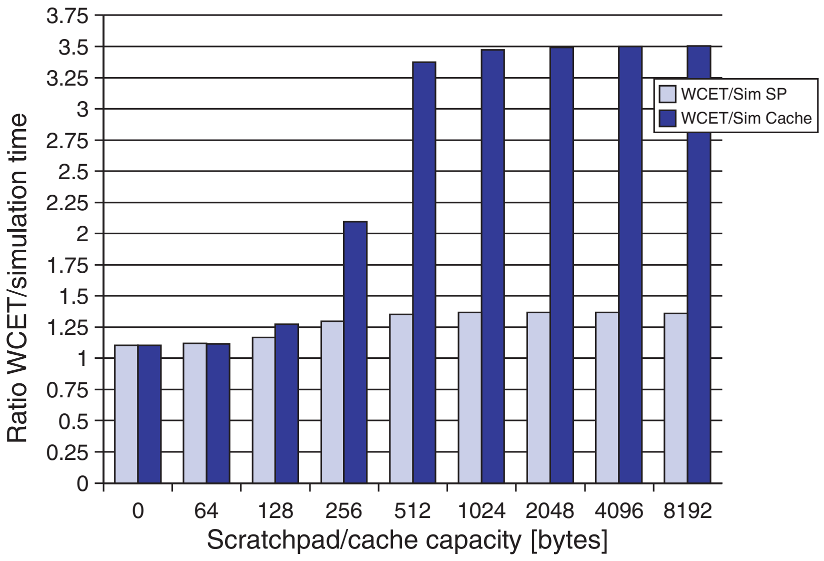


Fig. 4.35. Ratio of overestimation for scratchpad and cache for ADPCM

For Fast_IDCT, the final benchmark application, only a small overestimation of WCET to simulated cycles can be observed, highlighting the high precision of the used WCET analysis. This benchmark's execution time hardly depends on the values of the used input data set: all present data is processed according to the inverse fourier transform. Only two "if" statements are present in this application which lead to the very slight overestimation for the scratchpad case shown in Figure 4.36 a). The minimal overestimation is sustained for all scratchpad sizes. For a cache size of 64 bytes, the number of cache misses is accurately determined, since the small cache size and the fact that both instruction and data are accessed through the unified cache result in a high number of misses. Increasing the cache size results in strong improvements concerning the average case performance, but the WCET analysis values do not reflect this: for a cache size of 2 kB, the WCET is more than twice as high as the simulated cycles.

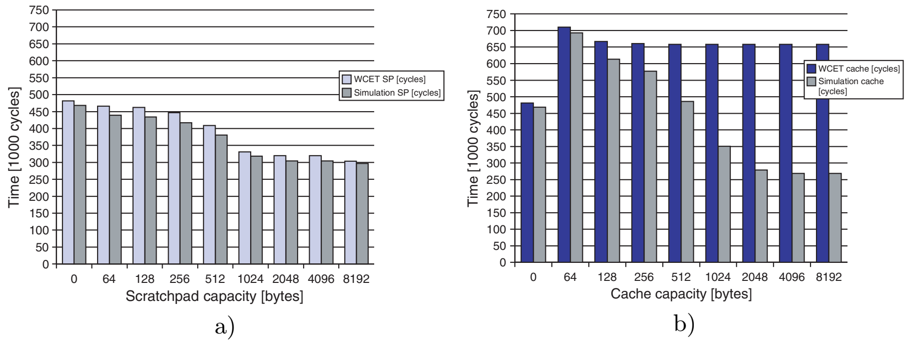


Fig. 4.36. Simulation and WCET analysis using scratchpad and cache for Fast_IDCT

Considering the ratios of WCET to simulated cycles shown in Figure 4.37, the very slight and constant overestimation of WCET compared to the actually simulated cycles of the scratchpad setup becomes clear. The cache ratio rises from initially very small ratios up to a level of nearly 2.5.

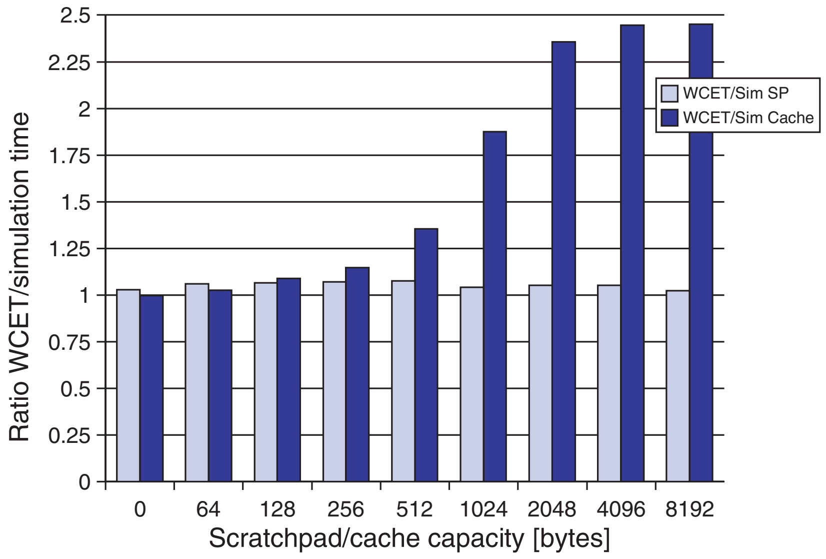


Fig. 4.37. Ratio of overestimation for scratchpad and cache for Fast_IDCT

In summary, for all considered benchmarks, the utilization of a scratchpad memory is beneficial both for the actually observed performance of the system and for the WCET analysis. Using a scratchpad memory allows the WCET analysis tool to generate upper bounds on the maximum execution time of the application whose overestimation only depends on how far the used input data set is from the worst case input set. This can be observed in particular for the Fast_IDCT benchmark: due to the fact that the performed computations hardly depend on the values of the input data, the overestimation of the WCET compared to the observed simulation cycle count is negligible. In contrast to that, the Multi_Sort benchmark shows a very much higher overestimation which is due to the fact that the performance of sorting algorithms is highly dependent on the input data. The fact that the WCET to simulation ratio stays constant for the scratchpad environment emphasizes the fact that the overestimation is only caused by the used input set and not by imprecisions in the workflow.

4.3.6 WCET Results for Dynamic Allocation

For the results presented in the following, the dynamic scratchpad allocation algorithm described in [VWM04b] was used instead of the static Bottom-Up allocation approach. This section first provides a summary of the dynamic allocation algorithm along with its requirements. After that, results for two benchmarks are presented that show how using the dynamic allocation algorithm and a scratchpad memory affects the WCET. These values are again compared to a system that uses a cache.

Static allocation techniques in general assign the most frequently accessed or executed memory objects to the scratchpad memory at compile time and load them onto the scratchpad before actually starting the application. In contrast to that, the dynamic allocation attempts to always keep the current working set of memory objects on the scratchpad at runtime. As an example, a program may have two or more "hot spots", e.g. innermost loops that contain frequently executed basic blocks. If the two hot spots don't fit on the scratchpad at the same time, then the static allocation has to decide which of the loop bodies should be allocated to the scratchpad. The dynamic allocation can choose to reuse the scratchpad space and thus allocate both hot spots to the scratchpad when they are executed: before executing the first hot spot, it is copied to the scratchpad and then executed from there. When the second hot spot is to be executed, it can replace hot spot number one in the scratchpad. In this way, the scratchpad can be more efficiently used than in the static allocation scheme. However, there is an increased overhead for the copying of instructions and data from main to the scratchpad and vice versa.

The approach presented in [VWM04b] uses a modified algorithm that is known from global register allocation for CISC processors. Instructions and data elements can be considered as values in the data flow graph of an application, whereas the scratchpad memory corresponds to the scarce resource that the objects have to be allocated to. This is very similar to the situation of allocating a program's currently live values to the small number of registers. It is vital for an efficient execution of the application that register allocation optimally exploits the register file by reusing the registers for different data as much as possible. The same is true for the dynamic scratchpad allocation algorithm: if a certain memory object is not required in the further course of the application, it should be replaced by a more promising object, at the same time keeping the copy costs in mind. The copying of objects to and from the scratchpad corresponds to the spilling of register values to and from memory. The dynamic allocation algorithm presents an extension of the register allocation approach since for register allocation, all considered memory objects are of uniform size.

The algorithm operates in two steps: first, the sets of memory objects that are to be allocated to the scratchpad memory and their copy points are determined. In a second step, scratchpad addresses are assigned to the chosen memory objects and the code is modified such that the copy operations are performed at the correct point in time during execution, and the instructions and data elements that were copied to the efficient scratchpad memory are subsequently accessed instead of the copies on the slower and larger main memory.

The considered memory objects consist of global variables, local non-scalar variables and instruction sequences in the form of traces. Trace generation is a well-known technique [TY96] that has been used to improve the performance of caches by trying to arrange the code in such a way that the percentage of straight-line code is increased. This involves analyzing the frequency of taken and untaken branches and modifying the code layout accordingly. The reason that traces are used for the dynamic allocation algorithm is that every trace has a single entry point and always ends with an unconditional jump. This attribute causes traces to be freely placeable anywhere within the address space without requiring additional modifications. The consideration of basic blocks instead of traces made it necessary in the previous sections to consider the influence of additional jump instructions to maintain correct control flow for the static scratchpad allocation algorithms. The models that consider basic blocks and their connections show an increased complexity, as shown in Sections 4.2.5 and 4.2.6. If traces are used instead, the modification of control flow is no longer necessary, thus simplifying the treatment of instruction memory objects at a finer granularity.

However, there is a drawback involved in using traces which becomes visible in particular for the 16 bit THUMB instruction set of the ARM architecture: in order to guarantee that traces can be freely placed anywhere in the processor's address space without modification, all jumps that leave the current trace have to be implemented as longjumps, i.e. as “BL” instructions, since they are the only jumps in THUMB mode that can cover the distance between two distant memory regions (cf. Section 3.3.1). Therefore, every conditional or unconditional jump that leaves one particular trace has to be replaced with a longjump, which requires two instruction fetches instead of just one. In addition, conditional jumps have to implemented using a conditional jump followed by a longjump if the current trace is to be left. These implementation details lead to a certain overhead when trace generation is performed using the THUMB instruction set. Postpass optimizations can be used to reduce the amount of overhead, however this aspect is not considered in this work.

The dynamic allocation algorithm first performs a liveness analysis of memory objects. Those objects that have non-overlapping lifetimes may share the same scratchpad space. An extended concept of DEF-USE chains is used to express the lifetimes of objects. Using flow constraints that consider the liveness information of memory objects, an ILP formulation of the allocation problem is generated and solved. The solution specifies which memory objects are to be copied to the scratchpad and back to main memory at which point in time during execution of the application. In the following step, the objects' addresses in the scratchpad memory are determined. To do this, another ILP problem is generated and solved.

Except for using a dynamic allocation algorithm to fill the scratchpad, the experimental setup used in this section is identical to the method described above for the static scratchpad allocation. The executable generated using the dynamic allocation algorithm is analyzed in the same way as the statically allocated applications: a simulation run using a typical input data set is performed to determine the average case number of cycles required to execute the application with varying scratchpad capacities. Then, WCET analysis is performed to determine whether the benefits concerning predictability of a scratchpad memory can also be obtained when it is allocated in a dynamic way. The annotation information described in the previous section also has to be provided for the dynamic allocation. The dynamic copying of code and data to the scratchpad and subsequent execution from there make the annotation more complex, however the amount of information that has to be specified for aiT to generate valid results remains roughly the same. For that reason, the technical details concerning annotation for the dynamic allocation are omitted here.

In the static approach, all memory objects were assumed to be present in their corresponding memory when the actual application was started. For the dynamic allocation, the overhead required to copy instructions and data to the scratchpad and back to main memory is considered to be part of the application program's execution. To allow a fair comparison between static allocation on one hand and dynamic allocation on the other, the initial copying of memory objects to the scratchpad memory when the static approach is being used was considered to be part of the application for the results presented in this section. Furthermore, since the trace generation step is considered as a preparatory step and not part of the dynamic allocation algorithm, the presented values for the static allocation were also generated following a trace generation step. In this way, both static and dynamic allocation operate on the same input application.

Two benchmarks were chosen due to the properties found in their code: the Multi_Sort application executes a sequence of several sorting algorithms. Each of the sorting algorithms has its own hot spot or innermost loop, such that the dynamic allocation can be expected to show an improved performance compared to the static approach, since it can always keep the currently active hot spot in the scratchpad memory. The ADPCM benchmark has a different structure: the bodies of its innermost loops are larger than for the Multi_Sort application, thus they will not all fit into a small scratchpad memory. In addition, it was observed in the static case that the cache performance for the ADPCM application was pretty poor. It is thus interesting to see how the dynamic allocation approach handles this benchmark.

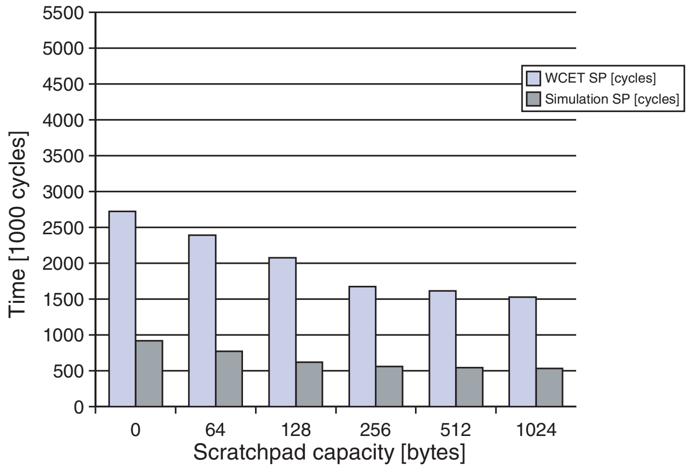


Fig. 4.38. Simulation and WCET for Multi_Sort with traces using a scratchpad with static allocation

Figure 4.38 shows the results of applying the static allocation algorithm by Steinke et al. [SWLM02] to the Multi_Sort application after the initial trace generation step. For a single scratchpad memory partition, this algorithm generates results that are very similar to those generated by the Bottom-Up approach. The memory objects that were chosen for static allocation to the scratchpad memory are copied at the beginning of the Multi_Sort application's main function. Looking at the simulated values, a steady decrease can be observed for a scratchpad size of up to 256 bytes. The savings concerning WCET follow the curve for the average case simulation, as in the previous section.

Comparing this result to Figure 4.39 shows the advantage of utilizing a dynamic scratchpad allocation algorithm: the cycle count that is obtained using a static allocation and 128 bytes of scratchpad memory can already be achieved with half the scratchpad capacity of 64 bytes for the dynamic allocation. This shows that it is beneficial to be able to adapt the contents of the scratchpad to the current working set of the application. In particular for a scratchpad capacity of 64 bytes, the static allocation achieves cycle savings of 16% compared to the case without scratchpad, whereas the dynamic allocation is capable of saving 34%. Since the dynamic allocation is mainly beneficial for small scratchpad memories and no significant gains were achieved past a size of 1 kB, only these scratchpad sizes are considered here. The results up to this point were already described in [VWM04b].

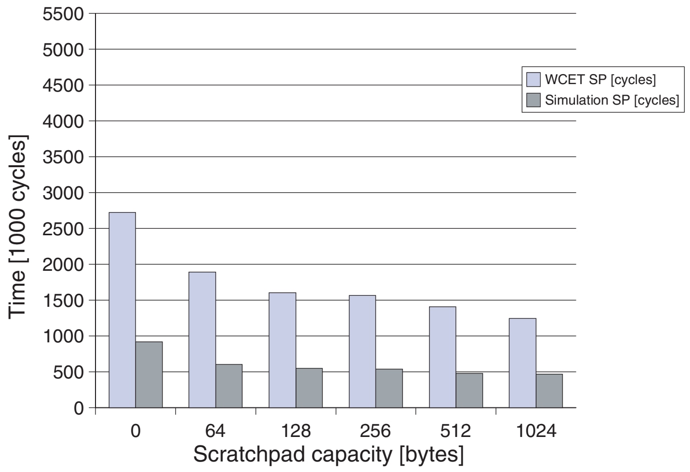


Fig. 4.39. Simulation and WCET for Multi_Sort using a scratchpad with dynamic allocation

The novel contribution of this work concerns the second set of bars in Figure 4.39. The results of WCET analysis show that the increased performance of the dynamic allocation algorithm also translates to improved upper bounds on the worst case execution time: while the average case simulation cycles are reduced by a maximum of about 48% for a scratchpad capacity of 1 kB compared to the case without scratchpad, the WCET values show an even higher reduction of about 54% for the same comparison. The reason for this behavior is due to the reduced access times of the scratchpad memory: if an additional 32 bit memory access is assumed in the WCET analysis that does not occur in the simulation run, then four additional cycles are counted in the WCET analysis for the main memory, compared to only one additional cycle for an access to the scratchpad. The improved average case performance of the application when a dynamic allocation approach is used thus directly translates to improved WCET figures as well: in contrast to the mentioned 54% reduction in WCET for the dynamic case, only 44% were achieved using the static approach. It can thus be concluded that the dynamic allocation algorithm is fully predictable, despite the fact that memory objects are copied to and from the scratchpad at runtime. The reason for this predictability is that all decisions concerning memory object placement are taken at compile time. Despite the slightly increased annotation effort, no further modification of the used WCET analysis techniques are required. Using the presented workflow, tight bounds can be determined for the WCET using the aiT tool without requiring any additional analysis modules.

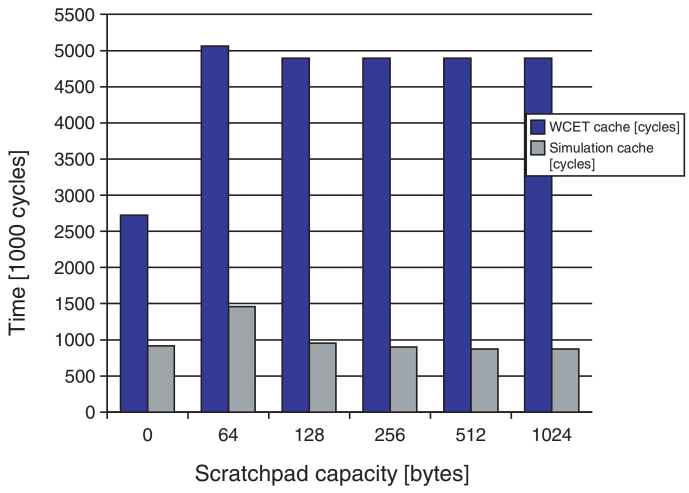


Fig. 4.40. Simulation and WCET for Multi_Sort using a cache

This is very much in contrast to a cache based system. Despite the fact that an additional complex cache analysis is required to analyze a cache's impact on WCET, the results both concerning average case simulation and WCET analysis are superior for the compiler controlled dynamic allocation approach. The results for a system utilizing a cache are shown in Figure 4.40. To make the results directly comparable, the executable used for cache simulation and WCET analysis was also generated using the preparatory trace generation step. As in the previous cases, very small cache sizes tend to lead to a performance degradation. In addition, the observed average case performance for all cache sizes is below that of the corresponding scratchpad values using the dynamic allocation approach. While a dynamically allocated scratchpad of 1 kB reduces the average case simulation time by 48%, a cache of the same capacity only results in savings of round about 5%. This effect is due to the fact that profitable memory objects may be evicted from the cache when other, rarely accessed memory objects are loaded. The dynamic allocation approach, on the other hand, is capable of consciously selecting those memory objects that will results in the maximum savings. While the cache's average execution time was superior to the statically allocated scratchpad in most observed cases, the dynamic allocation performs better, and is able to outperform the cache. Finally, the WCET analysis results for the cache setup are similar to the ones seen in the previous section: a high overestimation is obvious from the figure, and while the number of worst case misses can be reduced slightly for a 128 byte cache compared to 64 bytes, the overall results of the WCET cache analysis again show very loose upper bounds on the WCET.

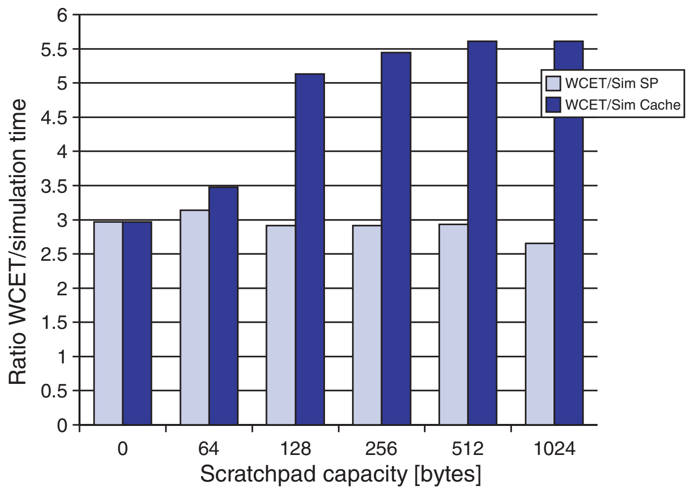


Fig. 4.41. Ratio of overestimation for dynamic scratchpad allocation and cache for Multi_Sort

Comparing the ratios of overestimation in Figure 4.41, it can be observed that for the dynamically allocated scratchpad, the values stay nearly constant at a value of around 3, whereas the cache analysis overestimates the WCET compared to the average case simulation by increasing factors of up to 5.5. These values underline the inherent predictability of a scratchpad memory even when a dynamic allocation algorithm is used: the improved performance of the system when larger scratchpad memories are used directly translates to better WCET analysis results. In contrast to that, the WCET values for the cache hardly change at all, despite the small, yet noticeable performance benefit observed during simulation.

The findings for the Multi_Sort benchmark were confirmed using ADPCM: Figure 4.42 a) shows the static allocation results using Steinke's algorithm following trace generation, whereas the right hand side shows the results for dynamic allocation. It can be seen that the ADPCM benchmark does not offer as much additional potential for the dynamic allocation. The dynamic scratchpad algorithm still outperforms its static counterpart, which is visible in particular for a scratchpad capacity of 128 bytes. The obtained WCET values scale with the achieved average case performance gains. The low overestimation of WCET over the simulated cycles are based on the fact that ADPCM hardly depends on the used input data set, as explained in the previous section.

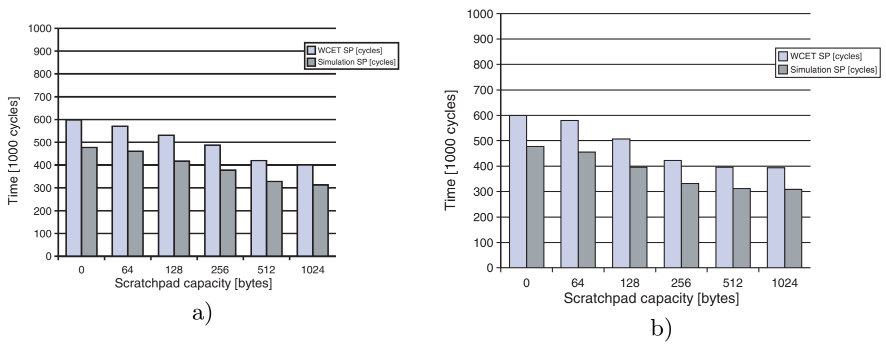


Fig. 4.42. Simulation and WCET for ADPCM with a scratchpad a) using Steinke's static algorithm b) using dynamic allocation

Figure 4.43 a) shows the results obtained for the cache: the performance gains do not translate to improved WCET bounds. For this benchmark in particular, the overhead of using a cache that is too small is significant. The ratio of WCET to simulation shows the same picture as before: while it stays the same for the scratchpad setup, it increases with larger cache sizes.

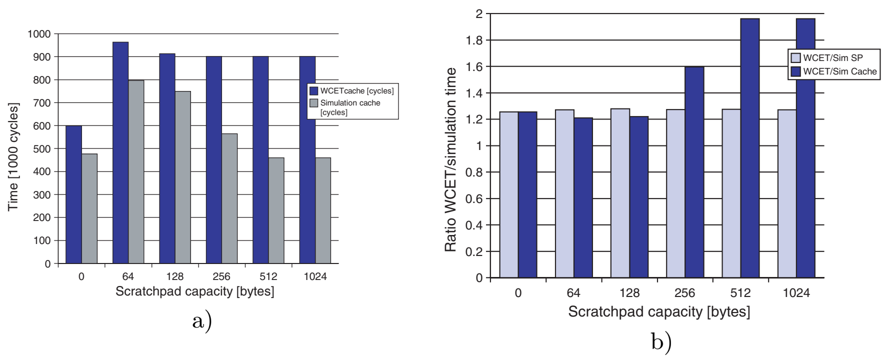


Fig. 4.43. a) Simulation and WCET for ADPCM with a cache, b) ratio of WCET for scratchpad and cache

These results show that the use of a scratchpad memory in conjunction with a dynamic allocation algorithm is beneficial both for the average case performance and for the obtainable results concerning WCET. Taking into account the high performance of the scratchpad together with its inherent predictability, it can be expected that in the future, a number of embedded devices will be equipped with scratchpad memories that are utilized using the presented allocation algorithms. In particular when real time aspects have to be considered during the design of an embedded system, the use of a scratchpad memory will result in WCET values that scale with the average case execution time.

5

Main Memory Optimizations

The requirements for storage capacities in all sorts of computing devices are increasing at a fast pace. New applications make larger memories necessary, mobile phones and PDAs are increasingly used to take pictures and to store music – the vast amount of user data requires increasing memory capacities to be available in the system. The necessary increase in memory sizes leads to several problems that need to be solved by system designers in order to allow further innovation, in particular in the embedded and portable market. One of the problems that designers are facing is the growing performance gap between processors and memories, which has been called the "Memory Wall" by Wulf et al. [WM95]. Since the speed of memories is not improving at the same rate as that of processors, there is an increasing gap between the capabilities of the processor and the memory system (cf. Figure 4.1 on Page 90). Introducing a hierarchy of memories is one possible solution to at least slow down this effect. Memory hierarchies consisting of scratchpad memories, caches and the register file are being described in this work.

Another crucial issue, in particular for portable embedded devices, is the energy dissipation. It has been shown in several publications that the memory subsystem consumes a considerable amount of the total system energy [KG97, KVIY00, Mos01]. This chapter in particular targets savings achievable by exploiting features of the main memory.

Most modern main memory chips support some kind of energy management, allowing the main memory to be put into a power down mode when it is not being used. The following section takes a look at previous work concerning power management and saving energy in the main memory. The subsequent section presents an optimization that exploits the presence of a scratchpad memory in order to shut down the main memory when instructions and data are accessed from the scratchpad and the main memory is thus not used for a certain amount of time. The optimization allocates memory objects to the scratchpad memory in in a way similar to that described in Section 4.2. However, a different cost function is being used here: in contrast to the multi memory allocation strategies where all energy savings were obtained from the reduced access-related energy per access of the scratchpad memory, the allocation considered in this section allocates objects to the scratchpad memory in such a way that the non-access related standby energy of the main memory in SDRAM technology is minimized. This is achieved by maximizing the time during which the main memory can be kept in the power down mode. In addition, the fact that connections between global variables and basic blocks that access these variables were modeled for the first time results in significant energy savings for one of the example applications.

The second part of this chapter takes into account the efficient utilization of a Flash memory found on most of today's embedded devices as a non-volatile memory. Flash memories in NOR technology (cf. Section 3.2.3 on Page 24) can also be used as instruction memories using the so-called Execute-In-Place (XIP) technique. The main memory can again be put into power down mode when the Flash memory is being accessed. As an additional benefit, the amount of main memory required for a system can be significantly reduced.

5.1 Related Work

With energy dissipation gaining importance in particular in the design of embedded devices, hardware vendors are providing means of controlling the energy required for the operation of a device when the full computing power is not required. These features are known as "Dynamic Power Management" (DPM). In general, they reduce the power dissipated by a system depending on the currently required performance. This is generally achieved by a power manager which observes and controls the transitions between different operating states at runtime. These power states may be implemented in different ways: It is possible to shut down parts of the system that are not required, thereby reducing the leakage power dissipation in these units. Current designs allow operation at a number of predetermined frequencies. The power manager can change the operating frequency during runtime with a certain overhead required for the transition. If the operating frequency is lowered, then the supply voltage can also be reduced, leading to significant energy savings since both the frequency and the square of the supply voltage are factors in the general energy equation

$$E = \alpha \cdot C_L \cdot (V_{DD})^2 \cdot f \cdot \#cycles \tag{5.1}$$

An illustrating example for the savings achievable by statically assigning voltages to tasks can be found in [IY98]. This contribution also presents an algorithm to statically determine the optimal operating voltage for a set of tasks running on a processor. Each possible operating voltage is linked to a corresponding operating frequency. If the worst case execution time of the tasks in the task set can be determined statically, then the allocation of voltages to the tasks can be solved in an optimal way by using a set of ILP equations. The paper concludes that even a small number of different voltages results in considerable energy savings. In another publication by the same authors [OIY99], a solution for the problem of assigning voltages to a set of tasks in a dynamic way is presented. Again, the set of tasks together with their worst case execution times are known. Different approaches are presented depending on whether the arrival time of tasks is known or not.

In a system that involves user interaction, tasks may be generated and terminated dynamically at runtime. In these systems, the behavior of the entire system is hard to predict, which makes power management more difficult. One way to decide when the system can be put into a lower power mode is to watch the system and heuristically assume that if the system was idle for a certain amount of time, than it will remain in the idle state for a sufficiently long time to make the transition to a low power mode worthwhile. As an alternative, the run time monitor can also keep track of the behavior of the system and predict the point in time when it will most probably become idle, e.g. when a set of input data has been handled and the system is waiting for new input. In both cases, wrong decisions taken by the monitor can cause an increase in the overall energy dissipation, since the transition between active and power down mode consumes a certain amount of time and energy, which has to be compensated by sufficiently long idle periods. More sophisticated methods make use of stochastic models to determine when it is worthwhile to put the system into power down mode.

A more complete overview over different power management techniques is provided in the survey paper [CY02]. It covers static techniques, applied at design time, as well as dynamic power management techniques that control the energy dissipation at runtime. The modeling of energy is done at different levels, from RTL to system and even a cluster level. Different implementations of dynamic voltage scaling are compared, and some system-level and operating-system supported techniques are mentioned. The survey also mentions some approaches for compiler-based power management.

In [BBM00], the authors consider dynamic power management on the system level, describing power-manageable components and how they can be configured to adapt to changing demands concerning computation power and energy dissipation. The Strong-ARM SA1100 processor with its three operating modes “Run”, “Idle” and “Sleep” and a hard disk with power management are provided as examples.

Several approaches have been developed to restrict the dominating main memory energy. Most system today employ a memory hierarchy that allows the frequently accessed memory objects to be stored in a more energy efficient, small memory that is closer to the processor. The advantages and disadvantages of several memory architectures that may be used for this purpose (caches, scratchpad memories) were already discussed in Section 4.3. This section deals with savings that are achievable by exploiting power management features present in modern main memories. The fact that in particular for embedded systems, the application to be executed on a device is usually known at design time can be exploited by performing a static power management at compile time.

In contrast to the multi-task static power management technique mentioned above [IY98], a compiler usually only has an influence on a single task executed on a system, meaning the scope has to be changed from the task- to the application level. For one considered application, the compiler has detailed knowledge of the execution and access frequencies of basic blocks and variables, respectively, and can take this into consideration during code generation. The gathered information can e.g. be used to determine regions in the code where only a low computing power is required, such as when the application is waiting for data. Since the processor considered in this work, the ARM7TDMI, does not provide power management features, it is not possible to modify the operating frequency or the voltage. However, since the memory hierarchy was found to consume a large percentage of the overall system energy [Mos01, KVIY00], it is worthwhile to consider power management within the compiler that targets the memory subsystem. In the presented approach, the compiler is able to exploit the power down modes provided by the used SDRAM memory chips. By allocating code and data to a small, energy efficient scratchpad memory, the main memory can be put into power down mode when the application is executed from the scratchpad. The objective function of the presented algorithm is to allocate those memory objects to the scratchpad memory that result in a maximum power down time of the main SDRAM memory.

The techniques and results presented in this section were first described in [Ker05], however some modifications to the cost function and the used ILP equation were performed in order to improve the obtained results.

5.2 Main Memory Power Management

As described in the sections on the behavior, timing and energy dissipation of DRAM memories in Chapter 3, the dynamic nature of DRAMs has to be considered in their behavioral and energy models, meaning that not only the action that is currently being performed is relevant, but past events also have to be accounted for in order to accurately capture their behavior. One obvious example for this is the burst access scheme: a faster burst access is only possible if the accessed addresses are contiguous. Another example that involves the consideration of the memory's behavior over time is the exploitation of power down modes: if the memory is not accessed for a longer period of time, it may be worthwhile with respect to the energy dissipation to put the memory into a power down mode. However, since the transition to and from power down takes a certain amount of time and energy, the memory should not be powered down unless it can remain in power down for a sufficiently long time.

Figure 5.1 shows the state machine representing the behavior of an SDRAM. The states can generally be split into two subsets: active and inactive. The inactive subset consists of two states: Deep Power Down (DPD) and Self Refresh. DPD allows the largest energy savings, at the cost of the memory losing all its information.

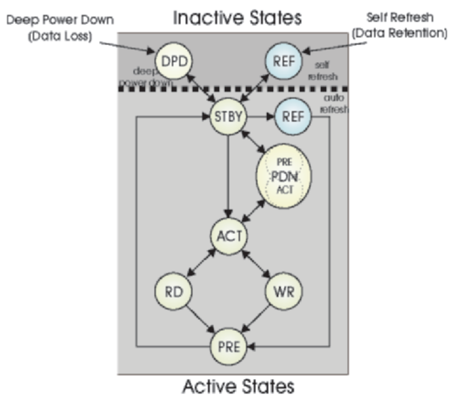


Fig. 5.1. SDRAM state machine, divided into inactive and active subsets

Using the Self Refresh mechanism, it is also possible to sustain the memory contents despite putting the memory into an inactive state. In Self Refresh mode, modern memories can take advantage of the Partial Array Self Refresh (PASR) and Temperature Compensated Self Refresh (TCSR) mechanisms. The former signifies that only those banks of the memory that contain live data are refreshed. Therefore, if some of the banks of the memory do not contain live data, the self refresh power can be reduced. TCSR, on the other hand, detects the temperature of the memory chip and controls the required refresh intervals accordingly: the maximum refresh frequency is required only if the memory or its environment is very hot. If the memory chip can be kept cool, less refresh operations per unit of time need to be performed, which effectively reduces the required refresh power. Since the energy savings achievable using PASR and TCSR are not very promising and are more suitably exploited using a run-time monitor rather than a compiler, they are not considered in this work. In addition, since the used compiler framework considers single application examples without sufficiently long idle periods, it is to be expected that the temperature readings from the memory chips will not change significantly during the execution of an application, and therefore not offer a lot of potential for TCSR optimization. Also, no means to estimate the temperature of the memory chip during execution of an application are integrated into our framework. For these reasons, self refresh mode is not considered in this work.

Putting the memory into DPD mode has a couple of disadvantages as well: Since a transition to DPD leads to a complete loss of information within the memory, any live data would have to be copied e.g. to a different memory before the transition to DPD can be performed. This represents a considerable overhead that can easily outweigh the benefits of DPD energy savings. Another drawback involved in using DPD during execution of a program is the fact that the transition from the DPD state to an active state takes a considerable amount of time ($150\mu s$, corresponding to 15,000 cycles at 100 MHz for the considered memory) during which the memory can not be used. During the active execution of an application, it is in general not acceptable for a memory to require such a long time to react to a request. The compiler may choose to insert wake-up dummy instructions that access the memory so that it can wake up in time for the first “real” access, but the memory would not be in the DPD state during this long transition anymore. In general, the overhead of changing between DPD and active states is too high to be beneficial. Therefore, only the energy saving potential in the active subset of the SDRAM state machine will be considered in this section.

Considering the states belonging to the active part of the state machine, the memory will usually be in the standby state from where it can be activated and subsequently accessed for reading or writing. To save energy, it can be put into a power down mode which is split into “precharge power down” and “active power down”. The actually assumed power down state depends on whether all banks of the memory are in the idle state (precharge power down) or not (active power down). As can be seen from the timing diagram of the considered memory [Mic04c], the transition from the active state to the power down state and back only requires three cycles, which is an acceptably short time even for an active device executing an application. During the transition, the energy dissipation of the memory can be modeled according to the assumptions for the standby state. Once the power down state has been reached, all in- and output buffers of the memory are deactivated, leading to considerable energy savings as shown in Table 5.1. It becomes evident that the power down mode is able to save up to 95% of the memory’s active power. If DPD was also considered, it would increase this factor to 99.9%. However, this additional gain of roughly 5 additional percentage points is outweighed by the disadvantages of the DPD for the considered setup.

Memory State	Power [mW]	Percentual savings
Active (access)	99.00	0.00%
Standby	66.63	32.70%
Power Down	3.60	96.36%
DPD	0.018	99.98%

Table 5.1. Average power values for different states of SDRAM memory [Mic04c]

5.2.1 Motivating Example

Assuming a system with a main memory and a scratchpad memory, the idea of the proposed optimization is to allocate memory objects, i.e. instructions as well as data, to the scratchpad memory in such a way that the main memory can be kept in the energy preserving power down mode for a maximum amount of time. This approach to energy minimization has been described for the first time in [Ker05], however some of the multi memory allocation equations were revised to generate the results presented here. The increasing impact of leakage energy on the energy dissipation of contemporary devices [Bor99] indicates that using leakage energy as a cost metric for energy optimization will gain importance in the future. The optimization goal of this approach differs from the the previous work on scratchpad allocation [SWLM02, WHM04], since in the previous work, only the access counts of memory objects and the resulting access-related energy dissipation were considered, but not the relationship of memory objects over time.

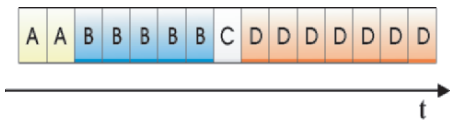


Fig. 5.2. Sequence of basic blocks of an example application in main memory

Consider the example given in Figure 5.2: A program consisting of four basic blocks A, B, C and D is executed from main memory in the sequence shown. In the previous scratchpad allocation approaches, the memory objects were weighted according to the number of accesses or their execution frequency and allocated to the scratchpad accordingly, leading to the distribution shown in Figure 5.3 a): Basic blocks A, B and D are stored in the scratchpad memory, only basic block C remains in main memory since it is executed only once. If the potential power down states of the main memory are taken into account, however, then the allocation shown in Figure 5.3 b) may actually be more beneficial, since the memory can be put in the power down mode (lightly shaded area) for a longer period of time. Additionally, only one transition from active to power down mode (darker shaded area) is required. In the optimization strategy considered in this section, the latter allocation is chosen since it results in higher savings concerning the standby energy of the main memory.

To summarize, the proposed optimization is aimed at reducing the standby energy of the main memory. This is in contrast to the previous scratchpad allocation techniques which were all aimed at reducing the access-related energy costs of memories. Based on the given example, an allocation of objects to the available memories based on the consideration of standby energy is

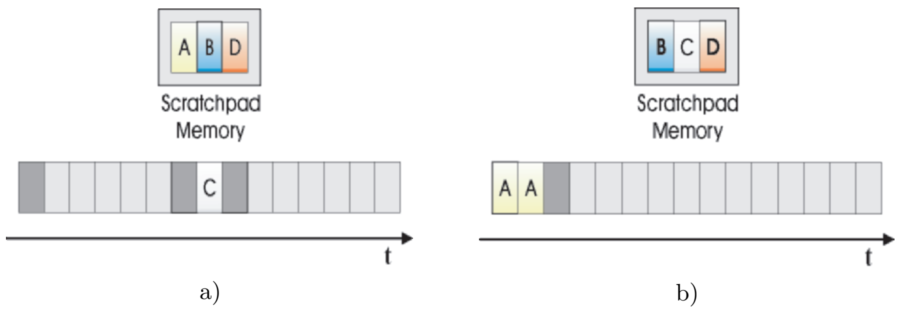


Fig. 5.3. Scratchpad allocation strategies: a) Traditional allocation optimizing for access energy, b) New allocation optimizing for main memory power down

expected to result in energy savings when the considered main memory provides a power down state.

A dynamic memory power manager is assumed to be present in the considered system that is capable of putting the main SDRAM memory into power down mode whenever it is not accessed. Since the presented optimization modifies the access sequence to the main memory in such a way that the power down times are maximized, it can be assumed that whenever one of the scratchpad partitions is being accessed, the main memory will not be used for a longer period of time. Thus, a transition to power down mode can be initiated whenever one of the scratchpad memories is accessed. The configuration of the dynamic memory power manager has to be adjusted so that it performs a transition to power down after a minimal time without activity on the SDRAM main memory. Using this optimistic method, it may occur in rare cases that a transition to power down is being done when the overhead for the transition is actually higher than the savings obtained by a short power down period. However, generating additional instructions to actively control the main memory states, and considering them in the model is not required in this setup.

5.2.2 Prerequisites

In the motivating example it was implicitly assumed that the main memory can be put to power down mode if memory objects are accessed from the scratchpad memory. This includes the assumption that no accesses to the main memory will take place during this time. There are two issues that require special attention in this context:

- Stack Accesses: if the stack is located in the main memory, and it is accessed while a basic block is executed from scratchpad memory, then the power down phase of the main memory is interrupted due to the access to the stack. To avoid this behavior, the stack was transferred to a special scratchpad partition for the experiments using the technique described in [Ste03]. It has to be ascertained that the stack will not exceed the bounds of the scratchpad partition, which is the case for the considered benchmarks.

- Global variables that are stored in the main memory and are accessed from within a basic block that is executed from the scratchpad memory will lead to the main memory being activated. To avoid this behavior, the relationship of basic blocks and the variables accessed by them is considered in the formulation of the optimization problem. The algorithm can thus allocate basic blocks and their related variables to the scratchpad together in order to prevent the main memory from being activated for a data access.

5.2.3 Memory Objects and Energy Functions

The sets V and BB contain all global variables and basic blocks of the application program, respectively:

$$V := \{v_1, \ldots, v_p\} \tag{5.2}$$

$$BB := \{bb_1, \ldots, bb_r\} \tag{5.3}$$

Together, BB and V form the set of memory objects O:

$$O := V \cup BB = \{v_1, \ldots, v_p, bb_1, \ldots, bb_r\} \tag{5.4}$$

The total number of memory objects is denoted as $n = p + r$. For every memory object o_i, there is a query function $Size(o_i)$ that returns the size of object o_i in bytes.

The set of m available scratchpad memory partitions is defined as

$$SPM := \{spm_1, \ldots, spm_m\} \tag{5.5}$$

Note that in contrast to the definition of MP used in Section 4.2, the main memory is not included in this set of scratchpad memory partitions. The size of scratchpad memory partitions is available by using function $Size(spm_j)$.

The number of cycles required to execute one basic block $o_i \in BB$ from the SDRAM main memory is denoted as $T_{SDRAM}(o_i)$. This time always includes accesses to variables that take place from within this basic block. The background energy dissipated in the main memory during the execution of this basic block consists of the standby and the refresh power that need to be spent over this time period:

$$\begin{aligned} &E_{SDRAM}(o_i) = \\ &(P_{SDRAM\_STBY} + P_{SDRAM\_REFRESH}) \cdot t_{CK} \cdot T_{SDRAM}(o_i) \end{aligned} \tag{5.6}$$

Cycles (denoted as T) are transformed into time values (denoted as t) by multiplying them with the duration of a single clock cycle t_{CK}. Note that since the optimization uses only the standby energy of the main memory as the objective function, the actual access energies of objects are not considered in the cost functions.

If the considered basic block o_i was instead executed from one of the available scratchpad memory partitions, the time required to fetch the basic block's instructions would be reduced, since scratchpad memories are generally faster than SDRAM main memories. If the number of cycles required to execute the basic block o_i from the scratchpad is denoted as $T_{SP}(o_i)$, then the main memory could be put into the power down state for this period of time, assuming that no global variables in the main memory are accessed from within o_i. The energy dissipation during the transition to power down is modeled, according to the vendor's specification, by assuming the memory is in the standby mode during this time. The overhead time for the transition is T_{OH} cycles, whereas for the remaining $T_{SP}(o_i) - T_{OH}$ cycles, the main memory is in power down mode. Figure 5.4 shows the states of the main memory when memory object o_i is executed from the scratchpad memory: The light grey area denotes the power down state, whereas the darker bars represent the transition from active to power down and vice versa. The number of cycles to change to power down and to wake up the memory are summarized in the value T_{OH}.

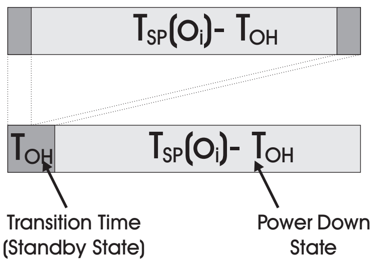


Fig. 5.4. Main memory power dissipation during scratchpad execution

The standby energy dissipated in the main memory while basic block o_i is executed from the scratchpad memory is expressed as

$$\begin{aligned} E_{SP}(o_i) = E_{OH} + P_{SDRAM\_REFRESH} \cdot t_{SP}(o_i) + \\ P_{SDRAM\_PDN} \cdot (t_{SP}(o_i) - t_{OH}) \end{aligned} \quad (5.7)$$

with

$$E_{OH} = P_{SDRAM\_STBY} \cdot t_{OH} \quad (5.8)$$

assuming that no other main memory accesses are required. E_{OH} models the energy dissipated during the transition to the power down mode.

Since dynamic profiling was performed in our experiments to determine accurate execution and access counts, the number of executions of basic block o_i can be determined as $\#exec(o_i)$. Also from dynamic profiling, the number of times that control flow passes along a particular edge connecting two nodes o_i and o_k is known. This value is attributed to the corresponding edge as $e(o_i, o_k)$. Finally, enprofiler also determines the number of accesses to variables from within each basic block (cf. Section 3.6.2 on Page 81 for a description of the statistics generated by enprofiler). The relationship between a basic block bb_i and a variable v_k is modeled as an additional, artificial dummy edge within the control flow graph. It is annotated with the number of accesses $\#acc(o_i, v_k)$. Assuming a number of l accesses to global variables when basic block o_i is executed once from the scratchpad memory, a pessimistic estimate for the additional background energy required can be expressed as

$$t_{DAT} = l \cdot t_{CK} \cdot (T_{DAT} + T_{OH}) \tag{5.9}$$
$$E_{DAT} = P_{SDRAM\_STBY} \cdot t_{DAT} \tag{5.10}$$

with T_{DAT} being the time it takes to access the global variable (which may depend on the length of the data type). To determine the total energy required in the SDRAM main memory when basic block o_i is executed from the scratchpad memory, but variables in the main memory are accessed, the energy required to access the variables has to be added to Equation 5.7. At the same time, the main memory's power down time is reduced by t_{DAT}, which is thus subtracted accordingly, leading to:

$$\begin{aligned} E_{SP}(o_i) = E_{OH} + E_{DAT} + P_{SDRAM\_REFRESH} \cdot t_{SP}(o_i) + \\ P_{SDRAM\_PDN} \cdot (t_{SP}(o_i) - t_{OH} - t_{DAT}) \end{aligned} \tag{5.11}$$

Figure 5.5 demonstrates the situation of the main memory having to transition to the active state and going back to power down mode after the variable access. Note that the effect of having to perform additional transitions from power down to the active state and vice versa is modeled within E_{DAT} by considering l times the overhead time t_{OH} and multiplying it with the standby power.

Up to this point, every basic block was assumed to require a certain overhead time and energy to transition to the power down state. This energy is not required for basic blocks o_k if the main memory is still in power down due to the previous basic block o_i also being executed from the scratchpad memory. In this case, the transition overhead E^{+}_{OH}, which is equivalent to $(P_{SDRAM\_STBY} - P_{SDRAM\_PDN}) \cdot t_{OH}$, can be subtracted from the energy equation 5.7 for the subsequent basic block o_k, which yields

$$E_{SP}(o_k) = (P_{SDRAM\_REFRESH} + P_{SDRAM\_PDN}) \cdot t_{SP}(o_k) \tag{5.12}$$

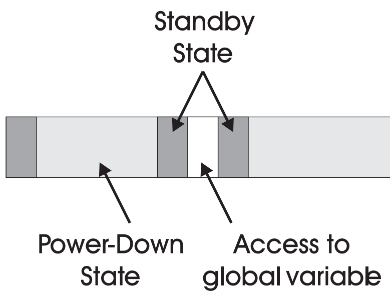


Fig. 5.5. Main memory status when a global variable is accessed in the main memory

assuming no variables are accessed from within basic block o_k. In a similar way, basic blocks accessing variables that are not stored in main memory, but are also allocated to a scratchpad memory partition, receive a benefit E^+_{DAT} since the main memory does not have to perform a transition to the active state. This benefit is given as

$$E^+_{DAT} = (P_{SDRAM\_STBY} - P_{SDRAM\_PDN}) \cdot t_{CK} \cdot (T_{DAT} + T_{OH}) \quad (5.13)$$

This equation expresses the energy savings obtained if one access to the corresponding variable goes to the scratchpad memory and thus one transition from power down to the active state and vice versa can be saved.

5.2.4 Binary Decision Variables

In order to formulate the optimization problem as a set of ILP equations, binary decision variables need to be defined. The results presented in the following section were generated using multiple scratchpad memory partitions using a setup similar to the one described in Section 4.2. Therefore, the model has to be able to handle partitioned scratchpad memories for allocation.

Assuming that m scratchpad memory partitions are present, the decision variables are arranged in a matrix

$$\tilde{O} := \begin{pmatrix} \tilde{o}_{1,1}, \ \ldots \ , \tilde{o}_{1,m} \\ \vdots \quad \ddots \quad \vdots \\ \tilde{o}_{n,1}, \ \ldots \ , \tilde{o}_{n,m} \end{pmatrix} \quad (5.14)$$

The meaning of the decision variables $\tilde{o}_{i,j}$ is defined as

$$\tilde{o}_{i,j} := \begin{cases} 1, \text{ if } o_i \in O \text{ is allocated to scratchpad partition } spm_j \in SPM \\ 0, \text{ otherwise} \end{cases} \quad (5.15)$$

The case of a memory object being allocated to the SDRAM main memory is captured in a separate decision variable which is defined as

$$\tilde{o}_{i,SDRAM} := \begin{cases} 1, \text{ if } o_i \text{ is assigned to the SDRAM main memory} \\ 0, \text{ otherwise} \end{cases} \tag{5.16}$$

Additional decision variables that denote control flow along an edge without having to leave the current scratchpad memory partition are used to model the size benefit that results when two basic blocks connected by a control flow edge are assigned to the same memory partition. As explained in detail in Section 4.2.6, a longjump instruction is always assumed to be required for each control flow edge. When two connected basic blocks are assigned to the same partition, then the size of the longjump instruction is subtracted since the longjump is not required. This correction is only done for the space constrained scratchpad memory partitions, since the main memory is assumed to be sufficiently large. The corresponding decision variables are defined as

$$x_{i,k,j} := \begin{cases} 1, \text{ if } o_i \text{ and } o_k \in O \text{ are both allocated to partition } spm_j \in SPM \\ 0, \text{otherwise} \end{cases} \tag{5.17}$$

In addition, decision variables are required to express the fact that whenever control passes from one scratchpad memory partition to another without touching the main SDRAM memory, then the SDRAM can remain in the power down mode and thus, the energy benefit E^+_{OH} may be subtracted from the objective function's energy value. The decision variable $y_{i,k}$ representing an edge between two basic blocks o_i and o_k is defined to take the value 1 when none of the two basic blocks are allocated to the main memory, and 0 otherwise:

$$y_{i,k} := \begin{cases} 1, \text{ if neither } o_i \text{ nor } o_k \in O \text{ are assigned to main memory} \\ 0, \text{ otherwise} \end{cases} \tag{5.18}$$

Due to the uniform modeling of both control flow and variable accesses as edges between memory objects, the variables $y_{i,k}$ can also be used to consider data accesses that go to scratchpad memory partitions and thus do not wake up the main memory. In this case, the energy benefit E^+_{DAT} is subtracted. Note that the introduction of this decision variable is a novel contribution of this work. In [Ker05], only the decision variables $x_{i,k,j}$ were used to represent the positive effect of not having to wake up the main memory. However, there is no reason that two memory objects connected by an edge have to be assigned to the same scratchpad memory partition in order to achieve the mentioned benefit concerning prolonged power down periods of the main memory: as long as none of the memory objects is assigned to the main memory, the SDRAM can stay in power down mode and the benefit E^+_{OH} or E^+_{DAT} may be subtracted from the objective function.

5.2.5 Objective Function

The objective function which allocates memory objects to the scratchpad memory in order to maximize the time the main memory can be kept in the power down state is formalized as a minimization function. It takes into account both the allocation of basic blocks and variables either to the main SDRAM memory or to one of the scratchpad memory partitions. Additionally, the benefit of not having to wake up the main memory when only scratchpad partitions are involved is also considered. Assuming $o_k \in V$ and $o_l \in BB$, the objective function takes the following form:

$$\begin{aligned}\text{Minimize} \quad & \sum_{i=1}^{p}\sum_{j=1}^{m} \#acc(v_i) \cdot E_{SP}(o_i) \cdot \tilde{o}_{i,j} + \\ & \sum_{i=1}^{r}\sum_{j=1}^{m} \#exec(bb_i) \cdot E_{SP}(o_i) \cdot \tilde{o}_{i,j} + \\ & \sum_{i=1}^{p} \#acc(v_i) \cdot E_{SDRAM}(o_i) \cdot \tilde{o}_{i,SDRAM} + \\ & \sum_{i=1}^{r} \#exec(bb_i) \cdot E_{SDRAM}(o_i) \cdot \tilde{o}_{i,SDRAM} - \\ & \sum_{i=1}^{n}\Bigg(\sum_{o_k \in Succ(o_i)} e(o_i, o_k) \cdot E^{+}_{OH} \cdot y_{i,k} + \\ & \sum_{o_l \in Succ(o_i)} e(o_i, o_l) \cdot E^{+}_{DAT} \cdot y_{i,l} \Bigg)\end{aligned} \tag{5.19}$$

The first four lines of the objective function determine the energy cost of each memory object individually: if it is allocated to one of the scratchpad memory partitions, the background energy E_{SP} is consumed in the main memory with every access or execution. The energy E_{SDRAM} is dissipated if it is allocated to the main memory.

The fifth and sixth lines determine the energy benefit that can be achieved if connected basic blocks are moved to scratchpad memory partitions, or if variables that are accessed from within a basic block are not kept on the main SDRAM memory. These relationships are expressed using the decision variables $y_{i,k}$ and $y_{i,l}$, respectively.

5.2.6 Constraints

Every memory object has to be allocated to one memory partition, either main memory or one of the scratchpad partitions:

$$\forall i : 1 \leq i \leq n : \sum_{j=1}^{m} \tilde{o}_{i,j} + \tilde{o}_{i,SDRAM} = 1 \tag{5.20}$$

The limited memory capacity of the scratchpad memory partitions (but not of the sufficiently large main memory) also has to be considered to generate valid results. If two memory objects o_i and o_k are connected by a control flow edge, i.e. o_k is a successor of o_i, and they are both allocated to the same scratchpad memory partition $spm_j \in SPM$, then the size of the longjump already considered in the model can be subtracted again to allow more memory objects to be allocated to the limited scratchpad space:

$$\begin{aligned}
&\forall j : 1 \leq j \leq m : \\
&\sum_{i=1}^{n} \Bigg(\tilde{o}_{i,j} \cdot Size(o_i) - \\
&\sum_{o_k \in Succ(o_i)} Size(longjump) \cdot x_{i,k,j} \Bigg) \leq Size(spm_j)
\end{aligned} \tag{5.21}$$

The correct setting of the decision variables modeling control flow within one scratchpad partition is provided by the following set of constraints, where m only specifies the number of scratchpad memory partitions:

$$\begin{aligned}
\forall i : 1 \leq i \leq n, 1 \leq k \leq n, \forall j : 1 \leq j \leq m : \\
\tilde{o}_{i,j} + \tilde{o}_{k,j} - 2 \cdot x_{i,k,j} \geq 0
\end{aligned} \tag{5.22}$$

This set of constraints will prevent variable $x_{i,k,j}$ to be set to 1 if the two memory objects o_i and o_k are not allocated to the same scratchpad partition. Setting any of the decision variables $x_{i,k,j}$ to the value 1 has a positive effect on the objective function's value, since it will cause more memory objects to fit onto the limited scratchpad space. An additional constraint that ensures the value 1 for $x_{i,k,j}$ is thus not required.

The additional benefit of not having to wake up the main memory when two basic blocks connected by a control flow edge both being allocated to arbitrary scratchpad partitions, but not to the main memory, is modeled by the decision variables $y_{i,k}$, with o_k being o_i's successor node. The correct setting of these decision variables can be achieved using ILP equations by defining

$$\begin{aligned}
\tilde{o}_{i,SDRAM} + y_{i,k} \leq 1 \\
\tilde{o}_{k,SDRAM} + y_{i,k} \leq 1
\end{aligned}$$

which can be summarized for all basic blocks to

$$\begin{aligned}
\forall i, k : 1 \leq i \leq n, o_k \in Succ(o_i) : \\
\tilde{o}_{i,SDRAM} + \tilde{o}_{k,SDRAM} + 2y_{i,k} \leq 2
\end{aligned} \tag{5.23}$$

Since a value of 1 for the decision variables $y_{i,k}$ will directly improve the value of the objective function, these constraints that force these decision variables to be set to 0 when one of the basic blocks o_i or o_k is assigned to the main memory are sufficient to achieve correct results.

The same reasoning also applies if object o_i is a basic block and o_k is a variable that is accessed from within this basic block: in this case, the benefit value E^{+}_{DAT} may be subtracted from the objective function that is to be minimized.

In total, the number of constraints and decision variables used in the presented model amounts to

$$\#Constraints(\text{PowerDown}) = n + m + (m+1) \cdot |E| \qquad (5.24)$$

$$\#DecisionVariables(\text{PowerDown}) = (m+1) \cdot (n + |E|) \qquad (5.25)$$

During the execution of the ILP solver, no difficulties with long solution times were encountered. The straightforward addition of the decision variables $y_{i,k}$ has again shown that the high flexibility is a clear benefit of utilizing ILP equations to model optimization problems.

5.2.7 Results for Main Memory Power Management

The results presented in this section were generated by exploiting the presence of a partitioned scratchpad memory in order to put the main memory to power down mode. The presented optimization solely considers the standby energy dissipation of the main SDRAM memory in the objective function. It was used in a setup similar to the one described in Section 4.2, consisting of several scratchpad memory partitions that memory objects may be allocated to. However, the main memory is not in SRAM technology, as in Section 4.2, but rather consists of a dynamic SDRAM memory that offers an energy saving power down mode exploited using the presented optimization. The SDRAM main memory used for the experiments presented in this section is a 128 MBit mobile SDRAM chip [Mic04a] operating at 3.3 Volts and 33 MHz to fit into the environment of the ARM7TDMI.

The standby energy dissipation of the SDRAM main memory is determined according to the data-sheet and profiling based methodology described in Section 3.4.3 on Page 51. Due to the highly dynamic nature of SDRAM memories, it is not possible to provide energy per access values, since the energy dissipation of SDRAM depends highly on the access pattern. The used values derived from the vendor's data sheet are summarized in Table 5.2. The names I_{DD1} through I_{DD5} are the common designations found in SDRAM data sheets.

Only the background, standby energy of the considered main memory in SDRAM technology is used as the metric for energy dissipation in this model. No notion of access related energy is included in the model, neither for the

Value name	Symbol	Current [mA]
active operating current	I_{DD1}	130.0
power-down standby current	I_{DD2}	0.45
active standby current	I_{DD3}	40.0
operating current	I_{DD4}	115.0
auto refresh current	I_{DD5}	3.0

Table 5.2. Data sheet values used for the experiments [Mic04a]

SDRAM nor for the scratchpad memory partitions. Since scratchpads are usually built using SRAM technology which has a very low standby energy compared to the access energy, the energy contribution of the scratchpad memory partitions can be neglected in this model. In the simulation run following the optimization, all energy contributions, including access related energy, are being considered. The optimization minimizes the standby energy of the main memory by exploiting the presence of the scratchpad memory partitions and keeping the main memory in the power down state.

The application benchmarks considered consist of an acceptably large number of memory objects. In addition, the proposed allocation algorithm considers the relationship of global variables and their accessing basic blocks. To allow the exploitation of this detail, global variables should be present in the benchmarks. The benchmarks used to generate results are summarized in Table 5.3.

Benchmark	Code Size [$bytes$]	Data Size [$bytes$]	Description
ME	796	4	Media application using intensive integer arithmetic
FIR	136	2732	Finite Infinite Response Filter Application
ADPCM	724	6928	Encoder and decoder using Adaptive Differential Pulse Code Modulation
Multi_Sort	716	1204	Sorting benchmark (combining several sorting algorithms)

Table 5.3. Selected benchmarks to evaluate the energy management aware allocation

All results generated using the proposed energy management aware allocation technique are compared to the Bottom-Up multi memory allocation technique since it provided the best results. The potentially large number of possible combinations of memory partitions was restricted in order to provide a concise overview over the obtained results. In this section, a maximum of two memory partitions is considered. The maximum size of each individual partition is 1024 bytes. In addition, the maximum combined size of the scratchpad partitions is kept below the total size of the considered benchmark, since otherwise all objects are allocated to the scratchpad and the main memory is not used at all. This represents a trivial solution in the considered setup which does not allow a comparison of the used allocation techniques.

Note that the results presented in this section for the Bottom-Up approach are not directly comparable to the ones shown in Section 4.2.9, since in this section, a main memory in DRAM technology is assumed in contrast to the SRAM in the multi memory allocation results. The energy dissipation of the SDRAM main memory and that of the scratchpad partitions is used to control the Bottom-Up optimization algorithm. It decides which memory objects are allocated to which partitions according to the ILP problem presented in Section 4.2.6. However, when the resulting application is later simulated, the main memory partition is considered to be put in the power down mode when it is not used. In contrast, the optimization algorithm presented in the previous section takes this effect into account and tries to allocate memory objects in such a way that the length of the power down times for the main memory are maximized.

The following figures show the overall energy savings that are achievable when scratchpad memory partitions of different capacities are present in the system. When only one single scratchpad partition is assumed, its size is given on the x-axis. If multiple scratchpad partitions are assumed, then both the total capacity and the two comprising scratchpad sizes are mentioned. The results are expressed as relative improvements compared to the case that no scratchpad is used. The proposed allocation algorithm that takes the power down of the main memory into account is labeled as "PowerDown Allocation", whereas the Bottom-Up approach is denoted as "MultiMemory Allocation".

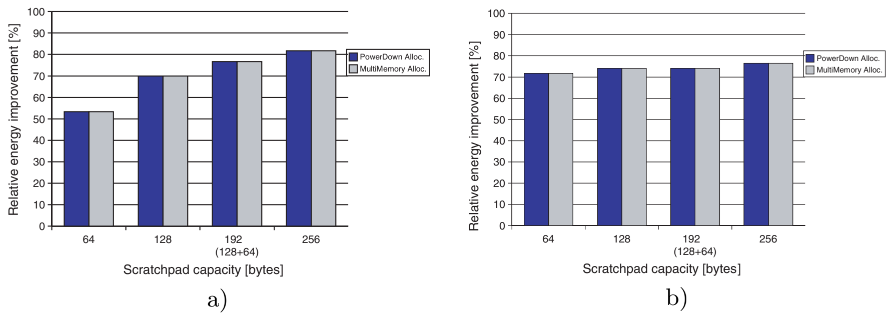


Fig. 5.6. Power down aware allocation for a) ME b) FIR

The ME application only contains one single scalar global variable. The optimization potential is therefore restricted to the allocation of instructions to the scratchpad memory in order to enable prolonged power down periods. As can be seen in Figure 5.6 a), both the PowerDown and the Bottom-Up multi memory allocation techniques generate similar, if not identical results: they both utilize the fast and energy efficient scratchpad memory to store instructions and thus reduce the energy dissipation within the memory subsystem. The motivation behind this utilization of the scratchpad resource is different, however: while the Bottom-Up model uses the scratchpad due to the reduced access related energy costs, the PowerDown approach uses the scratchpad memory in order to put the main memory into low power mode. Since the execution time of the application is reduced due to the use of the faster scratchpad memory, additional non access related energy dissipation can be saved. For large scratchpad partitions, both approaches obtain an improvement concerning energy dissipation of around 80%. This shows that the presented allocation performs at least as well as the multi memory allocation using the Bottom-Up model.

Nearly the same behavior can be observed for the FIR benchmark shown in Figure 5.6 b), which consists of fewer basic blocks, some of which are frequently executed and can reduce the overall energy dissipation by more than 70% even for a scratchpad partition of only 64 bytes. Increasing the scratchpad size does not lead to significant additional energy savings. Again, both the Bottom-Up allocation and the proposed power down aware technique yield similar results.

While the ADPCM benchmark uses a number of global variables, the high execution counts of the basic blocks limit the optimization potential for allocating variables to the scratchpad memory. For a total scratchpad capacity of 256 bytes, consisting of two 128 byte scratchpad partitions, Figure 5.7 a) shows that about 2 percentage points of additional savings could be obtained compared to the Bottom-Up approach, even though the allocation only differs in two basic blocks. This small change in the allocation results in a time prolonged by about 5% during which the main memory can stay in the power down state. In addition, a reduction of the execution time of around 3% can be observed. These two effects, shown in Figures 5.7 b) and c), respectively, together result in the achieved savings. For larger available scratchpad capacities, both approaches again save more than 80% of energy compared to a system without scratchpad.

For the Multi_Sort benchmark (cf. Figure 5.8), most scratchpad partition sizes result in similar results for the energy management aware allocation technique and for the Bottom-Up approach. However, increased savings of 5 percentage points can be observed for a scratchpad capacity of 768 bytes. The reason for these savings lies in the fact that the relationship between variables and their accessing basic blocks is taken into account in the model: in contrast to the Bottom-Up model, which is not aware of this connection between memory objects, the proposed allocation technique allocates a frequently accessed array to the scratchpad memory.

Since in this way, both the accessing basic blocks and the accessed data are on the scratchpad, the time that the main memory can be in the power down state is increased by more than 30%. The execution time is reduced by 9% compared to the Bottom-Up allocation.

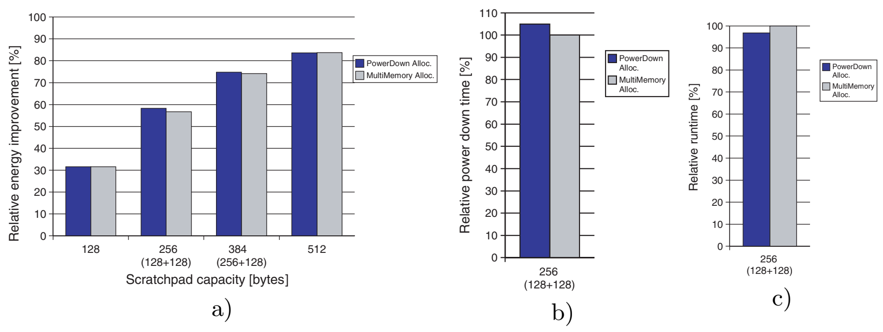


Fig. 5.7. Power down aware allocation for ADPCM: a) Results, b) power down intervals, c) execution time for 128 bytes capacity

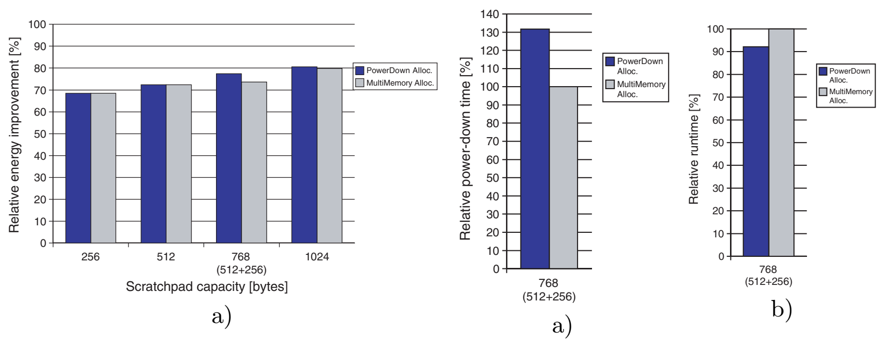


Fig. 5.8. Power down aware allocation for Multi_Sort: a) Results, b) power down intervals, c) execution time for 764 bytes capacity

In order to allow a clear presentation of the obtained optimization results and to provide a fair comparison and benefit analysis with respect to the previous work which serves as the baseline, burst mode accesses to the SDRAM main memory were not considered in the presented results since burst accesses were not supported by the previously studied allocation strategies. The energy savings achieved by taking the possibility of burst accesses into account when the SDRAM main memory is accessed would add an additional aspect to the analysis of the presented results, since the absolute number of possible burst accesses varies with the number of main memory accesses. The presented optimization exploits the presence of scratchpad memories, thus the number of accesses to the main memory will decrease for growing scratchpad sizes, and so will the potential for energy savings due to burst accesses. To demonstrate this point, some of the experiments involving the power down optimization were performed taking burst accesses into account. The results are shown in Figure 5.9 for the ME and the Multi_Sort benchmarks. The figures compare the results obtained by the power down optimization when bursts are considered to the case without bursts.

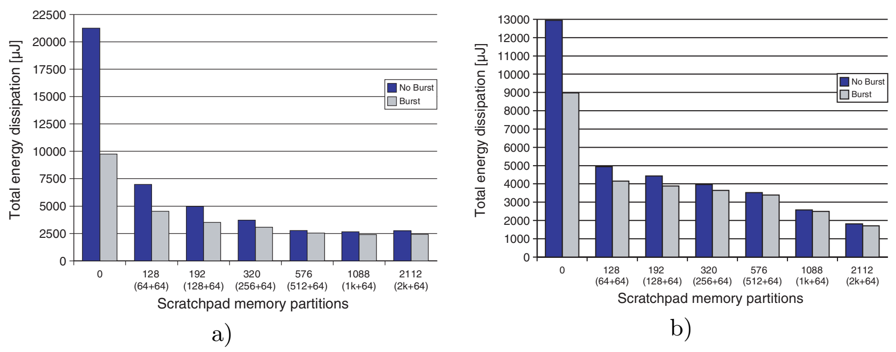


Fig. 5.9. Considering burst accesses for the PowerDown optimization a) ME b) Multi_Sort

While a large amount of energy can be saved when no scratchpad memory is present in the system (more than 50% for the ME application), the benefit obtained from burst accesses, as expected, is reduced significantly when scratchpad partitions of increasing capacities are added to the system. Considering burst accesses within the main memory would thus cause the savings obtained by the optimization and by the burst accesses to be overlayed, which would not allow a clear view on the achieved savings of the optimization. For this reason, and to be able to compare results to the previous work, burst mode accesses were not considered in the presented results.

In summary, the presented results show that the proposed memory allocation strategy that takes into account non-access related memory costs and tries to maximize the time that the main memory can be kept in the power down state performs at least as well as the previously presented Bottom-Up approach. Additional energy savings can be achieved when global variables are allocated to the scratchpad memory together with the basic blocks that access them. This leads to the situation where the main memory may not be required for a prolonged period of time, since both instructions and data can be read from the fast and energy efficient scratchpad. The consideration of non access-related standby energy represents a novel contribution leading to additional energy savings in the presence of main memories that offer power management features. A possible extension of the proposed models including a combination of both access and non access related energy dissipation is discussed in Chapter 8, “Future Work”.

5.3 Execute-In-Place using Flash Memories

Most currently available embedded systems offer a variety of memories to meet the user's needs to store large amounts of data on one hand, and to achieve low cost, small size and high performance on the other hand. Also, embedded systems require a non-volatile memory to store e.g. configuration information or the boot sequence instructions. This information should not be lost, even when the batteries are drained. The non-volatile memory utilized in most systems today is manufactured using Flash memory technology. Current Flash memory technology offers acceptably fast read accesses (though it is still significantly slower than e.g. SDRAM) and a low standby energy dissipation. On the downside, only a limited number of slow block-wise write accesses are possible, making Flash memories unsuitable for storing frequently changing data. The descriptions of Flash memory in Section 3.2.3 on Page 24 offer more detailed information.

In general, Flash memories can be used in several ways in an embedded system: in "Store-And-Download" architectures, parts of the content of the Flash memory are first copied to a different memory before actually accessing them. This mechanism is also used by the evaluation board and the supporting software considered in this work: a linker script contains information concerning the placement of memory objects in the main memory. A small Flash-based boot loader handles the task of copying the required instructions and data to the designated memory and then executes the application from there. In most systems, the entire operating system and applications are first transferred from Flash memory to the main memory at boot time, which is known as "fully shadowed code". As another option, the application or the operating system can request additional pages to be loaded from the Flash to the main memory ("demand paged code"). The full potential of Flash memories is not exploited in these cases, since Flash memories built in the NOR technology are suitable for use as Read-Only memories, e.g. to serve as instruction memory. This technique of directly executing instructions from the Flash memory is called eXecute In Place (XIP). As a relatively new development, Flash memories in NAND technology are now being fitted with additional circuitry to also allow them to be used as XIP memories [PSB$^+$03]. In the further course of this section, the special case of XIP using NAND Flash memories will not be considered. The main advantage of using XIP is the fact that there is no need to transfer the entire operating system or application program to the main memory (or to any other memory within the hierarchy) at startup, since the code can remain in the Flash and be accessed and executed from there.

The prolonged Flash access times and the increased energy dissipation of the Flash compared to the SDRAM main memory have to be traded against the advantage of not having to transfer the memory objects to the main memory. An optimization targeting a NOR-Flash memory's XIP capabilities thus has to be capable of taking timing as well as energy dissipation into account. In the end, those basic blocks that are frequently executed will be copied to the faster main memory. For the remaining instructions, the small number of accesses can not compensate the high copying overhead. They therefore remain in the Flash memory. During accesses to the Flash memory, the SDRAM main memory with its normally high standby energy dissipation may be put into power down mode, leading to additional energy savings. Finally, using parts of the Flash memory to execute code leads to reduced main memory requirements which translates to reduced cost of the device: since the Flash memory is always required to permanently store the operating system and the application software, it may just as well be used during normal operation of the device rather than only at boot time.

In order to integrate the XIP optimization into the encc compiler framework, the assumed operating frequency of the memory bus has to be adjusted to a value that allows both SDRAM main memory and Flash memory to be operated in realistic scenarios. In general, the external bus frequency depends on the internal frequency of the CPU: either, processor and bus operate at the same frequency, or the external bus frequency is the CPU frequency divided by an integer value. If the memory bus is too slow, then the speed difference (in terms of cycles per access) between Flash and SDRAM memories is negligible in the model and consequently, no advantage for the faster SDRAM memory can be determined. This will lead to optimization results that execute the entire application from the Flash memory using XIP due to the advantage of not having to copy objects at startup. In a realistic scenario, however, the speed and energy benefits of the SDRAM main memory may lead to energy savings when parts of the application are allocated to the SDRAM, despite the necessity to copy memory objects at startup. To make this effect apparent, the memories need to be operated at sufficiently high frequencies. Since the ARM7 CPU on the used ATMEL evaluation board only operates at 33 MHz, it is necessary to increase the processor speed in order to provide a sufficiently fast external bus frequency for the used modern SDRAM and Flash memories. However, changing the CPU operating frequency requires some thoughts concerning the used processor energy model that was established based on the slower processor.

The average power of the processor is given by the general equation

$$P = \alpha \cdot C_L \cdot (V_{DD})^2 \cdot f \tag{5.26}$$

with α denoting the switching activity, C_L the load capacitance of e.g. external bus wires, V_{DD} the supply voltage and f the frequency. Based on our energy model, the average power consumption in the considered ARM7TDMI processor operated at 3.3 Volts and 33 MHz is round about 150 mW. If the processor frequency is increased to 100 MHz, the average power is increased to about 450 mW according to Equation 5.26. Current processors that are capable of operating at a clock frequency of 100 MHz are usually implemented using a smaller feature size which in turn allows the supply voltage to be dropped to between 1.7 and 1.9 Volts. Since the operating voltage is squared in Equation 5.26, the factor of three introduced by increasing the frequency is thus compensated.

In addition, according to [ARM04b], the average power dissipated by a specific ARM processor is steadily decreasing with each generation, despite the fact that the operating frequencies are increased to higher and higher levels (cf. Table 5.4). This leads to the conclusion that by keeping the processor energy model unchanged, the energy dissipated within the processor is never underestimated. The influence of the used memories on the overall system energy dissipation may thus be even higher than what was measured using the unchanged processor energy model. Leaving the processor energy model untouched is thus a conservative and safe approximation.

Feature Size $[\mu m]$	Area $[mm^2]$	Power Consumption $[mW/MHz]$	Frequency (Worst Case)
0.35	2.14	2.07	45 MHz
0.25	1.0	0.80	60 MHz
0.18	0.5	0.25	84 MHz
0.13	0.3	0.06	125 MHz

Table 5.4. ARM7TDMI characteristics for different feature sizes

The following sections consider some of the preliminaries required to exploit the XIP capability of NOR Flash memories using an optimization within the compiler.

5.3.1 Analysis of the Copy Function

In most of the previously described experiments in this work, the startup code used by the ARM7 evaluation board was not included in the analysis, since minimization of the energy dissipation during the actual execution time of the application was the optimization goal. The XIP optimization presented in this section considers executing instructions directly from the Flash memory instead of first copying them to another memory in the system. This new perspective makes it necessary to also consider the startup process in order to determine the benefit achieved by not having to copy an object during the startup process.

In the startup code, the source address, the target address and the length of the block to be transferred are first loaded during an initialization phase. The information concerning the source and target locations of code and data is generated by the linker according to the used linker scripts. By performing a thorough analysis of the copy routine that is executed following the initial boot process of the processor, the time and the energy required to copy a memory object o_i can be determined. Since program objects are in general copied from the Flash memory to the SDRAM main memory, the overhead required to accesses these memories also has to be accounted for. This is done using the number of accesses and the Flash and SDRAM timing and energy models presented in the subsections of Chapter 3.

All considerations in the presented model are based on the actual copy routine found in the ARM software development toolkit, despite the fact that more efficient implementations of the copy routine are possible (cf. [Pet04] for an example). In the copy routine, the 32 bit ARM instruction set is used. Since the used instruction level energy model only covers the THUMB instruction set with a distinct energy value for each instruction [Ste03], one average value for all 32 bit ARM instructions I_{ARM} is assumed for the 32 bit instructions used within the copy routine.

For details concerning the analysis of the copy function and the energy equations, please refer to [Ker05]. For the further course of this work, it is sufficient to define the energy required to copy a memory object o_i from the Flash to the SDRAM main memory. It consists of the energy dissipated within the CPU and the energy required to access both the Flash and the SDRAM main memory. The copy energy can thus be written as

$$E_{COPY}(o_i) := E_{CPU}(o_i) + E_{MEM}(o_i) \tag{5.27}$$

5.3.2 Main Memory Partitioning

In order to efficiently exploit the XIP functionality of Flash memories built in NOR technology, it is essential that the main memory can be put into low power mode whenever fetching of instructions is performed from the Flash memory. Accesses to variables can make it necessary to wake up the main memory, as shown for the power down optimization in Figure 5.5 on Page 182. If a scratchpad memory is used as the alternative storage space, global variables can simply be allocated to the scratchpad along with the accessing basic block in order to avoid the problem of frequently having to wake up the main memory. Using a Flash memory instead of a scratchpad, this is not an option, since Flash memories are not suitable for storing data. On one hand, they are very slow since writing can only be performed in large blocks, on the other hand a typical Flash memory only survives 10^5 to 10^6 write operations before it becomes unusable [HP03]. In order to still allow at least parts of the SDRAM main memory to stay in power down mode for a long period of time, it was partitioned as shown in Figure 5.10: one SDRAM partition carries the global variables, the other carries the frequently executed instructions that are copied from the Flash memory during startup. In this way, the SDRAM instruction partition can stay in power down mode whenever instructions are fetched from the Flash memory. Since Flash is not usable as data memory, all data memory objects have to be copied to the SDRAM data partition at startup. The stack is also moved to the data partition of the SDRAM main memory. The SDRAM instruction partition can be chosen by the compiler to accommodate frequently accessed basic blocks and functions, since the initial overhead caused by the copy operation can be regained due to the faster and more energy efficient SDRAM accesses compared to the Flash memory. One additional improvement of this partitioned main memory is the fact that it allows the instruction partition of the main memory to be put into deep power down mode once no more instructions have to be executed from it. If all instructions until the very end of the application can be fetched from the Flash memory, then the SDRAM can be put into DPD mode, which corresponds to the partition being switched off and losing all its contents. If data was also stored in this partition, switching the memory off would not be feasible, since many algorithms use the main memory as the final storing place for results, which would be lost in case of a transition to DPD.

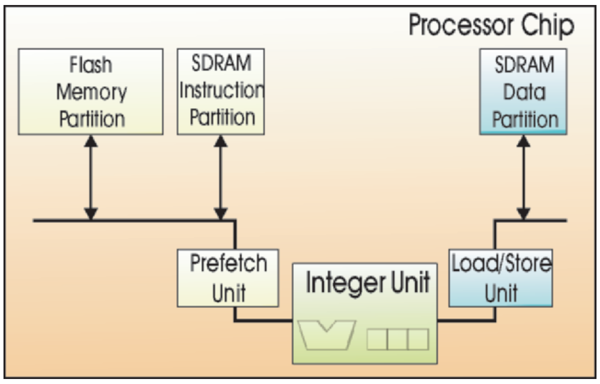


Fig. 5.10. Block diagram of the memory configuration for the XIP optimization

With this memory architecture in mind, the following experiments investigate the energy dissipation, the performance and the total amount of required main memory storage space using the following three allocation strategies for the code section of the applications under observation:

- full code shadowing: the entire code section is copied from the Flash memory to the SDRAM main memory on startup. The code is then executed from the SDRAM memory. This corresponds to the original behavior of the used ATMEL evaluation board and the associated software.
- full execute in place (XIP): all instructions remain on the Flash memory and are executed from there. Only data is initially copied to the data SDRAM partition. The SDRAM instruction memory is not required during the execution of the application and can thus remain switched off during the entire runtime.
- compiler based allocation: the compiler decides which instructions are to be copied to the faster SDRAM and which blocks should remain on the Flash memory. Following the final instruction fetch from the SDRAM instruction partition, it can be put into deep power down mode.

The data sections of the applications are always copied to the SDRAM data partition at startup. In the following sections, the optimization problem is formulated and a heuristic is presented to preselect a number of memory objects that lead to energy saving opportunities using the DPD of the SDRAM memory.

5.3.3 Prerequisites

The energy dissipated within the memories is first determined using a profiling simulation run and a subsequent analysis by enprofiler as described in Section 3.6. enprofiler determines the CPU and memory related energy dissipated during the execution of the application, taking into account both the access related and the non-access related energy values. In addition, the possibility of burst accesses for the SDRAM main memory and intrapage accesses for the Flash memory are considered. Note that the Flash memory described in Section 3.2.3, which was also assumed for the experiments, is not capable of operating in synchronous mode. Therefore, only intrapage accesses are used to enhance the performance of the Flash memory. Sequential intrapage accesses are detected by enprofiler and weighted accordingly. During the initial dynamic profiling run, enprofiler keeps track of the time and the energy it takes to execute a particular memory object from both SDRAM and Flash memory. The following values are determined:

- Executing memory object o_i from SDRAM: Since the application programs are compiled using encc, they use the 16 bit THUMB mode. The access related energy spent during the execution of a particular memory object o_i can be determined according to the number of random (r) and sequential (s) 16 bit read accesses:

$$\begin{aligned} E_{SDRAM\_ACC}(o_i) = r \cdot E_{SDRAM\_RND16\_RD} + \\ s \cdot E_{SDRAM\_SEQ16\_RD} \end{aligned} \tag{5.28}$$

  Using the timing information collected during the profiling run, the non-access related energy can be determined and added to the access related energy. The access related energy of the data partition is not considered in this section, since its contribution does not depend on the instruction allocation decisions taken by the compiler. The non-access related components have to be considered, however, since a substantially longer execution time will also cause the data partition to spend more standby energy.

  When an instruction o_i is executed from the SDRAM, the non-access related standby energy is determined by the fact that the memory is active (P_{SDRAM_STBY}) and needs to be refreshed constantly to keep its information ($P_{SDRAM\_REFRESH}$). The data partition is also assumed to be active, but in the power down mode. This state was chosen in order not to overestimate the data partition's contribution to overall energy, since not every instruction accesses a data element from the data partition. The letters I and D represent the standby energies of the instruction and data partitions, respectively, in the following equations.

$$\begin{aligned}&E_{SDRAM\_LEAKAGE}(o_i) = \\ &(P_{I\_SDRAM\_STBY} + P_{I\_SDRAM\_REFRESH}) \cdot t_{CK} \cdot T_{SDRAM}(o_i) + \\ &(P_{D\_SDRAM\_PDN} + P_{D\_SDRAM\_REFRESH}) \cdot t_{CK} \cdot T_{SDRAM}(o_i)\end{aligned} \tag{5.29}$$

To determine the overall cost for a memory object o_i, the access related costs are added to the non-access related energy and to the cost of executing o_i on the CPU. This value, $E_{SDRAM\_CPU}(o_i)$ is determined according to the instruction level energy model, taking into account the number of cycles required to access the SDRAM. This energy value has to be multiplied with the number of executions $\#exec(o_i)$ of this object. By adding the initial copy overhead caused by copying the object from Flash to SDRAM memory during startup, the overall energy dissipated by executing memory object o_i from the SDRAM memory is

$$\begin{aligned}E_{SDRAM}(o_i) = \#exec(o_i) \cdot (&E_{SDRAM\_LEAKAGE}(o_i) + \\ &E_{SDRAM\_ACC}(o_i) + E_{SDRAM\_CPU}(o_i)) + \\ &E_{COPY}(o_i)\end{aligned} \tag{5.30}$$

- Executing memory object o_i from Flash memory: To determine the energy dissipation when memory object o_i is executed from the Flash memory, s intrapage accesses are assumed for sequential accesses, since the used Flash memory does not support synchronous data transfers. Assuming r random and s sequential accesses thus yields

$$\begin{aligned}E_{FLASH\_ACC}(o_i) = r \cdot &E_{FLASH\_RND16\_RD} + \\ s \cdot &E_{FLASH\_SEQ16\_RD}\end{aligned} \tag{5.31}$$

The Flash memory itself does not have any non-access related energy costs in the used model. However, during execution of instructions from the Flash memory, non-access related energy is dissipated in the instruction and the data partition of the SDRAM memory. With T_{OH} representing the number of cycles it takes to perform a transition from the active state to the power down state or vice versa (during which the memory can be assumed to be in the standby state, as explained in Section 5.2.3), the standby energy of the SDRAM during execution of memory object o_i from the Flash memory can be written as:

$$\begin{aligned} &E_{FLASH\_LEAKAGE}(o_i) = \\ &(P_{I\_SDRAM\_STBY} + P_{I\_SDRAM\_REFRESH}) \cdot t_{CK} \cdot T_{OH} + \\ &(P_{I\_SDRAM\_PDN} + P_{I\_SDRAM\_REFRESH}) \cdot t_{CK} \cdot (T_{FLASH}(o_i) - T_{OH}) + \\ &(P_{D\_SDRAM\_PDN} + P_{D\_SDRAM\_REFRESH}) \cdot t_{CK} \cdot T_{FLASH}(o_i) \end{aligned} \tag{5.32}$$

It consists of the standby and the refresh power of the SDRAM instruction partition for the time T_{OH} required to perform the transition to power down mode. For the remaining time $T_{FLASH}(o_i) - T_{OH}$, the power down current and the refresh current are required. The data SDRAM partition is assumed to be in power down mode during the whole execution time of object o_i from the Flash memory.

The total energy dissipated by the system when instructions are fetched from the Flash memory and executed on the CPU is determined as follows:

$$\begin{aligned} E_{FLASH}(o_i) = \#exec(o_i) \cdot \Big(& E_{FLASH\_LEAKAGE}(o_i) + \\ & E_{FLASH\_ACC}(o_i) + E_{FLASH\_CPU}(o_i) \Big) \end{aligned} \tag{5.33}$$

Note that no copy costs are required in this case, since the instructions can be executed from the Flash memory directly.

This concludes the energy dissipation of memory objects when they are executed either from the SDRAM main memory after having been copied there, or from the Flash memory using XIP.

5.3.4 Preselection of Memory Objects to enable Deep Power Down

The actual optimization algorithm presented in the following section is capable of deciding whether it is beneficial to use the slower Flash memory to execute certain instructions and at the same time put the SDRAM instruction partition into power down mode. Deep power down mode, with its even higher energy saving potential, is not considered in the optimization itself. To allow the SDRAM instruction partition to go to deep power down when no more instructions are to be executed from it, and at the same time restrict the complexity of the final XIP optimization problem, a heuristic algorithm first selects those candidates for XIP execution on the Flash memory that allow the exploitation of the instructions partition's DPD.

The set S represents the execution of basic blocks over time, where each $s_j \in S$ stands for a sequence consisting of only one basic block that may be executed repeatedly:

$$S := \{s_1, \ldots, s_u\} \tag{5.34}$$

o_{s_j} designates the basic block executed in sequence s_j. The number of executions of a particular basic block $o_i = o_{s_j}$ within basic block sequence

s_j is written as $\#exec(o_{s_j})$. Figure 5.11 illustrates these notations. The total number of executions of a basic block o_i can thus be written as

$$\#exec(o_i) = \sum_{j=1}^{n} \#exec(o_{s_j}) \quad \text{with } o_{s_j} = o_i \tag{5.35}$$

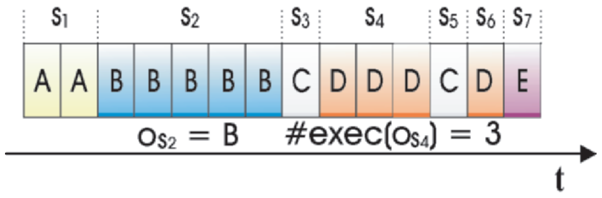


Fig. 5.11. Example basic block sequence for preselection algorithm

To determine basic blocks that should be allocated to the Flash memory in order to allow the instruction partition of the main SDRAM memory to be put to deep power down mode, the sequence of basic blocks is traversed from the last to the first basic block. In this way, the soonest point at which the instruction partition is no longer required and can thus be shut down is determined. All objects following this point in time are then executed from the Flash memory and do not need to be copied to the SDRAM at startup.

Choosing a basic block sequence to be executed from the Flash memory while the instruction partition of the SDRAM memory is in DPD mode requires the energy dissipation of this sequence to be re-calculated, since in contrast to the previous considerations, the DPD energy instead of the power down energy has to be assumed for the SDRAM instruction partition. Since it is difficult to integrate this aspect into a uniform ILP representation together with the consideration of the power down mode of the SDRAM main memory, a two step approach using a separate preselection of memory objects that allow the main SDRAM memory to go to deep power down was chosen. While the instruction partition is thus in DPD, the data partition of the SDRAM is still considered to be in the power down mode, as before, and thus consumes its power down and refresh energy.

$$\begin{aligned} E^{DPD}_{FLASH\_LEAKAGE}(o_{s_j}) = {} & (P_{D\_SDRAM\_PDN} + \\ & P_{D\_SDRAM\_REFRESH} + \\ & P_{I\_SDRAM\_DPD}) \cdot t_{CK} \cdot T_{FLASH}(o_{s_j}) \end{aligned} \tag{5.36}$$

The preselection algorithm first considers all basic blocks to be executed from the Flash memory. Therefore, the energy required to copy the basic blocks from the Flash to the main SDRAM memory is initially not considered in the energy equations. It is considered during the course of the algorithm, when basic blocks actually require copy operations. The cost of copying a set of memory objects $o_i \in M$ to the SDRAM memory is determined as

$$E_{COPY}(M) = \sum_{o_i \in M} E_{COPY}(o_i) \tag{5.37}$$

using $E_{COPY}(o_i)$ as defined above in Equation 5.27.

The initial energy dissipation for executing a basic block sequence from the SDRAM or the Flash memory is assumed to be

$$\begin{aligned} E^{DPD}_{FLASH}(o_{s_j}) = \#exec(o_{s_j}) \cdot (E^{DPD}_{FLASH\_LEAKAGE}(o_{s_j}) + \\ E_{FLASH\_ACC}(o_{s_j}) + E_{FLASH\_CPU}(o_{s_j})) \end{aligned} \tag{5.38}$$

$$\begin{aligned} E^{DPD}_{SDRAM}(o_{s_j}) = \#exec(o_{s_j}) \cdot (E_{SDRAM\_LEAKAGE}(o_{s_j}) + \\ E_{SDRAM\_ACC}(o_{s_j}) + E_{SDRAM\_CPU}(o_{s_j})) \end{aligned} \tag{5.39}$$

Finally, the preselection algorithm requires three sets, which are defined as follows:

- $A := \{o_i : o_i \text{ has pending executions }\}$
- $M := \{o_i : o_i \text{ is suitable for Flash execution }\}$
- $X := \{o_i : o_i \text{ is executed from Flash }\}$

Starting from the last executed basic block, the algorithm determines if the currently considered basic block is being executed at an earlier point in time. If it is, then this basic block is inserted into set A. If a basic block has no more pending executions and may thus be executed from the Flash memory, at the same time allowing the SDRAM instruction partition to be put into DPD mode, then the object is put into set M. Finally, if the energy dissipation is lower when an object and all the objects that are executed at a later point in time are executed from the Flash memory and the SDRAM is put into DPD mode, then all of these objects are inserted into set X. At the end of the algorithm, the set X thus contains those basic blocks from the end of the basic block sequence that should remain in the Flash memory instead of being copied to the SDRAM instruction partition.

Note that since the instruction sequence S is determined using dynamic profiling, it is possible that the preselection algorithm generates suboptimal results: assume that during a dynamic profiling run with a certain input data set, basic block sequence o_{s_j} is executed for the last time at point in time t. The preselection algorithm decides to put the SDRAM instruction partition into deep power down mode following this execution of sequence o_{s_j}. Due to a different input set and data dependencies in the control flow of the application, the sequence o_{s_j} is executed again at a later point in time $t + x$ during the actual execution of the application. Executing it from the SDRAM is not possible anymore, since the instruction partition is already in DPD mode.

However, each memory object also has a copy in the Flash memory. In order to generate valid code, the compiler has to ensure that only basic blocks in the Flash memory will ever be executed once the SDRAM instruction partition has been put to DPD. This is possible by performing the corresponding address transformations for all basic blocks in set X following the preselection algorithm. The presence in set X implies that this basic block is never copied to the SDRAM partition and is only executed from the Flash memory when the instruction partition has been put to DPD. In this way, if data dependencies cause a different behavior than expected by the preselection algorithm, correct, albeit not optimal code will result if the basic blocks in set X are modified accordingly.

The following example, using the sequence of basic blocks presented in Figure 5.11, shows how the algorithm selects basic blocks in detail. Figure 5.12 illustrates the steps of the algorithm.

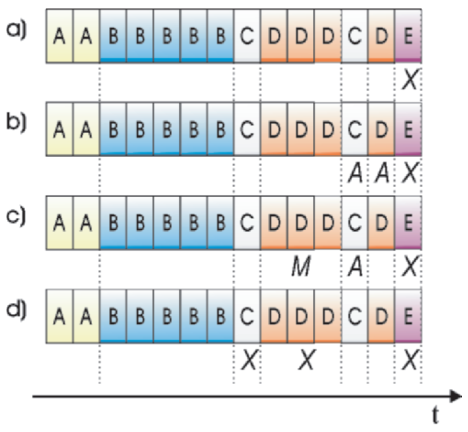

a)
A A B B B B B C D D D C D E
X
b)
A A B B B B B C D D D C D E
A A X
c)
A A B B B B B C D D D C D E
M A X
d)
A A B B B B B C D D D C D E
X X X
t

Fig. 5.12. Steps of the preselection algorithm

The algorithm starts at the end of the basic block sequence and first visits basic block “E”. This object is only executed once and thus has no executions pending at an earlier point in time. Since the object is only executed once, it is beneficial to execute this basic block from the Flash memory instead of first copying and then executing it from the SDRAM. It is therefore inserted into set X (Figure 5.12 a). After this decision has been taken, the benefit value is again initialized to “0”, and the algorithm assumes the SDRAM partition to be put into DPD mode before basic block “E” is being executed.

The next memory objects are “D” and “C”, both of which have executions pending earlier in the sequence. Therefore, these basic blocks are inserted into set A (Figure 5.12 b).

Subsequently, the second occurrence of basic block “D” is found. Since it has no further executions at an earlier point in time, it is removed from the set A and its energy dissipation both in SDRAM and in Flash are estimated using the equations presented above. In this example, we assume that since “D” is executed frequently, it is better executed from the faster SDRAM than from the Flash memory. Therefore, object “D” is inserted into the set M to mark that it may be executed from the Flash memory if this is later determined to be beneficial (Figure 5.12 c). At this point in the preselection algorithm, the overall benefit value is negative.

Finally, the last execution of basic block “C” is found. The benefit value from executing this object from the Flash memory is assumed to be so high that the overall benefit value for objects “C” and “D” being executed from the Flash memory is positive. Therefore, “C” is inserted into set X along with all potential candidates in set M (Figure 5.12 d). Once again, the benefit value is reinitialized and the algorithm continues to look for objects that should be executed from the Flash memory.

In this example, when the algorithm has reached the first basic block “A”, we assume a negative benefit value. Therefore, no new elements are inserted into the final Flash execution set X, and the result of the algorithm is that basic blocks “C”, “D” and “E” will remain in the Flash memory, and the SDRAM instruction partition will be put into DPD mode once object “C” is executed for the first time.

5.3.5 Formalization of the Preselection Algorithm

If a new sequence of basic blocks o_{s_j} is found that may potentially be executed from the Flash memory, it is inserted into set M and the energy benefit is adjusted by the difference between the SDRAM execution and the Flash execution. The benefit is thus calculated as:

$$E^+ = E^+ + \left(E^{DPD}_{SDRAM}(o_{s_j}) - E^{DPD}_{FLASH}(o_{s_j}) \right) \tag{5.40}$$

Since the SDRAM memory is in general faster and consumes less energy compared to the Flash memory, a negative benefit value will usually be obtained following this step. This can be compensated by not having to copy memory objects from Flash to SDRAM memory, which will be considered in the equations below.

At the beginning of the algorithm and after an object has been allocated to the set X, the energy benefit E^+ is initialized to the value 0.

The total number of executions of a particular basic block has already been introduced as $\#exec(o_i)$. The remaining number of executions (looking from the end of the execution to the beginning of the execution, i.e. in the direction of the preselection algorithm) is described by $\#exec^*(o_i)$. When the basic block sequence s_j is being considered by the algorithm, the number of executions of basic block o_i is reduced by the number of executions in the current sequence if $o_{s_j} = o_i$:

$$\#exec^*(o_i) = \#exec^*(o_i) - \#exec(o_{s_j}) \tag{5.41}$$

The algorithm can then determine pending executions of this basic block by checking whether $\#exec^*(o_i)$ is equal to 0. Depending on the outcome of this comparison, the algorithm proceeds:

- $\#exec^*(o_i) > 0$:
  Object o_i is inserted into set A. The algorithm continues with the subsequent basic block sequence s_{j+1}.
- $\#exec^*(o_i) = 0$:
  Since no executions of basic block o_i are pending, it is removed from set A (if it was previously included in set A). After this operation, the algorithm checks whether set A is the empty set:
  - $A \neq \emptyset$: The currently considered object o_i is inserted into set M.
  - $A = \emptyset$: The copy energy that can be saved by executing the currently considered object o_i and all remaining objects in set M from the Flash memory is added to the current energy benefit E^+. The sign of the benefit is then checked:
    - $E^+ + E_{COPY}(M) + E_{COPY}(o_i) \geq 0$: The advantage of not having to copy the objects in set M to the SDRAM memory outweighs the increased time and energy required to execute these objects from the Flash memory. Thus, all objects in set M and the currently considered object o_i are inserted into set X and are executed from the Flash memory. Following this step, the Energy benefit E^+ is reinitialized with the value 0, set M is the empty set. The algorithm continues with the subsequent basic block sequence s_{j+1}.
    - $E^+ + E_{COPY}(M) + E_{COPY}(o_i) < 0$: Proceed as in the case of $A \neq \emptyset$. The algorithm continues with the subsequent basic block sequence s_{j+1}.

The basic blocks in set X will be considered to have an energy dissipation of 0 when they are executed from the Flash memory in the subsequent optimization. This guarantees that they will be allocated to the Flash memory because the preselection algorithm determined that executing these objects from the Flash memory and keeping the SDRAM instruction partition in the DPD mode results in a higher benefit than what could be achieved by allocating them according to the XIP optimization which only assumes the SDRAM to be in power down mode.

5.3.6 Formalization of the XIP Allocation Problem

Considering the costs of executing basic blocks from the Flash or from the SDRAM as presented above, a minimization problem is formulated in ILP notation. The set of memory objects considered in this optimization is only formed by the r basic blocks of the application. Global variables are not considered at all, since they are always allocated to the data partition in SDRAM technology.

The energy costs for executing a basic block o_i from the SDRAM partition or from the Flash are determined as

$$\begin{aligned} E_{SDRAM}(o_i) = \#exec(o_i) \cdot \Big(& E_{SDRAM\_LEAKAGE}(o_i) + \\ & E_{SDRAM\_ACC}(o_i) + E_{SDRAM\_CPU}(o_i) \Big) + \\ & E_{COPY}(o_i) \end{aligned} \tag{5.42}$$

$$\begin{aligned} E_{FLASH}(o_i) = \#exec(o_i) \cdot \Big(& E_{FLASH\_LEAKAGE}(o_i) + \\ & E_{FLASH\_ACC}(o_i) + E_{FLASH\_CPU}(o_i) \Big) \end{aligned} \tag{5.43}$$

The energy values of those basic blocks that have been allocated to the Flash memory by the preselection algorithm are set to 0 for the Flash memory. This ensures that they will always be allocated to the Flash memory, since this causes no additional costs.

Additional adjustments of the energy dissipation are necessary since it is assumed in the model that for every successor of a basic block, a long jump to a different memory partition is required (cf. Section 4.2.3 for a detailed description). Since the long jump is the last instruction in a basic block, it is assumed that the long jump may be read sequentially both from the SDRAM and the Flash memory. As in the Bottom-Up model, the edges of the control flow graph are also considered in order to guarantee that the long jump energy is only considered where long jumps are actually required.

Two different effects have to be considered: if the control flow between two basic blocks stays on the same memory partition (i.e. either the SDRAM or the Flash memory), then the execution of an additional longjump is not required, meaning that the energy assumed for these longjump instructions can be subtracted. Depending on the memory partition, the energy required to execute one longjump instruction is defined as E_{SDRAM_JUMP} or E_{FLASH_JUMP}.

Additionally, when two connected basic blocks are both allocated to the Flash partition, then the additional energy benefit E^{+}_{OH} can also be subtracted from the energy equation, since the SDRAM instruction partition can remain in power down mode in this case (cf. Section 5.2.3 for a more detailed description). To model these refinements of the energy model, the variables $e(o_i, o_k)$ are required which contain the information how many times control flow passes along the edge that connects basic blocks o_i and o_k.

Decision Variables

Two decision variables are introduced for every memory object o_i to determine whether it is allocated to the SDRAM or the Flash memory:

$$\tilde{o}_{i,SDRAM} = \begin{cases} 1, & \text{if object } o_i \text{ is allocated to } SDRAM \\ 0, & \text{otherwise} \end{cases} \tag{5.44}$$

$$\tilde{o}_{i,Flash} = \begin{cases} 1, & \text{if object } o_i \text{ is allocated to } Flash \\ 0, & \text{otherwise} \end{cases} \tag{5.45}$$

To model the control flow along the edges as mentioned above, an additional set of decision variables is introduced. The variables are assigned the value 1 if the two basic blocks connected by the corresponding edge are allocated to the same memory partition:

$$x_{i,k,SDRAM} = \begin{cases} 1, & \text{if } o_i \text{ and } o_k \text{ are connected by an edge} \\ & \quad \text{and both allocated to } SDRAM \\ 0, & \text{otherwise} \end{cases} \tag{5.46}$$

$$x_{i,k,Flash} = \begin{cases} 1, & \text{if } o_i \text{ and } o_k \text{ are connected by an edge} \\ & \quad \text{and both allocated to } Flash \\ 0, & \text{otherwise} \end{cases} \tag{5.47}$$

Constraints

Two sorts of constraints have to be satisfied: on one hand, each memory object must be allocated either to the SDRAM or to the Flash memory.

$$\forall i, 1 \leq i \leq r : \tilde{o}_{i,SDRAM} + \tilde{o}_{i,Flash} = 1 \tag{5.48}$$

On the other hand, the decision variables for the edges may only be set to 1 if the corresponding object decision variables are set accordingly.

$$\begin{aligned} \forall i, k : 1 \leq i \leq r, 1 \leq k \leq r \ &: \\ \tilde{o}_{i,SDRAM} + \tilde{o}_{k,SDRAM} - 2 \cdot x_{i,k,SDRAM} &\geq 0 \\ \tilde{o}_{i,Flash} + \tilde{o}_{k,Flash} - 2 \cdot x_{i,k,Flash} &\geq 0 \end{aligned} \tag{5.49}$$

Note that considering the memory partition capacities is not required for this particular optimization, since in contrast to the small scratchpad memories considered in the previous sections, both Flash and SDRAM main memory are assumed to be sufficiently large to hold all basic blocks.

Objective Function

Finally, the objective function that is to be minimized by the ILP solver is presented:

$$\begin{aligned}
&\text{Minimize} \qquad (5.50)\\
&\sum_{i=1}^{r}(E_{SDRAM}(o_i)\cdot \tilde{o}_{i,SDRAM} + E_{Flash}(o_i)\cdot \tilde{o}_{i,Flash}) -\\
&\sum_{i=1}^{r}\sum_{o_k\in Succ(o_i)} e(o_i,o_k)\cdot E^{+}_{OH}\cdot x_{i,k,Flash} -\\
&\sum_{i=1}^{r}\sum_{o_k\in Succ(o_i)} e(o_i,o_k)\cdot (E_{SDRAM\_JUMP}\cdot x_{i,k,SDRAM} +\\
&\qquad\qquad E_{Flash\_JUMP}\cdot x_{i,k,Flash})
\end{aligned}$$

The first line of the objective function determines the energy contribution of basic blocks if they are executed from the Flash or from the SDRAM instruction partition. The following line subtracts the overhead energy E^{+}_{OH} required to wake up the SDRAM instruction partition if the SDRAM can remain in the power down mode. This is the case when basic blocks o_i and o_k are connected by an edge, such that o_k is the successor of o_i, and if they are both allocated to the Flash memory. The last two lines subtract the energy to execute the longjump instructions if they are not necessary due to two adjacent basic blocks being assigned to the same memory partition.

With $|E|$ being the number of edges in the control flow graph (including the “dummy” edges introduced to model the relationship of basic blocks and accessed variables), and r being the number of basic blocks, the size of the described ILP problem can be determined as

$$\#Constraints(\text{XIP}) = r + 2\cdot|E| \qquad (5.51)$$

$$\#DecisionVariables(\text{XIP}) = 2r + 2\cdot|E| \qquad (5.52)$$

5.3.7 Results for XIP

To evaluate and compare the results obtained using the optimization algorithm presented in the previous section, three experiments were performed for each benchmark application. First, only SDRAM main memory was assumed to be present in the system, and the entire code section of the application was copied to this SDRAM partition. The second experiment used only the Flash memory partition to execute instructions using the XIP functionality. Finally, the XIP optimization technique was used to allocate instructions to either the Flash or the SDRAM partition to maximize the determined benefit. Beside the different timing and energy properties of Flash and SDRAM Memory, the optimization takes into account the fact that instructions executed from the Flash partition do not have to be copied to the SDRAM instruction partition first. Furthermore, when instructions are executed from the Flash memory, the SDRAM main memory can be put in power down mode to save additional energy. If no more instructions are to be executed from the SDRAM instruction partition, it can even be switched off completely and be put into deep power down mode, which causes the main memory to lose all of its contents. Since in all the considered setups, data elements are always stored in a separate SDRAM data partition, the loss of instructions in the main memory does not imply a loss of data. For this reason, the deep power down mode can be exploited in this configuration.

For the Flash memory, the data sheet of a Micron Q-Flash MT28F640J3 chip was used [Mic04b]. Results were generated using the Flash energy and timing model presented in Sections 3.4.3 and 3.3.2, respectively, and the data sheet values shown in Table 5.5

Designation	Unit	Symbol	Value
Supply voltage	V	VDD	3.3
Read async. access time	ns	tAA	100
Read intrapage access time	ns	tAPA	20 / 10
Read burst access time	ns	tCLK	-
Read async. access current	mA	IDD1	9
Read intrapage access current	mA	IDD2	8
Read burst access current	mA	IDD3	-

Table 5.5. Micron QFlash characteristics [Mic04b]

The used main SDRAM memory consists of two 64 Mbit Mobile SDRAM chips [Mic04c] for the data and instruction partition. The corresponding data sheet values are provided in Table 5.6.

Value name	Symbol	Current $[mA]$
active operating current	I_{DD1}	50.0
power-down standby current	I_{DD2}	0.15
active standby current	I_{DD3}	35.0
operating current	I_{DD4}	80.0
auto refresh current	I_{DD5}	2.0

Table 5.6. Data sheet values used for the XIP experiments [Mic04c]

The benchmarks used to generate results are shown in Table 5.7, including information concerning code and data size. Detailed information is generated during a dynamic profiling run of the applications. As described in Section 3.6.2, the generated values include execution and access counts which are then used to generate an instance of the proposed ILP problem. Both the size and the execution frequency of code objects have an influence on their treatment by the allocation algorithm: objects that are large in size and are not frequently executed have the highest probability of being allocated to the Flash memory and executed using the XIP functionality. In contrast to that, small and frequently executed basic blocks will be copied to the faster SDRAM memory, since for them, the benefit achieved by the faster access times overcompensates the copy costs for these objects. In addition, the allocation algorithm also takes the potential for burst accesses into account, which may restrict the benefit of allocating very small single basic blocks to the SDRAM since they do not provide sufficient potential for beneficial burst accesses. As shown in Figure 5.9 in the previous section, considering burst accesses can lead to significant savings of up to 50%, depending on the control flow of the application under consideration.

Benchmark	Code Size [*bytes*]	Data Size [*bytes*]	Description
MPEG	15564	32032	MPEG2 decoding algorithm
ME	796	4	Media application using intensive integer arithmetic
ADPCM	724	6928	Encoder and decoder using Adaptive Differential Pulse Code Modulation
FIR	136	2732	Finite Infinite Response Filter Application
Fast_IDCT	1428	6552	Integer implementation of inverse discrete cosine transform (IDCT)
Multi_Sort	716	1204	Sorting benchmark (combining several sorting algorithms)
G.721	2784	2424	Encoding and decoding according to G.721 using “Adaptive differential Pulse Code Modulation”

Table 5.7. Selected benchmarks for XIP optimization

The experimental setup consists of a Flash memory partition with XIP capabilities and two SDRAM memory partitions, as shown in Figure 5.10. The partitioning of the main SDRAM was introduced for several reasons: writable data has to be allocated to an SDRAM memory due to the limited number of write cycles for Flash memory. However, data elements accessed in a large SDRAM partition can effectively prevent the memory from ever transitioning to a power down mode. By moving the data to a partition of its own, the instruction partition can be put into a low power mode while instructions are being executed from the Flash partition. Additionally, storing data on a separate SDRAM partition allows the SDRAM instruction partition to be put to deep power down mode when no more instructions are to be executed from it. For the initial experiments, the intrapage access times of the Flash memory were set to $20ns$, which represents a relatively slow memory. Results show that the achievable gains for this setting are not too high. By making the Flash memory faster, assuming $t_{APA} = 10ns$, it becomes more attractive with respect to overall energy dissipation and the savings by including an XIP-capable Flash memory in the system are increased.

Figure 5.13 shows the obtained results for the MPEG decoding benchmark. The left triplet of bars shows the results for energy dissipation, execution time and SDRAM instruction memory requirements in the case that only the SDRAM main memory is used to hold both instructions and data. The middle triplet of bars represents the results obtained by the proposed optimization which allocates memory objects to the SDRAM main memory or to the Flash in an appropriate way. Since this is expected to be the best configuration in all cases, all results were scaled such that the optimization results are at 100%. Since the optimization result bars always have a height of 100%, they carry no useful information and are thus omitted from the further figures. Finally, the right triplet of bars shows the results if all instructions are solely allocated to the Flash memory partition and executed from there.

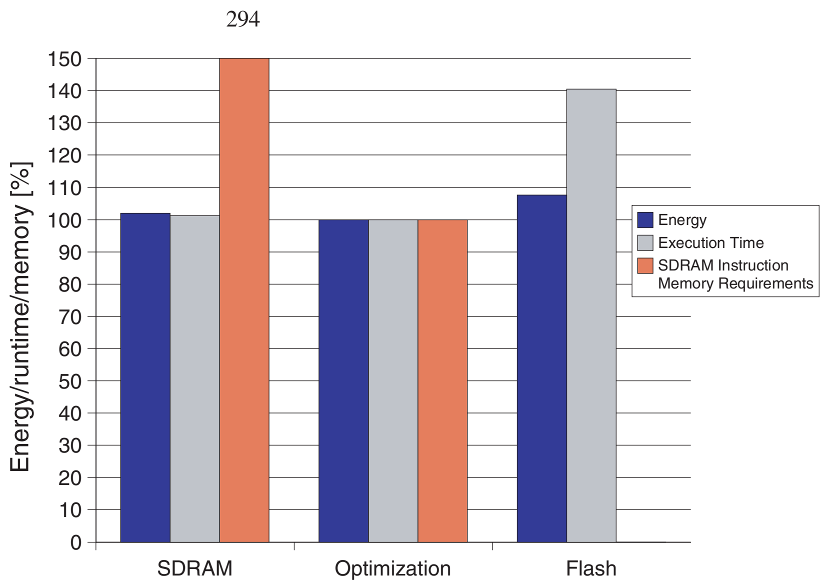
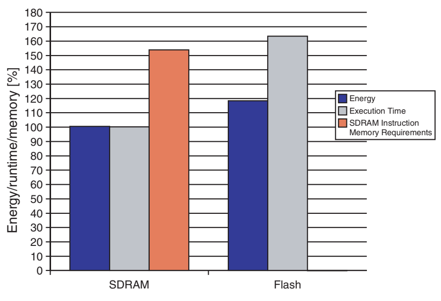


Fig. 5.13. SDRAM vs. Flash optimization vs. XIP for MPEG, $t_{APA} = 20ns$

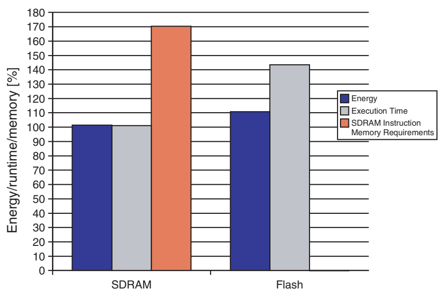
For the MPEG benchmark, the energy required for a pure SDRAM execution is slightly higher than the results obtained using the optimization. The same is also true for the execution time. Since this benchmark application has a high number of basic blocks and most of them are not executed frequently, more than two thirds of the total 70 functions are allocated to the Flash partition by the optimization, leading to savings in SDRAM capacity requirements of 66% when only XIP from the Flash memory is being used. Assuming that the Flash memory always has to have the capacity to accommodate the entire application in a non-volatile memory, the optimized version requires less than 4 kB of additional SDRAM space, whereas the entire program would require nearly three times that amount if all instructions were executed from the SDRAM.

For the version that only utilizes Flash memory to execute instructions, the energy dissipation is 6% above that of the optimized version. This is mainly due to the slower Flash memory accesses which result in the fact that the execution time is prolonged by nearly 40%. The rightmost bar in the figure has a height of zero: If all instructions are executed from the Flash, no SDRAM instruction partition is required.

Summarizing the results for this first benchmark, the optimization is capable of finding a good compromise between Flash and SDRAM execution of instructions. While for the pure Flash execution, the execution time is longer and consequently the energy dissipation higher, the optimized version that only allocates those instructions to the Flash memory that result in maximum energy savings is capable of reducing the overhead introduced by the slower Flash memory to zero. The optimized version even slightly outperforms the SDRAM-only allocation due to saved copy costs and power down/deep power down times of the SDRAM instruction partition. The most beneficial effect of utilizing the optimization algorithm however lies in the fact that only one third of the SDRAM main memory is required when the XIP functionality of the Flash memory is exploited. Since the memory requirements are increasingly becoming a main cost factor for embedded systems, it is mandatory to also exploit the Flash memories' XIP capabilities. The presented optimization is capable of providing a suitable allocation of instructions to both XIP-Flash and to SDRAM instruction memories.



a)



b)

Fig. 5.14. Flash memory XIP optimization for a) ME, b) ADPCM, $t_{APA} = 20ns$

The benchmarks ME, ADPCM and FIR, shown in Figures 5.14 and 5.15 a) show a similar behavior and are therefore discussed together. They all consist of a smaller number of basic blocks compared to the MPEG application. For all these benchmarks, the gains obtained concerning performance and energy dissipation compared to a pure SDRAM execution were found to be marginal. However, on average two thirds of the SDRAM capacity could be saved when the Flash memory is also utilized, which is significant for cost-sensitive embedded systems.

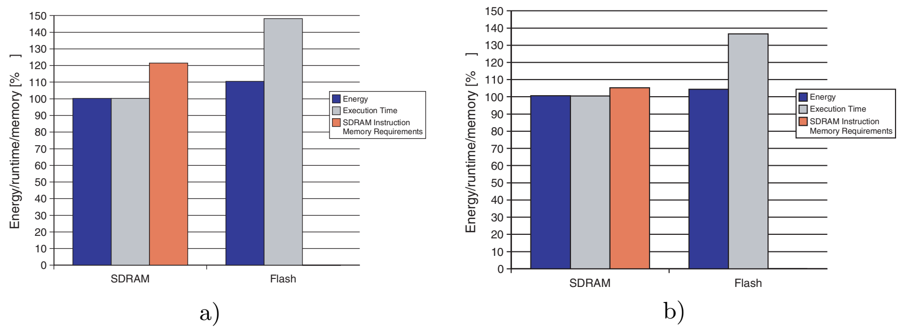


Fig. 5.15. Flash memory XIP optimization for a) FIR, b) Fast_IDCT, $t_{APA} = 20ns$

Compared to the Flash-only execution of instructions, the execution times are reduced by 30% to 40% using the optimization. This is due to the fact that many instruction accesses for the considered benchmarks can be performed using fast sequential burst accesses when executed from the SDRAM. An intrapage accesses from the Flash memory requires one additional cycle compared to burst accesses from the SDRAM. The energy gain of round about 10% is also mostly due to this reduced execution time.

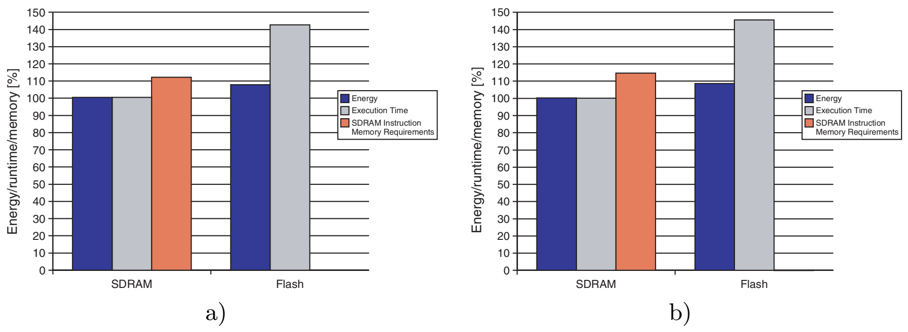


Fig. 5.16. Flash memory XIP optimization for a) Multi_Sort, b) G.721, $t_{APA} = 20ns$

For the remaining benchmarks, namely Fast_IDCT, shown in Figure 5.15 b), Multi_Sort and G.721 (Figure 5.16), only marginal improvements could be achieved compared to a pure SDRAM execution, however a potential to save up to 15% SDRAM capacity is present in these benchmarks. Compared to a pure XIP-execution from the Flash memory partition, the execution times were reduced by nearly one third, which is due to the faster access times of the SDRAM compared to the Flash memory. Energy was reduced by 6%, also mainly due to the saved execution cycles.

For the second set of experiments, the intrapage access times of the used Flash memory were assumed to be reduced to $10ns$ instead of $20ns$ as before. This is expected to lead to make the Flash more attractive and to have the optimization execute more instructions using XIP. The intrapage access time is the only parameter that was varied between the two sets of experiments, but the effects are clearly visible: for the two benchmarks shown in Figure 5.17, G.721 and FIR, the optimization allocates all instructions to the Flash memory and does not utilize the SDRAM partition at all. This leads to significantly longer execution times, but identical energy values compared to the SDRAM execution. This is due to the fact that the cost function used by the optimization algorithm only optimizes for minimal energy dissipation. Since no SDRAM is being used in the optimized version, no percentage can be supplied for the SDRAM-only execution of instructions.

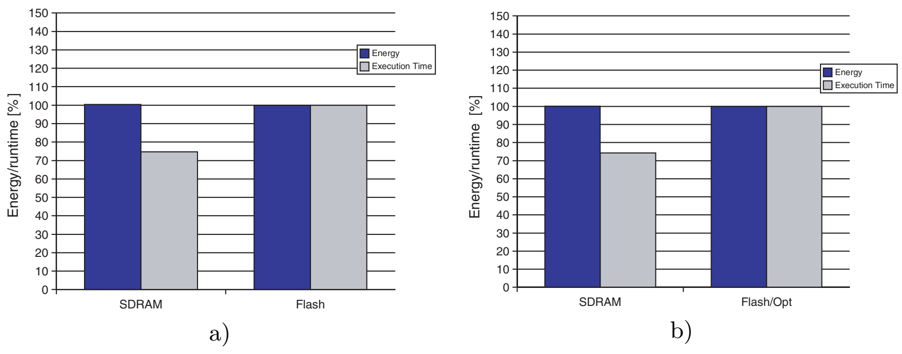
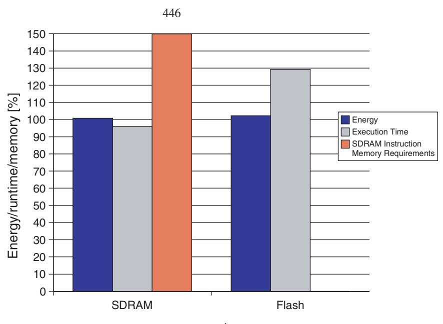
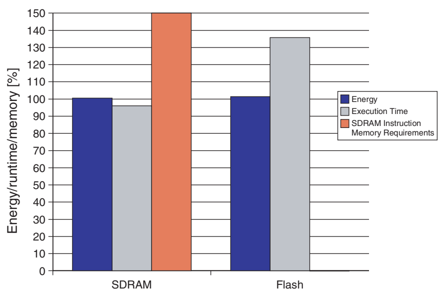


Fig. 5.17. Flash memory XIP optimization for a) G.721, b) FIR, $t_{APA} = 10ns$

The benchmarks Multi_Sort, ME (cf. Figure 5.18) and MPEG (Figure 5.19) are discussed together since they show similar results. In all of these benchmarks, the energy dissipation of the optimized version is slightly lower compared to both the pure Flash and the pure SDRAM execution. Despite the fact that a fast Flash memory with an intrapage access time of $10ns$ was used, the execution times of the optimized version still outperforms the pure Flash version by 30% on average. Compared to the SDRAM execution, the execution time is increased by only 4%, which is acceptable taking into account

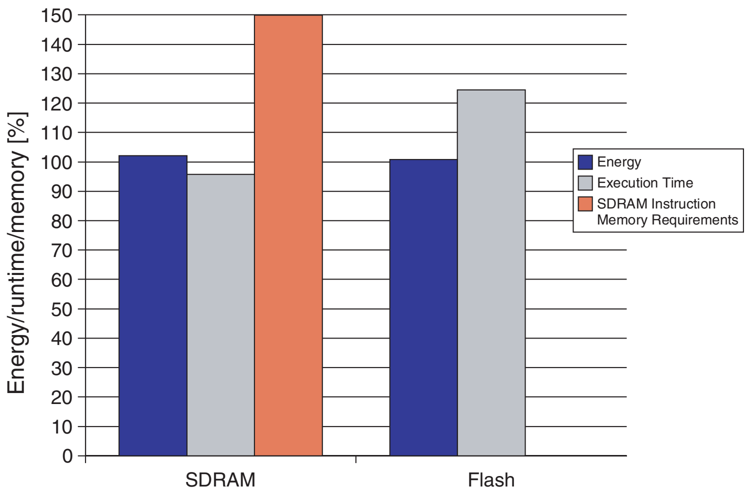


a)



b)

Fig. 5.18. Flash memory XIP optimization for a) Multi_Sort, b) ME, $t_{APA} = 10ns$



b)

Fig. 5.19. Flash memory XIP optimization for MPEG, $t_{APA} = 10ns$

the considerable amount of SDRAM capacity that can be saved if part of the program is executed from the Flash memory.

For the Fast_IDCT benchmark in Figure 5.20 a), nearly the entire program is executed from the Flash memory. Only 4% of the total program size is allocated to the SDRAM memory, leading to a percentually large decrease in the main memory requirements. Surprisingly, the energy dissipation could be reduced by 4% compared to a pure SDRAM execution, caused by the reduction of the initial copy costs from Flash to SDRAM memory. The optimization for

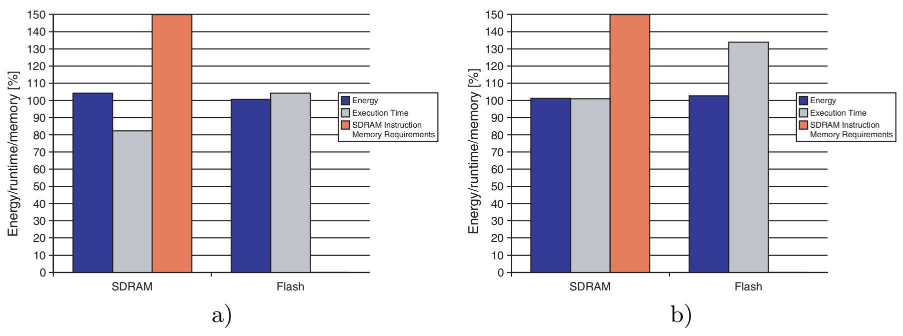


Fig. 5.20. Flash memory XIP optimization for a) Fast_IDCT, b) ADPCM, $t_{APA} = 10ns$

energy results in an execution time that is prolonged by 18% compared to SDRAM, but shorter by 4% compared to the Flash execution.

The ADPCM benchmark in Figure 5.20 b) is another benchmark where the copy costs have a relevant impact on the outcome of the optimization. This benchmark's execution time is shortest for the optimized version. In contrast to the other benchmarks where the slower Flash memory causes prolonged execution times compared to the pure SDRAM execution, ADPCM is relatively small and the execution counts of basic blocks are such that the copy costs are actually the dominating part of the overall execution time. Since these copy costs can be reduced by using XIP, the overall execution time can be reduced even compared to the faster SDRAM.

The influence of the basic blocks' execution counts on the allocation algorithm was further examined using the digital signal processing benchmark biquad_N_sections, a recursive filter application. By modifying the parameter N, it is possible to simulate a higher order filter and thus to increase the execution count of the basic blocks within the core of the filter routine. The results shown in Figure 5.21 were generated using a Flash memory with an intrapage access timing of $20ns$.

For small execution counts (N = 4), the copy costs are responsible for a high percentage of the execution time and the overall energy costs, whereas the execution of the application from the Flash memory has the same energy dissipation and execution time as the optimized version. This is also true for N = 25, however the energy costs caused by copying are compensated by the more frequent execution from the fast SDRAM. The same is true for the execution time: when using SDRAM, it is slightly below the results of the optimization, which still corresponds to executing the entire application from the Flash memory. Since no SDRAM memory is used by the optimization in these cases, the SDRAM bars are not shown in the figure.

For N = 100, both SDRAM and Flash execution are outperformed by the optimized version, which is capable of finding an allocation of memory

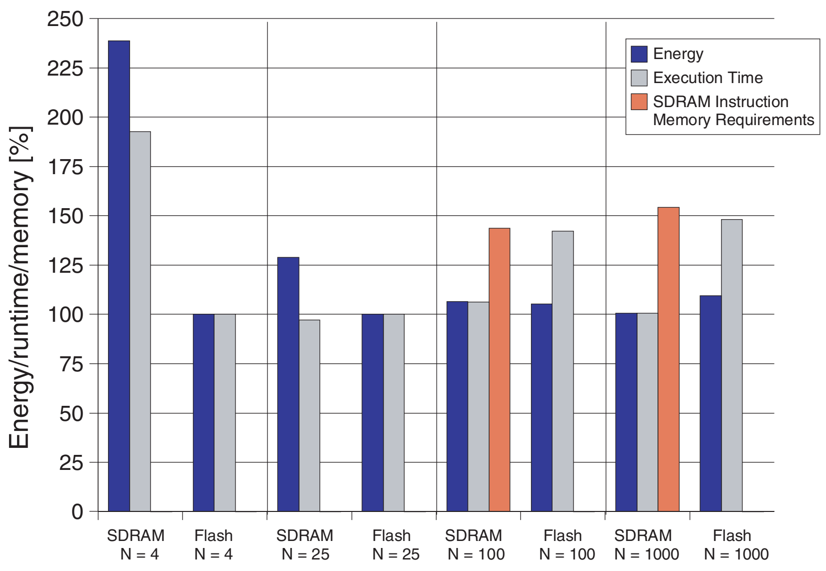


Fig. 5.21. Effect of execution counts on XIP optimization results for biquad_N_sections, $t_{APA} = 20ns$

objects to the Flash and to the SDRAM memory to minimize the energy dissipation. For the final results using N=1000, the slower access times of the Flash memory cause the pure Flash allocation strategy to consume more time and energy.

In summary, the presented optimization that targets the XIP capabilities of Flash memories built in NOR technology selects a suitable set of instruction memory objects that should be executed directly from the Flash memory instead of copying them to the SDRAM main memory at startup. In general, large basic blocks with few executions are likely to stay on the Flash memory, whereas basic blocks that are frequently executed can overcompensate the initial copy costs by being fetched from the faster and more energy efficient SDRAM instruction partition. As one main result of the optimization, utilizing the Flash memory's XIP capabilities can help to significantly reduce the main memory requirements of the applications. Since memory is one of the factors that determines the costs of a device, this result should encourage designers to take full advantage of the Flash memories present in most designs by considering the XIP option.

6 Register File Optimization

In the previous chapters, the memory hierarchy was shown to be responsible for a large amount of the overall energy consumed within an embedded computing system. Having looked at the use of scratchpad memories as an alternative to caches and at the energy saving potential offered by modern main memory technologies, this chapter considers the part of the memory hierarchy that is closest to the processor: the register file. It is essential for an efficient operation of the processor that the use of processor registers is as effective as possible. Especially in RISC architectures, all operations can usually only be performed on values stored within the register file. Only a couple of special operations (e.g. Load, Store, Push, Pop) are capable of accessing the memory at all.

Since all data in RISC architectures has to be made available in the register file in order to be processed, the register file represents an area of the processor with extremely high switching activity, and consequently a high amount of energy dissipation. For Motorola's M*Core architecture, 16% of the total processor power and as much as 42% of the data path power is consumed within the register file [SLAM98], which shows the significance of considering the register file during the design of an energy conscious system. In particular for embedded systems, the intended group of possible applications is known a priori. Therefore, typical use cases and scenarios can be used during the design of the system to determine the requirements concerning the register file.

The following section presents some related work dealing with the special properties of the register file. After possible physical implementations of the register file, a short overview over register allocation and lifetime analysis of variables within the compiler is provided. Following that, results are presented that show how the size of the register file influences code size, performance and energy. The behavior of one example benchmark application is studied in-depth. The conclusions drawn from this process are used to generate information within the compiler which helps the designer of an embedded system take decisions on the appropriate size of the register file which leads to an energy- and performance-optimized design.

The results in this chapter were presented for the first time in [WJS$^+$01]. This chapter gives an overview over the mentioned journal paper and extends the idea by providing a compiler guided estimation of the required register file size.

6.1 Related Work

The properties of the register file are important parameters for the design of embedded systems. In particular the size of the register file has a strong impact on the energy dissipation of the system. The fact that the compiler is responsible to generate code that efficiently exploits the available registers makes these considerations interesting in the context of compiler controlled energy savings. If too few registers are available in the architecture of the embedded system, the compiler has to insert so-called spill code to store values that are required at a later point in time in the main memory instead of the register file. This can lead to a severe loss of quality of the generated code, since additional instructions make the executable larger, the spill instructions increase the number of instructions that are executed without contributing to the actual results, and the additional accesses to main memory will cause both performance and energy penalties. Using a reconfigurable compiler, the quantitative impact of changes to the register file size can be studied.

Retargetable or reconfigurable compilers have been a topic of research for a long time, and they are even used in industry. However, due to the fact that the reconfigurability in general results in poor code quality, they are usually used only in the context of rapid prototyping. In the well-known Trimaran compiler [KGK99, Tri], the machine description language MDES is used to model the underlying hardware. Using this information, the compiler can generate code for that particular architecture.

A different approach was taken by [Leu97]: in this work, a compiler can be retargeted to different processors of the DSP domain described in the MIMOLA hardware description language [JM93]. It is possible to specify the architecture in the form of an executable specification and at the same time supply the necessary information for the compiler.

Another way of specifying the hardware architecture is followed by the LISA toolsuite, which allows the user to generate a simulator, a VHDL description to produce actual hardware, a compiler and other required tools using one single processor description supplied in the LISA language [CHB$^+$05]. Some additional information concerning the instruction set of the processor and the mapping of high-level constructs to assembly instructions is required in addition to the pure hardware description to allow the generation of compilation tools.

One of the vital aspects when designing a new architecture is the definition of the considered design space. A number of approaches to ASIP design where the design space consists of e.g. number and kind of functional units, issue width and the size of caches are summarized in a survey paper [JBK01]. Most of the mentioned approaches search the design space for an area-time tradeoff. The notion of energy dissipation is largely neglected in the reported strategies. Energy is only being considered on the basis of low level circuit energy models which are primarily non-application specific [KLMSP99, SRP$^+$95]. In contrast to that, this work presents results concerning both performance and energy when the size of the register file is modified. Additionally, information from the compiler can be used to automate the process of finding a suitable size for the register file size. The compiler guided selection of an appropriate register file size is a novel contribution of this work.

6.2 Implementation of the Register File

Since the register file is used to store information within the processor and is thus part of the memory hierarchy of a system, it can be implemented using the available standard building blocks that also make up memories. Since the register file is the part of the memory hierarchy that is closest to the processor, the requirements concerning throughput and efficiency are very high. In addition, the register file lies on the critical path for most CPU operations, since in RISC architectures, all operands are usually read from and written to the register file. Due to these special properties of the register file in the memory hierarchy, an efficient implementation is therefore mandatory in order to avoid a performance degradation of the system.

Since most arithmetic and logic operations require two operands, it is beneficial for the register file to have two read ports since this avoids an extra cycle to read the second operand. Since most ALU operations only generate one result value that has to be written back to a register, a single write port is usually sufficient for the register file of RISC architectures. For processors with a high degree of instruction level parallelism, the situation may be different. In this work, we only consider the RISC architecture as found e.g. in the ARM7 processor.

If the register file is configurable by the user, e.g. in an ASIP that can be tuned to the application that is to be executed, the register file is usually implemented using a multi port memory cell. This allows for an easy modification of the register file by simply instantiating a different memory within the processor. Since the efficiency and speed of the register file are vital, SRAM technology is usually chosen to implement registers. The advantages of DRAM (e.g. less space per bit, cf. Section 3.2.2) are usually more relevant to implement larger memories, where the complex surrounding logic does not dominate the total size of the memory. SRAM memory cells have the further advantage that they can be manufactured using the same technology as the transistors of the processor core, allowing the vendor to "drop in" the requested memory cell to implement the register file. This effectively helps to keep the production costs of the considered systems low.

6.3 Register Allocation and Lifetime Analysis

Register allocation is one of the tasks to be performed during code generation within the compiler [Muc97]. The task of register allocation is to determine a mapping of virtual registers to physical registers present in the processor. Virtual registers are used during code generation to represent data values and their dependencies. For every new value in an application's data flow graph, the compiler allocates a new virtual register. This means that the number of virtual registers used in a program can be very high. The large number of virtual registers then has to be mapped to the limited number of physical registers in the target hardware architecture.

It is essential for an acceptable performance of the application that the register allocation algorithm determines a suitable and efficient mapping of virtual to physical registers. If the lifetimes of two virtual registers are non-overlapping, i.e. they are never active or "live" at the same time, then these two virtual register values may be mapped to the same physical register. In this way, the reduced number of physical registers can be shared among different virtual register values. To determine which data values have non-overlapping lifetimes and can thus share a physical register, the register allocation algorithm first performs a lifetime analysis of the application program's data values.

The lifetime of a value starts when its value is being defined for the first time. It ends when the value is being read for the last time. Inbetween these two points, the value (or virtual register) is said to be "live". A simple example for lifetime analysis is shown in Figure 6.1. The left hand side of the figure shows the liveness information of the variables, starting from the top with a filled circle and ending with an unfilled circle. On the right hand side, the corresponding high-level source code is provided.

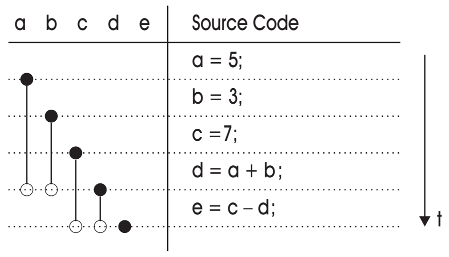


Fig. 6.1. Lifetime Analysis: an example

In this example, either the virtual register for variable a or b could share one physical register with d since they are not live at the same time: the value for d is generated after a and b have been read for the last time. Note that it is possible to map a and d to the same register despite the fact that d is generated by the same instruction that reads a. Since the operands have to be read and processed by the ALU before the results can be determined, this does not cause a problem. In the figure, the fact that register values a and b are not live after the addition operation is indicated by the unfilled circle.

In order to find a valid mapping of virtual to physical registers, the register allocation algorithm starts to determine the lifetimes of all values contained in virtual registers in a part of the program. There are different approaches that operate either on the basic block level or on entire functions [Muc97]. The lifetimes are used to generate a conflict graph which contains one node for every virtual register in the program. Two virtual registers are connected by an edge if their lifetimes overlap, i.e. if the values have conflicting lifetimes. The register allocation problem can thus be solved by assigning different registers to all nodes that are connected by an edge. This corresponds to the well-known graph coloring problem, which tries to assign a color to each node of a graph such that no two connected nodes have the same color. In the case of register allocation, each color corresponds to one physical register. Once a coloring for the entire conflict graph has been found, the register allocation problem is solved. Several other methods of performing register allocation are known [Muc97], however the graph coloring approach or related heuristics are the most commonly used register allocation algorithms.

For the example given above, the conflict graph along with a valid solution for the graph coloring is shown in Figure 6.2.

If the coloring of a conflict graph fails due to an insufficient number of colors (or physical registers), the lifetimes of variables have to be split in order to remove edges from the conflict graph. During code generation, this corresponds to a live data value being stored or "spilled" to the main memory in order to free the corresponding register. When the value is again required by the program, the value has to be read back from memory into a register. Since memory accesses are power and time consuming, spilling should be reduced to a minimum. It is therefore vital that the register allocation can allocate as many virtual registers to physical ones without requiring any additional spill code. If spilling can not be avoided, the algorithm has to determine one

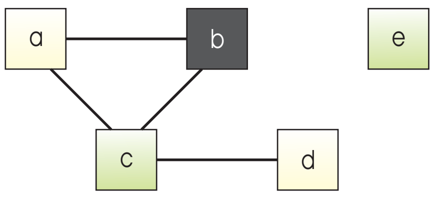


Fig. 6.2. Conflict Graph and Solution of Register Allocation

register to be spilled to memory that will cause the least spilling costs. This can be determined by considering the number of accesses to this value, or by analyzing the conflict graph to determine a candidate value that will cause the highest number of edges to be removed from the conflict graph.

Since register allocation in general is an NP-complete problem [Muc97], the encc compiler uses a heuristic algorithm to solve the graph coloring problem as presented in [App98]. The heuristic performs register allocation on the level of functions, not basic blocks. It assumes that all nodes in the conflict graph with a degree less than the number of available physical registers can always be colored. Once these candidate values have been assigned colors, the algorithm continues with the remaining nodes that have a higher degree. If it is not possible to color all nodes in this way, then one suitable node is chosen whose value is spilled to main memory. The data flow graph of the application is modified accordingly and the register allocation attempts to determine a valid coloring for the new conflict graph.

6.4 Workflow and Methodology

The results presented in this section exploit the fact that the encc compiler is parameterizable. By changing the configuration of the proposed target processor, it is possible to evaluate the effects of architectural changes on code quality by reconfiguring the compiler, generating code for the given number of registers and then simulating and analyzing the code. By comparing the code quality concerning code size, performance and energy consumption for different register file sizes, the designer can get a first idea of how many register should be present in the processor core to allow for an efficient execution of a program. From studying the source code of the example benchmarks and the profiling information retrieved from simulation, one common cause of performance degradation was identified. In Section 6.6, we will show how available information can be extracted from the compiler without having to generate code and perform simulation.

The workflow of the experiments is shown in Figure 6.3: Using the reconfigurability of the encc compiler, the number of registers available for the generated code is varied by changing the corresponding value in the compiler configuration file and rebuilding the compiler. This modified compiler is then used to compile the chosen benchmarks. The resulting assembly code is analyzed and executed using the instruction set simulator. Since only 16 bit THUMB instructions are generated by the encc compiler, the maximum number of registers had to be restricted to eight, since only three bits in the opcode are available to code the register number. The minimum number of registers that have to be available to generate valid and executable code for our set of applications was found to be three: a load operation from main memory that accesses a value within an array generally uses the register-register addressing mode since the offset in register-offset mode is limited to a maximum of 7 bits

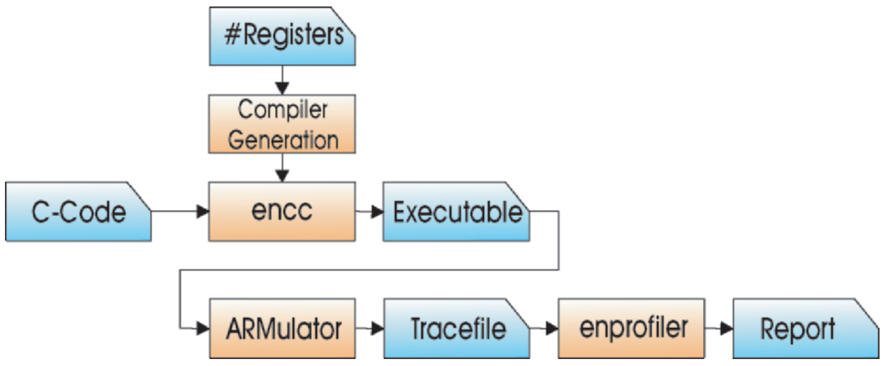


Fig. 6.3. Workflow for register file evaluation

in the THUMB instruction set. To load one value from an array, a minimum of two registers is thus required if the load operation overwrites the register containing the array's base address. If an operation with two operands is to be executed, a minimum of three registers is thus required to load the operand values into registers.

Reducing the number of registers in an architecture may severely degrade the achievable performance due to additional spill code that has to be inserted to store and retrieve values that can't be kept in the processor registers. However, there are also positive effects when the register file size is reduced: A smaller register file size will cause less chip area to be used and will also reduce power dissipation in the circuitry due to a reduction in the switched capacitance. If the instruction set architecture of the processor can be modified, using less registers generally results in shorter instruction words, since the operands in registers can be addressed with fewer bits. As an alternative to making the instruction words shorter, some processors take advantage of the unused patterns to code other useful information [KPL99].

These possible positive effects of reducing the size of the register file are difficult to evaluate without a complete hardware evaluation toolchain. Therefore, this work solely considers the effect of changes to the register file size on the compiler generated code for the ARM processor architecture.

The evaluation board that forms the basis of the hardware used for the experiments is described in-depth in Section 3.1. The evaluation board carries an ARM7TDMI processor with a small 4 KB scratchpad memory. Since program code and data together do not fit onto the scratchpad memory, the chosen setup consists of keeping the global data elements in the main memory while instructions are allocated to the scratchpad memory. In this way, instruction fetches will only account for a small part of the overall memory energy, and it can be expected that the effect of additional data accesses due to spilling will be clearly visible, since main memory accesses consume a considerable amount of energy. If the scratchpad allocation algorithms presented in Chapter 4 were used instead, the allocation results may be different for every register file size, which would make it difficult to determine whether the observed effects stem from the register file size or the allocation algorithm.

6.5 Benchmark Suite

The benchmarks used to investigate the effect of changing the register file size are shown in Table 6.1. They cover the domains of digital signal processing and multimedia, along with standard sorting algorithms. Since the memory footprint of the applications is not relevant when the size of the register file is modified, the memory space occupied by instructions and data is not provided in this table. Note that all used benchmarks make use of arrays, so that a minimum of three registers is always required to generate valid code.

Benchmark Name	Domain
insertion_sort	standard sorting algorithm
lattice	Lattice filter application
ME	Media application
biquad_N_sections	Filter application
matrix-mult	Multiplication of two matrices

Table 6.1. Benchmarks used for the register file size experiments

The insertion_sort algorithms sorts a given array of integers, lattice determines the output of a lattice filter. The ME mediabench benchmark has already been used in the previous chapters. It consists mainly of arithmetic operations on integer data. The biquad_N_sections program, part of the DSP-kernel benchmark suite [ZVSM94], performs the filtering of input values through N biquad IIR sections. Finally, matrix_mult implements the multiplication of two 2D matrices.

Note that the ARM7TDMI does not feature a floating point unit. Since the use of the data types `float` or `double` would result in inefficient library calls (as shown using the reference implementation of the IDCT benchmark on Page 129), only data of type integer is used in the experiments.

6.5.1 Results for the Ratio of Spill Code to Total Code Size

When the number of available registers is decreased, the compiler may have to insert additional spill code into the application's executable. This addition of spill code first of all leads to an increase in the application's code size. The spilling instructions are made up of memory load and store operations. The absolute number of instructions related to spilling is counted during register allocation within the compiler. Since the absolute number of spilling related instructions is not very meaningful, the ratio of spill instructions to total code size is presented in Figure 6.4.

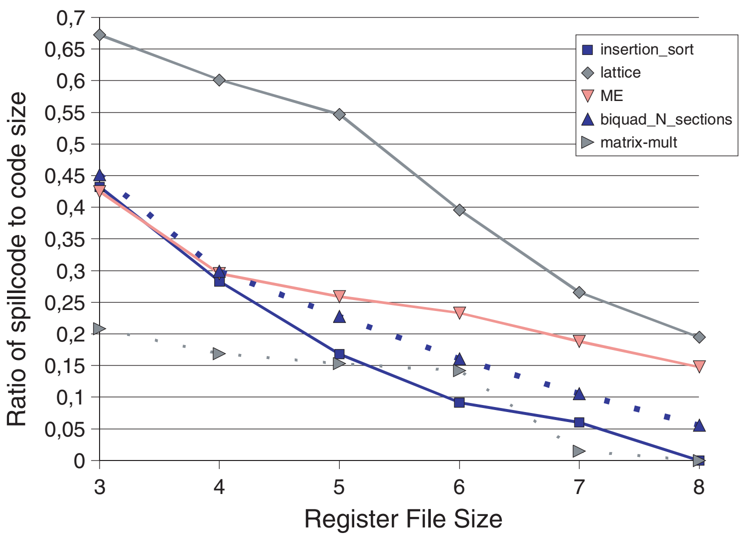


Fig. 6.4. Ratio of spill instructions to total number of instructions

As expected, the ratio of spill code increases strongly with decreasing number of registers. If e.g. only three registers are available, the code size for the lattice benchmark is increased by nearly 70%. On average, for a register file size of three, the code size is increased by about 44% compared to the theoretical code that could be obtained when no spilling occurs. For the maximum of eight registers, this overhead is reduced to only 8%, and for two of the considered benchmarks, no spill code at all is required. Note that the THUMB instruction set with its maximum register file size of eight registers frequently requires spill code to be added to the program, whereas the 14 freely usable registers of the 32 bit ARM mode are usually sufficient to generate code without spilling.

The 'saturation' point of the benchmark applications, i.e. the point where no more spill code is present in the assembly code, varies between seven and 19 registers. This could be determined using the compiler despite exceeding the number of possible registers, since the sole generation of assembly code is possible for an arbitrary number of registers. However, it is not possible to translate this assembly code to machine code due to the lack of bits in the instruction words to code the larger register numbers. Despite the fact that this code can not be translated into an executable, the value of 19 registers provides an upper bound for the register file size for this set of applications: if an ASIP designer wants to make sure that his application performance is not impaired by spill code at all, he can always determine this 'saturation' value for a particular application. This can either be achieved by simulation, as in this section, or the compiler can output information about the number of simultaneously live values, as we have done in Section 6.6.

This initial static analysis of the code generated for different register file sizes is useful when minimal code size, which is a static property of the code, is the target. Code size, or the memory footprint of an application is an important aspect in the development of embedded systems, since the memory is an important factor driving the cost of a system. In a setting where the designer has an influence on the register file, e.g. using an ASIP, it is therefore advisable to avoid code size increases due to spilling. Of course, the costs of adding another register to the register file also have to be taken into account. The designer thus has to determine a suitable trade-off between these two costs.

6.5.2 Results for the Number of Cycles

Beside the memory requirements, performance is another vital optimization goal for embedded systems. If performance is to be evaluated, a purely static analysis of the generated code is insufficient. Spill code introduced within the innermost loop of an application has a strong negative effect on the application's performance, whereas the performance loss may not even be noticed in code that is rarely executed.

To be able to estimate the influence of spill code on performance, the benchmark applications were executed using the ARM instruction set simulator after compilation. Using enprofiler, the cycle count of the applications was determined. The obtained results for the considered benchmarks are shown in Figure 6.5. To provide better comparability, the performance values for the different benchmarks were normalized such that the relative performance using the minimum of 3 registers corresponds to the value zero and the obtained performance improvement by adding registers is shown in percent.

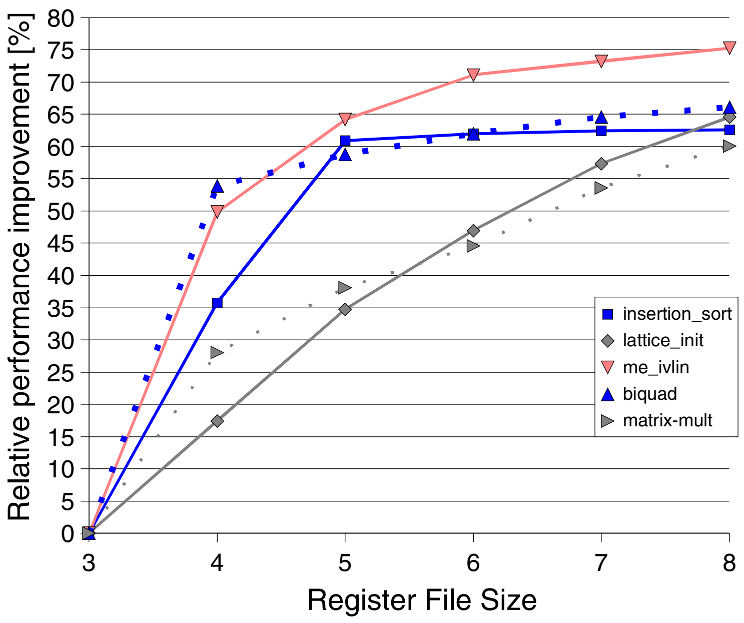


Fig. 6.5. Number of cycles over number of registers

As expected, the size of the register file has a strong impact on the number of executed cycles: for the ME application, the number of cycles could be reduced by 75% by providing eight registers instead of three. For all considered benchmarks, the performance is at least increased by 60%. In particular for the matrix multiplication and the lattice filter, the saved cycles depend in a nearly linear way on the number of registers for the observed range. For every added register, a performance improvement of on average 10% was obtained.

The impact of spill instructions on the performance is very strong due to the fact that the instructions used to access main memory require more than one cycle to execute on the ARM7: a store operation to the main memory takes two cycles, a load operation even requires three cycles (cf. Section 3.3.1). The waitstates of the used memory have to be added to these cycle counts. For the considered memory setup with 32 bit accesses to the main memory requiring three additional waitstates, a load operation takes a total of 6 cycles. The negative impact of spill instructions on performance is thus reinforced by the fact that particularly slow instructions are being added.

Some of the curves in figure 6.5 show a remarkably sharp bend, which is obvious in particular for the insertion_sort benchmark. When the number of registers is increased from four to five registers, the instruction count decreases substantially, whereas an increase from five to six registers hardly changes the number of executed cycles.

To understand the cause for this behavior, the innermost loop of the application has to be considered, since it is the part of the application that is executed most frequently and therefore has a strong influence on the dynamic behavior of the program. The innermost loop is shown in the original C source code and in the assembly code generated assuming eight available registers in Figures 6.6 a) and b), respectively.

For this small code fragment, it is possible to determine which C variable is stored in which physical register of the ARM7 processor by simply comparing the two code fragments. For example, `indx2` is stored in register `r3` since it is the result of the first subtraction in C as well as in the assembly code. `indx` is kept in register `r6`, but since this variable is not read again within the loop (it is only present in the loop initialization code), it is not live within the loop and can therefore be neglected. Continuing the analysis of the assembly code, the mapping shown in table 6.2 can be derived. The expression `(indx2-1)*4` is used to address the one-dimensional array `This[]` whose elements of type `int` require four bytes each.

Table 6.2 provides an explanation of the sharp bend in the curve for the insertion_sort benchmark: five registers (`r0`, `r2`, `r3`, `r4`, `r7`) are required to hold the values that are live within the innermost loop. Reducing the number of registers to four requires spill code to be inserted within the innermost loop, which inevitably leads to a severe performance degradation.

The shapes of the other applications' curves can be explained in a similar way, i.e. by analyzing the required number of live values within the innermost loop. To give one further example, the program biquad_N_sections contains

```
/* find the insertion point */
for (indx2 = indx - 1; indx2 > 0;)
{
  int temp_val = This[indx2 - 1];
  if (temp_val > cur_val)
  {
    This[indx2--] = temp_val;
  }
  else
    break;
}
```

a)

```
;;      for (indx2=indx-1;indx2>0;)
        SUB r3,r6,#1
        CMP r3,#0
        BLE LL8_0

LL12_0
;;      int temp_val=This[indx2-1];
        SUB r2,r3,#1
        LSL r2,r2,#2
        LDR r7, [r0, r2]

;;      if (temp_val>cur_val)
        CMP r7,r4
        BLE LL8_0

LL10_0
;;      This[indx2--]=temp_val;
        MOV r2,r3
        SUB r3,r3,#1
        LSL r2,r2,#2
        STR r7, [r0, r2]

;;      for (indx2=indx-1;indx2>0;)
        CMP r3,#0
        BGT LL12_0
LL8_0
   . . .
```

b)

Fig. 6.6. Inner loop of *insertion_sort*: a) C code, b) assembly code

Variable	Register
indx2	r3
indx2-1, (indx2-1)*4	r2
This[]	r0
temp_val	r7
cur_val	r4

Table 6.2. Mapping of variables to registers

two frequently executed `for`-loops, each of them containing a statement of the form `array[loop_counter] = value`. Each of these statements requires four registers to hold the simultaneously live values. Therefore, it is logical for this application to show a bend at the transition point from three to four registers (cf. Figure 6.5).

As a conclusion, it is possible to determine at least a lower bound for the register file size solely by analyzing the application code. Since this procedure is rather tedious for large applications, Section 6.6 will show how this can be automated using information the compiler collects during liveness analysis and register allocation.

6.5.3 Results for Energy Consumption

Beside performance in terms of number of cycles, energy is one of the prime optimization goals for embedded systems. Using the energy model for the ARM7 processor presented in Section 3.4.2 and the access parameters of the considered memory, enprofiler analyzes the total energy dissipation of the application that was compiled using different sizes for the register file size.

The energy values shown in Figure 6.7 are again normalized such that the energy consumption using the minimum of three registers is represented as the relative value 0. The general behavior of the curves is obvious: for very small register file sizes, the energy consumption is high since a lot of spill code has to be executed. The memory accesses to the spilled values in the main memory cause an additional contribution to the energy consumption, since the instructions that access main memory consume a lot more energy than e.g. ALU instructions. This is on one hand due to their longer execution times, on the other hand, these instructions cause a higher switching activity in particular on the high-capacitative external bus wires.

The fact that the number of registers is an important factor during the design process of an ASIP is underlined by the results provided in Table 6.3 which shows the relative energy savings possible by increasing the number of available registers one by one. The average energy savings possible in the

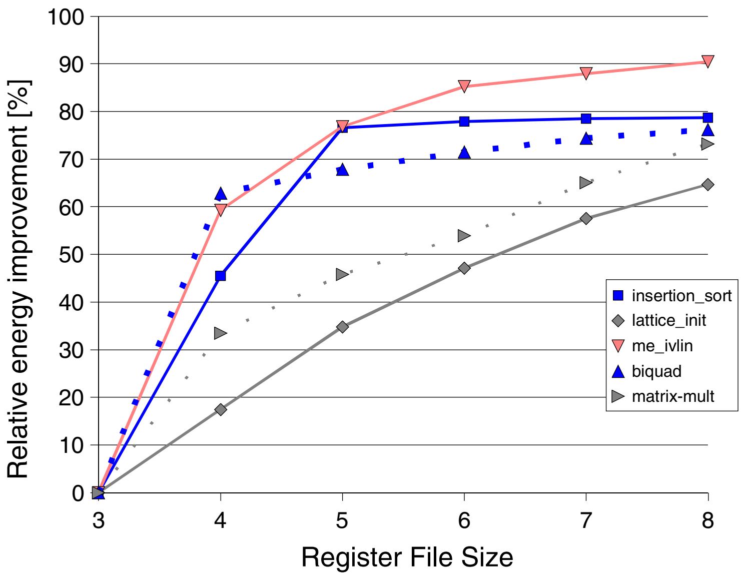


Fig. 6.7. Relative energy consumption over number of registers

benchmark	$3 \rightarrow 4$	$4 \rightarrow 5$	$5 \rightarrow 6$	$6 \rightarrow 7$	$7 \rightarrow 8$
insertion_sort	45.47%	57.11%	5.60%	2.63%	0.92%
lattice	17.45%	20.97%	18.92%	19.78%	16.84%
ME	59.28%	43.16%	36.29%	18.24%	20.47%
biquad_N_sections	62.89%	13.35%	11.26%	10.42%	6.71%
matrix-mult	33.44%	18.57%	14.99%	24.26%	23.28%
Average improvement	43.71%	30.63%	17.41%	15.06%	13.64%

Table 6.3. Energy savings by changing register file size

range from three to eight registers amount to approximately 22%. Since all curves are starting to level out even in the range we have considered here, it is clear that the positive contribution of additional registers becomes smaller the more registers are already present in the design. One should bear in mind, however, that each additional register will also consume additional energy due to added capacitances and required chip area. This energy has not been considered in our approach, but may be the subject of future work.

6.6 Compiler Guided Choice of Register File Size

The knowledge that the compiler has about the generated code for different sizes of the register file can be exploited to gain information concerning a useful range of register file sizes. Since lifetime analysis is performed within the compiler during the register allocation phase as described above in Section 6.3, the compiler is aware of the number of simultaneously live values for every point in the program. This number is known as the "register pressure". If no spill code is to be introduced into the application program, then the maximum register pressure found within the application program determines the minimum number of registers that have to be present in the processor. However, this value is usually too large to be practically relevant.

In order to reduce the maximum register pressure, it is possible to perform instruction scheduling prior to register allocation. The encc employs a simple heuristic which attempts to move the start ("DEF") and the end of the lifetime ("USE") of values closer together. The simple, yet effective instruction scheduling moves all instructions that decrease the register pressure (i.e. less values are live after execution of this instruction) as far towards the beginning of the considered basic block as possible without violating any data dependency constraints. Those instructions that increase the register pressure are moved toward the end of the basic block. These steps effectively move the definition and the use of values closer together, thus reducing the maximum number of values that are live at the same time.

Even the register pressure that was reduced using instruction scheduling is usually too high to serve as an indication for an upper bound of the register file size. Since the execution frequency of basic blocks has a strong impact on both performance and energy, the compiler's knowledge concerning the most frequently executed basic blocks should also be taken into account. By using either static or dynamic profiling (cf. Section 3.6.1), the compiler is aware of the execution count of basic blocks. This information together with the maximum register pressure found within a basic block can be used to find a useful range for the register file size.

It is important to consider the position of this analysis in the code generation process: after register allocation, spill code will already have been inserted, and no more than eight values will ever be live at the same time since otherwise the code could not be executed on the processor. The generation of register pressure information thus has to be performed before the actual register allocation step, but following code selection (so that the structure of the code is already fixed and the virtual registers can be used instead of application program variables) and instruction scheduling (to reduce unnecessarily high register pressure).

For the insertion_sort benchmark, the results generated by the compiler are shown in Figure 6.8

```
*** Register Pressure Information ***
NodeList:
BBNode: 'LL12', IterationCount: 2600, MaxRegPressure: 4
BBNode: 'LL10', IterationCount: 1300, MaxRegPressure: 5
BBNode: '_M_10', IterationCount: 1000, MaxRegPressure: 5
BBNode: 'LL26', IterationCount: 1000, MaxRegPressure: 4
BBNode: 'LL23', IterationCount: 1000, MaxRegPressure: 4
BBNode: 'LL18', IterationCount: 1000, MaxRegPressure: 4
BBNode: 'LL22', IterationCount: 1000, MaxRegPressure: 2
BBNode: 'LL14', IterationCount: 550, MaxRegPressure: 4
BBNode: 'LL15', IterationCount: 500, MaxRegPressure: 4
BBNode: 'LL21', IterationCount: 500, MaxRegPressure: 1
. . .
 *** End of Register Pressure Information ***
```

Fig. 6.8. Register pressure information for insertion_sort

The compiler generates one line for each basic block. To enable the task of finding relevant basic blocks, the information is sorted by the iteration count of the basic blocks such that the most frequently executed basic blocks appear at the beginning of the list. Note that the basic blocks from the innermost loop shown in Figure 6.6 can be found as the first entries in the list. The maximum register pressure together with the iteration count provide information concerning the number of required registers for an efficient execution of the application. In the insertion_sort example, the most frequently executed basic block `LL12` contains four simultaneously live values. The following two basic blocks each contain 5 live values. It is these basic blocks `LL10` and

`_M_10` that are responsible for the sharp bend in the curve of the insertion_sort performance graph. The following basic blocks have both a smaller iteration count and require less registers. They therefore do not have an impact on the recommended register file size for this application.

To summarize, the designer has to partition the basic blocks according to their iteration counts. By only considering the most frequently executed basic blocks, the number of required registers to avoid spill code to be inserted within the innermost loops can be estimated. Of course, the information produced by the compiler could still be refined to e.g. contain the number of accesses to potentially spilled values or the number of spill instructions that would be generated if a value has to be spilled. Refining the information obtained from the compiler will be part of the future work concerning the size of the register file.

7 Summary

Concerning the development of embedded systems, the past years have seen a large increase in the number of new devices. Innovative designs provide a good potential on the highly competitive market. Depending on the application, different requirements have been put forward for embedded systems. The most important properties that are being asked of systems today are high performance, low energy dissipation and, in particular for real time capable systems, timing predictability. Since to achieve a high degree of flexibility, a majority of embedded systems uses processors that execute applications written in software, the compiler used to generate the executable code is of vital importance to achieve satisfactory results concerning the mentioned requirements. The contributions made in this work cover all three aspects, and in particular include the consideration of the memory hierarchy in the given system in order to generate code that optimally exploits the architectural features of the different available memories.

Models are required to integrate the properties of the system under consideration into the used compiler. To generate code, a precise definition of the instruction set architecture is required, which is provided for the ARM processor. Timing and energy models for the processor are also provided in order to optimize for performance and energy dissipation and to evaluate the generated code. Since this evaluation is performed using instruction set simulation, simulation models are also required.

To also consider the used memory architecture, models of the memories are required. Again, the important aspects that have to be covered to be able to optimize code taking into account the properties of the memory hierarchy are the access times and the energy dissipation of the used memories. Since in particular the behavior of dynamic RAMs depends on the current circuit state, some effort is required to integrate these memories into the toolchain used by the compiler environment.

The first set of optimizations considered in this work deals with the exploitation of multiple scratchpad memory partitions. Partitioning of scratchpad memories is considered for two main reasons: on one hand, using several smaller memories compared to one large memory leads to reduced energy dissipation, since the energy and access times depend on the size of the considered memory. On the other hand, recent designs by ARM Ltd. feature a partitioned scratchpad memory in the form of a so-called Tightly Coupled Memory (TCM). It can thus be observed that there is a trend towards partitioned scratchpad memories in industry.

In contrast to caches, scratchpad memories require support from the programmer or the compiler to be exploited. In this work, the compiler automatically assigns memory objects, i.e. instructions and data objects, to the scratchpad memories. This is first formalized in a simple Base model that is represented in the form of an ILP problem. The Base model considers basic blocks and global data as independent objects and allocates them to the scratchpad memory according to their execution or access frequency. This model is then refined to consider the relationship of basic blocks and functions: while functions can be considered as atomic units that can be allocated to any memory partition without modification, basic blocks require a special treatment when their successors are not allocated to the same memory to preserve the correctness of control flow. In the Top-Down model, additional decision variables and constraints are used to achieve the selection of either a complete function or of several individual basic blocks within that function. The Bottom-Up model in addition also considers the advantage of allocating blocks of contiguous basic blocks to the same scratchpad memory partition. The edges of the control flow graph are used in this case in order to avoid a strong increase in the complexity of the ILP representation. Results show that all three approaches are capable of saving significant amounts of energy: for the smallest scratchpad capacity of 64 bytes and one particular application, the Top-Down approach saves 16% of the energy dissipated in the memory subsystem, whereas the more precise Bottom-Up allocation saves up to 33%. For larger scratchpad memory capacities, the Bottom-Up approach can save up to 97% of the energy spent in the considered memory setup for one considered application, which translates to about 80% of total energy savings for the complete system. For the other considered benchmarks, the maximum energy savings in the memory subsystem amount to about 80%. For two benchmarks, considerably lower savings were determined. In one case, this is due to calls to libraries whose code can not be allocated to the scratchpad, in the other case memory accesses to a global data array are not detected due to the use of pointers instead of array accesses. The direct comparison of Base, Top-Down and Bottom-Up approach reveals that the Bottom-Up allocation outperforms the Base case by up to 17% for small scratchpad capacities. The results of the Top-Down allocation are between the Base and the Bottom-Up model, as expected.

To generate code for the Harvard-style ARM TCM architecture, the Bottom-Up ILP model was modified so that instructions are allocated to one scratchpad memory partition and data to the other. This effectively simplifies the ILP problem, however it also leads to less energy savings, since the compiler can not freely allocate instructions and data to the optimal partitions. A direct comparison of the Bottom-Up and the ARM-TCM allocation techniques shows that for some memory configurations, nearly twice the amount of energy can be saved when the Bottom-Up technique is used. If the considered system only supports the allocation of instruction and data to separate memory partitions, then the ARM-TCM allocation model presented in this work can be used to optimally allocate memory objects to the TCM partitions.

Finally, the effect of leakage currents when many small scratchpad memory partitions are present in the system is considered. By extending the ILP representation of the allocation problem, each memory is modeled to introduce a leakage energy into the system if it is being used by the allocation, i.e. if at least one object is allocated to it. When the leakage energy assumed for each of the memories is increased, the ILP solver decides to allocate objects to less memory partitions in order to save the leakage energy. While for small leakage values, up to ten scratchpad memory partitions are allocated and used, only one scratchpad memory and the main memory is used for very large leakage values. In this way, the compiler can help the designer of a system to decide on the number of scratchpad memory partitions that should be used in the system.

Scratchpad memories are being used to improve the performance and reduce the energy dissipation of embedded systems. They are better suited for this purpose than caches since they are smaller and require less energy per access than a cache of comparable capacity. One additional advantage is that due to the allocation of memory objects to the scratchpad memory at compile time, a scratchpad memory offers full predictability. This is in strong contrast to a cache which takes decisions concerning hits and misses dynamically at runtime, which is hard to predict at design time. During the design phase of safety critical real-time systems, the designer has to be able to provide a guaranteed upper bound on the worst case execution time (WCET) of the system. The closer this upper bound is to the actual worst case performance of the system, the less resources are required to implement the system. Tight upper bounds can thus help to make a system cheaper. The WCET can be determined at design time using WCET analysis tools which analyze the architecture and the application and generate a guaranteed upper bound for the maximum execution time. A lot of effort has gone into the integration of caches into these WCET analysis tools. The used algorithms are very complex, yet they frequently strongly overestimate the WCET. The quality of the WCET analysis results also depends on the architecture of the cache: if LRU replacement is used in set-associative caches, then the analysis can determine which set will be replaced in case of a miss. The ARM processors, however, use a random replacement policy which leads to less predictability and thus loose upper bounds on the WCET. The comparison of caches and scratchpad memories performed in this work underlines the fact that scratchpad memories are fully predictable, while the use of caches is not useful if timing predictability is a design concern. Both an average case simulation and a WCET analysis is performed for scratchpad memories and caches of varying capacities. Memory objects are allocated to the scratchpad memories using both the static Bottom-Up approach mentioned above and a dynamic allocation technique.

In the static case, the cache was found to outperform the scratchpad allocation in the average case for some benchmarks, while the use of a scratchpad memory resulted in tighter upper bounds on the WCET. To be more precise, the performance benefit obtained by using a scratchpad memory has a direct influence on the WCET as well: the savings that were observed concerning average case execution time were also found in the WCET. For the cache, the situation is quite different. While for small cache sizes, the observed average case performance is degraded due to a large number of conflict misses, larger cache sizes lead to substantial savings in the average case, as expected. The WCET, however, does not profit from these improvements: it usually stays at the same high level as for a very small cache. This means that the used cache analysis is unable to determine an acceptably high number of guaranteed misses.

For the dynamic scratchpad allocation, the scratchpad retains its full predictability even if memory objects are copied from the main memory to the scratchpad and vice versa at runtime. Using this allocation technique, however, the situation for the cache is even worse, since even in the average case, the scratchpad memory shows superior performance. As a conclusion, scratchpad memories do not require any additional complex analysis techniques, they can simply be introduced as distinct memory regions in the WCET analysis tool. While a lot of effort has been spent to improve the quality of WCET analysis when caches are present in a system, the use of a scratchpad is capable of producing superior results at lower computational cost.

Beside the highly efficient scratchpad memories, optimization potential can also be found in the main memories: in particular modern main memories in SDRAM technology offer power management features like power down modes that can be efficiently exploited using a compiler. In the considered setup, a scratchpad memory is assumed to be present in the system along with an SDRAM main memory. The compiler again allocates memory objects to the scratchpad partition, however a different objective function is used: since the main memory can be put in the power down mode if instructions and data are both accessed from the scratchpad memory, the compiler optimization tries to allocate objects in such a way that the main memory can be kept in the power down mode for a maximum amount of time. To achieve this, a model containing all non access related energy costs is constructed. In particular, this formalized model also considers the dependencies between variables and their accessing basic blocks. By allocating both basic blocks and variables to the scratchpad at the same time, the optimization can prevent an activation of the main memory solely to perform an access to a data element when instructions are fetched from the scratchpad memory. The results obtained by this model are compared to the Bottom-Up allocation approach which uses the access related “per-access” costs instead of the standby energy in the objective function. Despite this difference in the two optimization approaches, the obtained results are surprisingly similar. Total energy savings of more than 80% can be achieved compared to a system without scratchpad by exploiting the scratchpad memory and the power down mechanism of the SDRAM main memory. In several cases, the new power down optimization was able to outperform the Bottom-Up approach, since it was able to keep the main memory in the power down state for a prolonged time. In one case, a global array was allocated to the scratchpad instead of main memory. This had the effect described above: the main memory stays in power down mode for a longer time, since the global variable is accessed on the scratchpad memory partition. The general idea of formulating an optimization function based on reducing the standby energies of memories is therefore a promising option when memories with power management features are being used.

Most embedded systems deployed today are equipped with a non volatile Flash memory to permanently store application programs and configuration information. Usually, the Flash memory is only used at startup: following the boot procedure, the operating system copies all instructions and data from the Flash memory to the volatile main memory and accesses them from there. The Flash memory is thus not used during normal operation of the device, which is a waste of resources. Modern Flash memories are capable of directly serving as instruction memories. This property is known as eXecute-In-Place (XIP). Using XIP, instructions can be fetched from the Flash memory instead of first being copied to e.g. the main memory and then being executed from there. The fact that Flash memories are in general slower than the main instruction memory partition in SDRAM technology leads to a tradeoff that can be solved by the compiler: if an object is frequently executed, it may be beneficial to copy it to the faster SDRAM memory, if it is only executed once, it may just as well be executed directly from the Flash memory, saving the copy overhead. Since Flash memories should in general not be used for frequent writing, it is not viable to allocate writable data to the Flash memory. As long as instructions are executed from the Flash memory, the main SDRAM memory can again be put in the power down mode to save energy. To avoid the situation where the main memory has to be activated due to data accesses, the main SDRAM memory was partitioned into one data and one instruction partition. An optimization problem was then formalized that either copies instructions to the SDRAM instruction partition or executes them directly from the Flash memory, taking into account both the access and the non access related energy contributions. Since the data is stored in a separate SDRAM partition, the instruction partitions of the SDRAM main memory can be shut off completely when the last instruction has been fetched from it and its contents are thus not required any longer. To also take these improvements into account, a heuristic algorithm was implemented that first selects those objects from the end of the application that should be executed from the Flash memory so that the SDRAM can go to deep power down.

The results of the optimization are compared to using only SDRAM or using only Flash memory as instruction memory. In general, the optimized version can be expected to have similar execution time compared to a pure SDRAM execution. When the performance of the optimized version is below that of the SDRAM version, however, the requirements concerning the capacity of the SDRAM instruction partition are reduced significantly, in particular if a relatively fast Flash memory is used. For one benchmark, an increase in the number of cycles by 5% resulted in 96% savings concerning the SDRAM instruction partition's size. For most of the other benchmarks, considerable savings concerning the required SDRAM memory could be obtained. Since the size of the main memory is one of the cost factors in embedded systems, and since the Flash memory always has to be present in the system to store the application when no supply voltage is available, the proposed optimization strategy leads to substantial savings.

Finally, the compiler is also responsible for managing one of the most effective memories in the memory hierarchy: the register file. During register allocation, the compiler assigns all values that occur within a program to the physically available registers of the target processor. If an insufficient number of registers is available, then additional code has to be inserted to spill the values to memory. In this work, results concerning the influence of the register file size on the quality of the generated code are presented. It was found that for a very small register file, the code size can be increased by up to 70% compared to the case where no spilling is required. Concerning performance, an average of about 60% of the cycles could be saved when eight registers were available in the processor instead of the minimum three. Average energy savings of 22% were observed in the same range. In addition to these experimental results, the information available within the compiler concerning basic block execution counts and the number of simultaneously live data values are used to provide the designer with an information concerning the recommended register file size.

8

Future Work

Despite the solutions and improvements achieved by the optimizations presented in this work, there is always a need to perform more research and to strive for optimality concerning the exploitation of available resources. This chapter lists further contributions that were not investigated and makes suggestions for possible research and experimental work concerning the optimization of compiler generated code taking into account the properties of the memory architecture to achieve the objectives of high performance, low energy dissipation and predictable access timing.

3 – Models and Tools

In this work, only the ARM7 processor architecture was considered. It would make sense to additionally consider processors from other domains, e.g. DSPs or VLIW architectures in order to validate the results determined for the RISC class of machines. To evaluate results concerning these new classes of processors, it would be useful to resort to the large amount of work that has been performed concerning automatic generation of processor simulators from architecture description languages.

Concerning memory models, a unified way of specifying the access as well as non access related properties of different memory architectures would be helpful in the integration of additional memory models.

The memory hierarchy simulator MEMSIM is a versatile tool that should be extended to simulate more memory architectures, e.g. victim caches, loop buffers or take into account the energy dissipation of way-management in caches. Additionally, the simulation model should be extended to allow the consideration of both multi-process and multiprocessor simulation.

A unified configuration mechanism for the encc compiler framework would be helpful in performing further experiments. Currently, the configuration is somewhat complex due to the incremental development process of the compiler. Also, additional analysis techniques should be added to the compiler to be able to detect array references performed using pointers.

This would e.g. lead to better results concerning the FFT benchmark in the multi memory optimization chapter. Finally, cache optimization algorithms should be integrated into the encc compiler framework to allow a better comparison between caches and scratchpad memories. Currently, the "loop tiling" optimization which is expected to increase the cache performance is being studied and implemented.

4.2 – Multi Memory Optimizations

The allocation of memory objects to partitioned scratchpad memories has been implemented by generating an ILP formulation of the allocation problem and solving it using an ILP solver. Since the allocation algorithms that were studied in this work belong to the class of assignment problems, efficient approximation algorithms exist to solve these problems.

They should be implemented and compared to the optimal results generated using ILP. Despite the fact the during the experiments, no problems with the execution times of the ILP solver were encountered, the general concern about this topic remains.

As a new optimization objective, a dynamic allocation of objects to partitioned scratchpad memories could be used. The dynamic approach used for a single scratchpad would have to be extended to also find an allocation of objects to several partitions. In this case, the execution times of optimal solving algorithms may become a problem. In this case, a heuristic may have to be used.

4.3 – Impact of Scratchpad Allocation Techniques on WCET

While the WCET results obtained for a scratchpad are better than those for a cache, the annotation effort that has to be taken for every new application is still quite high, despite the fact that the annotation process is mostly automated. A fully automatic extraction of all required annotation information from the executable of the application would help reduce the manual annotation effort. Information that can not be determined from the executable alone, like loop bounds that include complex expression, should be supplied in a single configuration file for the considered benchmark.

Using a full featured cache analysis to validate the comparison between cache and scratchpad concerning WCET is highly desirable. The complexity of cache analyses is quite high, and considering the fact that the ARM caches use an unpredictable random replacement policy, the results for the cache may not improve very much.

An improvement of the obtained results for the scratchpad can be expected if the notion of WCET is also used within the compiler: for the experiments presented here, the compiler allocated memory objects to the scratchpad using energy dissipation as the cost function. If, in contrast to that, the compiler allocated those objects to the scratchpad that lie on the critical path of the application, an improvement concerning WCET results is to be expected.

5.2 – Main Memory Power Management

In the presented optimization, only the standby energy dissipation of the SDRAM main memory was considered in the cost function, since the optimization goal was to minimize the standby energy by keeping the main memory in the power down mode. In the Bottom-Up model that was used for comparison, only access related energy dissipation was considered. Despite the fact that these two optimizations target different contributions to the energy budget, the obtained results are surprisingly similar. A unified model that takes into account both access and non access related energy contributions could be formulated and solved. The methodology should be straightforward, however, the similarity of results obtained from the two models mentioned above leads to the assumption that no significant changes would be achieved concerning the allocation of memory objects to the scratchpad memory.

5.3 – Execute-In-Place using Flash Memories

The straightforward implementation of the actual XIP optimization problem and the separate choice of basic block sequences that enable the main memory to be shut off completely is not a very elegant solution. A uniform model should be generated that takes into account both the power down and the deep power down states of the main SDRAM instruction memory partition. Using additional decision variables to correct the energy dissipation of objects that are executed from the Flash memory when the SDRAM partition is in deep power down mode should be straightforward.

6 – Register File Optimization

The information generated by the compiler could be refined to also include the number of accesses to different registers, the number of different expressions in the basic block and information concerning the spill costs of the virtual registers. More detailed information would help the designer to take a decision concerning the register file size based on the compiler output. Furthermore, the architectural effects of modifying the register file size should be considered and traded against the observed effects concerning the quality of the code generated by the compiler.

References

[ABC03] Federico Angiolini, Luca Benini, and Alberto Caprara. Polynomial-time algorithm for on-chip scratchpad memory partitioning. In *Proccedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 318–326, October 2003.

[Abs04a] AbsInt Angewandte Informatik GmbH. aiT Reference Manual, 2004.

[Abs04b] AbsInt Angewandte Informatik GmbH. aiT: Worst Case Execution Time Analyzers. http://www.absint.com/ait, 2004.

[Alb99] David H. Albonesi. Selective cache ways: on-demand cache resource allocation. In *Proceedings of the International Symposium on Microarchitecture*, pages 248–259, November 1999.

[ANS] ANSI (American National Standards Institute). ISO/IEC 9899:1999 (or: C99), "The ANSI C standard". http://www.ansi.org.

[Apa04] The Apache XML project. Xerces C++-Parser. http://xml.apache.org/xerces-c, 2004.

[App98] Andrew W. Appel. *Modern Compiler Implementation in C.* Cambridge University Press, Cambridge, New York u.a., 1998.

[ARM] ARM Ltd. The ARM9 processor core family. http://www.arm.com/products/CPUs/families/ARM9Family.html.

[ARM98a] ARM Ltd. ARM Software Development Toolkit Version 2.50 User Guide, ARM Document Number DUI 0040D, 1998.

[ARM98b] ARM Ltd. ARM710T Datasheet, ARM Document Number ARM DDI 0086B, 1998.

[ARM00] ARM Ltd. ARM946E-S Technical Reference Manual, ARM Document Number DDI 0155A, 2000.

[ARM01] ARM Ltd. ARM7TDMI Reference Manual, ARM Document Number DDI 0210B, 2001.

[ARM04a] ARM Ltd. ARM1136JF-S and ARM1136J-S Technical Reference Manual, ARM Document Number DDI 0211E, 2004.

[ARM04b] ARM Ltd. Cost-effective Per-Use Access to ARM Technology, ARM Foundry Program Flyer. http://www.arm.com, 2004.

[Aus] Todd Austin. SimpleScalar LLC. http://www.simplescalar.com.

[B0̈2] Helmut Bähring. *Microcomputing systems. Volume II: Buses, Memories, Peripherals and Microcontrollers (Mikrorechner-Systeme. Band*

II: Busse, Speicher, Peripherie und Mikrocontroller) (in German language). Springer-Verlag, 2002.

[BBM00] Luca Benini, Alessandro Bogliolo, and Giovanni De Micheli. A Survey of Design Techniques for System-Level Dynamic Power Management. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(3):299–316, June 2000.

[BMP00] Luca Benini, Alberto Macii, and Massimo Poncino. A recursive algorithm for low-power memory partitioning. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, pages 78–83, July 2000.

[Bor99] Shekhar Borkar. Design Challenges of Technology Scaling. *IEEE Micro*, 19(4):23–29, July 1999.

[BSL$^+$02] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad Memory: A Design Alternative for Cache On-chip memory in Embedded Systems. In *Proceedings of the Tenth International Workshop on Hardware/Software Codesign (CODES)*, May 2002.

[BTM00] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, pages 83–94, June 2000.

[CFW$^+$94] Francky Catthoor, Frank Franssen, Sven Wuytack, Lode Nachtergaele, and Hugo De Man. Global Communication and Memory Optimizing Transformations for Low Power Signal Processing Systems. In *Proceedings of the Workshop on VLSI Signal Processing*, pages 178–187, October 1994.

[CH98] Keith D. Cooper and Timothy J. Harvey. Compiler-Controlled Memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 2–11, October 1998.

[CHB$^+$05] Jianjiang Ceng, Manuel Hohenauer, Gunnar Braun, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. C Compiler Retargeting Based on Instruction Semantics Models. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, pages 1150–1155, March 2005.

[CKI$^+$01] Lakshmi N. Chakrapani, Pinar Korkmaz, Vincent J. Mooney III, Krishna V. Palem, Kiran Puttaswamy, and Weng-Fai Wong. The emerging power crisis in embedded processors: what can a (poor) compiler do? In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 176–180, November 2001.

[CKL00] Naehyuck Chang, Kwanho Kim, and Hyung Gyu Lee. Cycle-Accurate Energy Consumption Measurement and Analysis: Case Study of ARM7TDMI. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, pages 185–190, July 2000.

[CY02] Wissam Chedid and Chansu Yu. Survey on Power Management Techniques for Energy Efficient Computer Systems. Technical report, Department of Electrical and Computer Engineering, Mobile Computing Research Lab, Cleveland State University, September 2002.

[EH] Jan Edler and Mark D. Hill. Dinero IV Trace-Driven Uniprocessor Cache Simulator. http://www.cs.wisc.edu/˜markhill/DineroIV.

[Fer97] Christian Ferdinand. *Cache Behavior Prediction for Real-Time Systems*. PhD thesis, Universität des Saarlandes, Saarbrücken, 1997.

[FHP00] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a Simple, Efficient Code Generator Generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, June 2000.

[Gai] Jiri Gaisler. LEON2 SPARC processor. http://www.gaisler.com/products/leon2/leon.html.

[GAV95] Antonio Gonzales, Carlos Aliagas, and Mateo Valero. A data cache with multiple caching strategies tuned for different types of locality. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 338–347, July 1995.

[GCC05] GCC Home Page. GNU Project - Free Software Foundation (FSF). http://gcc.gnu.org, 2005.

[GLMS02] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Springer, 2002.

[Hel04] Urs Helmig. Compiler supported Optimization of Accesses to Partitioned Memories (Compilergestützte Optimierung von Zugriffen auf partitionierte Speicher) (in German language). Master's thesis, Embedded Systems Group, Department of Computer Science XII, University of Dortmund, March 2004.

[His05] Jason D. Hiser. *Effective Algorithms for Partitioned Memory Hierarchies in Embedded Systems*. PhD thesis, Department of Computer Science, University of Virginia, USA, April 2005.

[HLTW03] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The Influence of Processor Architecture on the Design and the Results of WCET Tools. *Proceedings of the IEEE*, 91(7), July 2003.

[HP03] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, Third Edition*. Morgan Kaufmann Publishers, 2003.

[HPH$^+$00] Ulrich Heinkel, Martin Padeffke, Werner Haas, Thomas Bürner, Herbert Braisz, Thomas Gentner, and Alexander Grassmann. *The VHDL Reference*. John Wiley & Sons, Ltd, 2000.

[HS89] Mark D. Hill and Alan Jay Smith. Evaluating Associativity in CPU Caches. *Transactions on Computers*, pages 1612–1630, December 1989.

[ILO05] ILOG. ILOG CPLEX: High Performance Software for Mathematical Programming and Optimization. http://www.ilog.com/products/cplex, 2005.

[IY98] Tohru Ishihara and Hiroto Yasuura. Voltage Scheduling Problem for Dynamically Variable Voltage Processors. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, pages 197–202, August 1998.

[JBK01] Manoj Kumar Jain, M. Balakrishnan, and Anshul Kumar. ASIP Design Methodologies: Survey and Issues. In *Proceedings of the IEEE/ACM International Conference on VLSI Design*, pages 76–81, January 2001.

[JM93] R. Jöhnk and Peter Marwedel. MIMOLA Reference Manual Version 3.45. Technical Report 470, Embedded Systems Group, Department of Computer Science XII, University of Dortmund, March 1993.

[Ker05] André Kernchen. Compiler supported energy reduction for SRAM and Flash-based memory technologies (Compilergestützte Energiereduktion von SDRAM und Flash-basierten Speichertechnologien) (in German language). Master's thesis, Embedded Systems Group, Department of Computer Science XII, University of Dortmund, January 2005.

[KG97] Milind B. Kamble and Kanad Ghose. Analytical energy dissipation models for low power caches. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, pages 143–148, August 1997.

[KG02] Arvind Krishnaswamy and Rajiv Gupta. Profile Guided Selection of ARM and Thumb Instructions. In *Proceedings of the conference on Languages, Compilers, and Tools for Embedded Systems (LCTES) and Software and Compilers for Embedded Systems (SCOPES)*, pages 56–64, June 2002.

[KGK99] Hansoo Kim, Kanchi Gopinath, and Vinod Kathail. Register allocation in hyper-block for EPIC processors. In *Proceedings of the International Conference on Parallel Computing: Fundamentals & Applications (ParCo)*, pages 550–557, August 1999.

[KHZA00] Ronny Krashinsky, Seongmoo Heo, Michael Zhang, and Krste Asanović. SyCHOSys: Compiled Energy-Performance Cycle Simulation. In *Workshop on Complexity-Effective Design, held in conjunction with the 27th Annual International Symposium on Computer Architecture (ISCA)*, June 2000.

[KKS01] Mahmut Kandemir, Ismail Kadayif, and Ugur Sezer. Exploiting Scratch-Pad Memory Using Presburger Formulas. In *Proceedings of the Internationational Symposium on System Synthesis (ISSS)*, pages 7–12, September 2001.

[KLMSP99] Johnson Kin, Chunho Lee, William H. Mangione-Smith, and Miodrag Potkonjak. Power Efficient Mediaprocessors: Design Space Exploration. In *Proceedings of the Design Automation Conference (DAC)*, pages 321–326, June 1999.

[KPL99] Young-Jun Kwon, Danny Parker, and Hyuk Jae Lee. TOE: Instruction Set Architecture for Code Size Reduction and Two Operations Execution. In *International Workshop on Compiler and Architecture Support for Embedded Systems (CASES)*, October 1999.

[KVIY00] Mahmut Kandemir, Narayanan Vijaykrishnan, Mary J. Irwin, and Wu Ye. Influence of Compiler Optimizations on System Power. In *Proceedings of the 37th Design Automation Coference (DAC)*, pages 304–307, June 2000.

[Leu97] Rainer Leupers. *Retargetable Code Generation for Digital Signal Processors.* Kluwer Academic Publishers, 1997.

[Leu00] Rainer Leupers. *Code Optimization Techniques for Embedded Processors.* Kluwer Academic Publishers, 2000.

[Leu01] Rainer Leupers. LANCE: A C Compiler Platform for Embedded Processors. In *Embedded Systems/Embedded Intelligence*, Nürnberg, Germany, February 2001.

[LLM$^+$01] Markus Lorenz, Rainer Leupers, Peter Marwedel, Thorsten Dräger, and Gerhard P. Fettweis. Low-Energy DSP Code Generation Using a Genetic Algorithm. In *Proceedings of the International Conference on Computer Design (ICCD)*, pages 431–437, September 2001.

[LMD$^+$04] Markus Lorenz, Peter Marwedel, Thorsten Dräger, Gerhard Fettweis, and Rainer Leupers. Compiler based Exploration of DSP Energy Savings by SIMD Operations. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 839–842, January 2004.

[LMW95] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Performance Estimation of Embedded Software with Instruction Cache Modeling. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 380–387, November 1995.

[LMW96] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, pages 254–263, December 1996.

[LNC03] Hyung Gyu Lee, Sungyuep Nam, and Naehyuck Chang. Cycle-accurate Energy Measurement and High-Level Energy Characterization of FPGAs. In *Proceedings of the Fourth International Symposium on Quality Electronic Design (SQED)*, pages 267–273. IEEE, March 2003.

[Lor03] Markus Lorenz. *Performance- and Energy-Efficient Compilation for Digital SIMD Signal Processors using Genetic Algorithms (Performance- und energieeffiziente Compilierung für digitale SIMD-Signalprozessoren mittels genetischer Algorithmen) (in German language).* PhD thesis, Embedded Systems Group, Department of Computer Science XII, University of Dortmund, January 2003.

[LSI04] LSI Logic Corporation. ARM Processor Cores. http://www.lsilogic.com/products/arm, 2004.

[Lun02] Thomas Lundqvist. A WCET Analysis Method for Pipelined Microprocessors with Cache Memories. Technical report, Dept. of Computer Engineering, Chalmers University of Technology, June 2002.

[LWD02] Markus Lorenz, Lars Wehmeyer, and Thorsten Dräger. Energy aware Compilation for DSPs with SIMD Instructions. In *Proceedings of the conference on Languages, Compilers, and Tools for Embedded Systems (LCTES) and Software and Compilers for Embedded Systems (SCOPES), published in ACM SIGPLAN Notices, Volume 37, Issue 7*, pages 94–101, June 2002.

[Mac02] Philip Machanik. Approaches to Addressing the Memory Wall. Technical report, University of Brisbane, November 2002.

[Mar93] Peter Marwedel. *Synthesis and Simulation of VLSI-Systems (Synthese und Simulation von VLSI-Systemen) (in German language).* Carl Hanser Verlag, 1993.

[MCB$^+$03] Paul Marchal, Francky Catthoor, Davide Bruni, Luca Benini, Jose Ignacio Gomez, Luis Piuel, and Henk Corporaal. SDRAM-Energy-Aware Memory Allocation for Dynamic Multi-Media Applications on Multi-Processor Platforms. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, pages 516–521, March 2003.

[Men] Mentor Graphics. ModelSim. http://www.mentor.com.

[Mic01] Micron Technology Inc. Calculating DDR Memory System Power. Technical Report TN-46-03, Micron Technology Inc., 2001.

[Mic04a] Micron Technology Inc. 128Mbit Mobile SDRAM Data Sheet, MT48 series. http://www.micron.com/pdf/datasheets/dram/mobile/128Mbx16x32Mobile.pdf, 2004.

[Mic04b] Micron Technology Inc. Data Sheet for Micron Q-Flash MT28F640J3. http://www.micron.com/pdf/datasheets/flash/qflash/MT28F128J3_15.pdf, 2004.

[Mic04c] Micron Technology Inc. MT48H4M16LF Data Sheet - 64Mbit Mobile SDRAM. Document No. Y25L. http://www.micron.com/pdf/datasheets/dram/mobile/Y25L_64Mb.pdf, 2004.

[MMC00] Afzal Malik, Bill Moyer, and Dan Cermak. A programmable unified cache architecture for embedded applications. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 165–171, November 2000.

[MMP03] Alberto Macii, Enrico Macii, and Massimo Poncino. Improving the Efficiency of Memory Partitioning by Address Clustering. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, pages 18–24, March 2003.

[Mos01] Vasily G. Moshnyaga. Reducing Cache Energy through Dual Voltage Supply. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 302–305, January 2001.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[MWV$^+$04] Peter Marwedel, Lars Wehmeyer, Manish Verma, Stefan Steinke, and Urs Helmig. Fast, predictable and low energy memory references through architecture-aware compilation. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 4–11, January 2004.

[NCB$^+$95] Lode Nachtergaele, Francky Catthoor, Florin Balasa, Frank Franssen, Eddy De Greef, Hans Samsom, and Hugo De Man. Optimization of memory organization and hierarchy for decreased size and power in video and image processing systems. In *Proceedings of the IEEE International Workshop on Memory Technology, Design and Testing*, pages 82–89, August 1995.

[OIY99] Takanori Okuma, Tohru Ishihara, and Hiroto Yasuura. Real-Time Task Scheduling for a Variable Voltage Processor. In *Proceedings of the 12th International Symposium on System Synthesis (ISSS)*, pages 24–29, November 1999.

[Pal03] Samir Palnitkar. *Verilog HDL*. Prentice Hall, 2003.

[PB00] Peter Puschner and Alan Burns. A Review of Worst-Case Execution-Time Analysis. *Journal of Real-Time Systems*, 18(2/3):115–128, May 2000.

[PCD$^+$01] Preeti Ranjan Panda, Francky Catthoor, Nikil D. Dutt, Koen Danckaert, Erik Brockmeyer, Chidamber Kulkarni, Arnout Vandecappelle, and Per G. Kjeldsberg. Data and memory optimization techniques for embedded systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 6(2):149–206, April 2001.

[PDN97] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. Architectural exploration and optimization of local memory in embedded systems. In *Proceedings of the International Symposium on System Synthesis (ISSS)*, pages 90–97, September 1997.

[PDN99a] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. Local memory exploration and optimization in embedded systems. *IEEE Transactions on Computer Aided Design*, 18:3–13, January 1999.

[PDN99b] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. *Memory Issues in Embedded Systems-On-Chip*. Kluwer Academic Publishers, 1999.

[Pet04] Klaus Petzold. Scratchpad Allocation Strategies for Multi-Process Systems (Scratchpad-Allokations-Strategien für Multiprozess-Systeme) (in German language). Master's thesis, Embedded Systems Group, Department of Computer Science XII, University of Dortmund, October 2004.

[PHM00] Stefan Pees, Andreas Hoffmann, and Heinrich Meyr. Retargetable Compiled Simulation of Embedded Processors Using a Machine Description Language. *IEEE Transactions on Design Automation of Electronic Systems*, 5(4):815–834, October 2000.

[PMP04] Kimish Patel, Enrico Macii, and Massimo Poncino. Synthesis of Partitioned Shared Memory Architectures for Energy-Efficient Multi-Processor SoC. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, pages 700–701, February 2004.

[PRASA99] Vijay S. Pai, Parthasarathy Ranganathan, Hazim Abdel-Shafi, and Sarita Adve. The Impact of Exploiting Instruction-Level Parallelism on Shared-Memory Multiprocessors. *Transactions on Computers, Special Issue on Caches*, pages 218–226, February 1999.

[PSB$^+$03] Chanik Park, Jaeyu Seo, Sunghwan Bae, Hyojun Kim, Shinhan Kim, and Bumsoo Kim. A Low-cost Memory Architecture with NAND XIP for Mobile Embedded Systems. In *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 138–143, October 2003.

[Pus99] Peter Puschner. Real-Time Performance of Sorting Algorithms. *Real-Time Systems*, 16(1):63–79, January 1999.

[Saa03] Alexandre Saad. Java-based Functionality and Data Management in the Automobile. Prototyping at BMW Car IT GmbH. http://www.javaspektrum.de. *JavaSpektrum*, 2:49–53, March 2003.

[SBM99a] Tajana Simunic, Luca Benini, and Giovanni De Micheli. Cycle-Accurate Simulation of Energy Consumption in Embedded Systems. In *Proceedings of the Design Automation Conference (DAC)*, pages 867–873, June 1999.

[SBM99b] Tajana Simunic, Luca Benini, and Giovanni De Micheli. Energy-Efficient Design of Battery-Powered Embedded Systems. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, pages 212–217, August 1999.

[SBT00] Akshaye Sama, M. Balakrishnan, and J. F. M. Theeuwen. Speeding up Power Estimation of Embedded Software. In *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, pages 191–196, July 2000.

[SC99] Wen-Tsong Shiue and Chaitali Chakrabarti. Memory Eploration for Low Power, Embedded Systems. In *Proceedings of the Design Automation Conference (DAC)*, pages 140–145, New Orleans, June 1999. ACM/IEEE.

[SCG95] Simon Segars, Keith Clarke, and Liam Goudge. Embedded Control Problems, Thumb, and the ARM7TDMI. *IEEE Micro*, 15(5):22–30, October 1995.

[Sch03] Marc Schuller. Studying the switching activity of RISC processors using the Leon processor (Untersuchung der Schaltaktivität von RISC-Prozessoren am Beispiel des Leon Prozessors) (in German language). Master's thesis, Institut für Technische Informatik, Universität Stuttgart, 2003.

[Sed98] Robert Sedgewick. *Algorithms.* AddisonWesley, Massachusetts, 1998.

[SGW+02] Stefan Steinke, Nils Grunwald, Lars Wehmeyer, Rajeshwari Banakar, M. Balakrishnan, and Peter Marwedel. Reducing Energy Consumption by Dynamic Copying of Instructions onto Onchip Memory. In *Proceedings of the International Symposium on System Synthesis (ISSS)*, pages 213–218, October 2002.

[SJC+03] Hojun Shim, Yongsoo Joo, Yongseok Choi, Hyung Gyu Lee, and Naehyuck Chang. Low-Energy Off-Chip SDRAM Memory Systems for Embedded Applications. *ACM Transactions on Embedded Computing Systems (TECS)*, 2(1):98–130, Februrary 2003.

[SKWM01] Stefan Steinke, Markus Knauer, Lars Wehmeyer, and Peter Marwedel. An Accurate and Fine Grain Instruction-Level Energy Model supporting Software Optimizations. In *International Workshop on Power And Timing Modeling, Optimization and Simulation (PATMOS)*, pages 3.2.1–3.2.10, September 2001.

[SLAM98] Jeff Scott, Lea Hwang Lee, John Arends, and Bill Moyer. Designing the Low-Power MCORE Architecture. In *Proceedings of the IEEE Power Driven Microarchitecture Workshop*, pages 145–150, June 1998.

[SNF04] Julian Seward, Nick Nethercote, and Jeremy Fitzharding. Valgrind-system for debugging and profiling x86-Linux programs. http://valgrind.kde.org, 2004.

[SRP+95] Deo Singh, Jan Rabaey, Massoud Pedram, Francky Catthoor, Suresh Rajgopal, Naresh Sehgal, and Thomas J. Mozden. Power conscious CAD Tools and Methodologies: A Perspective. In *Proceedings of the IEEE*, pages 570–594, April 1995.

[SS99] Giannis Sinevriotis and Thanos Stouraitis. Power Analysis of the ARM 7 Embedded Microprocessor. In *Proceedings of the ninth International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 261–270, October 1999.

[SSR01] Florian Schintke, Jens Simon, and Alexander Reinefeld. A Cache Simulator for Shared Memory Systems. In *Computational Science - ICCS 2001, appeared in Lecture Notes in Computer Science, volume 2074*, pages 569–578, May 2001.

[SSWM01] Stefan Steinke, Rüdiger Schwarz, Lars Wehmeyer, and Peter Marwedel. Low power code generation for a RISC processor by register pipelining. Technical Report 754, Embedded Systems Group, Department of Computer Science XII, University of Dortmund, March 2001.

[Ste03] Stefan Steinke. *Analysis of the energy saving potential in embedded systems through energy optimizing compliation techniques (Untersuchung des Energieeinsparungspotenzials in eingebetteten Systemen durch energieoptimierende Compilertechnik) (in German language).* PhD thesis, Embedded Systems Group, Department of Computer Science XII, University of Dortmund, April 2003.

[SWLM02] Stefan Steinke, Lars Wehmeyer, Bo-Sik Lee, and Peter Marwedel. Assigning Program and Data Objects to Scratchpad for Energy Reduction. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, pages 409–415, March 2002.

[Sym] Symphony EDA. VHDL Simili. http://www.symphonyeda.com/products.htm.

[Syn] Synopsys Inc. Discovery Verification Platform - VCS http://www.synopsys.com/products/simulation/simulation.html.

[Syn96] Synopsys Inc. *Power Products Reference Manual Version 3.5*. Synopsys, 1996.

[TFW00] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and Precise WCET Prediction by Separated Cache and Path Analyses. *Real-Time Systems*, 18(2/3):157–179, May 2000.

[The00] Michael Theokharidis. Measuring Energy consumption of ARM7-TDMI Processor Instructions (Energiemessung von ARM7TDMI Prozessor-Instruktionen) (in German language). Master's thesis, Embedded Systems Group, Department of Computer Science XII, University of Dortmund, November 2000.

[TL98] Vivek Tiwari and Mike Tien-Chien Lee. Power Analysis of a 32-bit Embedded Microcontroller. *VLSI Design Journal*, 7(3), 1998.

[TM04] Yudong Tan and Vincent Mooney. Integrated Intra- and Inter-task Cache Analysis for Preemptive Multi-tasking Real-Time Systems. In *Proceedings of the 8th Workshop on Software and Compilers for Embedded Systems (SCOPES), in: Lecture Notes on Computer Science, LNCS3199*, pages 182–199, September 2004.

[TMW94a] Vivek Tiwari, Sharad Malik, and Andrew Wolfe. Compilation Techniques for Low Energy: An Overview. In *Proceedings of the IEEE Symposium on Low Power Electronics (SLPE)*, October 1994.

[TMW94b] Vivek Tiwari, Sharad Malik, and Andrew Wolfe. Power Analysis Of Embedded Software: A First Step Towards Software Power Minimization. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 384–390, November 1994.

[TMW96] Vivek Tiwari, Sharad Malik, and Andrew Wolfe. Instruction Level Power Analysis and Optimization of Software. *Journal of VLSI Signal Processing Systems, Special issue on technologies for wireless computing*, 13(Issue 2-3):223–238, August/September 1996.

[Tri] Trimaran. Trimaran Homepage. http://www.trimaran.org.

[TY96] Hiroyuki Tomiyama and Hiroto Yasuura. Optimal code placement of embedded software for instruction caches. In *Proceedings of the European Design and Test Conference (ED&TC)*, pages 96–101, March 1996.

[UNS02] Sumesh Udayakumaran, Bhagi Narahari, and Rahul Simha. Application Specific Memory Partitioning for Low Power. In *Proceedings* *of the Workshop on Compilers and Operating Systems for Low Power (COLP)*, pages 03.1 – 03.8, September 2002.

[VSM03] Manish Verma, Stefan Steinke, and Peter Marwedel. Data Partitioning for Maximal Scratchpad Usage. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 77–83, January 2003.

[VWM04a] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Cache Aware Scratchpad Allocation. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, pages 1264–1269, February 2004.

[VWM04b] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Dynamic Overlay of Scratchpad Memory for Energy Minimization. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 104–109, September 2004.

[VWM04c] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Efficient Scratchpad Allocation Algorithms for Energy Constrained Embedded Systems. *Proceedings of the Workshop on Power Aware Computer Systems (PACS), published in Lecture Notes on Computer Science (LNCS), Volume 3164*, 3164:41–56, January 2004.

[WHM04] Lars Wehmeyer, Urs Helmig, and Peter Marwedel. Compiler-optimized Usage of Partitioned Memories. In *Proceedings of the 3rd Workshop on Memory Performance Issues (WMPI2004), published ACM International Conference Proceedings Series*, pages 114–120, June 2004.

[WJ94] Steven J.E. Wilton and Norman P. Jouppi. An Enhanced Access and Cycle Time Model for On-Chip Caches. Technical Report 93/5, Western Research Laboratory, July 1994.

[WJ96] Steven J.E. Wilton and Norman P. Jouppi. CACTI: An Enhanced Cache Access and Cycle Time Model. *IEEE Journal of Solid-State Circuits*, 31(5):677–688, May 1996.

[WJS$^+$01] Lars Wehmeyer, Manoj K. Jain, Stefan Steinke, Peter Marwedel, and M. Balakrishnan. Analysis of the Influence of Register File Size on Energy Consumption, Code Size and Execution Time. *IEEE Transactions on Computer Aided Design, Special Issue on Software and Compilers for Embedded Systems*, 20(11):1329–1337, November 2001.

[WL02a] Jens Wagner and Rainer Leupers. A Fast Simulator and Debugger for a Network Processor. In *Proceedings of Embedded Intelligence*, February 2002.

[WL02b] Jens Wagner and Rainer Leupers. Advanced Code Generation for Network Processors with Bit Packet Addressing. In *Proceedings of the Workshop on Network Processors (NP1)*, pages 91–115, February 2002.

[WM95] William A. Wulf and Sally A. McKee. Hitting the Memory Wall: Implications of the Obvious. *ACM Computer Architecture News*, 23(1): 20–24, March 1995.

[WM04] Lars Wehmeyer and Peter Marwedel. Influence of Onchip Scratchpad Memories on WCET prediction. In *Proceedings of the Workshop on Worst Case Execution Time Analysis at the ECRTS conference*, pages 29–32, June 2004.

[WM05] Lars Wehmeyer and Peter Marwedel. Influence of Memory Hierarchies on Predictability for Time Constrained Embedded Software. In

Proceedings of the Design, Automation and Test in Europe Conference (DATE), pages 600–605, March 2005.

[XMBH] Min Xu, Milo Martin, Doug Burger, and Mark Hill. WWW Computer Architecture Page. http://www.cs.wisc.edu/˜arch/www.

[YTIE97] Hiroto Yasuura, Hiroyuki Tomiyama, Akira Inoue, and Fajar N. Eko. Embedded System Design Using Soft-Core Processor and Valen-C. In *Proceedings of the 4th Asia Pacific Conference on Hardware Description Languages (APCHDL)*, pages 121–130, August 1997.

[ZKSI03] Wei Zhang, Mahmut Kandemir, Anand Sivasubramanian, and Mary J. Irwin. Performance, Energy, and Reliability Tradeoffs in Replicating Hot Cache Lines. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 309–317, October 2003.

[ZVSM94] Voijin Zivojnovic, J. Martinez Velarde, Chris Schläger, and Heinrich Meyr. DSPstone: A DSP-Oriented Benchmarking Methodology. In *Proceedings of the International Conference on Signal Processing Applications and Technology (ICSPAT)*, pages 715–722, October 1994.

Index