Dynamic Scratchpad Memory Management
based on Post-Pass Optimization
Post-Pass 최적화를 통한 Scratchpad 메모리용 동적 관리 기법

2008년 2월

서울대학교 대학원
전기·컴퓨터공학부
Bernhard Egger
Dynamic Scratchpad Memory Management
based on Post-Pass Optimization

Post-Pass 최적화를 통한 Scratchpad 메모리용 동적 관리 기법

지도교수 이재진

이 논문을 공학박사 학위논문으로 제출함

2007년 10월

서울대학교 대학원
전기컴퓨터공학부
Bernhard Egger

Bernhard Egger의 공학박사 학위논문을 인준함

2007년 11월

위원장

부위원장 이재진

위원장 심현식

위원 127

위원 THOMAS GROSS

위 원
Abstract

Contemporary portable devices are getting more powerful and include an ever increasing number of features, yet, despite advancing in terms of usability and functions, battery life still remains a major concern. There exists a plethora of techniques to reduce the energy consumption of battery-powered devices, for example, putting the processor core into a sleep state, reducing the processor’s supply voltage, or dim the backlight after a certain period of inactivity.

In this thesis, we focus on reducing the energy consumption of the memory system by exploiting scratchpad memory (SPM) on embedded processor cores. Scratchpad memories as a replacement or addition to traditional hardware caches are not new. SPM offers a number of advantages over caches that make them interesting for embedded systems. First, accesses to the SPM have a constant and known latency - a property which is important for systems running real-time tasks. Second, thanks to the simpler structure, accessing data in an SPM requires significantly less energy than a set-associative cache of the same size. Therefore, it is not surprising that SPMs
are a hot research topic in the embedded systems area. However, while portable systems have evolved to the point where they run full-featured operating systems with virtual memory and preemptive multitasking, research on SPM allocation techniques has focused mainly on one application to be run on an a priori known hardware configuration.

In this thesis, we introduce dynamic SPM allocation techniques for systems with virtual memory and preemptive multitasking. We show how to generate SPM-optimized binaries that are independent of the processor’s actual SPM size. We achieve this by clustering frequently accessed code and data into a pageable region. At runtime, an SPM manager (SPMM) exploits the memory management unit (MMU)’s page fault exception handler to track accesses to code and data in the pageable region. Whenever a page fault exception occurs, the SPMM loads the requested page into the SPM and resumes execution of the application.

The SPM-optimized binaries are generated by a postpass optimizer. The postpass optimizer classifies basic blocks and data blocks based on profiling information into two regions, pageable and cacheable. For optimal performance, temporally local code and data should be placed in the same page to avoid unnecessary page fault exceptions. The postpass optimizer, therefore, employs several optimization techniques such as loop detection and function splitting to achieve good clustering.

The binary images generated by the postpass optimizer neither depend on the presence nor the size of the SPM. This makes them well-suited for
multitasking environments. While in the single-task scenario the whole SPM is under control of the application, the SPM needs to be managed as a global resource in a multi-process environment. We propose three different SPM sharing strategies for systems with preemptive multitasking and show how they can be integrated into an existing operating system.

Processor cores that contain both cache and SPM often access both memories in each access to keep the latency short. These unnecessary accesses waste a lot of energy. We propose a horizontally partitioned memory system where the address translation is serialized with the memory access. On the instruction side, the original cache is replaced with a big SPM and a small direct-mapped cache, while on the data side, a tiny SPM is added to the original cache.

We have evaluated the dynamic SPM allocation techniques introduced in this thesis on our cycle-accurate processor core simulator with fifteen single-process and ten multi-process benchmarks for portable devices. We analyze code only, data only, and code plus data SPM allocation in a single-process environment and investigate the effect of the MMU page size and the horizontally partitioned memory system on the performance of the proposed techniques. To show the effectiveness of the multi-process SPM sharing strategies, we have implemented a small runtime environment (RTE) with virtual memory and preemptive multitasking.

The obtained results show that the proposed SPM allocation techniques in conjunction with the horizontally partitioned memory system achieve a
significant reduction in energy consumption and a substantial improvement in runtime performance.

**Keywords:** Code placement, compilers, data placement, heterogeneous memory, multitasking, paging, portable systems, postpass optimization, scratchpad, victim cache, virtual memory

**Stud-Nr:** 2003-30778
Contents

Abstract i

1 Introduction 1

1.1 Memory Hierarchies 2

1.1.1 Caches vs. Scratchpad Memories 3

1.2 Motivation 6

1.3 Related Work 9

1.4 Contributions 13

1.5 Organization of this Thesis 16

2 The Horizontally Partitioned Memory Subsystem 17

2.1 Limitations of Existing Memory Subsystems 18

2.2 Horizontally Partitioned Memory Subsystem 20
2.2.1 Replacing the Instruction Cache ........................................... 23
2.2.2 Replacing the Data Cache .................................................. 24

3 Generating SPM-Optimized Images .......................................... 26

3.1 The Postpass Optimizer ......................................................... 27
  3.1.1 Overview ................................................................. 27
  3.1.2 Constant Data .......................................................... 29
  3.1.3 PC-Relative Data Table Accesses ...................................... 30
  3.1.4 Separation of Code and Data .......................................... 32
  3.1.5 Workflow ................................................................. 33

3.2 Placement of Code and Data ................................................. 34
  3.2.1 Code Classification ...................................................... 35
  3.2.2 Data Classification ..................................................... 37

3.3 Pageable Code and Data Placement ....................................... 39
  3.3.1 Code Clustering .......................................................... 39
  3.3.2 Data Clustering .......................................................... 46

3.4 SPM-Optimized Binaries ...................................................... 47

4 Runtime SPM Management ..................................................... 51
  4.1 Single-Process SPM Management ........................................ 51
4.1.1 Runtime SPM Management ...................... 53

4.2 Multi-Process SPM Management .................... 58
  4.2.1 The SPM Manager ............................. 59
  4.2.2 SPM Sharing Strategies ...................... 61
  4.2.3 Real-Time Considerations .................... 70

5 Evaluation Environment ................................. 72
  5.1 Simulation Environment ............................ 72
  5.2 Single-Process SPM Management .................... 75
    5.2.1 Performance Metrics ......................... 75
    5.2.2 Benchmarks ................................ 79
  5.3 Multi-Process SPM Management ..................... 80
    5.3.1 The Runtime Environment .................... 80
    5.3.2 Performance Metrics ......................... 81
    5.3.3 Benchmarks ................................ 82

6 Experimental Results ................................ 86
  6.1 Single-Process SPM Management .................... 86
    6.1.1 Code Placement .............................. 86
    6.1.2 Data Placement .............................. 104
List of Figures

1.1 Cache and SPM architecture ........................................... 3

1.2 Die area and access energy for SPM, a direct-mapped cache,
   and a 4-way set-associative cache .................................. 5

2.1 ARM11 L1 cache block diagram ...................................... 19

2.2 On-chip memory architecture ........................................ 21

2.3 Each TLB entry contains an additional SPM flag ............... 21

3.1 The postpass optimizer ................................................. 28

3.2 Constant data extraction from local constant pools .......... 29
3.3 Separating local data pools from code. (a) original functions with local data pools (b) after separating the local data pools from their functions and grouping code and data independently into pages. (c) When placing the code, data pages are inserted (and cloned), as necessary, in such a way that all references can be resolved.

3.4 Code placement example.

3.5a The code clustering algorithm in pseudocode, first part.

3.5b The code clustering algorithm in pseudocode, second part.

3.6 Data structures of SPM-optimized binaries.

4.1 Virtual memory organization of SPM-optimized binaries.

4.2 Operation of the SPM manager.

4.3 Global SPM sharing strategy.

4.4 Divided SPM sharing strategy.

4.5 Computing a new SPM allocation for the divided SPM strategy depending on the policy. $s_{cur}[i]$ contains the number of blocks currently assigned to process $i$. The policies maximum-workingset and on-demand are also shown.

4.6 Hybrid SPM sharing strategy.
4.7 Moving a shared pool of two blocks from \( p \) to \( q \). (a) before (b) after moving the shared pool. 

5.1 Modified tiny PTE format to accommodate page sizes down to 64 bytes. 


6.2 Memory system: no data cache and no minicache. Code clustering enabled. 

6.3 TLB performance for varying page sizes. 

6.4 Memory system: no data cache and with minicache. Code clustering enabled. 

6.5 The effect of the minicache on SPM-unaware binaries. 

6.6 Memory system: with data cache and minicache. Code clustering enabled. 

6.7 The effectiveness of the thrashing-protection heuristics. 

6.8 Comparison against a direct-mapped cache. 

6.9 Data only placement for a horizontally partitioned memory system with a comparable die area. 

6.10 Code and data placement for a horizontally partitioned memory system with a comparable die area.
6.11 Comparison of a cache-optimized against an SPM-optimized image. ................. 112

6.12 Code and data placement for given hardware configurations. ................. 115

6.13 Energy consumption, throughput, and pagefaults for multi-process benchmarks. ..................... 117

6.14 Energy consumption, throughput, and pagefaults for multi-process benchmarks. ..................... 118
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Cycle Access Times for $\mu$TLB, Caches, and SPM</td>
<td>22</td>
</tr>
<tr>
<td>2.2</td>
<td>Cycle Access Time</td>
<td>22</td>
</tr>
<tr>
<td>2.3</td>
<td>Instruction Side Die Area Requirements of the Horizontally</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>Partitioned Memory Subsystem Compared to Various Instruction Cache Sizes</td>
<td></td>
</tr>
<tr>
<td>2.4</td>
<td>Data Side Die Area Requirements of the Horizontally Partitioned Memory Subsystem Compared to Various Data Cache</td>
<td>25</td>
</tr>
<tr>
<td>4.1</td>
<td>Properties of the Proposed SPM Sharing Strategies. (MWS: Maximum-Workingset Policy, OD: On-Demand Policy, n: Number of SPM-Optimized Processes)</td>
<td>70</td>
</tr>
<tr>
<td>5.1</td>
<td>Access Latencies in CPU Cycles</td>
<td>75</td>
</tr>
<tr>
<td>5.2</td>
<td>Per-Word Access Energy and Power Parameters</td>
<td>78</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Contemporary portable devices are getting more powerful and include an increasing number of features. While just a few years ago, tech-savvy consumers carried around a mobile phone, an MP3 player, a PDA, and a portable TV all as separate devices performing one or two specific functions, these days it is hard to find a mobile phone without a built-in digital camera, an MP3 player, a personal organizer, a web browser, an email client, and many more features.

Despite this ongoing digital convergence, today’s portable devices still do not live up to consumers’ expectations concerning battery life. Not surprisingly is reducing the energy consumption and thereby increasing the running time of these devices a much researched topic, and the possibilities are manifold.
1.1 Memory Hierarchies

In this thesis, we focus on increasing the performance and reducing the energy consumption by optimizing the placement of code and data in the memory hierarchy of the device. Memory systems are typically organized hierarchically with the fastest, most expensive, and usually also smallest memory located closest to the CPU, and the slowest, cheapest, and biggest memory farthest from the CPU. Memories such as static random access memories (SRAM) provide the necessary short access latency and bandwidth to deliver instructions and data at a pace that contemporary CPUs require, but they are expensive. On the other end of the spectrum are hard disks or flash memories that are cheaper but an order of magnitude slower than SRAMs. Designing hierarchical memory systems, therefore, enables system integrators to obtain satisfactory performance while keeping the manufacturing cost reasonable.

The prevalent memory hierarchy in portable devices consists of a non-volatile solid state memory or hard disk, synchronous dynamic random access memory (SDRAM), and the on-chip memory system. The on-chip, or level-one (L1) memory system comprises of SRAM memory implemented as cache, scratchpad memory (SPM), or both.
Many of today’s state-of-the-art processors for mobile and embedded systems feature cache and scratchpad memory. Some examples of processors with SPM are ARMv6 cores [5], the Intel IXP Network Processor [10], Intel’s XScale [17], or Phillips’ LPC3180 microcontroller [32]. Both cache and scratchpad memories are made of SRAM cells. Caches are composed of tag and data arrays plus management logic (Figure 1.1 (a)) that makes them mostly transparent to the software. They automatically store frequently used data on an on-demand basis. Whenever the CPU requests a datum, the hardware cache first checks whether it already has the datum stored. If yes, the datum is returned to the CPU (cache hit), typically with one-cycle latency. If not, a cache miss has occurred, and the cache requests the datum.
from the next level in the memory hierarchy. The latency of a cache miss in embedded systems is usually an order of magnitude slower than a cache hit.

Scratchpad memory (SPM), on the other hand, consists of a simple array of SRAM cells and includes neither a tag array nor complex comparator logic (Figure 1.1(b)). Usually the SPM is mapped into the physical address map as a contiguous block of fast memory. Unlike caches, it is the application or operating system’s (i.e., the programmer’s) responsibility to determine what parts of the code and/or data are placed in the SPM at what points in time.

Regardless of the need of management code, SPM has a number of advantages over a cache. First, data residing in the SPM is returned to the CPU with a short and constant latency. This is in contrast to hardware caches, where a datum may first have to be fetched from the external memory on a cache miss before it can be delivered to the CPU. The constant and a priori known access time of SPMs makes them attractive for real-time systems. Another advantage is that an SPM of a certain size requires less die area than a cache with the same capacity. This is because of the missing tag array and tag comparison logic. The simpler structure of SPM also leads to a reduced energy consumption per access. The reduced energy consumption is especially prominent when comparing an SPM to an $n$-way set-associative cache since these caches access all $n$ tag and data arrays in parallel on each access. Figure 1.2 compares the die area and the energy
Figure 1.2: Die area and access energy for SPM, a direct-mapped cache, and a 4-way set-associative cache.
consumption of an SPM, a direct-mapped and a 4-way set-associative cache for 0.13 \( \mu \text{m} \) technology. These properties make SPM attractive for embedded systems. Placing the most frequently accessed parts of the program into the SPM can reduce both the energy consumption and the execution time of an application [6, 29].

1.2 Motivation

Despite the advantages of SPM, L1 memory is usually implemented as a hardware cache in desktops and contemporary high-end embedded CPUs, such as Intel’s XScale processor [17]. There are two main reasons for this: first, to run an application efficiently on a cached architecture, little or no information about the application’s behavior is necessary. The cache management logic automatically stores recently used data in the cache. The data remains in the cache until it gets replaced with more recent data, i.e., a cache automatically adapts to the application course. This is in contrast to the SPM, where profiling data about frequently executed code blocks and/or frequently accessed data must be obtained and analyzed before an SPM-optimized binary can be generated. Second, binary compatibility is an important property for applications that run on various hardware configurations. However, with very few exceptions [11, 28], SPM allocation techniques require knowledge of the SPM size at compile-time in order to compute the optimal SPM allocation. That is, to accommodate for the various configurations of cores and SPM sizes, application vendors have
to produce several different SPM-optimized binaries to cover all possible configurations. This is not only cumbersome, but also complicates the installation process for the end user.

In embedded systems, on the other hand, the use of scratchpad memory is widespread. Embedded systems often serve a specific purpose. Software is configured and installed on the device before it ships; it is rarely changed thereafter. This allows designers to customize embedded applications to a particular configuration and take full advantage of the SPM.

Contemporary portable devices diverge more and more from embedded systems. The user can download and run applications from the internet, and the device runs a full-featured operating system with a scheduler, virtual memory, and even a file system. Processes are created and destroyed on the user’s demand and at arbitrary times. Furthermore, the varying hardware configurations of the devices make it impractical for applications to be tailored to one specific SPM size. In other words, binary portability is also becoming increasingly important for embedded systems.

Several studies have discussed various SPM allocation techniques. They consider either code, data, or both and put the most beneficial blocks into the SPM to achieve maximum performance or energy savings. Static SPM allocation techniques set up the contents of the SPM when the application starts running and do not modify them afterwards, while dynamic techniques copy blocks to and from the SPM to adapt to the course of the application at runtime. For most studies, the size of the SPM must be
known at compile time, and only very few consider multiple tasks.

In this thesis, we present dynamic SPM allocation techniques for code and data that do not require knowledge of the SPM size at compile-time. We show that by replacing both the on-chip instruction and data caches by an SPM and a smaller cache for single as well as multi-process situations a substantial reduction in energy consumption and improvement in runtime performance are possible. We discuss SPM-sharing strategies and show how they can be integrated into existing operating systems with preemptive multitasking and virtual memory. To make our approach independent of the availability of the application’s source code, we have developed a postpass optimizer that generates the SPM-optimized application binaries based on profiling information. The presented techniques require that the SPM is physically addressed and can be mapped into the virtual address space. While such architectures exist (for example, ARM11 [5] cores), we introduce a horizontally partitioned memory subsystem replacing the on-chip instruction and data cache in order to further reduce the energy consumption.

Our method is outlined as follows: A postpass optimizer generates application binaries optimized for memory systems with both SPM and caches. Independent of the SPM size and based solely on profiling information, it classifies the code and data of an application binary into a pageable and a cacheable region. The former is copied on demand to the SPM before execution. The latter, the cacheable region, is placed at a fixed location in the
external memory and cached by the respective cache. Since the code/data classification is independent of the SPM size, the generated SPM-optimized binaries are portable across varying hardware configurations. At runtime, the instruction and data SPMs are managed by an SPM manager (SPMM). It allocates both SPMs to the running processes depending on the SPM sharing strategy, tracks page accesses by intercepting MMU pagefault exceptions, and copies frequently executed pages to the SPMs on-demand. Both the pageable region and the SPMs are logically divided into pages the size of one MMU memory page.

1.3 Related Work

Existing work on SPM allocation can be roughly divided into two classes: statically allocated and dynamically managed scratchpad memories. Static SPM allocation techniques initialize the scratchpad memory with the designated program parts at load time. The contents of the SPM do not change at runtime. In dynamic SPM allocation techniques the contents of the SPM change while the program executes. The program points at which code and data blocks are moved back and forth from the SPM to the main memory are determined at compile-time. Both static and dynamic SPM allocation techniques can be further classified into approaches that consider only code, only data, or both.

Static SPM allocation techniques are presented in [1, 2, 5, 28, 39]. Except
for Nguyen’s work [28], all of these techniques require knowledge of the SPM size at compile time. Angiolini et al. [1][2] present SPM allocation schemes that select code blocks which promise the highest energy savings using an algorithm based on dynamic programming. While [1] requires special hardware support to split the SPM into several partitions, [2] uses a postpass optimizer to modify the necessary instructions so that the application runs on a unified SPM. Banakar et al. [6] solve the static assignment with a knapsack algorithm, both for code and data blocks. Verma et al. [39] select memory objects based on a cache conflict graph obtained through cache hit/miss statistics. The optimal set of memory objects is obtained by solving an integer linear program (ILP) variant of the knapsack algorithm. In [28], Nguyen et al. delay the decision which blocks should go to the SPM until the application is loaded, making their approach independent from the actual scratchpad memory size. Some profiling information has to be embedded into the application binary, but the authors report only a minimal increase in image size.

Dynamically allocated SPM algorithms are presented in [7][10][11][18][19][20][22][36][37]. Kandemir et al. [19][20] focus on data arrays accessed from well-structured loop kernels. Arrays are split into tiles that are transferred to the SPM independently, so that arrays that are bigger than the size of the SPM can be allocated. Also Li et al. [22] assign data arrays to the SPM. To determine the most beneficial set, the authors split the SPM into several chunks of different sizes. The allocation of data blocks to the SPM chunks is computed by a graph-coloring algorithm. In [36], Steinke et al.
present a technique that dynamically copies code blocks to the SPM. An ILP computes the optimal set of blocks. Udayakumaran et al. \cite{37} focus on performance optimization and consider local and global data. They construct a *data program relationship graph (DPRG)* from the program’s control-flow graph (CFG), which then guides a greedy heuristical algorithm to determine the most promising candidates. In \cite{10}, Egger et al. copy loop nests on demand to the SPM. Also in their approach, an ILP computes the most beneficial set of loops for a given SPM size. Janapsatya et al. \cite{18} introduces the so-called *concomitance* metric which indicates how correlated in time the execution of code blocks is. Blocks with a strong correlation are copied together to the SPM at runtime. Finally, Dominguez et al. \cite{9} propose an SPM allocation scheme for heap data. Promising candidates are assigned a fixed-size *bin* that can hold up to \( n \) elements of a dynamically allocated variable. At runtime, the heap manager allocates an object to the SPM only if there is free space in its predetermined bin.

The horizontal partitioning of memory architectures has recently been examined in \cite{7,11,34}. Inspired by the memory architecture of the Intel XScale architecture with a large main data cache and a 2 KB minicache, Shrivastava et al. \cite{33} show that by cleverly allocating the data objects to one of the caches, a substantial amount of energy can be saved. Egger et al. \cite{11} present a dynamic SPM allocation technique for a horizontally partitioned memory system consisting of an SPM and a small cache. At runtime, an SPM manager intercepts MMU page faults to load frequently executed code into the SPM on demand which makes their approach inde-
dependent of the SPM size. Cho et al. propose a similar approach for the data side. After profiling accesses to global data and stack, an ILP model computes the optimal location of each data block for each function. At runtime, blocks are mapped to the SPM using the MMU.

To this day, the use of SPM in multitasking environments has not been widely studied. Poletti et al. propose an API that helps the programmer move blocks back and forth between the SPM and the main memory, thereby placing the burden of deciding which blocks should go to the SPM on the programmer. Verma et al. chose an automatic approach. They present three sharing strategies: non-saving, saving, and hybrid. The non-saving strategy divides the scratchpad evenly between the applications, and no runtime support is needed. In the saving approach, the whole scratchpad memory is given to the currently active task. The SPM is treated as part of a process’ context, i.e., the contents of the SPM are saved and restored at each task switch. The hybrid approach is a mixture of the non-saving and the saving method where parts of the SPM are assigned exclusively to the processes and a common area that is shared and needs to be saved/restored at each task switch. They chose a static allocation which makes this approach unsuitable for dynamic process creation/destruction.

In the unified second-level cache is replaced by SRAM that is managed by software, and the main memory serves as a paging device for the SRAM. Pages are copied in (and out) of the SRAM whenever one of the first-level caches misses. In principle, it is possible to apply the same con-
cept to the first-level caches, however, since the latency of the first-level cache is much more critical than that of the second-level one, unmodified software will probably not run at a satisfactory performance anymore. Applying the code clustering techniques described in our work would be one possibility to overcome this limitation.

The SPM management techniques presented in this thesis are independent of the SPM size at compile-time. A postpass optimizer classifies code and data that is likely to reduce the energy consumption when placed in the SPM and puts it into a specific region in the binary image. At runtime, the SPM manager decides if and which pages to load. The proposed multi-process SPM sharing strategies support dynamic creation/destruction of processes which makes them well suited for portable devices such as smartphones with varying hardware configurations where the processes are started and ended on the user’s demand.

1.4 Contributions

The contributions of this thesis are as follows.

- We introduce a dynamic SPM allocation technique that loads pages on demand. Thus, our approach is independent of the SPM size. The SPM-optimized binaries generated by our postpass optimizer are binary portable across various hardware configurations. At runtime, thrashing-protection heuristics attempt to minimize thrashing so that
the applications run efficiently even on smaller SPM sizes.

- We propose a horizontally partitioned memory system for contemporary embedded processors with an MMU. The instruction cache is replaced with a direct-mapped, physically addressed minicache and scratchpad memory with one-cycle access latency supporting CPU clock frequencies of up to 1.5 GHz. The presence of the minicache enables SPM-unaware programs to run with reasonable performance. To the best of our knowledge, this work, along with our preliminary results [1], presents the first approach to access physically addressed SPM in a virtual memory environment.

- By using a postpass optimizer, we are able to generate SPM-optimized binaries to which the source code is not readily available. Unlike previous work, the SPM-optimized binaries run unmodified with no performance degradation on systems without any SPM. We, therefore, achieve total memory architecture independence: SPM-optimized application binaries run on processors with or without SPM, and the proposed memory architecture runs SPM-optimized as well as SPM-unaware binaries.

- We show that by using the data cache as a victim buffer for the SPM, considerable additional energy savings are possible. We provide an in-depth analysis of the effect of the MMU’s page size on our dynamic SPM management technique.
• We introduce an SPM management technique for multitasking systems with dynamic process creation and destruction. The proposed technique is independent of both the number of processes and the concrete hardware configuration.

• We develop, analyze, and implement three different SPM sharing strategies for multitasking systems and analyze them in terms of performance, algorithmical complexity, and task-switching overhead.

We evaluate the proposed dynamic SPM allocation technique on our cycle-accurate ARM9E-S simulator [35] that has been extended to include a model for the horizontally partitioned memory subsystem. For the evaluation, we use fifteen embedded applications, including an H.264 video decoder, the standard ISO MP3 decoder, an MPEG-4 video encoder/decoder, a public-key encryption/decryption program (PGP), and several applications from MediaBench [21] and MiBench [13]. We analyze the effect of the MMU page size and discuss code only, data only and code plus data SPM allocation. To evaluate the multi-process SPM sharing strategies, we have implemented a small RTE with virtual memory and preemptive multitasking.

With a single process and code only SPM allocation, we achieve a 35% reduction in energy consumption and a 31% improvement in runtime performance on the horizontally partitioned memory architecture with an MMU page size of 256 bytes compared to a fully cached system. With data only SPM allocation, the energy consumption is reduced by 10% and the run-
time performance increases by 17%. For code and data SPM allocation, the reduction in energy consumption is 40% and the runtime performance improvement 34%.

To evaluate the multi-process SPM sharing strategies, we run multi-process benchmarks comprising of several single-process SPM-optimized applications. We compare the energy consumption and throughput of the horizontally partitioned memory system with a fully cached processor core. For the overall best multi-process strategy, the global SPM sharing strategy, we achieve a 47% improvement in throughput and a 32% reduction in energy consumption.

1.5 Organization of this Thesis

The remainder of this thesis is organized as follows: Chapter 2 discusses limitations of existing memory systems and presents a horizontally partitioned memory subsystem and its implementation. Chapter 3 explains the process of generating SPM-optimized binaries using our postpass optimizer. In chapter 4, single-process SPM management techniques for code and data are discussed, and SPM management techniques for operating systems with preemptive multitasking are introduced. Chapter 5 explains the evaluation environment including the simulator and the benchmark applications. Chapter 6 presents the results. Finally, chapter 7 concludes this thesis.
Chapter 2

The Horizontally Partitioned Memory Subsystem

Many of today’s state-of-the-art processors for portable devices contain both cache and scratchpad memory. Some examples of processors with cache and SPM are ARMv6 cores [5], the Intel IXP Network Processor [16], Intel’s XScale [17], or Phillips’ LPC3180 microcontroller [32]. However, all of these processors suffer from one of two shortcomings: either, the SPM access latency is higher than that of the cache when the MMU is turned on, or both memories are accessed in parallel.

Both flaws severely affect the usefulness of SPM with regard to runtime performance and energy reduction. Since the cache access latency in case of a hit is usually one cycle, a higher SPM access time will result in at least double the latency compared to a cache hit which strongly affects the
performance of the application. Similarly, accessing both memory structures in every access to provide one-cycle access latency hurts the energy consumption since a datum can only reside in one of the memories.

To obtain maximal energy savings, we propose a horizontally partitioned memory system that serializes the address translation with the SPM/cache access. In this chapter, we show that such a memory hierarchy is feasible and implementable with minimal effort.

### 2.1 Limitations of Existing Memory Subsystems

Existing memory subsystems of embedded cores that support physically addressed scratchpad memory mapped into a virtual address space are restricted in one of two ways: either the SPM access latency is longer than one cycle, or the SPM is accessed simultaneously with the cache in each request. These double accesses waste energy because only one of the memory structures can contain the requested datum.

In the first case, the MMU first translates the virtual address (VA) to a physical address (PA). The PA is then compared to the SPM base register, and the SPM is accessed only if the PA lies within the SPM’s address range. Normally, the VA-to-PA address translation requires one cycle if the translation lookaside buffer (TLB) hits. An SPM access can thus be handled
in two cycles: one cycle for the address translation and one cycle for the actual SPM access. Our experiments with an ARM926EJ-S development board show that the instruction SPM latency is one cycle when the MMU is turned off and two cycles when it is turned on. The additional cycle is caused by the foregoing VA-to-PA translation.

Other designs, such as the ARM11 core [5], access cache and SPM simultaneously (Figure 2.1). At the same time, the address translation is performed by a MicroTLB (μTLB) [26], which is basically a fully associative cache with 2 to 16 entries providing fast lookups of recently used page table entries. The usual cache hit signals plus special SPM range hit signals are then used to select the correct datum from one of the cache sets or the SPM. While the latency of the cache and the SPM is one cycle, both are active in every memory request, which wastes a significant amount of energy. The XScale’s horizontally partitioned, virtually addressed cache architecture seems to suffer from the same problem [17].
2.2 Horizontally Partitioned Memory Subsystem

In [11], we have proposed a horizontally partitioned, on-chip memory subsystem for the instruction side of a Harvard architecture. The original 4-way set-associative instruction cache is replaced by an SPM and a direct-mapped minicache. In this thesis, we extend the idea to the data side where we place a small SPM alongside with the original 4-way set-associative cache (Figure 2.2). Both SPMs and caches are physically addressed. The address translation is serialized with the SPM/cache access. The translated VA, the PA, is sent either to the cache or the SPM depending on the SPM flag (see below). The serialization of the address translation and the SPM/cache access enables the memory system to fetch the datum from the correct memory, and thereby eliminating the unnecessary duplicated memory access mentioned above.

When fetching an instruction or reading/writing a datum, the \( \mu \)TLB first translates the VA of the instruction/datum into a physical address. The SPM flag stored in the VA’s TLB entry determines whether the instruction/datum is to be loaded from the SPM or the cache (Figure 2.3). If the SPM flag is set, the address is located in the SPM address range, and only the SPM is accessed. If the SPM flag is clear, then the instruction/datum must be loaded from the cache, in which case the SPM is not accessed. Since the SPM flag is evaluated inside the \( \mu \)TLB along with the
VA-PA-address pairs, the the time- and energy-wise more expensive full 32-bit address comparison of the physical address with the SPM base address register performed in current cores is only necessary when the main TLB misses (Figure 2.2). The SPM flag is computed by the MMU for each address translation. Therefore, the entries of the unified TLB must also contain the SPM flag.

Compared to a traditional cache, serializing the address translation with the SPM or cache access increases the latency of an instruction fetch/data access by approximately the access time of the $\mu$TLB. With the current 0.13-µm manufacturing process, however, core clocks of up to 1.5 GHz can easily be supported with a one-cycle latency. Table 2.1 shows cycle access times.
Table 2.1: Cycle Access Times for $\mu$TLB, Caches, and SPM

<table>
<thead>
<tr>
<th>Memory structure</th>
<th>Cycle access time [ns]</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-entry $\mu$TLB</td>
<td>0.22</td>
</tr>
<tr>
<td>16-entry $\mu$TLB</td>
<td>0.22</td>
</tr>
<tr>
<td>cache, 512B, direct-mapped, 16B lines</td>
<td>0.21</td>
</tr>
<tr>
<td>cache, 1KB, direct-mapped, 16B lines</td>
<td>0.22</td>
</tr>
<tr>
<td>cache, 2KB, direct-mapped, 32B lines</td>
<td>0.22</td>
</tr>
<tr>
<td>cache, 4KB, direct-mapped, 32B lines</td>
<td>0.23</td>
</tr>
<tr>
<td>cache, 8KB, direct-mapped, 32B lines</td>
<td>0.25</td>
</tr>
<tr>
<td>cache, 16KB, direct-mapped, 32B lines</td>
<td>0.33</td>
</tr>
<tr>
<td>cache, 512B, 4-way, 16B lines</td>
<td>0.27</td>
</tr>
<tr>
<td>cache, 1KB, 4-way, 16B lines</td>
<td>0.28</td>
</tr>
<tr>
<td>cache, 2KB, 4-way, 32B lines</td>
<td>0.28</td>
</tr>
<tr>
<td>cache, 4KB, 4-way, 32B lines</td>
<td>0.29</td>
</tr>
<tr>
<td>cache, 8KB, 4-way, 32B lines</td>
<td>0.31</td>
</tr>
<tr>
<td>cache, 16KB, 4-way, 32B lines</td>
<td>0.32</td>
</tr>
<tr>
<td>SPM, 1KB</td>
<td>0.21</td>
</tr>
<tr>
<td>SPM, 2KB</td>
<td>0.21</td>
</tr>
<tr>
<td>SPM, 4KB</td>
<td>0.24</td>
</tr>
<tr>
<td>SPM, 8KB</td>
<td>0.32</td>
</tr>
<tr>
<td>SPM, 16KB</td>
<td>0.35</td>
</tr>
</tbody>
</table>

Table 2.2: Cycle Access Time

<table>
<thead>
<tr>
<th>8-entry $\mu$TLB</th>
<th>0.22ns</th>
</tr>
</thead>
<tbody>
<tr>
<td>16KB SPM</td>
<td>0.35ns</td>
</tr>
<tr>
<td>512B direct-mapped cache</td>
<td>0.21ns</td>
</tr>
<tr>
<td>Total latency ($\mu$TLB → SPM or minicache)</td>
<td>0.57ns</td>
</tr>
</tbody>
</table>

(i.e., the minimal time between two subsequent requests) for the $\mu$TLB, and various cache and SPM sizes. Table 2.2 lists a concrete example of a horizontally partitioned memory hierarchy with an 8-entry $\mu$TLB in front of a 16 KB SPM and a 512-byte direct-mapped cache. The numbers were obtained with CACTI [40].

Memory systems that feature both a cache and an SPM in a setup similar to the ARM11 architecture [3] (Figure 2.1) can be easily modified. However, as we will show in the results section (Chapter 6), significant energy savings
can also be achieved by replacing the original cache architecture with a horizontally partitioned memory system. The following sections explain why the optimal configurations for the instruction and the data side are different.

2.2.1 Replacing the Instruction Cache

Based on trace profiles, code of applications typically run on portable systems can be well classified into frequently and infrequently executed code. It is therefore possible to dynamically load almost all frequently executed code into the SPM and only a relatively small number of fetches to infrequently executed code are not covered by the SPM. While replacing the original cache with a large SPM results in a simpler design that yields good results for SPM-optimized applications, the small cache serves an important purpose when running SPM-unaware binaries or when the size of the available SPM is so small that parts of the frequently executed code cannot be loaded into the SPM to avoid frequent copy-in/copy-out operations. For the instruction side, we therefore replace the original instruction cache with a 512- or 256-byte direct-mapped minicache and place as much SPM into the remaining space as possible.

Because of the missing cache control logic and the simpler design, scratchpad memories are more efficient than caches in terms of both energy and die area. We can therefore replace the cache with scratchpad memory that is larger than the original cache and add a 512-byte minicache, yet still achieve
Table 2.3: Instruction Side Die Area Requirements of the Horizontally Partitioned Memory Subsystem Compared to Various Instruction Cache Sizes

<table>
<thead>
<tr>
<th>Instruction Cache configuration</th>
<th>SPM size area [KB] [mm²]</th>
<th>Minicache size area [KB] [mm²]</th>
<th>Total area [mm²]</th>
<th>Die area reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1KB, 4-way, 16B lines</td>
<td>0.37</td>
<td>512 0.10</td>
<td>0.32</td>
<td>13%</td>
</tr>
<tr>
<td>2KB, 4-way, 32B lines</td>
<td>0.42</td>
<td>512 0.10</td>
<td>0.39</td>
<td>7%</td>
</tr>
<tr>
<td>4KB, 4-way, 32B lines</td>
<td>0.53</td>
<td>512 0.10</td>
<td>0.49</td>
<td>8%</td>
</tr>
<tr>
<td>8KB, 4-way, 32B lines</td>
<td>0.73</td>
<td>512 0.10</td>
<td>0.70</td>
<td>4%</td>
</tr>
</tbody>
</table>

A reduction in the required die area. Table 2.3 lists the configuration of the instruction side of our proposed horizontally partitioned memory system for various cache sizes.

For example, a 4-KB, 4-way set associative cache with a line size of 32 bytes occupies a die area of 0.53 mm². Replacing the 4-KB cache with a 6-KB SPM (0.39 mm²) plus a 512-byte direct-mapped cache with a line size of 16 bytes (0.10 mm²) requires a die area of only 0.49 mm², or 8% less than the cache, and yields a total on-chip memory size of 6.5 KB.

### 2.2.2 Replacing the Data Cache

In contrast to code, data accesses typically show less locality. Furthermore, the data clustering techniques presented in chapter 3 only consider static data, i.e., data that is present in the object files at compile-time either as real data or as zero-initialized data. A large portion of the data accesses, however, are generated by stack and heap accesses, both of which our technique does not consider. For this reason, the optimal distribution of die area
between scratchpad and cache memory is different from the instruction side. We add a very small SPM (0.5 or 1 KB) and leave the original data cache unchanged. The increased die area requirements are mostly canceled out by the smaller die area occupied by the horizontally partitioned memory system on the instruction side.

Table 2.4: Data Side Die Area Requirements of the Horizontally Partitioned Memory Subsystem Compared to Various Data Cache Sizes

<table>
<thead>
<tr>
<th>Data Cache configuration</th>
<th>Data Cache area [mm²]</th>
<th>SPM size [KB]</th>
<th>SPM area [mm²]</th>
<th>Total area [mm²]</th>
<th>Die area increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>1KB, 4-way, 16B lines</td>
<td>0.37</td>
<td>0.5</td>
<td>0.05</td>
<td>0.42</td>
<td>14%</td>
</tr>
<tr>
<td>2KB, 4-way, 32B lines</td>
<td>0.42</td>
<td>0.5</td>
<td>0.05</td>
<td>0.47</td>
<td>12%</td>
</tr>
<tr>
<td>4KB, 4-way, 32B lines</td>
<td>0.53</td>
<td>1</td>
<td>0.10</td>
<td>0.63</td>
<td>19%</td>
</tr>
<tr>
<td>8KB, 4-way, 32B lines</td>
<td>0.73</td>
<td>1</td>
<td>0.10</td>
<td>0.83</td>
<td>14%</td>
</tr>
</tbody>
</table>
Chapter 3

Generating SPM-Optimized Images

Contrary to caches, the contents of the SPM are not automatically managed by the hardware. To efficiently run applications on hardware with SPM, frequently accessed code/data blocks should be copied to the SPM before they are accessed. In this chapter, we describe how our postpass optimizer generates SPM-optimized binaries by first determining the preferred storage location of each code/data block, then clustering temporally local blocks together, and finally generating an SPM-optimized executable image of the application.
3.1 The Postpass Optimizer

This section describes SNACK-pop, the postpass optimizer used for generating the SPM-optimized binary image of an application. It operates on the compiled object files of the application and provides means to analyze code and/or data block access profiles. Code and/or data can be freely relocated within the binary image with SNACK-pop taking care of maintaining a correct control flow and locating the data such that all references can be resolved. The postpass optimizer can also inject code into binary images, for example, an SPM manager into standalone SPM-optimized binaries.

3.1.1 Overview

Using a postpass optimizer has several advantages. First, any binary can be optimized without requiring access to the source code and recompiling the application. Second, a postpass optimizer enables whole program optimization, including libraries, which is impossible at the source level. Finally, since optimizations concerning code layout are of a rather low-level nature, postpass code/data arrangement is well-suited for this purpose. Figure 3.1 shows the organization of our postpass optimizer called SNACK-pop. SNACK-pop is part of our Seoul National University Advanced Compiler tool Kit [35]. It operates on the ARM/Thumb instruction set and includes support for the DSP extensions.

The input to the postpass optimizer are application binaries and libraries
Figure 3.1: The postpass optimizer.

in the ARM ELF file format [3]. SNACK-pop disassembles the object files into code and data segments and resolves all references to symbols. Code blocks are further divided into functions composed of basic blocks. Branches with hard-coded offsets are resolved and replaced by relocation information to enable SNACK-pop to freely relocate code.

The following sections, section 3.1.2 to 3.1.4, discuss in detail a few points that need special attention when disassembling or re-assembling the binary image. Section 3.1.5 describes the complete workflow from the unoptimized ELF object files to the SPM-optimized application binary.
3.1.2 Constant Data

Constant data residing in local data pools requires special attention. Most
ARM compilers place constant data used in a function into the function’s
costant pool. Consider, for example, a function \textit{foo()} that contains a call
to the \textit{strcm}
}
string. If both `foo()` as well as `strcmp()` are executed frequently, both will be dynamically loaded into the SPM at runtime. If `strcmp()` is not currently present in and has to be loaded into the SPM before execution, the code block containing `foo()` (and the constant string) might get evicted from the SPM to make room for `strcmp()`. One of the arguments of `strcmp()`, however, points to the constant string, and accessing it would cause the SPM manager to load the function containing the referenced data (i.e., `foo()`) into the SPM. In the worst case, when only a small amount of SPM is available, this could lead to a situation where the application does not terminate because of thrashing.

To avoid such scenarios, whenever SNAV-POP detects the passing of pointers that point to a function’s constant pool, it extracts the referenced constant data from the function and places it in a global data region. The call is then modified to pass a pointer to the global data (Figure 3.2 (b)).

### 3.1.3 PC-Relative Data Table Accesses

Data accesses to tables using PC-relative addressing also require special care, because the offset between the instruction accessing the data and the data itself must remain constant. This hinders free relocation of basic blocks and data blocks. An example of such a table access is shown below
The `add` instruction computes the starting address of the data table (r0=0x40), which is not necessarily identical to the start address of the actual data (0x50) to be accessed. The following `ldrb` instruction uses the starting address in r0 as the base register for the data access. The problem is that the table offset is computed relative to the location of the `add` instruction. In order to independently move both the code block containing the `add`, `ldrb` sequence as well as the data block containing the table, SNACK-pop inserts a symbol to the beginning of the data table, and adds a relocation with the correct offset to the `add` instruction as follows

```
0x20  add  r0, pc, #0x18 ; = 0x40
0x24  ldrb  r2, [r0, ...]
      ...
      ...
0x50  dcd  0x07031975
0x54  dcd  0x22051975
```

The base address of the table access is now computed relative to the data table, and no longer relative to the `add` instruction. This relocation is resolved when SNACK-pop generates the SPM-optimized binary.

```
0x20  add  r0, pc, {relocation to .datatab - 0x10}
0x24  ldrb  r2, [r0, ...]
      ...

.datatab
0x88  dcd  0x07031975
0x8c  dcd  0x22051975
```
3.1.4 Separation of Code and Data

If a function is copied to the instruction SPM, its local data pool is also moved to the instruction SPM. ARM processors enable data reads (and writes) from and to the instruction SPM, however, such accesses incur an additional one-cycle latency. Furthermore, these local data pools can be of considerable size, e.g., data tables containing precomputed values. To maximize the amount of code in the instruction SPM, it is, therefore, desirable to separate the code from its local data pool, and place the data in a separate memory page that can be mapped to the data SPM/cache independently of the referencing code page. The problem when separating code from data are immediate offsets in instructions. In the ARM instruction set, load/store instructions have a range of +/-4KB, for example.

The postpass optimizer separates a function’s code from its local data pool. It then combines the local data pools of spatially local code into separate memory pages. In the final stage of the image generation when the absolute virtual addresses of the code are known, the postpass optimizer injects those data pages in between the code pages in such a way that no immediate offsets exceeds their range. If a data page has already been placed and is too far away for the immediate offset of an instruction, the postpass optimizer clones that data page and places the clone close to the referring instruction. These clones do not increase the size of the binary image, since the clones exist only in the virtual address space and are mapped to the original physical page. To avoid aliasing problems, only read-only data pools
Figure 3.3: Separating local data pools from code. (a) original functions with local data pools (b) after separating the local data pools from their functions and grouping code and data independently into pages. (c) When placing the code, data pages are inserted (and cloned), as necessary, in such a way that all references can be resolved.

are separated from their functions. Figure 3.3 illustrates the principle.

### 3.1.5 Workflow

To generate an SPM-optimized binary, the original image is first run on a functional simulator with various training inputs. The instruction and data traces of the profile runs are then fed into the postpass optimizer to generate an SPM-optimized binary. It computes the average number of instruction fetches for each basic block and the average number of read and
write operations from/to a data block. For each block, the desired location is computed by solving one of the equations described in section 3.2.Blocks which are to be placed in the SPM are then clustered into memory pages based on temporal locality. This process is described in greater detail in Section 3.3. Once the placement of code and data is done, SNACK-pop generates a new ELF binary and inserts the data structures required by the runtime SPM manager (the region table and the block table, see section 3.4) that contain the location and size of of two regions, the paged and the cached region, as well as additional information on the paged region.

3.2 Placement of Code and Data

In order to efficiently run SPM-optimized applications on the proposed horizontally partitioned memory subsystem, the postpass optimizer separates frequently from infrequently accessed code or data blocks. Since the actual size of the SPM is not known at compile-time, SNACK-pop computes the preferred storage location based on approximate energy per access values. This section describes the computation of the storage location for both code and data blocks and explains how temporally local code and data is clustered together into the same memory page to maximize the runtime performance and minimize the energy consumption.
3.2.1 Code Classification

Based on trace profiles, SNACK-pop determines the preferred location for each block $b_i$ using the following heuristics:

$$ Location_{\text{code}}(b_i) = \begin{cases} \text{cached} & \text{if } E_{\text{cached}}^{\text{code}}(b_i) < E_{\text{paged}}^{\text{code}}(b_i) \\ \text{paged} & \text{otherwise} \end{cases} \quad (3.1) $$

with

$$ E_{\text{cached}}^{\text{code}}(b_i) = A_i \cdot e_{i-spm} + M_{\text{code}} \cdot S_i (e_{\text{ext}} + e_{i-spm}) \quad (3.2) $$

$$ E_{\text{paged}}^{\text{code}}(b_i) = A_i \cdot e_{i-cache}(1 + m_{i-cache} \cdot l_{i-cache} \cdot e_{\text{ext}}) \quad (3.3) $$

where

- $A_i$ number of instructions fetched from block $b_i$
- $S_i$ size of block $b_i$ in words
- $M_{\text{code}}$ average number of page misses for code pages
- $m_{i-cache}$ instruction cache miss ratio
- $e_{i-spm}$ instruction SPM access energy per instruction
- $e_{\text{ext}}$ external memory access energy per instruction
- $e_{i-cache}$ instruction cache access energy per instruction
- $l_{i-cache}$ linesize of the instruction cache

The first term in Eq. (3.2), $A_i \cdot e_{i-spm}$, represents the energy required to execute block $b_i$ from the instruction SPM. The second term, $M_{\text{code}} \cdot S_i (e_{\text{ext}} + e_{i-spm})$, computes the cost of copying block $b_i$ from main memory to the instruction SPM. The empirical factor $M_{\text{code}}$ is used to consider the fact that a code block might get copied to the SPM several times. Note that at this
point, we only consider the pure copy cost and not the additional overhead of the SPM management at runtime because this overhead only occurs once for each page, and not for every single block. The energy consumed when executing block $b_i$ from the instruction cache is computed by Eq. (3.3). Note that both $M_{code}$ and the cache miss ratio $m_{i-cache}$ are empirical factors since the sizes of the instruction SPM and the instruction cache are not known at this point. Similarly, exact values for $e_{i-spm}$, $e_{ext}$, and $e_{i-cache}$ are not known. Our experiments show that as long as the ratio between these three values is reasonable, the classification algorithm produces good results. For this reason, we also do not distinguish between read and write operations for SPM and cache.

The location function is evaluated for each basic block in a function. We then perform function splitting similar to [30]: all basic blocks with $Location_{code}(b_i) = paged$ are extracted from $f_k$ and placed into a separate code segment called $f_k^{paged}$. The remaining blocks, that is those with $Location_{code}(b_i) = cached$, are placed in a code segment labeled $f_k^{cached}$. The reordering might invalidate some fall-through edges in the control flow graph. Additional branch instructions are inserted as needed to restore the correct control flow.

After all functions have been processed, SNACK-pop groups all cached code segments without any further processing into the cached code region. The paged code segments, $f^{paged}$, are clustered into memory pages based on temporal locality. This process is described in detail in section 3.3.
3.2.2 Data Classification

Similar to the code classification presented in the previous section, the post-pass optimizer determines the storage region for each data block $d_i$ using the following heuristics:

$$\text{Location}^\text{data}(b_i) = \begin{cases} \text{cached} & \text{if } E^\text{data}_{\text{cached}}(d_i) < E^\text{data}_{\text{paged}}(d_i) \\ \text{paged} & \text{otherwise} \end{cases}$$

(3.4)

with

$$E^\text{data}_{\text{paged}}(d_i) = A_i \cdot e_{d-\text{spm}} + (1 + w) \cdot M_{\text{data}} \cdot S_i(e_{\text{ext}} + e_{d-\text{spm}})$$

(3.5)

$$E^\text{data}_{\text{cached}}(d_i) = A_i \cdot e_{d-\text{cache}}(1 + m_{d-\text{cache}} \cdot l_{d-\text{cache}} \cdot e_{\text{ext}})$$

(3.6)

where

$A_i$ number of read and write accesses to block $d_i$

$S_i$ size of block $d_i$ in words

$M_{\text{data}}$ average number of page misses for data pages

$m_{d-\text{cache}}$ data cache miss ratio

$e_{d-\text{spm}}$ data SPM access energy per word

$e_{\text{ext}}$ external memory access energy per word

$e_{d-\text{cache}}$ data cache access energy per word

$l_{d-\text{cache}}$ linesize of the data cache

$w$ 0 if the block is read-only, 1 otherwise

The first term in Eq. (3.5), $A_i \cdot e_{\text{spm}}$, represents the energy required to access the data in block $d_i$ from the data SPM. The second term, $(1 + w) \cdot$...
$M_{data} \cdot S_i(e_{ext} + e_{d-spm})$, computes the cost of copying block $d_i$ from main memory to the SPM and back if the block is read-write. Also in the data classification computation, the empirical factor $M_{data}$ is used to consider the fact that a data block might get copied to the SPM and written back to main memory several times. The energy consumed when accessing block $d_i$ from the data cache is computed by Eq. (3.6).

This computation is performed for each global and local data block. Global data blocks are read-only or read-write data blocks located in one of the data (including the zero-initialized) sections of the original binary image. Local data blocks are data blocks contained in functions’ constant pools. Local data blocks are always read-only and typically contain constants and addresses to global variables. Some library functions also place lookup tables in local data blocks. Both local and global data blocks are then extracted from the binary image. While global data blocks can be placed virtually anywhere in the SPM-optimized image, local data blocks need to be located close to the referencing code because of the limited range of immediate values in the instruction encoding. Section 3.3.2 explains the process in more detail.

After all code and data blocks have been processed and their storage location determined, the postpass optimizer applies clustering heuristics to minimize the number of memory pages copied into/out of the respective scratchpad memories as described in the next section.
3.3 Pageable Code and Data Placement

Extracting the frequently accessed code and data blocks and placing them into the pageable code region is one important step to efficiently run an SPM-optimized binary on the proposed horizontally partitioned memory subsystem. However, the temporal relationship between functions and data blocks has not been considered yet. Since the pageable region is logically divided into pages which are loaded as a whole into the SPM on-demand by the SPMM, it is important to place temporally local blocks together into the same memory page to reduce the number of copy in/out operations.

Intuitively, a good block placement algorithm should (1) allocate the pageable blocks into as few pages as possible and (2) cluster temporally local blocks together in as few pages as possible.

The following two sections explain how code, respectively data blocks are clustered into pages.

3.3.1 Code Clustering

The problem of allocating code blocks to as few pages as possible can be mapped to Knapsack, a well-known NP-hard problem. If the temporal relationship between code blocks (i.e., the call graph) is also considered, the problem becomes harder than Knapsack. Therefore, we have developed the following heuristics that work reasonably well for a wide range of benchmarks. Figure 3.4 illustrates the steps of the heuristics on a running
example. The circles represent functions and the size of the function is denoted by the number inside the function. The weight on the edges between two functions denotes the number of dynamic calls. We assume a page size of 128 bytes. Figures 3.5a and 3.5b contain the most important parts of the algorithm in pseudocode form. The algorithm starts with the function called CodeClustering() at the top of the listing.

In the first step, SNACK-pop detects loops by inspecting the dynamic call graph (DCG) (Figure 3.5a, DetectLoops). What is denoted a loop is not a loop in the traditional sense. Informally speaking, if a function \( f \) that is called \( k \) times calls another function, \( g \), \( i \) times with \( i \gg k \), then the function \( f \) most probably contains a loop. It is possible that \( f \) does not contain a loop, but rather calls \( g \) at various locations inside \( f \). However, detecting such code patterns as loops does no harm. On the contrary, it allows us to consider loops that other detection techniques cannot handle.

The formal definition of a loop is as follows: let \( |a \to b| \) denote the weight of the edge \( a \to b \) in the DCG (that is, the number of calls from \( a \) to \( b \)). \( |* \to b| \) denotes the sum of the weights of all incoming edges to \( b \), i.e., the total number of calls to \( b \). We define that a function \( f \) is a loop header if there exists a function \( g \) such that

\[
\frac{|f \to g|}{|* \to f|} \geq \text{threshold}
\]

(i.e., the number of calls \( f \to g \) divided by the number of all incoming calls to \( f \) exceeds a certain threshold value). In Figure 3.4 (b), the functions \( c \) and \( f \) have been identified as loop headers for a threshold value of 5.
Figure 3.4: Code placement example.
CodeClustering

begin
  headers := DetectLoops()
  for each loop header h do
    loop[h] := LoopClosure(h, | s → h |)
  LCG := ComputeLoopCallGraph()
  for each loop l in a depth-first traversal of LCG do
    bin[h] := AssignFunctionsToBins(l, pagesize)
    for each loop l in a depth-first traversal of LCG do
      ReduceFragmentation(l)
      place all functions that are not part of any loop in an extra bin
  end

DetectLoops

begin
  headers := ∅
  for each function f do
    for each callee g of f do
      if |(| f → g | / | s → f |) ≥ threshold | then
        headers := headers ∪ {f}
    return headers
  end

LoopClosure(function f, threshold)

begin
  loop := loop ∪ {f}
  for each callee h of f do
    if |(| f → g | ≥ threshold) then
      loop := loop ∪ LoopClosure(h, threshold)
    return loop
  end

Figure 3.5a: The code clustering algorithm in pseudocode, first part.
ComputeLoopCallGraph
begin
  $LCG := \emptyset$
  for each loop $l$ do
    for each loop $m$ of $f$ do
      if $(l \neq m) \land (m \text{ completely contained in } l)$ then
        $LCG := LCG \cup \{l \rightarrow m\}$
    return $LCG$
end

AssignFunctionsToBins(loop $l$)
begin
  $bin := \emptyset$
  for each function $f$ in $l$ that is not placed in any bin yet do
    $bin := bin \cup \{f\}$
    $bin\text{-maxsize} := \lceil\frac{bin\text{-size}}{\text{pagesize}}\rceil \cdot \text{pagesize}$
  return $bin$
end

ReduceFragmentation(loop $l$)
begin
  for each function $f$ in $l$ do
    candidates := $\emptyset$
    for each inner loop $il$ of $l$ do
      if $(bin[il]\text{-maxsize} - bin[il]\text{-size} \geq f\text{-size})$ then
        candidates := candidates $\cup \{il\}$
      end
    if candidates $\neq \emptyset$ then
      best-fit $f$ into candidates
      $l := l - \{f\}$
      $bin[l]\text{-size} := bin[l]\text{-size} - f\text{-size}$
      $bin[l]\text{-maxsize} := \lceil\frac{bin[l]\text{-size}}{\text{pagesize}}\rceil \cdot \text{pagesize}$
    end
end

Figure 3.5b: The code clustering algorithm in pseudocode, second part.
For each loop header \(hd\), the members of the loop are identified by computing the closure of the loop, \(\text{closure}(hd, hd)\). The closure is recursively defined by

\[
\text{closure}(hd, f) = \bigcup_{h \in H} \text{closure}(hd, h)
\]

with

\[
H := \left\{ h \mid |f \rightarrow h| \geq 1 \right\}
\]

i.e., the loop consists of all functions \(h\) that are called at least as many times as the header of the loop (Figure 3.5a, LoopClosure). In Figure 3.4 (b), \(\text{closure}(c, c) = \{c, e, f, g, h\}\) and \(\text{closure}(f, f) = \{f, h\}\). Note that function \(g\) is not a member of \(\text{closure}(f, f)\) because the number of calls to \(f\), \(|* \rightarrow f|\) is larger than the number of calls from \(f\) to \(g\). Function \(g\) is, however, a member of \(\text{closure}(c, c)\).

After detecting all loops in the DCG, we build the loop call graph (LCG). The LCG is a directed graph with the loops as the nodes and an edge between loop \(l_1\) and \(l_2\) if loop \(l_2\) is an inner loop of \(l_1\) (Figure 3.5b, ComputeLoopCallGraph). Figure 3.4 (c) shows the LCG of the running example with loop \(f\) being an inner loop of loop \(c\).

The LCG is then traversed in a depth-first manner (i.e., the innermost loops are processed first). For each loop \(l_i\), a bin \(b_{l_i}\) is allocated. We insert all functions \(f^{\text{paged}}\) to bin \(b_{l_i}\) that are contained in \(l_i\) and have not yet been allocated to any other bin (Figure 3.5b, AssignFunctionsToBins). After all nodes in the LCG have been processed, the maximum size of each bin is defined by rounding up the sum of all functions in the loop bin to the next
multiple of a memory page

\[ \text{size}_{\max}(b_i) = \left\lceil \sum_{f \in b_i} \frac{\text{size}(f)}{\text{pagesize}} \right\rceil \cdot \text{pagesize} \] (3.7)

here, \text{pagesize} denotes the size of one memory page. Figures 3.4(d) and (e) show the state of the LCG and the associated bins after processing \( f \) and \( c \), respectively. Note that even though loop \( c \) contains the functions \( c, e, f, g \) and \( h \), only \( c, e \), and \( g \) are assigned to bin \( b_c \) because \( f \) and \( h \) have already been placed in bin \( b_f \).

Now, we consider all non-leaf nodes of the LCG (i.e., loops containing inner loops). To reduce the internal fragmentation of the loop bins without destroying the close temporal relationship between the functions in a loop bin, we push functions allocated to the outer loop bin into the bins of its inner loops using the bestfit algorithm \[8\] as long as the size of the inner loop bin \( b_{li} \) does not exceed \( \text{size}_{\max}(b_{li}) \) (Figure 3.5b: ReduceFragmentation).

After no more functions can be pushed to inner loops’ bins, the maximum size of the outer loop bin is recalculated according to Eq. (3.7). In Figure 3.4(f), function \( g \) is placed into \( b_f \) and the maximum size of bin \( b_c \) is reduced to 128 bytes.

Functions that belong to the \textit{paged} region, but are not part of any loop, are placed last. For each unplaced function \( f_k^{paged} \), we follow the DCG up towards the root. For each caller \( g \) encountered on the way up, we compute the loop closure \( \text{closure}(g, g) \) with a threshold of one. If the closure includes both \( f_k^{paged} \) and an existing loop \( l \), we try to place \( f_k^{paged} \) in the bin \( b_l \) of loop \( l \). Any remaining functions are allocated to an extra bin.
It is possible that a loop consists of more than one memory block. In such cases, the functions of the loop are sorted according to the number of fetches per word, \( fpw(f) = \frac{\text{fetch}_f}{\text{size}_f} \). The functions are then placed in descending order of their \( fpw \) value. The memory blocks of the loop are indexed with the so-called loop block index. The loop block index is defined as the pair \( lbi(b_k) = \langle l, i \rangle \). For each block \( b_k \), \( l \) denotes the index of the loop \( l \), and \( i \) denotes the index of the block within loop \( l \). The first memory block of loop 3, for example, has an LBI of \( \langle 3, 1 \rangle \), the second one \( \langle 3, 2 \rangle \), and so on. The LBI is used by the thrashing-protection heuristics of the runtime SPM manager (Section 4.1.4).

### 3.3.2 Data Clustering

For practical reasons, local data is clustered differently from global data. Local data, i.e., data blocks that have been extracted from constant pools of functions, is subject to stringent restrictions concerning its placement. Blocks containing local data must be placed within the limits imposed by the intermediate offsets’ ranges of the referencing instructions. Global data, on the other hand, can be placed freely.

For each loop bin \( b_i \) generated during code clustering (see Section 3.3.1), two data bins are generated: \( db_i^{\text{paged}} \) and \( db_i^{\text{cached}} \). The first, \( db_i^{\text{paged}} \) contains all local data blocks \( db_i \) for which \( Location^{\text{data}}(db_i) = \text{paged} \) holds. Accordingly, \( db_i^{\text{cached}} \) consists of all blocks \( db_i \) with \( Location^{\text{data}}(db_i) = \text{cached} \). Data blocks in \( db_i^{\text{paged}} \) are sorted by the number of accesses per
word $apw(db) = (\text{read}_{db} + \text{write}_{db})/\text{size}_{db}$. The data blocks with the lowest $apw$ value are placed last.

Global data blocks $db_i$ with $Location^{data}(db_i) = paged$ are clustered in two different bins, $db^{paged}_{readonly}$ and $db^{paged}_{readwrite}$. The former, $db^{paged}_{readonly}$, contains read-only data blocks, and the latter read-write blocks. Inside these two data bins, data blocks with the lowest $awp$ value are placed last. The separation of read-only from read-write data reduces the number of blocks that need to be written back to the external memory by the SPMM whenever they are evicted from the SPM.

### 3.4 SPM-Optimized Binaries

After the postpass optimizer has separated frequently from infrequently executed code, determined the location of each block, and clustered code and data blocks based on temporal locality, memory pages that contain code or data to be copied into the respective SPM before accessing them are placed into the pageable region. Because the range of the intermediate offset in load/store instructions is limited, data pages containing cacheable data might also be placed in the pageable region in order to guarantee that the distance between the referencing instruction and the datum does not exceed the range of the intermediate offset. The postpass optimizer computes the final layout of the pageable region and places any other code and data in the cacheable region. If the SPM-optimized binary is intended to run without
any support from a runtime environment, the postpass optimizer optionally injects a small SPM manager (SPMM) into the SPM-optimized binary that manages the SPM of the standalone application at runtime.

SNACK-pop also inserts two additional data structures that allow the SPMM to setup the MMU’s page table mappings when the binary is first loaded and manage the SPM while the application runs. Specifically, the postpass optimizer adds the region table and the block table (Figure 3.6). The region table contains the location and size of both the pageable and cacheable region. The block table contains one entry per block of the pageable region. Each entry contains the type of the block (code, read-only data, or read-write data), the preferred storage location (SPM, or cache), the block’s size, its loop block index (LBI), the virtual address of the block, and the address of the page table entry (PTE) in the pagetable. The type

Figure 3.6: Data structures of SPM-optimized binaries.
field is used by the SPMM to determine which SPM the block should be loaded to (code blocks are loaded into the instruction SPM, whereas read-only and read-write blocks are loaded into the data SPM) and also whether the block needs to be written back to main memory whenever it is evicted from the SPM (read-write data blocks). The storage location field is used to mark cached pages that have been placed in the pageable region only because of immediate offset restrictions of one of the referencing instructions. Because of internal fragmentation some blocks might not be completely filled. Hence, the size field allows the SPMM to copy only as many bytes as needed. The address of the PTE and the virtual address of each block are computed and stored in the block table by the SPMM at runtime when the application is loaded. The loop block index, finally, is used by the SPMM’s thrashing-protection heuristics (Section 4.1.1) to map the least frequently accessed pages of loops that contain more blocks than SPM blocks are available to the cache.

Figure 3.6 shows an example of an SPM-optimized binary and its additional data structures. The SPM-optimized binary image consists of a cached region starting at 0x000, and a paged region at 0x200. The region table contains the type, the virtual address, and the size of each region. The block table contains eight entries, one for each page in the pageable region. Blocks $b_1$ and $b_3$ both belong to loop 1 with $b_1$ being the first block, and $b_3$ the second block of loop 1. The second loop, loop 2, consists of the blocks $b_6$, $b_7$, and $b_8$ (in this order). Block $b_2$ is a read-only data block that will be copied to the SPM on-demand. The same applies to block $b_4$, but
since it is a read-write block, the SPMM will have to write it back to main memory whenever it evicts the block from the SPM. Block $b_5$, finally, will be mapped as cacheable by the SPMM and is never copied to the SPM.
Chapter 4

Runtime SPM Management

When SPM-optimized binaries are run on a system with SPM, an SPM manager (SPMM) loads pages into the SPM on-demand. In this chapter, we describe in detail the dynamic SPM management techniques for single-process, as well as multi-process environments.

4.1 Single-Process SPM Management

In a single-process environment, the running application has complete control over all available resources in the system. On cached cores, binaries run without further software support. On cores equipped with scratchpad memory, however, the SPM has to be explicitly managed either by the running application or by a dedicated SPM manager that is part of the runtime environment.
For embedded systems, the set of running tasks and the size of the SPM are often known when the system is built. For such systems, particularly well-suited SPM management techniques are those where the optimal storage location of each code and/or data block is solved by an integer linear programming (ILP) formulation and the SPM is managed by the running task. These techniques, however, have the disadvantage that the optimized binaries are tailored to exactly one hardware configuration, and might run only inefficiently (or not at all) on different configurations.

To overcome this problem, we propose an SPM management technique that depends neither on a certain size of the SPM nor on a predetermined set of running applications. An SPM manager (SPMM) manages the SPM at runtime as a global resource. The SPMM uses demand-paging techniques similar to those in virtual memory systems [1] to track the course of the application and load pages into the SPM on demand.

An SPM-optimized application contains a special code and data region, the pageable region, several data structures necessary for managing the SPM, and the SPMM. The pageable region consists of code and/or data that should be located in the SPM for maximal energy savings. The SPM management technique presented in this chapter assume that both SPM and cache are available, and are specifically optimized for the horizontally partitioned memory system (see Chapter 2).
After an SPM-optimized binary image has been loaded, control is passed to the SPMM initialization code before the actual application entry point is called. By default, the whole virtual address space is mapped cacheable. The SPMM extracts the location of the pageable region from the region table that is embedded in the SPM-optimized binary and disables the page table mappings to all pages whose block table entry’s location field is set to SPM, i.e., accessing any datum to such a page will trigger an MMU page fault. The global heap and stack are mapped to cacheable memory regions; accesses to heap or stack are covered by the data cache (if present).
The SPMM computes and stores the addresses of the PTEs in the page tables and each block’s virtual address in the block table. Figure 4.1 displays a possible memory allocation after loading an SPM-optimized binary. The cached and the pageable region have been mapped to virtual addresses 0x200, and 0x800, respectively. The addresses of the eight blocks in the pageable region are stored in the block table along with their PTE addresses (shown in italic).

After the binary has been loaded and the page tables have been setup, the application starts running (Figure 4.2 (a) and (b)). As soon as the program counter (PC) reaches code or a memory operation accesses a datum located in the paged region, the memory access fails because the memory is not mapped. The MMU signals a pagefault exception to the CPU which then executes the pagefault execution handler (Figure 4.2 (c)(1)). The SPMM, intercepting the pagefault exception, determines in which page the the fault has occurred, and copies that page to the SPM (Figure 4.2 (c)(2)). It then modifies the PTE and restarts the aborted instruction. (Figure 4.2 (d)(3+4)). The application now runs without further interruption until another disabled memory page is accessed. It is worthwhile noting that as long as a page is present in the SPM, its PTE remains valid and accessing that page will not generate a pagefault, i.e., no additional cost occurs when the course of the application accesses an already loaded page.

Usually, the number of pages in the paged region exceeds the number of available pages in the SPM. Therefore, pages residing in the SPM may need
Figure 4.2: Operation of the SPM manager.
to be evicted before a new page can be loaded into the SPM. If the evicted page is a code page or a read-only data page, the SPMM does not need to copy the page back to main memory; it simply overwrites the old page with the contents of the new one. In the case of a read-write data page, the SPMM stores the page back to main memory before it is overwritten in the SPM. The PTE of the evicted page is invalidated to trigger another page fault as soon as that page is accessed again. For this purpose, the SPMM keeps track of which pages in the SPM are occupied and which pages are free. The SPMM uses a round-robin policy for page replacement.

If the hardware has both an instruction SPM as well as a data SPM, the SPMM manages both SPMs independently. Code blocks are loaded into the instruction SPM, and data blocks are stored in the data SPM.

**Shadow Copies of the Pageable Region**

To copy an unmapped page into one of the SPMs with a sequence of load/store instructions, the SPMM needs to be able to access the unmapped block. To avoid a lot of expensive MMU page table operations, the SPMM maintains a *shadow copy* of the pageable area. The shadow copy is located at a different virtual address than the pageable region, but points to the same physical memory. The shadow copy is mapped uncachable and enables the SPMM to access unmapped blocks in the virtual address space. If the data cache is used as a victim buffer for pageable code blocks (see Section 6.1.1), then a second, cacheable shadow copy is maintained.
Thrashing-Protection Heuristics

The postpass optimizer classifies code into pageable and cacheable regions without knowing the actual SPM size. If the number of blocks in the pageable regions exceeds the number of blocks available in the SPM, thrashing can occur. An application thrashes if newly loaded pages trigger the removal of pages residing in the SPM only to be accessed shortly after. If an application thrashes, lots of page faults and page copy-in/copy-out operations severely affect both the runtime performance as well as the energy consumption.

The SPMM contains thrashing-protection heuristics to protect against simple forms of thrashing: The working set of a loop consists of all pageable blocks of that loop. The information which block belongs to which loop is stored in the block table’s LBI field. To protect the application from thrashing while executing a loop, the SPMM maps all blocks $b_k$ where the block index $i$ in $lb_i(b_k) =< l, i >$ is larger than $N_{SPM}$, the number of SPM pages, to the cache. This reduces the number of pageable blocks in the working set of the loop to the exact size of the SPM. If a loop $l$ contains (possibly several) inner loops $il$, the LBI of its first block can be set to one or to $1 + \max(lb_i(b_i)) \ \forall b_i \in il$, i.e., to the size of its biggest inner loop plus one. In the former case, inner loops are not considered and the LBIs of outer loops are low. This setting can still lead to thrashing if the outer loop is executed frequently. In the latter case, loops with big inner loops get assigned high LBIs. Thrashing cannot occur but the high number of blocks
mapped to the cached area may lead to bad instruction cache performance. The optimal setting depends on the application.

The presented single-task dynamic SPM management technique exploits the MMU to catch accesses to pages that are copied on demand into the SPM at runtime. Only the virtual-to-physical address mapping changes; from the application’s point of view all addresses remain constant and no code patching is necessary. If no SPM is present, the pageable region is simply mapped to the external memory and the application runs just as well as an unmodified binary. Since our demand-paging technique is independent of the SPM size, it is directly applicable to multiple tasks. This is the topic of the next section.

4.2 Multi-Process SPM Management

For contemporary portable systems featuring a full-fledged operating system with preemptive multitasking, the set of concurrently running tasks is not known when the system is built or a single application is compiled. All SPM management techniques published so far consider only single processes, or an a priori known set of tasks and cannot adapt dynamically to new processes.

In this section, we introduce an SPM management technique for multi-process systems with virtual memory and preemptive multitasking. The SPM is managed by an multi-process SPM manager (SPMM) that treats
the SPM as a global resource. Whenever a new process is created, destroyed or requires more SPM, the SPMM re-distributes the available SPM among the processes depending on the SPM sharing strategy. We discuss three SPM sharing strategies, namely the global, the divided, and the hybrid SPM sharing strategy. We analyze each of these strategies with regard to several preferable properties of multi-process SPM allocation such as fairness, computational complexity, or easy adaptation to joining/leaving processes.

As in the single-task scenario, an SPM-optimized binary of an application must be available to exploit the advantages of SPM. The runtime environment/SPMM presented in this section uses the identical SPM-optimized images as in the single-process scenario (Section 4.1).

### 4.2.1 The SPM Manager

The SPM is managed by an SPM manager (SPMM) that is part of the runtime environment (RTE). The minimal runtime environment (RTE) consists of a scheduler, a process loader, and the SPM manager (SPMM). Whenever an application is started, the SPMM checks whether the newly created process is SPM-optimized or SPM-unaware. This is accomplished by checking for the paged code region that is only present in SPM-optimized images and contains the most frequently executed code (The process of computing and generating the paged code region is described in Chapter 3). If the application is SPM-optimized, its process’ virtual memory mappings are initialized
in such a way that accesses to code and data located in the paged region triggers an MMU page fault in the same way as described in the single-process runtime SPM management (Section 4.1.1). As soon as an SPM-optimized process becomes ready to run, the SPMM allocates the SPM depending on the currently active SPM sharing strategy (see Section 4.2.2). Whenever the control flow reaches unmapped code or data located in the paged region, the MMU generates a page fault exception. The SPMM then loads the requested page into the SPM, and modifies and enables its page table entry (PTE) before the aborted instruction is restarted. If a page needs to be evicted from the SPM, the SPMM disables its PTE.

As long as a page resides in the SPM, instruction fetches from that page incur no additional overhead because they do not trigger a page fault. This means that accesses to pages residing in the SPM go unnoticed by the SPMM. This is a slight disadvantage because—unless the MMU provides page reference or aging bits—it makes it harder for the SPMM to employ more sophisticated page replacement strategies than simple round-robin replacement.

Integration into Existing Operating Systems

The SPMM can be built as a module to facilitate easy integration into existing operating systems. The SPMM needs only be invoked by the OS for the following five events: (a) a new process is created (b) a process exits (c) a process changes its ready-to-run status (d) a process is scheduled (e)
a page fault exception for an SPM page occurs.

**Effect on Virtual Memory Systems**

The SPM can be regarded as an additional layer in the memory hierarchy of a virtual memory system with paging, and thus does not hinder paging of memory pages to an external storage medium. Minor modifications to the page fault exception handler are necessary to redirect those page fault exceptions to the SPM that are caused by pages to be run from the SPM.

### 4.2.2 SPM Sharing Strategies

Where a page is placed in the SPM and which page is evicted from it when a page fault exception occurs is determined by the SPM sharing strategy. We propose three different strategies for multi-process SPM allocation: *global*, *divided*, and *hybrid*. In the *global* SPM sharing strategy, the SPM is treated as a global resource and shared among all running processes. The *divided* SPM sharing strategy divides the SPM into $n$ disjoint regions and assigns one region to each SPM-optimized process. The *hybrid* SPM sharing strategy is a combination of the first two strategies: part of the SPM is divided into $n$ disjoint regions while the remaining portion is shared among all processes.

We have designed all three strategies with the following goals in mind: easy adaptation, maximum preservation, computational complexity, small
task-switching overhead, and fairness. Adaptation is important because processes are created and destroyed at arbitrary times. An SPM sharing strategy must be able to easily adapt at runtime to a varying number of SPM-optimized processes. Whenever a process joins or leaves, the SPM is redistributed among all active processes. Ideally, the new allocation preserves as much of the old state as possible. For example, a recently allocated page should not be evicted from the SPM before all the older pages have been replaced. Similarly, if a process has to release a few of its pages because a new process joins, then these pages should be the ones that would be replaced next because they have not been recently loaded into the SPM. Another example is when a process $p$ has to surrender pages to another process $q$. These pages may still contain code of $p$, even though they now belong to $q$. An SPM sharing strategy with good preservation will not evict that code until process $q$ actually requires the page.

Since a new SPM allocation needs to be computed at runtime whenever a process joins or leaves, it is important that the allocation algorithms do not require complex calculations. Similarly, the task-switching overhead introduced by the SPM sharing strategy should be kept as small as possible. Finally, an SPM sharing strategy should be fair, that is, a single process should not be able to cling to the whole or an over-proportional part of the SPM and thereby discriminating other processes.
Global SPM Sharing

In the global SPM sharing strategy, the SPM is shared among all processes (Figure 4.3). The SPMM maintains a single round-robin pointer, \textit{next}, that points to the next block to be replaced. Whenever a pagefault occurs, the requested page is loaded into the designated SPM page, and the \textit{next} pointer is moved to the following page in the list.

The global SPM sharing strategy satisfies all but one goal. It easily adapts to joining or leaving processes and preserves the existing SPM allocation. It does not require any computation when the number of running task changes, and there is no task switching overhead generated by the global strategy. It is, however, not particularly fair because a process with a large working set can request as many pages needed, thereby evicting most or all pages of the other processes.
Divided SPM Sharing

For the divided SPM sharing strategy, the SPMM splits the SPM into $n$ disjoint regions where $n$ is the number of active SPM-optimized processes. These regions are private to the process they are assigned to, i.e., other processes cannot load pages into another process’ region. The size of each private region is determined by the SPM division policy. We have implemented two different policies, the maximum-workingset (static), and the on-demand (adaptive) policy.

For each process $p$, the SPMM maintains a pointer $next_p$ that points to the next page to be replaced (Figure 4.4). Whenever a process requests a page, the SPMM loads the code into the designated page. If there is no free page left, the SPMM replaces the oldest page within $p$’s region in a round-robin fashion.
The divided SPM sharing strategy is fair because each process can only occupy up to $s(p)$ pages. Like the global strategy, the divided strategy does not cause any task switching overhead for managing the SPM.

Easy adaptation and maximum preservation can be accomplished if we allow the disjoint regions to be discontiguous. To do so, blocks of the SPM are managed with a ring implemented with a doubly-linked list. The pseudocode in Figure 4.5 illustrates the process of computing a new SPM allocation. Assume we have $n$ running processes, and $s_{\text{cur}}[i]$ stands for the number of SPM blocks currently allocated to process $i$. When a process joins or leaves, the SPMM first computes the new number of SPM blocks $s_{\text{new}}[i]$ allocated to each process. It then removes extra blocks from all processes $i$ where $s_{\text{new}}[i] < s_{\text{cur}}[i]$. By following the round-robin pointer $\text{next}[i]$, the SPMM removes the $s_{\text{new}}[i] - s_{\text{cur}}[i]$ oldest blocks from $i$ and leaves the more recently allocated blocks allocated to $i$. Finally, the SPMM assigns blocks to all processes $i$ where $s_{\text{cur}}[i] < s_{\text{new}}[i]$. The new blocks are inserted into process $i$'s region at $\text{next}[i]$, i.e., on a page fault, process $i$ will first allocate the newly assigned blocks before evicting its own code blocks. The complexity of the divided SPM sharing strategy depends on the sharing policy. Static policies, i.e., policies that do not change the SPM distribution unless the number of running processes changes, have a very little overhead. Adaptive policies monitor the behavior of the running processes and recompute a new SPM distribution when required and thus incur a higher overhead.
procedure AllocateSPM(Thread list $P$)
begin
  compute $s_{new}$ according to the active policy

  for $i := 1$ to $n$ do begin
    if ($s_{new}[i] < s_{cur}[i]$) then begin
      remove $s_{cur}[i] - s_{new}[i]$ blocks from process $i$
    end else if ($s_{cur}[i] < s_{new}[i]$) then begin
      assign $s_{new}[i] - s_{cur}[i]$ blocks to process $i$
    end
    $s_{cur}[i] := s_{new}[i]$
  end

policy max_working_set(Thread list $P$, distribution $s_{new}$)
begin
  for $i := 1$ to $n$ do begin
    $s_{new}[i] := \sum_{\forall p \in P} \text{max working set}[p] \cdot \# \text{of SPM pages}$
  end
end

policy on_demand(Thread list $P$, distribution $s_{new}$)
begin
  for $i := 1$ to $n$ do begin
    $s_{new}[i] := \sum_{\forall p \in P} \phi_P[i] \cdot \# \text{of SPM pages}$
  end

  for $i := 1$ to $n$ do $\phi_P[i] := \phi_P[i]$
end

Figure 4.5: Computing a new SPM allocation for the divided SPM strategy depending on the policy. $s_{cur}[i]$ contains the number of blocks currently assigned to process $i$. The policies maximum-workingset and on-demand are also shown.
The Maximum-Workingset Policy. In this policy, the size of a process’ private region is proportional to the maximum working set of that process (computed by the postpass optimizer at compile-time, see Chapter 3). The maximum-workingset policy is static, i.e., the distribution of the SPM does not change unless a new process joins or a running process exits.

The On-Demand Policy. During its execution, a process may go through various phases with different working sets. The on-demand policy distributes the SPM according to the current working sets of the running processes, $ws(p)$, by keeping track of at the average number of pagefaults, $\phi PF(p)$, over the last $k$ epochs of process $p$. $\phi PF(p)$ depends on the relationship of $s(p)$ and $ws(p)$: if $s(p) \geq ws(p)$, process $p$ will generate no or only few compulsory misses, and if $s(p) < ws(p)$, (possibly many) capacity misses occur. The on-demand policy constantly measures the number of pagefaults and updates $\phi PF(p)$. Immediately before a process is scheduled, $\phi PF(p)$ is compared to $\phi PF_{last}(p)$, the average number of pagefaults used when the current SPM distribution was calculated. If $\phi PF(p)$ differs significantly from $\phi PF_{last}$, then the SPM distribution is re-computed. The on-demand policy is adaptive as it matches the SPM distribution to the current working set of the running processes.
Hybrid SPM Sharing

The hybrid SPM sharing strategy is a mixture between the global and the divided SPM sharing strategy. A part of the SPM, the shared pool, is shared between all processes. The remaining blocks are distributed according to the divided SPM sharing strategy (Figure 4.6). The hybrid strategy can be considered the common case because a shared pool size of zero blocks yields the identical SPM allocation as the divided strategy. Likewise, with a shared pool size equal to the number of SPM blocks, the allocation is the same as with the global SPM sharing strategy.

The hybrid strategy incurs a small overhead at each task switch when the blocks of the shared pool are moved from the old to the newly scheduled process (Figure 4.7). Assume that the size of the shared pool is two blocks, i.e., $s_{\text{shared}} = 2$, and that these blocks are allocated to the currently running
process, \( p \). For efficiency reasons, the hybrid SPM sharing strategy maintains a per-process \( \text{pool.end}[p] \) pointer that is always exactly \( s_{\text{shared}} \) blocks ahead of \( \text{next}[p] \). Whenever the running process allocates a new block, both pointers \( \text{next}[p] \) and \( \text{pool.end}[p] \) are advanced by one block, i.e., at any time the shared pool is made up from the \( s_{\text{shared}} \) oldest blocks of the currently running process.

Moving the shared pool from \( p \) to \( q \) consists of the following steps: first, the \( s_{\text{shared}} \) oldest blocks of \( p \) are removed. Since \( \text{pool.end}[p] \) is always \( s_{\text{shared}} \) blocks ahead of \( \text{next}[p] \), the SPMM simply links the block preceding \( \text{next}[p] \) to \( \text{pool.end}[p] \) (Figure 4.7 [upper row]). The two blocks are then inserted into \( q \)’s list of SPM blocks as shown in the lower row of Figure 4.7.

In terms of adaptation, preservation, and computational complexity, the
Table 4.1: Properties of the Proposed SPM Sharing Strategies. (MWS: Maximum-Workingset Policy, OD: On-Demand Policy, n: Number of SPM-Optimized Processes)

<table>
<thead>
<tr>
<th>Property</th>
<th>SPM sharing strategy</th>
<th>SPM sharing strategy</th>
<th>SPM sharing strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>global</td>
<td>divided</td>
<td>hybrid</td>
</tr>
<tr>
<td></td>
<td>MWS OD</td>
<td>MWS OD</td>
<td>MWS OD</td>
</tr>
<tr>
<td>Easy adaptation</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Maximum Preservation</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Fairness</td>
<td>no</td>
<td>yes</td>
<td>somewhat</td>
</tr>
<tr>
<td>Computational complexity</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Task-switch overhead</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

The hybrid SPM sharing strategy is equal to the divided strategy. The hybrid strategy is fair because a process can only allocate up to $s_{cur[p]} + s_{shared}$ blocks. At each task switch, the hybrid strategy incurs a small overhead caused by moving the shared pool from the old to the newly scheduled process.

Table 4.1 summarizes the properties of the SPM sharing strategies discussed in this section.

4.2.3 Real-Time Considerations

To give an estimation of the worst-case execution time (WCET), we have to assume that a page fault occurs every time the control flow reaches code in the paged region. A guaranteed lower WCET is difficult to assess because the number of active processes changes dynamically at runtime (Section 5.3.1 quantifies the penalty incurred for a single page fault). This
WCET estimation might not be good enough for real-time tasks with tight timing constraints. It is, however, easy to accommodate the RTE to such tasks. A real-time task requests $n$ SPM pages that are exclusively under the control of the task. The RTE removes these pages from the list of available pages and recomputes a new SPM allocation. This approach works with all proposed SPM sharing strategies.
Chapter 5

Evaluation Environment

5.1 Simulation Environment

We evaluate the effectiveness of the proposed SPM management techniques on SNACK-armsim, a cycle-accurate architecture simulator that models the ARM9E-S core and supports the ARMv5TE instruction set. It includes timing models for the pipelined ARM9E-S core, the MMU with the unified TLB, caches with \( \mu \)TLBs, scratchpad memory, the AMBA AHB bus, and external memory.

For this work, we have extended SNACK-armsim in the following ways:

1. Models for the proposed horizontally partitioned on-chip memory system with an SPM and a cache as presented in Chapter 2.
2. TLB entries are extended to include the \textit{SPM flag} (see Figure 2.3). The flag is
computed whenever the MMU translates a virtual into a physical address. Based on the SPM flag, either the cache or the SPM is accessed. Both \( \mu \)TLBs contain 16 entries, and the unified TLB has 64 entries. (3) To accommodate the MMU to page sizes of 1024, 512, 256, 128, and 64 bytes, we extend the address field of tiny page table entries to include bits 9...6 (Figure 5.1). Standard tiny PTEs in ARMv5 architecture support only a page size of 1024 bytes. Bits 31...10 contain the physical address, bits 5 and 4 the permission bits, bit 3 and 2 determine whether the page is cacheable or bufferable, and the last two bits contain the tiny page selector (11b). For 1024-byte pages, physical addresses are 1-KB aligned, i.e., the bits 9...0 of the address are always zero. Sixty-four byte pages must be aligned at 64 byte boundaries. Thus, bits 5...0 of the address are zero and do not interfere with the access permission bits, the cacheable/bufferable flags, or the tiny page selector. Note that the PTEs itself do not include the SPM flag.

Reducing the page size increases the size of the page table. On ARM architectures, a 1-MB area of memory mapped with tiny (1024 byte) pages consists of 1024 entries. Each entry is 4 bytes wide. Thus, the size of the
(second-level) page table is 4 KB. A page size of 512 bytes doubles the size of the page table to 8 KB. Accordingly, 256-byte pages require 16 KB, 128-byte pages 32 KB, and for 64-byte pages, the second-level page table consumes as much as 64 KB per page table.

For the simulations, the processor core clock in SNACK-armsim is set to 200 MHz. The latencies of the cache, the SPM, the unified and the \( \mu \)TLBs, and the external memory (SDRAM) are shown in Table 5.1. Cache, unified TLB, and both \( \mu \)TLBs have a hit access latency of one. The latencies vary in case of a miss: the \( \mu \)TLBs have a miss latency of 2 plus that of the following unified TLB access. That is, three cycles in total if the unified TLB hits. If the unified TLB misses, the virtual address is sent to the MMU, which then performs a page table walk and computes the SPM flag. The page table walk consists of one or two nonsequential memory accesses. If a cache misses, it incurs a miss latency of two cycles plus the latency of an eventual writeback (in case the line is dirty) plus the following burst access to the external memory to fill the cache line.

For the reference cases, we assume a flat memory space without MMU or TLBs, i.e., the data cache and the instruction cache are physically addressed. This is an over-optimistic assumption as in a real ARM926E-J core (and also ARM11 cores), the MMU must be turned on to enable the data cache. With an MMU turned on, the reference cases would each consume between 5 and 10\% more energy. Nevertheless, we compare our SPM-optimized binaries against reference cases without TLB access energy.
Table 5.1: Access Latencies in CPU Cycles

<table>
<thead>
<tr>
<th>Memory</th>
<th>Hit</th>
<th>Miss</th>
<th>Memory</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache</td>
<td>1</td>
<td>2 + writeback + line fetch</td>
<td>SPM</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>unified TLB</td>
<td>1</td>
<td>3 + MMU page table walk</td>
<td>SDRAM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>μTLB</td>
<td>1</td>
<td>2 + unified TLB access</td>
<td>non-sequential</td>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>sequential</td>
<td>24</td>
<td>24</td>
</tr>
</tbody>
</table>

because we want to make sure that the proposed techniques also achieve energy savings if there is no such architectural limitation.

5.2 Single-Process SPM Management

5.2.1 Performance Metrics

For the single-process benchmarks, we use the total execution time as the performance metric and the total energy consumed by the core and the memory subsystem as the energy metric. The execution time is computed by dividing the measured number of core clocks by the core clock frequency

\[ T_{total} = \frac{\# \text{core clocks}}{\text{core frequency}} \]

The consumed energy is computed by summing up the the core energy, the on-chip memory system with both μTLBs, the unified TLB, the instruction and the data cache, the SPM, the off-chip bus, and the external memory (SDRAM)

\[ E_{total} = E_{core} + E_{unified TLB} + E_{i-\mu TLB} + E_{d-\mu TLB} + \]
\[ E_{icache} + E_{dcache} + E_{SPM} + E_{ext\_static} + E_{ext\_dynamic} \]

The core energy is computed by

\[ E_{core} = T_{total} \cdot P_{core} \cdot f_{core} \]

where \( f_{core} \) is the core frequency in MHz and \( p_{core} \) the power per MHz parameter from Table 5.2 (a). The energies consumed by the TLBs, the caches and the SPM, respectively, are computed by

\[ E_{TLB} = e_{TLB}(hit + miss \cdot linesize) \]
\[ E_{cache} = e_{cache}(hit + miss \cdot linesize) \]
\[ E_{SPM} = e_{SPM}(read + write) \]

where \( e_{TLB}, e_{cache}, \) and \( e_{SPM} \) are taken from Tables 5.2 (a) and (b). \( Hit, \ miss, \) and \( linesize \) denote the number of hits, the number of misses and the linesize for the corresponding memory structures, respectively. The \( \mu \)TLBs and the unified TLB are modeled as caches with a 8-byte linesize. The cache energy is computed accordingly with the corresponding linesize. The SPM energy is simply the access energy multiplied by the sum of reads and writes.

The SDRAM energy is composed of static and dynamic energy [25]. We have modeled the low-power 64-\( \)MB Samsung K4X51163PC SDRAM [33] with a memory bus frequency \( f_{mem} = 66 \) MHz and a supply voltage \( V_{dd} = 1.8 \) V. The static energy consumption, \( E_{ext\_static}, \) includes the standby power
and the power to periodically refresh the SDRAM cells and is computed by

\[ E_{ext\_static} = T_{total} \cdot P_{standby} \]

where \( P_{standby} \) is the static power consumption of the SDRAM (Table 5.2 (a)). The dynamic energy,

\[ E_{ext\_dynamic} = e_{read\_random} \cdot \text{read\_random} + e_{read\_burst} \cdot \text{read\_burst} + \\
e_{write\_random} \cdot \text{write\_random} + e_{write\_burst} \cdot \text{write\_burst} \]

includes both SDRAM dynamic energy and the memory bus energy. The energies \( e_{read/write\_random/burst} \) denote the per-word access energy for a random/burst read/write access, respectively.

Table 5.2 lists the values used for the energy calculations. All energy parameters are the energy required per word (4-byte) access, including the values for SDRAM read/write burst. The cache, SPM, minicache, and TLB access energies were computed for 0.13 \( \mu \)m technology using CACTI [40]. In our calculations, we do not distinguish between read and write operations for the caches, the SPM, and the TLBs, even though the write access energy is slightly lower than that of a read access. The core power consumption for a 0.13 \( \mu \)m ARM926EJ-S core without caches was taken from [34]. The static and dynamic energy of the SDRAM were computed using the System Power Calculator from [24], bus energy was taken from [34].

77
## Table 5.2: Per-Word Access Energy and Power Parameters

<table>
<thead>
<tr>
<th>4-way Assoc. Cache</th>
<th>Direct-Mapped Cache</th>
<th>Minicache</th>
<th>TLB</th>
</tr>
</thead>
<tbody>
<tr>
<td>linesize [words]</td>
<td>linesize [words]</td>
<td>energy [nJ]</td>
<td>energy [nJ]</td>
</tr>
<tr>
<td>1 4-way 8 0.538 1 direct 8 0.197</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 4-way 8 0.542 2 direct 8 0.203</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 4-way 8 0.550 4 direct 8 0.215</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 4-way 8 0.564 8 direct 8 0.237</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Minicache type entries energy [nJ]</td>
<td>TLB 2-way 64 0.141</td>
<td></td>
<td></td>
</tr>
<tr>
<td>256 direct 2 0.193 unified TLB 2-way 64 0.141</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>512 direct 4 0.196 µTLB full 16 0.125</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### (a)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5 0.121 4 0.145 read random 11.747</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0.128 6 0.160 write random 10.397</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 0.134 8 0.175 read burst 3.373</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 0.139 10 0.183 write burst 1.659</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Core model power [mW/MHz] SDRAM power [mW] |
|----------|------|----------|------|
| ARM926EJS 0.360 standby 9.600 |

### (b)
5.2.2 Benchmarks

We use 15 embedded applications to evaluate our work. These include nine benchmarks from MiBench [14] and MediaBench [21], a H.264 video decoder [15], the official ISO MP3 decoder [27], MPEG-4 XviD encoding/decoding [41], and a public key encryption/decryption tool, Pretty Good Privacy (PGP) [31]. We chained the benchmarks Quicksort, Dijkstra, SHA, ADPCM-enc, ADPCM-dec, and Bitcount together into one benchmark called Combine. Each of the smaller benchmarks is executed once in 

Combine to represent an embedded application with multiple phases. Table 5.3 summarizes the characteristics of each benchmark. We set $M_{\text{code}} = 4$, $m_{i\text{-cache}} = 0.02$, $M_{\text{data}} = 4$, $m_{d\text{-cache}} = 0.05$, and $\text{threshold} = 4$ for the clustering algorithm (Section 3.3) for all benchmarks.

We compare the horizontally partitioned memory system with our dynamic SPM management technique to a fully-cached system. For the fully-cached system, the reference case, both the instruction and the data cache are, if present, 4-way associative physically indexed, physically tagged caches. The instruction cache was chosen amongst 1-, 2-, 4- and 8-KB caches and set to the smallest cache that achieves a cache miss ratio below 2%. Accordingly, the data cache size was set to the smallest cache achieving a miss ratio below 5%. Also for the data cache, the possible cache sizes were 1, 2, 4, and 8 KB. Table 5.3 lists the total code and data size, the number of dynamic instructions, the number of dynamic reads and writes, and the cache configuration of the reference case for each benchmark. While the
Table 5.3: Properties and Configuration of the Reference Case for Single-Process Benchmarks

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>combine</td>
<td>10</td>
<td>49</td>
<td>191.2</td>
<td>272.7</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>epic</td>
<td>17</td>
<td>3</td>
<td>329.9</td>
<td>217.6</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>unepic</td>
<td>16</td>
<td>3</td>
<td>30.6</td>
<td>25.0</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>ft</td>
<td>12</td>
<td>4</td>
<td>91.4</td>
<td>57.2</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>h264</td>
<td>115</td>
<td>151</td>
<td>64.7</td>
<td>79.7</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>mp3</td>
<td>23</td>
<td>90</td>
<td>82.1</td>
<td>139.7</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>mp4d</td>
<td>36</td>
<td>495</td>
<td>54.4</td>
<td>56.3</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>mp4e</td>
<td>39</td>
<td>495</td>
<td>35.6</td>
<td>50.4</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>pgpd</td>
<td>48</td>
<td>289</td>
<td>59.2</td>
<td>41.0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>pgpe</td>
<td>41</td>
<td>289</td>
<td>9.6</td>
<td>10.5</td>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>

Static data size does not include the memory required for the stack, the heap, and memory mapped I/O. Read and write operations to any of these areas are included in the dynamic number of reads and writes.

5.3 Multi-Process SPM Management

5.3.1 The Runtime Environment

We have implemented a minimal RTE consisting of a loader, a scheduler, and the SPMM. The scheduler is a preemtising round-robin scheduler with a scheduling frequency of 100Hz. All processes have equal priority. Processes in the scheduler queue are always ready-to-run because there are no data dependencies between processes and all I/O data is placed in external memory. The loader loads processes from a RAM file system and assigns stack and heap areas to newly created processes.
Whenever a process accesses an unmapped instruction, on average, the penalty incurred by loading the corresponding page into the SPM is 69 instructions, or 240 core clocks (1.2 ms). The interrupt handler is responsible for 7 instructions, and the SPMM requires 48 instructions for managing the SPM, advancing the round robin pointers and disabling/enabling the memory mappings. Copying a page of 256 bytes requires 14 load/store multiple instructions.

### 5.3.2 Performance Metrics

As in the single-process scenario, we use the total energy consumed by the core and the memory subsystem as the energy metric and the total execution time as the performance metric. Additionally, we define the *throughput* of the RTE as the amount of work per time. The amount of work, i.e., running a benchmark from start to the end, is constant, thus

\[
\text{throughput} = \frac{c}{\text{execution time}}
\]

The simulator computes the total number of core clocks from the start till the end of a run. The end of a run is reached as soon as the last single process applications of a multi-process benchmark ends. The execution time is computed by dividing the measured number of core clocks by the core clock frequency, and the energy consumption is calculated by summing up the core energy, the on-chip memory system with both μTLBs (if present), the unified TLB, the instruction and the data cache, the SPM
(if present), the off-chip bus, and the external memory (SDRAM). The energy consumption of the various memories is computed according to the equations given for the single-process scenario (see Section 5.2.1).

5.3.3 Benchmarks

Lacking a benchmark suite consisting of multiprocess applications, we use 15 embedded applications to construct representative multiprocess benchmarks. The applications include nine benchmarks from MiBench [14] and MediaBench [21], a H.264 video decoder [15], the official ISO MP3 decoder [27], MPEG-4 XviD encoding/decoding [31], and a public key encryption/decryption tool, Pretty Good Privacy (PGP) [31]. We chained the applications quicksort, dijkstra, SHA, ADPCM-enc, ADPCM-dec, and bitcount together into one application called combine. Each of the smaller applications is executed once in combine to represent an embedded program with multiple phases.

Each of the multi-process benchmarks comprises of several applications. Dsp represents a multiprocess DSP application and contains combine and fft. High load consists of seven concurrently executing processes to simulate a situation where the system is under high load. Internet 1 and 2 represent scenarios using encrypted email (PGP) and internet browsing (unepic). The multiprocess benchmarks multimedia 1 to 5 were generated by randomly selecting applications. The single applications contained within are started dynamically at random points in time. Table 5.4 summarizes the charac-
Table 5.4: Properties of the Multiprocess Benchmarks and Reference Case

Instruction Cache Size

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>dsp</td>
<td>combine</td>
<td>0</td>
<td>21</td>
<td>4</td>
<td>8</td>
<td>256</td>
</tr>
<tr>
<td></td>
<td>fft</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>high load</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>combine</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>fft</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>mp3</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>mp4d</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>pgpd</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>pgpe</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>unepic</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>internet 1</td>
<td>pgpd</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>pgpe</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>unepic</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>internet 2</td>
<td>combine</td>
<td>0</td>
<td>13</td>
<td>4</td>
<td>256</td>
</tr>
<tr>
<td></td>
<td></td>
<td>epic</td>
<td>159</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>fft</td>
<td>102</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>multimedia 1</td>
<td>mp5</td>
<td>0</td>
<td>103</td>
<td>4</td>
<td>256</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mp4d</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>pgpd</td>
<td>50</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>pgpe</td>
<td>179</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>unepic</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>multimedia 2</td>
<td>combine</td>
<td>63</td>
<td>133</td>
<td>4</td>
<td>256</td>
</tr>
<tr>
<td></td>
<td></td>
<td>epic</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>fft</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>mp4d</td>
<td>112</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>pgpe</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>unepic</td>
<td>139</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>multimedia 3</td>
<td>epic</td>
<td>36</td>
<td>230</td>
<td>4</td>
<td>256</td>
</tr>
<tr>
<td></td>
<td></td>
<td>h264</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>mp4e</td>
<td>93</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>pgpe</td>
<td>147</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>unepic</td>
<td>43</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>multimedia 4</td>
<td>fft</td>
<td>0</td>
<td>141</td>
<td>4</td>
<td>256</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mp3</td>
<td>69</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>pgpd</td>
<td>53</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>pgpe</td>
<td>141</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>unepic</td>
<td>189</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
characteristics of each benchmark. The first column shows the eight multiprocess benchmarks. The second column lists the single process applications of each benchmark. The third column shows the starting time of each application. A starting time of zero implies that the application is started as soon as the scheduler starts running. A starting time of \( x \) ticks denotes that the application is started after \( x \) scheduler ticks.

The reference case for our measurements is defined by running all benchmarks in the RTE on an ARM926EJ-S core with virtually indexed, virtually tagged caches. The instruction cache size of the reference case is set to the smallest cache that achieves a cache miss ratio of about 1\%. The size of the data cache is fixed to 16KB. Column five in Table 5.4 lists the instruction cache size of the reference case for each benchmark. The multiprocess benchmarks are composed of the original (SPM-unaware) single process applications.

The benchmarks are run in the RTE on a ARM926EJ-S core with the horizontally partitioned memory system presented in Chapter 2 and compared to the reference case. The SPM-aware single process applications are generated by the postpass optimizer. Columns six and seven in Table 5.4 list the size of the SPM and the cache.

A theoretical lower bound, the so-called ideal case, is obtained by running each benchmark with the original (SPM-unaware) applications on the horizontally partitioned memory architecture. We assume that all instruction fetches are covered by the SPM. The SPM is nowhere big enough to
hold all code at once, but we assume that it always contains the required instructions, i.e., no page faults occur and no code is copied from the external memory into the SPM. In practice, this lower bound is unachievable, but we will show that the proposed SPM sharing strategies approach the lower bound if the number of page faults is reasonably small.
Chapter 6

Experimental Results

6.1 Single-Process SPM Management

In this section, we present the results of code clustering, data clustering, and finally code and data clustering for single processes.

6.1.1 Code Placement

The instruction and data cache size of the reference case and the corresponding configuration of the horizontally partitioned memory system for each benchmark are shown in Table 6.1. The data side remains unchanged, only the instruction cache is replaced with an instruction SPM and a direct-mapped minicache. Column six lists the die area requirements of the horizontally partitioned memory system compared to that of the instruction
Table 6.1: Configuration of the Horizontally Partitioned Memory System for Code Placement

<table>
<thead>
<tr>
<th>Application</th>
<th>Reference Case</th>
<th>Horiz.Part.Mem.System</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>cache</td>
<td>icache</td>
</tr>
<tr>
<td>combine</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>epic</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>unepic</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>ftt</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>h264</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>mp3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>mp4d</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>mp4e</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>pgpd</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>pgpe</td>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6.2: Code Placement for 256-byte Pages

<table>
<thead>
<tr>
<th>Application</th>
<th>reference</th>
<th>cached</th>
<th>paged</th>
<th>without</th>
<th>with</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>i-cache</td>
<td>size</td>
<td>#instr.</td>
<td>size</td>
<td>#instr.</td>
</tr>
<tr>
<td></td>
<td>[KB]</td>
<td>[KB]</td>
<td>[K]</td>
<td>[KB]</td>
<td>[M]</td>
</tr>
<tr>
<td>combine</td>
<td>1</td>
<td>4.52</td>
<td>6.1</td>
<td>4.84</td>
<td>191.2</td>
</tr>
<tr>
<td>epic</td>
<td>2</td>
<td>9.55</td>
<td>15.7</td>
<td>7.37</td>
<td>329.9</td>
</tr>
<tr>
<td>unepic</td>
<td>2</td>
<td>10.30</td>
<td>24.7</td>
<td>5.51</td>
<td>30.6</td>
</tr>
<tr>
<td>ftt</td>
<td>4</td>
<td>4.81</td>
<td>3.8</td>
<td>6.34</td>
<td>91.4</td>
</tr>
<tr>
<td>h264</td>
<td>8</td>
<td>83.49</td>
<td>127.6</td>
<td>31.44</td>
<td>64.5</td>
</tr>
<tr>
<td>mp3</td>
<td>4</td>
<td>4.88</td>
<td>2.9</td>
<td>17.42</td>
<td>82.1</td>
</tr>
<tr>
<td>mp4d</td>
<td>4</td>
<td>20.05</td>
<td>18.4</td>
<td>15.92</td>
<td>54.4</td>
</tr>
<tr>
<td>mp4e</td>
<td>2</td>
<td>23.95</td>
<td>20.9</td>
<td>14.63</td>
<td>35.6</td>
</tr>
<tr>
<td>pgpd</td>
<td>1</td>
<td>38.99</td>
<td>46.0</td>
<td>9.01</td>
<td>59.2</td>
</tr>
<tr>
<td>pgpe</td>
<td>1</td>
<td>34.87</td>
<td>48.1</td>
<td>8.99</td>
<td>9.6</td>
</tr>
</tbody>
</table>

Avg. page fill ratio 98% 91%

cache.

Table 6.2 shows the results of the code placement algorithm for a 256-byte page size with and without clustering. The size of the reference instruction cache is listed in the second column. Columns three to six show both the size (in Kilobytes) and the number of dynamic instructions for the cached and paged code regions. For the cached code region, the number of dynamic instruction is shown in thousands and for the paged region in millions.
On average, more than 99.9% of all dynamic instruction fetches are placed in the paged code region and only a few thousand fetches are covered by the minicache. Note that the actual number of fetches from the minicache will be higher for two reasons: first, the SPM manager itself is located in the cached region. The more pagefaults occur, the more often the SPMM is invoked and causes cache accesses. Second, the *thrashing-protection heuristics* in the SPMM (see Section 4.1.1) maps the last few pages of loops that are larger than the entire SPM to cached code regions.

Without clustering, the postpass optimizer generates less pages in the pageable area than with clustering (Table 6.2 columns seven and eight). The pages without clustering (98% fill ratio) are less fragmented than the pages after clustering (91%). The reason is that without clustering, the postpass optimizer allocates all pageable code into one single, big bin. When dividing the bin into pages, only the very last page can possibly be fragmented. With clustering, however, the postpass optimizer assigns one bin to each loop. Since each loop bin is divided into pages, more fragmentation occurs.

**The Effect of Clustering**

Figure 6.1(a) compares the normalized energy consumption of the reference image to SPM-optimized images when code clustering is disabled. The total energy consumption is split up into CPU core, SDRAM (includes static and dynamic energy), TLB, instruction cache, and SPM energy. The CPU
Figure 6.1: Memory system: no data cache and no minicache. Code clustering disabled.
core-energy consumption is directly proportional to the execution time, i.e.,
the execution time is represented by the fraction of the CPU Core bar.
Figure 6.1 (b) shows the normalized number of external memory accesses.
The total number is split up into instruction fetch, data read, and data write
accesses. Figure 6.1 (c) displays the number of page faults for the different
MMU page sizes on a logarithmic scale.

The reference case, denoted ref, is the original (SPM-unaware) binary
image run on a standard ARM926EJ-S core with an instruction, but no
data cache. For each application, the reference instruction cache, SPM,
and minicache sizes are set to the corresponding values in Table 3.3. The
SPM-optimized images are generated by our postpass optimizer with code
clustering disabled (see Section 3.3.1). Figure 6.1 shows the results of each
application for an MMU page size of 64, 128, 256, 512 and 1024 bytes
(denoted 64b, 128b, 256b, 512b, and 1024b, respectively).

The number of page faults is directly related to the size of the working
set. If the number of pages in the working set exceeds the number of pages
available in the SPM, the application will thrash. Intuitively, we expect
better results for smaller MMU pages. Without clustering, temporally local
code is potentially scattered all over the pageable code region, that is, with
larger page sizes, chances increase that the loaded page contains only a small
part of actually executed code, and the rest of the page consists of code that
is not executed. However, smaller pages also cause more page faults and,
at a certain page size, the advantage of a smaller pages is canceled out by

90
the increasing overhead of the SPMM. Furthermore, the performance of the \( \mu \) TLBs with its 16 entries also decreases with smaller page sizes, because the working set consists of more pages than for larger page sizes.

We observe the expected behavior for \textit{fft}, \textit{h264}, \textit{mp3}, \textit{pgpd}, and \textit{pgpe}. \textit{Combine}, \textit{fft}, \textit{pgpd}, and \textit{pgpe} start thrashing at 1024 byte pages, which results in a significantly higher execution time, energy consumption, and also more external memory accesses. \textit{H264}, \textit{mp3}, \textit{mp4d}, and \textit{mp4e} suffer from a high number of page faults (Figure 6.1(c)) and consume significantly more energy than the reference case for all page sizes.

From Figure 6.1(b) and (c) we observe that, even without clustering or a minicache, the number of external memory accesses decreases as long as the number of page faults is reasonably small. Without a minicache, a high number of page faults will inevitably lead to poor performance and lots of external memory accesses because the SPM manager is located in the external memory. We place the SPMM in external memory and not in the SPM because of the following considerations: it only makes sense to place the SPMM in the SPM when it is accessed very frequently and this is only the case when an application thrashes. The SPM pages occupied by the SPMM, however, are not available to the application any more, which will increase the thrashing. Furthermore, some applications might only thrash if the SPMM is placed in the SPM.

Figures 6.2(a)-(c) display the results for the same hardware configuration with code clustering enabled. Figure 6.2(c) shows the relative number of
Figure 6.2: Memory system: no data cache and no minicache. Code clustering enabled.
Figure 6.3: TLB performance for varying page sizes.

Page faults with clustering compared to no clustering. For the applications that suffer from a high number of page faults without clustering (h264, mp3, mp4d, and mp4e), we observe that clustering effectively reduces the size of the working set. Reducing the number of page faults has a direct impact on execution time, energy consumption and the external memory accesses as can be seen by comparing, for example, mp3 in Figure 6.1 (a) and (b) with Figure 6.2 (a) and (b).

Clustering can also cause thrashing as can be observed in Figure 6.2 (c) for epic with 1024, or fft with 512- and 1024-byte pages. This is because of the increased fragmentation whose effect grows with larger page sizes.

**TLB Performance**

Figure 6.3 shows the performance of the 16-entry, fully associative instruction \( \mu \)TLB and the 64-entry, two-way unified TLB on a logarithmic scale in dependence on the page size. The instruction \( \mu \)TLB is accessed for
each instruction fetched by the core. The number of instruction µTLB hits is labeled \textit{Inst µTLB Hit}. µTLB misses are handled by the unified TLB, which either hits or misses, i.e., the number of µTLB misses is the sum of \textit{Unified TLB Miss} and \textit{Unified TLB Hit}.

Not surprisingly, the performance of the µTLB suffers with smaller MMU page sizes. One factor is that the number of virtual-to-physical address translations doubles whenever we divide the page size by two. For example, the virtual-to-physical address translations of a working set of 4-KB code can be cached by the TLB with 4 entries for an MMU page size of 1024 bytes. For 512-byte pages we need 8 entries, for 256-byte pages 16, for 128-byte pages 32, and for 64-byte pages 64 entries to cache the address translations of 4 KB of code. A second reason is that the SPMM invalidates the affected entry in the TLB whenever it replaces a page in the SPM, namely the entry of the evicted page.

\textbf{The Minicache}

With only a few thousand instruction fetches from external memory, one could assume that the addition of a minicache is not necessary. This is, however, not so, as we show in this section.

The purpose of the minicache is twofold: First, it caches the SPM manager that is always placed in the cacheable code region. For applications with few page faults, executing the SPMM from the minicache (and not the SPM) incurs only a negligible performance penalty. Applications with
Figure 6.4: Memory system: no data cache and with minicache. Code clustering enabled.
a high number of page faults benefit more from executing the SPMM in
the SPM. However, the SPM area occupied by the SPMM itself (about 250
bytes) is not available for the application, which only further increases the
number of page faults. Second, one of the goals of the proposed memory
system is that it also runs SPM-unaware binaries with acceptable perfor-
mance. Without any instruction caching at all, SPM-unaware applications
are executed directly from external memory with an unacceptable increase
in execution time and energy consumption (Section 6.1.1).

Figure 6.4 shows the results for a setup with a 512-byte, direct-mapped
instruction minicache, SPM, and no data cache. Compared to the identical
setup without a minicache (Figure 6.2), the energy consumption and the
number of external memory accesses drop below the reference case for at
least one page size for all benchmarks except h264. For h264, even with
the minicache and for all page sizes, the reference image consumes less
energy and generates less external memory accesses as the SPM-optimized
binary. This is because h264 contains one big loop with a code size of 13.5
KB. The loop is bigger than the available SPM (10 KB), so the SPMM’s
thrashing-protection heuristic (Section 4.1.1) will map the last 3.5 KB of
code as cacheable pages. However, because the loop calls several inner loops,
the working set still exceeds the number of available SPM pages and h264
thrashes.
SPM-Unaware Binaries

As mentioned above, one of the design goals of the proposed memory architecture was that SPM-unaware applications run with acceptable performance and energy consumption. In contrast to previous work with only SPM, unoptimized applications can still profit from the minicache. Figure 6.5 shows the effect of the minicache on the execution time and energy consumption of unoptimized binaries. Without a minicache, the unoptimized applications run more than 13-fold slower and consume almost 10 times more energy than the reference runs. With the minicache, however, we only suffer a 2.9-fold increase in runtime with 3.2 times more energy consumed.
Figure 6.6: Memory system: with data cache and minicache. Code clustering enabled.
Using the Data Cache as a Victim Buffer

The proposed horizontally partitioned memory architecture is completed by adding a data cache. The SPMM uses the data cache as a victim cache for pageable code. Whenever the SPMM copies a code page from external memory to the SPM, the page is loaded through the data cache. Unlike a traditional victim cache, pages evicted from the SPM are not written back to the data cache. To study the effect of data cache pollution caused by caching the pageable region, the data cache size is set to the smallest cache size that achieves a miss ratio below 5% (Table 5.3).

The reference images run on an ARM926EJ-S core with an instruction and a data cache. For each application, the instruction and data cache sizes are set to the corresponding values in Table 5.3. For the SPM-optimized images, the instruction cache has been replaced with an SPM and a 512-byte minicache and the data cache is identical to that of the reference case.

Table 6.3 lists the average latency for handling a page fault in terms of core clocks and executed instructions. The SPM management code consisting of the low-level interrupt handler, the page replacement, and the page table TLB management requires 55 instructions independent of the page size. The block copy routine executes more instructions as the page size

<table>
<thead>
<tr>
<th>Page size</th>
<th>1024 bytes</th>
<th>512 bytes</th>
<th>256 bytes</th>
<th>128 bytes</th>
<th>64 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg. number of core clocks</td>
<td>1263</td>
<td>809</td>
<td>673</td>
<td>647</td>
<td>560</td>
</tr>
<tr>
<td>Avg. number of instructions</td>
<td>157</td>
<td>104</td>
<td>82</td>
<td>71</td>
<td>64</td>
</tr>
</tbody>
</table>
increases.

Figure 6.6 shows the final results for our horizontally partitioned memory system in conjunction with the SPMM. With the addition of the data cache, all but one benchmark consume less energy than the reference case for at least one page size setting. Exploiting the data cache as a victim buffer increases the data cache miss rate slightly, especially for benchmarks with a relatively small data cache that suffer from a high number of page faults (h264, mp4d, and mp4e) (Figure 6.6(c)). However, even though the code blocks pollute the data cache up to a certain degree, our experiments show that the performance gain by far outweighs the additional cost caused by an increased number of data cache accesses. Note that again the number of page faults is not reduced by adding a data cache, but the number of external memory accesses can be significantly reduced.

Table 6.4 lists the average reduction in execution time and energy con-

Table 6.4: Memory System: with Data Cache and Minicache. Code Clustering Enabled

<table>
<thead>
<tr>
<th>Pagesize</th>
<th>64b</th>
<th>128b</th>
<th>256b</th>
<th>512b</th>
<th>1024b</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benchmark</td>
<td>time</td>
<td>energy</td>
<td>time</td>
<td>energy</td>
<td>time</td>
</tr>
<tr>
<td>combine</td>
<td>96%</td>
<td>77%</td>
<td>93%</td>
<td>74%</td>
<td>93%</td>
</tr>
<tr>
<td>epic</td>
<td>111%</td>
<td>86%</td>
<td>85%</td>
<td>63%</td>
<td>84%</td>
</tr>
<tr>
<td>unepic</td>
<td>167%</td>
<td>135%</td>
<td>100%</td>
<td>74%</td>
<td>99%</td>
</tr>
<tr>
<td>fft</td>
<td>86%</td>
<td>90%</td>
<td>52%</td>
<td>53%</td>
<td>41%</td>
</tr>
<tr>
<td>h264</td>
<td>121%</td>
<td>135%</td>
<td>112%</td>
<td>120%</td>
<td>95%</td>
</tr>
<tr>
<td>mp3</td>
<td>94%</td>
<td>101%</td>
<td>82%</td>
<td>83%</td>
<td>42%</td>
</tr>
<tr>
<td>mp4d</td>
<td>106%</td>
<td>119%</td>
<td>75%</td>
<td>84%</td>
<td>65%</td>
</tr>
<tr>
<td>mp4e</td>
<td>111%</td>
<td>119%</td>
<td>82%</td>
<td>85%</td>
<td>79%</td>
</tr>
<tr>
<td>pgpd</td>
<td>86%</td>
<td>65%</td>
<td>84%</td>
<td>62%</td>
<td>83%</td>
</tr>
<tr>
<td>pgpe</td>
<td>59%</td>
<td>65%</td>
<td>48%</td>
<td>52%</td>
<td>47%</td>
</tr>
<tr>
<td>Geom. mean</td>
<td>101%</td>
<td>96%</td>
<td>79%</td>
<td>73%</td>
<td>69%</td>
</tr>
</tbody>
</table>
Table 6.5: Number of Page Faults Without and With Thrashing Protection

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Thrashing Protection off</th>
<th>Thrashing Protection on</th>
<th>Benchmark</th>
<th>Thrashing Protection off</th>
<th>Thrashing Protection on</th>
</tr>
</thead>
<tbody>
<tr>
<td>combine</td>
<td>21</td>
<td>18</td>
<td>mp3</td>
<td>157946</td>
<td>19449</td>
</tr>
<tr>
<td>epic</td>
<td>689</td>
<td>689</td>
<td>mp4d</td>
<td>49453</td>
<td>47720</td>
</tr>
<tr>
<td>unepic</td>
<td>27</td>
<td>26</td>
<td>mp4e</td>
<td>19185</td>
<td>19185</td>
</tr>
<tr>
<td>fft</td>
<td>60241</td>
<td>27</td>
<td>pgpd</td>
<td>126</td>
<td>125</td>
</tr>
<tr>
<td>h264</td>
<td>191921</td>
<td>66805</td>
<td>pgpe</td>
<td>211</td>
<td>211</td>
</tr>
</tbody>
</table>

sumption for each application depending on the MMU’s page size. Overall, with a page size of 256 bytes, on average, we achieve a 31% improvement in performance and a 35% reduction in energy consumption compared to a fully-cached core. Furthermore, our horizontally partitioned memory system requires 8% less die area than the corresponding memory system with an instruction and a data cache.

Effectiveness of the Thrashing-Protection Heuristics

Table 6.5 shows the effectiveness of the thrashing-protection heuristics. In Figure 6.7, for each benchmark, the normalized execution time, energy, and number of page faults are shown for the reference case and the SPM-optimized binary with disabled, and enabled thrashing-protection heuristics, respectively. For combine, epic, unepic, pgpd, and pgpe, the working set fits into the number of available SPM pages, therefore, turning the thrashing protection on has no big effect. For fft, h264, and mp3, the thrashing-protection heuristics significantly reduce the number of page faults. The heuristics do not work well for mp4d and mp4e. This is because the current implementation considers each loop’s working set size independent of

101
whether this loop has inner loops or not. Even so, the thrashing-protection heuristics affects neither the performance nor the energy consumption negatively for all benchmarks.

**Comparison against a Direct-Mapped Instruction Cache**

Set-associative caches require significantly more die area and energy per access than direct-mapped ones, the former because of the more complex control logic and the latter because of the parallel look-ups. Because of
the better performance, however, many embedded processors contain set-associative caches ([4], [5], [17]). In this section, we compare the reference case with its 4-way set-associative cache against a direct-mapped cache that requires a comparable die area and the horizontally partitioned memory architecture. On the 4-way set-associative cache and the direct-mapped cache, we run the original binary. On the horizontally partitioned memory architecture, we run the SPM-optimized binary with a page size of 256 bytes. Table 6.6 lists the corresponding direct-mapped cache for various sizes of a 4-way set-associative cache. A 1-KB 4-way set-associative cache with a die area of 0.369 mm² and an access energy of 0.538 nJ/word, for example, is replaced with a 4-KB direct-mapped cache with a die area of 0.331 mm² and an access energy of 0.215 nJ/word.

Figure 6.8 shows the normalized execution time and energy consumption for the reference case, a direct-mapped cache with comparable die area,
and the horizontally partitioned memory architecture. Thanks to its larger size, the direct-mapped cache outperforms the 4-way set-associative cache for most benchmarks. Compared to the horizontally partitioned memory system, it achieves a slightly better reduction in energy consumption only for benchmarks that suffer from a high number of page faults (h264, mp4d, mp4e). Compared to the direct-mapped cache, the horizontally partitioned memory architecture achieves, on average, a 16% improvement in runtime performance and a 14% reduction in energy consumption.

### 6.1.2 Data Placement

The instruction and data cache size of the reference case and the corresponding configuration of the horizontally partitioned memory system for each benchmark are shown in Table 6.7. The instruction side remains unchanged, only the data cache is replaced. We run all benchmarks with two different configurations of the data side: the first one, so-called “die” lists a feasible configuration whose die area requirements come closest to that of the original data cache. For die, the original data cache is replaced with an SPM and a cache half the size of the original cache. The configuration entitled “inc” leaves the original data cache unchanged and adds a small data SPM (0.5 or 1KB). Both configurations of the horizontally partitioned memory system require slightly more die area than the original data cache.

Table 6.8 shows the results of the data placement algorithm for local, and local plus global data. Local data consists of constant pools and other data.
Table 6.7: Configuration of the Horizontally Partitioned Memory System for Data Placement

<table>
<thead>
<tr>
<th>Application</th>
<th>Reference Case</th>
<th>Horiz.Part.Mem.System</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>icache [KB]</td>
<td>dcache [KB]</td>
</tr>
<tr>
<td></td>
<td>dSPM [KB]</td>
<td>dcache [KB]</td>
</tr>
<tr>
<td></td>
<td>die area [KB]</td>
<td>“die”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>die area [%]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dSPM [KB]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cache [KB]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“inc”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>die area [%]</td>
</tr>
<tr>
<td>combine</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>epic</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>unepic</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>fft</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>h264</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>mp3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>mp4d</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>mp4e</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>pgpd</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>pgpe</td>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>

local to functions, whereas global data includes the data sections (read-only, read/write, zero initialized) in the binary image. Table 6.8 (a) shows the result of local data placement. The total size of the local data is shown in the second column. The third column lists the size of the local data allocated to the pageable region, and the fourth column contains the number of memory operations to the paged local data. Accordingly, columns five and six list the same information for data placed in the cached region. Table 6.8 (b) displays the data placement for local and global data. For local data only, the data placement algorithm assigns 99.15% of all local data operations to the pageable region by placing 4.7% of the local data. For all benchmarks, the size of the pageable local data is extremely small, yet covers most of the memory accesses. If we consider local and global data, 0.15% of the data that cover 23.39% of all memory accesses are placed in the pageable region. The size of the data placed in the pageable region is still very small (0.85KB on average). 76% of the memory accesses are dispersed over 99.85% of the
Table 6.8: Data Placement

(a)

| Application | total size [KB] | local data | | | | cached size [KB] | | | |
|-------------|----------------|------------|------------|------------|------------|----------------|------------|------------|------------|------------|----------------|------------|
|             | size [KB] | paged | #mem op | size [K] | | | #mem op | size [K] | | | | |
| combine     | 1.92     | 0.16 | 22608 | 1.77 | 1.8 | | | | | | | |
| epic        | 1.43     | 0.02 | 2887  | 1.40 | 0.9 | | | | | | | |
| unepic      | 1.64     | 0.02 | 338   | 1.63 | 1.0 | | | | | | | |
| fft         | 1.57     | 0.08 | 1436  | 1.49 | 0.1 | | | | | | | |
| h264        | 32.59    | 0.29 | 1292  | 32.31 | 3.4 | | | | | | | |
| mp3         | 1.60     | 0.38 | 5989  | 1.22 | 0.6 | | | | | | | |
| mp4d        | 9.80     | 0.27 | 3426  | 9.53 | 2.9 | | | | | | | |
| mp4e        | 9.69     | 0.30 | 3411  | 9.39 | 2.6 | | | | | | | |
| pgpd        | 47.80    | 0.09 | 242   | 47.71 | 8.3 | | | | | | | |
| pgpe        | 47.95    | 0.13 | 291   | 47.82 | 12.5 | | | | | | | |

Data placement 4.70% 99.15%

(b)

| Application | total size [KB] | local + global data | | | | cached size [KB] | | | |
|-------------|----------------|---------------------|------------|------------|----------------|------------|------------|------------|------------|------------|----------------|------------|
|             | size [KB] | paged | #mem op | size [K] | | | #mem op | size [K] | | | | |
| combine     | 49.18    | 0.59 | 23697 | 48.59 | 21128 | | | | | | | |
| epic        | 3.34     | 0.14 | 2896  | 3.20 | 146 | | | | | | | |
| unepic      | 3.44     | 0.02 | 338   | 3.42 | 162 | | | | | | | |
| fft         | 3.54     | 0.13 | 1475  | 3.41 | 128 | | | | | | | |
| h264        | 151.34   | 1.77 | 1606  | 149.57 | 2490 | | | | | | | |
| mp3         | 90.72    | 2.73 | 10218 | 87.98 | 8512 | | | | | | | |
| mp4d        | 496.76   | 1.20 | 3564  | 493.56 | 3268 | | | | | | | |
| mp4e        | 495.65   | 1.55 | 3822  | 494.11 | 3259 | | | | | | | |
| pgpd        | 286.80   | 0.22 | 248   | 286.58 | 12222 | | | | | | | |
| pgpe        | 286.95   | 0.20 | 293   | 286.65 | 1766 | | | | | | | |

Data placement 0.15% 23.30%
local and global data. Data blocks with a relatively low number of accesses per word are not assigned to the pageable region because the copy-in and copy-out cost outweighs the benefit of the energy-wise cheaper data SPM accesses.

Figure 6.9 shows the results of the horizontally partitioned memory system for data only with an MMU page size of 256 bytes. In this setup, the instruction side is not modified. We compare the reference image (first bar of each benchmark) to the configurations “die” (bars 2 and 3) and “inc” (bars 4 and 5) (see Table 6.7), both for local (bars 2 and 4) and for local+global data (bars 3 and 5). Figures 6.9 (a), (b), and (c) show the energy consumption, the number of external memory accesses, and the number of on-chip memory accesses normalized with respect to the reference case. The runtime performance is proportional to the energy consumed by the CPU core (first segment of the bars in Figure 6.9 (a)). Figure 6.9 (d) shows the performance (miss rates) of both the instruction and the data cache. 100% hit ratio for both caches is located in the middle of the chart. The instruction miss ratio is shown in the lower half, and the data cache miss rate in the upper half. For all benchmarks, “inc” performs better both in terms of energy consumption as well as runtime performance than “die” even though it features a bigger data SPM. Because of the reduced data cache size in the “die” configuration, the data cache has a significantly lower hit rate (Fig. 6.9 (d)) which then causes a higher number of external memory accesses (Fig. 6.9 (b)) compared to the reverence or the “inc” case.
The reason for the much better performance of *unepic*, *fft*, and *pgpe* is not mainly caused by data paging. When comparing the instruction cache miss ratios in Fig. 6.9(d), we see that for those benchmarks the miss rate is significantly lower compared to the reference case, even though the instruction side is identical among all configurations. When the postpass optimizer extracts local data from functions and clusters frequently accessed blocks together, it also has to reorganize the layout of the functions in such a way that all references to data in constant pools can be satisfied (Section 3.1.4). By doing so, it also splits the functions into a paged and a cached part as explained in Section 3.2.1 and generates a final code layout that has a much better cache affinity than the reference image.

With data placement alone, on average, the configuration “die”-local (resp., -global) achieves an 11% (10%) reduction in energy consumption and a 17% (17%) improvement in performance. For “inc”, the energy consumption is reduced by 16% and the performance improves by 20% on average for both the local and the global data placement technique.

### 6.1.3 Code and Data Placement

Figure 6.10 shows the results of the horizontally partitioned memory system for both code and data with an MMU page size of 256 bytes. In this setup, the instruction side is set to the configuration listed in Table 6.1. The SPMM uses the data cache as a victim for code pages and the thrashing-protection heuristics are turned on. As in the previous section, we compare
Figure 6.9: Data only placement for a horizontally partitioned memory system with a comparable die area.
the reference image (first bar of each benchmark) to the configurations “die”
(bars 2 and 3) and “inc” (bars 4 and 5) (see Table 6.7), both for local (bars 2 and 4) and for local+global data (bars 3 and 5).

With code and data placement, the horizontally partitioned memory system achieves better results overall than with only data placement. This is not surprising, given that code placement alone performs well. Compared to data only placement, the difference in energy consumption between the configurations “die” and “inc” is much bigger. This is caused by the SPMM that uses the data cache as a victim buffer for code blocks. The smaller data cache size of “die”, therefore, suffers more as can be seen by inspecting the data cache miss rates in Fig.6.10 (d).

With code and data placement, on average, the configuration “die”-local (resp., -global) achieves an 34% (33%) reduction in energy consumption and a 32% (31%) improvement in performance. For “inc”, the energy consumption is reduced by 40% and the performance improves by 34% on average for both the local and the global data placement technique.

The horizontally partitioned memory system with and instruction and a data SPM outperforms both the instruction side-only and data-side only configuration. In comparison with data only placement, the “inc” configuration with code and data placement gains an additional 24% on energy consumption and 14% on runtime performance. Compared to code only placement, the “inc” configuration achieves an extra 5% reduction of energy consumption and a 3% improvement in runtime performance.
Figure 6.10: Code and data placement for a horizontally partitioned memory system with a comparable die area.
Figure 6.11: Comparison of a cache-optimized against an SPM-optimized image.

**Comparison with a Cache-Optimized Image**

As mentioned in Section 6.1.2, accessing code and data from the SPM rather than the cache is not alone responsible for the partly significant improvement in runtime performance. Part of the improvement is accountable to the better code and data layout of SPM-optimized binaries even when the image is run on a core with only caches and no SPM. Figure 6.11 compares the normalized execution time and energy consumption of the reference case with the SPM-optimized image run on the same hardware as the reference case (i.e., no SPM) and the identical SPM-optimized image run on the horizontally partitioned memory architecture (denoted *reference, cacheopt, and horizpart*, respectively). Again, the execution time is proportional to the energy consumed by the CPU core.

Thanks to the separation of frequently and infrequently accessed code
and data, the SPM-optimized image performs notably better than the reference image for all benchmarks, yet is significantly outperformed by the same image run on the horizontally partitioned memory system. Compared to the reference case, on average, the SPM-optimized binaries run on cached-only cores achieve a 19% improvement in runtime performance and a 15% reduction in energy consumption. The SPM-optimized binaries run on the horizontally partitioned memory system achieve, on average, a 34% improvement in runtime performance and a 40% reduction in energy consumption.

6.1.4 Exploiting SPM on Given Hardware

Contemporary CPUs for portable devices such as the ARM11 core often feature both caches and SPM. In this section we show that the dynamic SPM management techniques presented in this thesis can be directly applied to those architectures. While the performance improvement is independent of the concrete implementation of the CPU (given that both the SPM and the cache have the same latency), significant energy savings can only be achieved if only the affected memory is accessed.

Figure 6.12 compares running an SPM-unaware reference image with an SPM-optimized binary on identical hardware. We run both images on a 4, 8, and 16 KB setup. For each setup, the instruction cache, the instruction SPM, the data cache and the data SPM are of identical size. For each benchmark in Figure 6.12 the first bar denotes the reference case with all
4-KB memories, the third bar the 8-KB, and the fifth bar the 16-KB setup. The results of running the SPM-optimized binaries are shown in the second, the fourth and the six bar. Each reference case is normalized to 100% and compared to the corresponding result of the SPM-optimized binary.

On average, the SPM-optimized binary achieves a 25% reduction in energy consumption and a 18% improvement in runtime performance for the 4-KB, a 24% reduced energy consumption and a 6% improved runtime performance for the 8-KB, and finally a 25% reduction in energy consumption and a 3% improvement in runtime performance for the 16-KB setup. While the reduction in energy consumption stays more or less the same across all configurations, the performance improvement diminishes as the size of the L1 memories is increased. This is caused by saturation of the caches and/or the SPMs, i.e., the working set of the application fits completely into the on-chip memories.

Overall, the presented dynamic single-process SPM allocation techniques work well for a wide variety of benchmarks and memory configurations. In the next section, we discuss the results obtained when running the same SPM-optimized binaries in a multi-process environment.

6.2 Multi-Process SPM Management

This section presents the results obtained by running the multi-process benchmarks (Table 5.4) in a simple RTE with virtual memory and pre-
Figure 6.12: Code and data placement for given hardware configurations.
emptive scheduling. The multi-process benchmarks consist of a number of single-threaded applications.

The reference case is obtained by running the multi-process benchmark with the SPM-unaware reference images. To evaluate the proposed SPM sharing strategies, we run the multi-process benchmark with SPM-optimized binaries. The reference case is run with a 4-KB 4-way set-associative instruction and a 16-KB 4-way set-associative data cache. The horizontally partitioned memory system consists of a 6-KB instruction SPM, a 256-byte direct-mapped instruction minicache and a 16-KB 4-way set-associative data cache. We do not consider data paging.

Figures 6.13 and 6.14 compare the normalized energy consumption (a), the throughput, and the number of page faults of the reference case denoted ref, to the ideal case (ideal), and the three SPM sharing strategies, dynamic, hybrid (with a pool size of 1/4th, 2/4th, and 3/4th of the available SPM size and the on-demand policy), and the global SPM sharing strategy, denoted global. The execution time is represented by the fraction of the CPU core energy bar, since the energy consumed by the CPU core is directly proportional to the execution time.

The global strategy clearly outperforms the other sharing strategies. Figures 6.13 and 6.14 (c) shows that the performance of a sharing strategy mostly depends on the number of page faults. Even though the currently running process can evict all blocks of other processes which then have to load them again when scheduled, the effect is not as big as expected. One
Figure 6.13: Energy consumption, throughput, and pagefaults for multi-process benchmarks.
Figure 6.14: Energy consumption, throughput, and pagefaults for multiprocess benchmarks.
reason is the relatively long epoch (the scheduler runs at a frequency of 100Hz), another that all processes behave “reasonable”, i.e. do not thrash when scheduled.

For most benchmarks, the global strategy comes very close to the ideal case. A comparison of the global SPM sharing strategy with the (in practice unachievable) ideal case shows that the global SPM strategy exploits 87% of the runtime improvements and 89% of the energy savings on average.

On average, the divided SPM sharing strategy achieves a 19% increase in throughput and a 13% reduction in energy consumption (Table 6.9). The hybrid SPM sharing strategy achieves a 32%, 39%, and 43% improvement in throughput and a 23%, 27%, and 30% reduction in energy consumption for a shared pool size of $1/4^{th}$, $2/4^{th}$, and $3/4^{th}$ of the SPM size, respectively. Finally, the global SPM sharing strategy achieves a 47% improvement in throughput and a 32% reduction in energy consumption over a fully-cached ARM926EJ-S core.
Table 6.9: Energy Consumption and Execution Time of the Different SPM Sharing Strategies in Comparison With a Fully-Cached System. $E$ Stands for Energy, and $T$ Denotes the Throughput.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>ideal</th>
<th>divided</th>
<th>hybrid with on-demand policy</th>
<th>global</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$E$</td>
<td>$T$</td>
<td>$E$</td>
<td>$T$</td>
</tr>
<tr>
<td>dsp</td>
<td>58%</td>
<td>148%</td>
<td>69%</td>
<td>126%</td>
</tr>
<tr>
<td>high load</td>
<td>60%</td>
<td>154%</td>
<td>105%</td>
<td>96%</td>
</tr>
<tr>
<td>internet 1</td>
<td>69%</td>
<td>118%</td>
<td>69%</td>
<td>117%</td>
</tr>
<tr>
<td>internet 2</td>
<td>67%</td>
<td>123%</td>
<td>79%</td>
<td>107%</td>
</tr>
<tr>
<td>multimedia 1</td>
<td>48%</td>
<td>227%</td>
<td>86%</td>
<td>140%</td>
</tr>
<tr>
<td>multimedia 2</td>
<td>64%</td>
<td>135%</td>
<td>94%</td>
<td>98%</td>
</tr>
<tr>
<td>multimedia 3</td>
<td>66%</td>
<td>130%</td>
<td>101%</td>
<td>96%</td>
</tr>
<tr>
<td>multimedia 4</td>
<td>46%</td>
<td>237%</td>
<td>56%</td>
<td>202%</td>
</tr>
<tr>
<td>Geo.mean</td>
<td>59%</td>
<td>154%</td>
<td>81%</td>
<td>119%</td>
</tr>
</tbody>
</table>

120
Chapter 7

Conclusions and Future Directions

7.1 Conclusions

In this thesis, we have presented dynamic scratchpad memory (SPM) management techniques based on postpass optimization for contemporary portable systems with a memory management unit (MMU). Our techniques successfully reduce the energy consumption and increase the runtime performance by loading frequently accessed code and data on demand into the instruction and the data SPM, respectively. These days, portable devices such as smart phones run a full-featured operating system with virtual memory and preemptive multitasking. Unlike traditional embedded systems, knowledge of the exact hardware specifications and the set of running tasks is not
available at compile-time anymore. Therefore, we propose SPM allocation techniques that operate independently of a specific hardware configuration.

At compile-time, a postpass optimizer classifies code and data blocks as pageable or unpageable. Pageable blocks are blocks that likely yield an energy reduction when placed and subsequently accessed in the SPM. Pageable code and data are then grouped into pages the size of an MMU page. Since the minimal transfer unit from main memory to SPM is one page, good code and data clustering is necessary to keep the number of page transfers minimal. Code blocks are clustered into pages based on loop detection. Starting with the innermost loops first, the postpass optimizer places a loop’s code into a bin whose size is a multiple of the MMU page size. Code of outer loops is first placed in the bins of its inner loops if there is any room left before a bin for the outer loop is allocated. Data blocks are clustered into pages based on the location of the code referencing them. Two data blocks that are referenced from two different code blocks placed in the same code memory page are also placed in the same data memory page. After clustering code and data into memory pages, the postpass optimizer orders the pages based on their position in the loop control graph and the total number of accesses. Each page is assigned an value. Pages with high energy benefit when accessed in SPM are assigned a lower value than pages with a lower energy benefit.

At runtime, the SPM is managed by an SPM manager (SPMM). When an SPM-optimized binary is loaded, the SPMM disables the memory blocks of
the paged region in the MMU’s page tables. Whenever the application tries to execute code or access data located in an unmapped page, the processor signals a page fault exception that is handled by the SPMM. The SPMM loads the requested page into the SPM, enables the page in the page tables, and restarts the aborted instruction.

In a single-process environment, the SPMM assigns the whole SPM to the running application. For multiple running processes, the SPM has to be treated as a shared resource and needs to be managed by the operating system. We have developed an SPMM for multitasking systems and present and evaluate three multi-process SPM sharing strategies. The *global* SPM sharing strategy loads pages of the currently running process into the SPM without restriction. In the *divided* SPM sharing strategy, the SPM is distributed amongst all running processes, and each process can only load as many of its blocks into the SPM as it has assigned. The *hybrid* SPM sharing strategy is a mixture between the global and the divided strategy. A part of the SPM is distributed to each running process as in the divided strategy and the remaining SPM blocks are assigned to the currently running process on each task switch.

Contemporary portable devices execute all kinds of different processes obtained from different sources. We cannot expect that all applications that ever run on the device can be optimized for SPM. Processors with SPM only, however, would execute SPM-unaware images with an unacceptably low performance. Designs with caches and SPM, therefore, deliver good
performance for unoptimized binaries while offering SPM to SPM-optimized binaries. However, current processor designs have one major flaw: either the latency to access the SPM is higher than that of the cache, or both memories are accessed in each request. While the SPM management techniques presented in this thesis work for out-of-the box hardware as well, the energy savings will be minimal if both the SPM and the cache are accessed simultaneously. Therefore, we propose a horizontally partitioned memory system consisting of an SPM alongside with a cache in which the MMU’s address translation is serialized with the on-chip memory access. Based on the physical address, either only the cache or only the SPM is accessed. This saves a considerable amount of energy.

We have evaluated the proposed dynamic SPM allocation technique on the horizontally partitioned memory subsystem using fifteen embedded applications, including an H.264 video decoder, an MP3 decoder, an MPEG-4 video encoder/decoder, and a public-key encryption/decryption tool. We analyze the effect of the MMU page size and discuss code only, data only and code plus data SPM allocation. To evaluate the multi-process SPM sharing strategies, we have implemented a small RTE with virtual memory and preemptive multitasking.

With a single process and code only SPM allocation, we achieve a 35% reduction in energy consumption and a 31% improvement in runtime performance on the horizontally partitioned memory architecture with an MMU page size of 256 bytes compared to a fully cached system. With data only
SPM allocation, the energy consumption is reduced by 10% and the runtime performance increases by 17%. For code and data SPM allocation, the reduction in energy consumption is 40% and the runtime performance improvement 34%.

To evaluate the multi-process SPM sharing strategies, we run multi-process benchmarks comprising of several single-process SPM-optimized applications. We compare the energy consumption and throughput of the horizontally partitioned memory system with a fully cached processor core. For the overall best multi-process strategy, the global SPM sharing strategy, we achieve a 47% improvement in throughput and a 32% reduction in energy consumption.

The results of the dynamic SPM management techniques based on post-pass optimization presented in this thesis show that significant improvements in runtime performance and a big reduction in energy consumption both in a single-process as well as a multi-process environment are possible even if the size of the SPM is not known at compile time. The best results are achieved with the proposed horizontally partitioned memory subsystem. While SPM-unaware applications run with acceptable performance on the horizontally partitioned memory system, to fully exploit its benefits it is inevitable to generate SPM-optimized binaries based on profile data. Furthermore, the proposed memory system requires modifications to a well-established memory hierarchy which further hinders its industrial adoption. We therefore believe that the traditional memory hierarchy consisting of
caches still provides the best overall performance, but the horizontally partitioned memory subsystem presents a first attempt for a less energy-hungry memory subsystem featuring both caches and SPM. However, the dynamic SPM management techniques presented in this thesis are not dependent on the horizontally partitioned memory subsystem. The techniques both for single- and multi-process scenarios introduce a first approach to achieve binary portability, that is, SPM-optimized application binaries that run on various hardware, and also standalone or as one process in a multitasking environment.

7.2 Future Directions

The techniques presented in this thesis are merely a first step towards SPM-optimized applications in multi-processing environments. In themselves, they still leave a lot of room for improvement, but also lead to several directions of research worth pursuing.

7.2.1 Improvements to the Postpass Optimizer

Code and Data Clustering

While the postpass optimizer separates hot from cold code within functions, the basic allocation unit remains a function. An analysis of the pageable blocks generated by the postpass optimizer has shown that there still exists a
considerable gap inside pageable functions between very frequently and less frequently executed code blocks. To further improve the temporal locality and thereby reduce the working set, the code clustering algorithm needs to place the basic blocks directly. For data blocks, the temporal locality of date pages can be further improved by mixing local and global block together into one block. Local data (i.e., constant pools) often contains references to global data. By placing the local data along with the referenced global data, the number of data pages loaded into the SPM can be further reduced.

**Heap and Stack Data**

Currently, our postpass optimizer does not allocate stack and heap data to the SPM. Our trace analysis is restricted to data that is available at compile time, that is either global data or data located in constant pools of functions. We have proposed a first approach that includes stack in [7]. This approach does not handle heap data either and requires the SPM size to be known at compile-time. We can directly extend our work to include stack data pages. However, without any knowledge of the current call path, we do not expect good results. In order for stack pages to be loaded into the SPM only when doing so will result in reduced energy consumption requires that the SPM is aware of the current call path. For heap data, the postpass optimizer will need to identify the code location that allocates a frequently accessed block of heap data and replace that code with SPM-aware heap allocation code. Until now, our postpass optimizer does not modify code
(except inserting a couple of branches to maintain a correct control flow). Therefore, extending our work to include heap data is not straightforward.

7.2.2 Improvements to the SPM Manager

SPM Page Replacement

The SPMM evicts pages based on a simple round-robin replacement strategy. To apply more advanced page replacement strategies such as least recently used (LRU) or most recently used (MRU) that offer a better page replacement, some modifications to the SPMM will be necessary. Once a page is loaded into the SPM and its page table entry is enabled, accesses to this page go unnoticed by the SPMM. Since the MMU on ARM architectures does not offer an aging bit of recently accessed pages, implementing the LRU or MRU replacement strategy is not straightforward. Second-chance replacement can be implemented relatively easy by disabling but not evicting pages from the SPM. Accesses to such pages will generate a page fault which will be interpreted by the SPMM as setting its reference bit.

Prefetching

Some benchmarks contain huge basic blocks that span over more than one MMU page. If the first page of such a basic block is loaded, it is inevitable that the remaining pages will also be loaded. In such situations, the SPMM
might apply prefetching to keep the number of page faults low. In multi-process environments depending on the SPM sharing strategy, experiments will have to show whether prefetching achieves performance gains or whether it merely pollutes the SPM.

### 7.2.3 Further Directions

**Clustering at Load-Time**

Even though the postpass optimizer orders pages according to their access frequencies and the SPMM maps excess pages to the cache if not enough SPM is available, the number of page faults for some benchmarks is still relatively high. An interesting future direction is to do the clustering when the binary image is loaded on the device. At that point in time, the exact size of the SPM and cache is known and the pageable region could be tailored to that size. The clustering algorithm will need to be very efficient in order not to consume too much energy or execution time. The postpass optimizer can assist the clustering algorithm by generating freely positionable code.

**Dynamic SPM Allocation based on Performance Counters**

If the processor offers performance counters, the SPMM can decide at runtime which blocks are worth loading into the SPM and which are not. If the minimal transfer unit is one memory page, dynamic SPM allocation based on performance counters can be combined with clustering at load-
time, that is, the SPMM does not only decide which blocks to load or not, it also modifies the block layout to achieve optimal energy savings and runtime performance. Dynamic SPM allocation that is based on performance counters can be implemented, for example, in a virtual machine such as a JVM.

Applying Clustering Techniques to Desktop Computers and Servers

Code and data clustering methods that reduce the size of the working set could also prove interesting for standard desktop computers and servers. Considering the trend for more and more virtualization, there is never enough physical main memory available. By applying the clustering techniques presented in this thesis to server application or applications run in virtual machines, the amount of memory pages swapped out to disk (paging) can be reduced.
Bibliography


[6] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M.Balakrishnan, and Peter Marwedel. Scratchpad memory: A design alternative for cache on-chip


[27] MP3 Reference Decoder.


초록

최근 휴대용 기기들이 점점 강력해지고 더욱 더 많은 부가 기능을 제공하고 있다. 하지만 사용의 편리성과 기능의 발전에 반해 배터리의 수명은 여전히 중요한 문제로 남아있다. 배터리로 구동되는 휴대용 기기의 에너지 소모를 줄이기 위한 연구는 이미 많이 존재하는데, 프로세서를 유휴 상태로 유지하여 프로세서 공급 전압을 줄인다거나, 특정 대기 시간이 지난 후에 화면의 backlight 를 어둡게 하는 방법들이 그 예이다.

본 논문은 임베디드 프로세서 코어에 내장된 scratchpad memory(SPM)을 이용하여 메모리 시스템의 에너지 소모를 줄이는 방법에 초점을 맞춘다. 임베디드 시스템에서 SPM은 전통적인 하드웨어 캐시를 대체하거나 캐시와 더불어 사용되어 왔으며 캐시에 비해 많은 장점을 가지고 있다. 첫 번째로, SPM은 미리 알 수 있는 고정된 latency로 접근할 수 있다. 이 점은 실시간 작업이 주된 시스템에서 중요한 특성이 다. 두 번째로, SPM은 구조가 간단하기 때문에 크기가 같은 set-associative 캐시에 비해 접근 당 소비 에너지가 매우 작다. 따라서 SPM은 저전력 임베디드 시스템 분야에서 많이 연구되어 왔다. 하지만 휴대용 기기의 점점 가상 메모리(virtual memory)와 선점형 멀티테스킹 (preemptive multitasking) 등의 완전한 기능을 갖춘 운영체제를 도입하고 있는데 반해 SPM의 관리 기법에 대한 이전의 연구는 주로 이미 알려진 하드웨어 구성에 대해 동작하는 한 개의 응용 프로그램에만 초점을 맞쳐져 있었다.

본 논문은 가상 메모리와 선점형 멀티테스킹 기능을 갖춘 시스템에서
동작하는 동적 SPM 관리 기법과, 프로세서의 실제 SPM 크기에 독립적 이면서 SPM에 최적화된 실행 이미지를 어떻게 생성하는가에 대한 연구 결과를 소개한다. 기본적인 방법은 자주 접근되는 코드와 데이터를 pageable region으로 놓는 것이다. 실행 시, SPM 관리자는 pageable region 안의 코드와 데이터 접근을 추적하기 위해 memory management unit(MMU)의 page fault 기능을 이용한다. Page fault가 발생할 때마다, SPM 관리자는 요청되었던 페이지를 SPM에 로드하고 사용할 수 있도록 해당 fragment를 관리한다.

SPM에 최적화된 실행 이미지는 postpass optimizer에 의해 생성된다. postpass optimizer는 basic block가 data block에 대한 profiling 정보를 바탕으로 pageable과 cacheable의 두 영역으로 분류한다. 최적의 성능을 위해 같은 시간대에 접근되는 코드와 데이터를 동일 페이지에 배치시켜 불필요한 page fault를 방지한다. 따라서 postpass optimizer는 만약 할만한 코드와 데이터의 배치를 얻기 위해 loop detection과 function splitting 등의 여러 가지 최적화 기법을 적용한다.

Postpass optimizer로 생성된 실행 이미지는 멀티테스팅 환경에 적절하게 동작하기 위해서 SPM의 존재 여부나 크기에 종속되지 않는다. 단일 프로세스 시나리오는 응용 프로그램이 SPM 전체를 사용하지만, 다중 프로세스 환경은 그 안에서 SPM이 공유 자원으로 다루어져야 한다. 본 논문은 단일 프로세스 환경의 SPM 관리 기법뿐만 아니라 선행형 멀티 테스팅 시스템에서 SPM을 공유하는 세 가지 기법을 제안한다. 그리고 그 기법들이 기존의 운영체제 안에 어떻게 구현될 수 있는지 설명한다.

캐시와 SPM을 모두 포함하고 있는 프로세서 코어는 짧은 latency를 유지하기 위해 메모리 접근요청이 있으면 캐시와 SPM을 동시에 접근한
다. 이러한 불필요한 동시 접근은 많은 에너지 낭비를 초래하는데 본 논문은 이를 해결하기 위해 물리 메모리를 접근하기 위한 주소 변환이 캐시 및 SPM 접근에 순차적이어서 캐시나 SPM 중 한 쪽을 선택적으로 접근할 수 있는 메모리 시스템(vertically partitioned memory system)을 제안한다. 명령어 쪽의 메모리는 기존의 명령어 캐시 크기 가 큰 SPM과 상대적으로 크기가 작은 direct-mapped 캐시로 대체된 다. 반면에 데이터 쪽의 메모리는 기존의 데이터 캐시 크기가 매우 작 은 SPM이 추가된다.

본 논문에서 소개한 동적 SPM 관리 기법은, 휴대용 기기에 대표적으 로 사용되는 15개의 단일 프로세스와 10개의 다중 프로세스 벤치마크를 사이클 단위의 정확성을 가진 프로세스 코어 시뮬레이터에서 실행하여 그 성능을 측정하였다. 단일 프로세스 환경에서 코드만, 데이터만, 또는 코드와 데이터를 동시에 SPM에 할당하여 제안한 방법의 효율성을 실험을 통해 보였으며, MMU의 페이지 크기가 성능에 끼치는 영향을 분석하였다. 다중 프로세스 SPM 공동 기법의 효과를 보여주기 위해 가상 메모 리와 선점형 멀티테스킹 기능을 갖춘 실행 환경을 구현하였다.

실험을 통해 얻어진 결과는 본 논문에서 제안한 SPM 관리 기법과 메 모리 시스템이 에너지 소비와 실행시간의 감소에 매우 효과적임을 보여 준다.

주요어: 코드 베치, 컴파일러, 데이터 베치, 이중 메모리, 멀티테스킹, 페이지징, 이동 시스템, 포스트페스 최적화, 스크래치패드 메모리, 비타 캐시, 가상 메모리

학번: 2003-30778
Acknowledgments

“You are crazy! You will never make it!” was the prevalent reaction of my friends and colleagues back home in Switzerland when I told them that I would move to Korea to pursue my PhD at Seoul National University (SNU). Now, almost exactly five years after I have enrolled as a PhD student, it is not without a certain pride that I can say I have proven them wrong.

This achievement, however, would not have been possible without the immense help and support I have received from professors, co-workers, family and friends during my time as a graduate student. First and foremost I would like to thank my advisors, Professor Heonshik Shin and Professor Jaejin Lee. Had Professor Shin not replied to my very first email and encouraged me to get my PhD at SNU, I would probably not be here right now. He then provided a temporary home in his computer system laboratories (CSLAB) and helped my to overcome the first obstacles of life in Korea. After a little bit less than a year, I moved to Professor Jaejin Lee’s Advanced Compiler Research Lab (ACRL) where I have spent four very interesting, challenging, and sometimes stressful years. Professor Lee was an
excellent advisor and mentor, always treating me with respect and accepting my different cultural background. Without his knowledge, encouragement and the countless discussions I would never have finished. Having experienced the culture shock between Asian and Western countries first hand, he was more than just an advisor to me, more like a senior or even, if I may say so, a friend.

I also thank the members of my PhD committee, Professor Kern Koh, Professor Sang Lyul Min, and Professor Thomas Gross for their helpful comments and feedback on the first drafts of this thesis.

My studies would not have been half as fun without my fellow students in the ACRL. These include Chihun Kim, Choonki Jang, Seungkyun Kim, Kwangsub Kim, Jungwon Kim, Kiwon Kwon, Jongyeong Lee, Taejun Ha, Yoonsung Nam, Posung Chun, and Junghyun Kim. Special thanks go to Seungkyun, Kwangsub, and Jungwon - without their help in deciphering Korean regulations or writing the Korean abstract, I would not have been able to submit this thesis. I would like to mention Chihun and Choonki who have been great friends and contributors to the postpass optimizer, codename mordor. Having been the oldest member and first PhD graduate of the ACRL, I hope that I was a good friend, colleague, and role model for the other students. I also thank Hyeonjin Choi who has brought more than just a few faxes to my desk.

Special thanks go to my friends. With Simon Bühlmann I have shared lots and lots of Korean-style barbecues, Soju, Altang - not to forget the
hangovers. The chat logs of my numerous and endless discussions with Thomas Frey, who has been a very good friend since high school, have achieved an impressive size. I would also like to mention Jürg Randegger who was a great roommate during our time at the Swiss Federal Institute of Technology (ETH Zürich) and always has a free space for me on his couch during my visits to Switzerland. Finally, I thank Miriam Kägi who was and always will be a very special friend.

Last but by no means least, my deepest thanks go to my family. To my parents, Jürg and Annerös Egger, for their unconditional love and support during the last 32 years of my life and in every of my decisions. To my gorgeous younger sister, Regula, with her lovely daughter Emilia and my two younger brothers, Thomas and Martin, for being who you are. It is an honor to be your brother and your son.

Bernhard Egger, Seoul, January 2008