UNIVERSITY OF CALIFORNIA

Irvine

Memory Optimizations and Exploration for Embedded Systems

A dissertation submitted in partial satisfaction of the

requirements for the degree Doctor of Philosophy

in Information and Computer Science

by

Preeti Ranjan Panda

Committee in charge:

Professor Nikil D. Dutt, Chair

Professor Alexandru Nicolau

Professor Daniel D. Gajski

1998

©1998

PREETI RANJAN PANDA

ALL RIGHTS RESERVED

The dissertation of Preeti Ranjan Panda is approved,

and is acceptable in quality and form for

publication on microfilm:

_____

_____

_____

Committee Chair

University of California, Irvine

1998

Dedication

To my parents,
Prahallad Panda and Sanjukta Panda:
The eternal sources of wisdom and inspiration

Contents

List of Figures vi

List of Tables ix

Acknowledgment x

Curriculum Vitae xii

Abstract xv

Chapter 1 Introduction 1
1.1 Embedded System Synthesis 1
1.2 Embedded System Architecture 3
1.3 Thesis Contributions 6
1.4 Thesis Outline 7

Chapter 2 Off-chip Memory Access Optimizations 8
2.1 Incorporating Off-chip Memory Accesses into Synthesis 9
2.1.1 Memory Access Optimizations in DRAMs 11
2.1.2 Representing Memory Accesses for HLS 16
2.1.3 Incorporation of Memory Models in HLS 18
2.2 Power Optimizations in Memory Accesses 25
2.2.1 Mapping Arrays into a Single Memory 27
2.2.2 Mapping Arrays into Multiple Memories 37
2.2.3 Mapping into Multiport Memories 47
2.3 Summary and Status 48

Chapter 3 Data Organization: The Processor Core/Cache Interface 49
3.1 Embedded Processor-based Systems 49
3.2 The Data Organization Strategy 52
3.2.1 Problem Description 52
3.2.2 Memory Organization of Scalar Variables 54
3.2.3 Memory Organization for Array Variables 61
3.2.4 Experiments 66
3.2.5 Mapping into Set-Associative Caches 71

3.3 Data Alignment Technique for Loop Blocking 74
3.3.1 Loop Blocking 74
3.3.2 Data Alignment Strategy 76
3.3.3 Experiments 82
3.3.4 Discussion 86
3.4 Summary and Status 88

Chapter 4 On-chip Vs. Off-chip Memory: Utilizing Scratch-Pad Memory 90
4.1 Scratch-Pad Memory 90
4.2 Problem Description 92
4.3 The Partitioning Strategy 96
4.3.1 Features Affecting Partitioning 96
4.3.2 Formulation of the Partitioning Problem 101
4.3.3 Solution of the Partitioning Problem 102
4.4 Context Switching 103
4.4.1 Programs with Priorities 105
4.5 Generalized Memory Hierarchy 105
4.6 Experiments 106
4.7 Summary and Status 109

Chapter 5 Memory Architecture Exploration 111
5.1 Motivation 111
5.2 Illustrative Example 113
5.3 Exploration Strategy 115
5.4 Memory Performance Estimation 116
5.4.1 Refinement to self-spatial locality 119
5.4.2 Refinement to group-spatial locality 120
5.4.3 Refinement incorporating cache conflicts 122
5.4.4 Computation of Processor Cycles 126
5.5 Experiments 127
5.6 Summary and Status 135

Chapter 6 Case Study: MPEG Decoder 137
6.1 Overview of MPEG Decoder 138
6.2 Memory Exploration of Sample Routines 139
6.2.1 Dequant Routine 139
6.2.2 PLUS Routine 144
6.3 Memory Exploration of MPEG Decoder 147
6.4 Summary and Status 156

Chapter 7 Conclusions 158
7.1 Summary of Contributions 158
7.2 Future Directions 160

Bibliography 162

List of Figures

1.1 Embedded Processor-based Synthesis Flow 4

2.1 Shaded blocks represent the synthesized hardware/off-chip memory interface 9
2.2 Typical DRAM Organization 11
2.3 (a) Timing diagram for Memory Read cycle (b) Model for Memory Read operation 13
2.4 (a) Code for *FindAverage* (b) Treating the memory accesses as independent Reads (c) Timing diagram of *page mode read* cycle (d) Treating the memory accesses as one page mode read cycle 15
2.5 CDFG nodes used for memory operations 16
2.6 CDFG models of memory operations 17
2.7 Ordering of memory operations determines possibility of exploiting R-M-W mode 19
2.8 CDFG Transformation for page mode operation 22
2.9 Synthesis model of a memory-intensive system 27
2.10 Tile-based mapping of a 4×9 array – with tiles of size 2×3 . . . 29
2.11 Example (SOR) – Access Pattern of Array 'u' 31
2.12 Selection of Memory Mapping Strategy 33
2.13 Transition Counts for the different mapping techniques for example SOR 35
2.14 Architecture involving multiple memory modules 38
2.15 Mapping of Arrays to Multiple Physical Memories 39
2.16 Partitioning of an array into multiple array partitions 41
2.17 Deriving logical memories from behavior 43

3.1 Shaded blocks represent the Processor Core/Data Cache/Off-chip Memory subsystem 50
3.2 $a[i], b[i]$ and $c[i]$ map into the same cache line 52
3.3 (a) Access Sequence (b) Closeness Graph for $M = 3$ 55
3.4 Procedure for clustering the variables of the Closeness Graph 57
3.5 (a) Interference Graph (b) Memory Assignment 59
3.6 Procedure for assigning clusters to memory locations 61
3.7 (a) Code showing arrays accessed in loops (b) Interference Graph . 62
3.8 Cost function for expected number of conflicts 65

3.9 Procedure for assigning addresses to arrays 65
3.10 Comparison of cache hit ratios in direct-mapped cache 68
3.11 Comparison of speed-up in direct-mapped cache 69
3.12 (a) Mapping into 4-element direct-mapped cache (b) Mapping into 4-element 2-way set-associative cache (c) Unfavourable Interference Graph for 2-way set-associative cache (d) Optimal data placement for 2-way set-associative cache 72
3.13 Procedure for selecting tile size 76
3.14 (a) Self-interference in a 30×30 tile: Rows 1 and 5 of tile interfere in cache (b) Avoiding self-interference by padding 79
3.15 Algorithm for computing pad size for array 80
3.16 Multiple Tiled Arrays 82
3.17 Matrix multiplication on SS-5: Variation of data cache miss ratio with array dimension 83
3.18 Performance of L-U Decomposition on SS-5: Variation of (1) Data Cache Miss Ratio, (2) Normalized Instruction Count, (3) Normalized Processor Cycles, and (4) MFLOPs with array size 85

4.1 Shaded blocks are targetted by the optimization techniques presented in this chapter 91
4.2 Block Diagram of Typical Embedded Processor Configuration . . . 92
4.3 Division of Data Address Space between SRAM and DRAM 93
4.4 Histogram Evaluation Example 95
4.5 Example Life-Time Distribution 98
4.6 (a) Example Loop (b) Computation of LCF values 100
4.7 Memory space partitioned between data from different programs . 104
4.8 Generalized Memory Hierarchy 105
4.9 Performance Comparison of Configurations A, B, C, and D 109

5.1 Procedure CONV 113
5.2 Memory access pattern in CONV example 113
5.3 Overall algorithm for memory architecture exploration 116
5.4 (a) Example loop (b) General n-level loop nest 118
5.5 (a) Example code (b) Shaded region involved in one iteration (c) Cache mapping 123
5.6 Cache conflict between scalars and arrays 124
5.7 Variation of memory performance with line size for fixed cache size of 1 KB (*Histogram* Benchmark) 128
5.8 Variation of memory performance with different mixes of cache and Scratch-Pad memory, for total on-chip memory of 2 KB (*Histogram* Benchmark) 129
5.9 Variation of memory performance with total on-chip memory space (*Histogram* Benchmark) 131

5.10 Memory exploration for *Lowpass* benchmark 133
5.11 Memory exploration for *Beamformer* benchmark 134

6.1 Overview of MPEG Decoder structure 138
6.2 Code for *Dequant* Routine 140
6.3 Variation of memory performance with line size for fixed cache size of 1 KB (*Dequant* Routine). No Scratch-Pad memory. 141
6.4 Variation of memory performance with different mixes of cache and Scratch-Pad memory, for total on-chip memory of 1 KB (*Dequant* Routine) 143
6.5 Code for *PLUS* Routine 144
6.6 Variation of memory performance with line size for fixed cache size of 1 KB (*PLUS* Routine). No Scratch-Pad memory. 146
6.7 Variation of memory performance with different mixes of cache and Scratch-Pad memory, for total on-chip memory of 1 KB (*PLUS* Routine) 147
6.8 Variation of memory performance with line size for fixed cache size of 1 KB (*MPEG Decoder*). No Scratch-Pad memory. 151
6.9 Variation of memory performance with different mixes of cache and Scratch-Pad memory, for total on-chip memory of 1 KB (*MPEG Decoder*) 153
6.10 Variation of memory performance with total on-chip memory space (*MPEG Decoder*) 155

List of Tables

2.1 Transition Count Reduction for Image Processing Examples 36
2.2 Transition Count Reduction due to Multiple Memories 46
2.3 Transition Count Reduction for a Specific Library 47

3.1 Ratio of Execution Times: TSS/DAT, ESS/DAT, and LRW/DAT . 87

6.1 Characteristics of Routines in the MPEG Decoder 149
6.2 Array Accesses in the MPEG Decoder 150

Acknowledgment

I take this opportunity to express my gratitude to my advisor, Professor Nikil Dutt for his role in making the Ph.D. program at U. C. Irvine an utterly enjoyable experience for me. The importance of staying focussed on a research topic for an extended period cannot be overemphasised, and it is my fortune to have worked with Professor Dutt, who helped me remain on course right from the beginning. I have learned many things from him during this fruitful relationship. My technical writing has improved by leaps and bounds over the last few years as a result of the innumerable iterative refinement cycles our papers have undergone. I have acquired most of my presentation skills from the several rehearsals of my talks that Professor Dutt has patiently endured and constructively criticised. His vast knowledge and experience were extremely useful in our technical discussions. Further, the 'Dutt Library' in his office is a veritable gold mine. It pains me to have to part with the hundreds of books, journals, and conference proceedings I have steadily filched from him over the years.

It is my fortune to have been co-supervised by Professor Alex Nicolau. I have greatly benefitted from the discussions I have had with him. His technical insights, cogent arguments, phenomenal vocabulary, and sense of humour are all worthy of appreciation. I am deeply indebted to him for his guidance and support.

Professor Daniel Gajski has been a very valuable source of inspiration throughout my graduate program. He has led an infinite number of interesting and informative discussions during the friday afternoon CADLAB meetings, and has contributed in many ways towards the broadening of my perspective. I thank him for serving on my committee.

I have had the opportunity to work closely with Professor Hiroshi Nakamura from the University of Tokyo when he was a visiting faculty at U. C. Irvine. It has been a great honour and pleasure to work with him. He was the primary contributor of many of the ideas discussed in Chapter 3. My thanks are due to Professor Tomas Lang for his detailed critique of my research. I thank Professor Lang and Professor R. Jayakrishnan for serving on my Ph.D. candidacy examination committee.

My heartfelt thanks are due to many friends in Irvine who made the Ph.D. program a cherishable experience for me. Prashant Mundkur, who is virtually pursuing several simultaneous Ph.D.'s on wide ranging topics from image processing to theoretical physics to film studies, has always been around to enhance the quality of an otherwise mundane day; Chittur Subbaraman's irrational vegetarianism arguments and Ashish Das' tandoori dishes are now folklore; Nupur Pande and Priyadarshini Rao have shared many intricacies of Indian classical music; and Mahadevan "Mad" Iyer has provided many hilarious antics, all at his own expense. I have gained numerous insights from my interaction with the members of U. C. I. CADLAB: Gaurav Aggarwal, Roger Ang, Ana Azevedo, Smita Bakshi, Antenor Carvalho, En-Shou Chang, Viraphol Chaiyakul, Rainer Dömer, Nong Fan, Vijay Ganesh, Alina and Peter Grun, Sumit Gupta, Tedd Hadley, Ashok Halambi, Asheesh Khare, Jon Kleinsmith, Pradip Jha, Erica Juan, Champaka and Logie Ramachandran, Aniruddha Mitra, Hartej Singh, Min Xu, and Jianwen Zhu. Finally, I thank my parents for their continued encouragement all through my career.

Curriculum Vitae

Preeti Ranjan Panda

1969	Born Berhampur, India
1990	B. Tech. in Computer Science and Engineering, Indian Institute of Technology, Madras, India
1990-1993	Software Design Engineer, Texas Instruments (India), Bangalore, India.
1993-1994	Teaching Assistant, Department of Information and Computer Science, University of California, Irvine, USA
1994-1997	Research Assistant, Department of Information and Computer Science, University of California, Irvine, USA
1995	M.S. in Information and Computer Science, University of California, Irvine, USA
1998	Ph.D. in Information and Computer Science, University of California, Irvine, USA Dissertation: *Memory Optimizations and Exploration for Embedded Systems*

Publications

Book Chapters

1. "Improving Cache Performance through Tiling and Data Alignment," P. R. Panda, H. Nakamura, N. D. Dutt, and A. Nicolau, *Solving Irregularly Structured Problems in Parallel*, G. Bilardi, et al., eds., Lecture Notes in Computer Science, Volume 1253, Springer-Verlag, Heidelberg, Germany, 1997.

Journal Papers

1. "Memory Data Organization for Improved Cache Performance in Embedded Processor Applications," P. R. Panda, N. D. Dutt, and A. Nicolau, *ACM Transactions on Design Automation of Electronic Systems*, 2(4), October 1997.
2. "Low Power Memory Mapping through Reducing Address Bus Activity," P. R. Panda and N. D. Dutt, *To Appear in IEEE Transactions on VLSI Systems.*
3. "Incorporating DRAM Access Modes into High-Level Synthesis," P. R. Panda, N. D. Dutt, and A. Nicolau, *To Appear in IEEE Transactions on CAD.*
4. "Estimating the Complexity of Synthesized Designs from FSM Specifications," B. Mitra, P. R. Panda, and P. P. Chaudhary, *IEEE Design and Test of Computers, March 1993.*

Conference Papers

1. "Exploiting Off-chip Memory Access Modes in High-level Synthesis," P. R. Panda, N. D. Dutt, and A. Nicolau. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*, pages 333–340, San Jose, CA, November 1997.
2. "A Data Alignment Technique for Improving Cache Performance," P. R. Panda, H. Nakamura, N. D. Dutt, and A. Nicolau, In *Proceedings IEEE International Conference on Computer Design*, pages 587–592, October 1997.
3. "Architectural Exploration and Optimization of Local Memory in Embedded Systems," P. R. Panda, N. D. Dutt, and A. Nicolau, In *Proceedings 10th International Symposium on System Synthesis*, pages 90–97, September 1997.

4. "Efficient Utilization of Scratch-Pad Memory in Embedded Processor Application," P. R. Panda, N. D. Dutt, and A. Nicolau, In *Proceedings European Design and Test Conference*, March, 1997.

5. "Behavioral Array Mapping into Multiport Memories Targeting Low Power," P. R. Panda and N. D. Dutt, In *Proceedings of the 10th International Conference on VLSI Design*, pages 268–272, Hyderabad, India, January 1997.

6. "Memory Organization for Improved Data Cache Performance in Embedded Processors", P. R. Panda, N. D. Dutt, and A. Nicolau, In *Proceedings 9th International Symposium on System Synthesis*, pages 90–95, November 1997.

7. "Low Power Mapping of Behavioral Arrays to Multiple Memories", P. R. Panda and N. D. Dutt, In *Proceedings International Symposium on Low Power Electronics and Design* , Monterey, CA, August 1996.

8. "Reducing Address Bus Transitions for Low Power Memory Mapping," P. R. Panda and N. D. Dutt. In *Proceedings, European Design and Test Conference*, pages 63–67, Paris, France, March 1996.

9. "1995 High Level Synthesis Design Repository," P. R. Panda and N. D. Dutt, In *International Symposium on System Synthesis*, pages 170–174, Cannes, France, September 1995.

10. "Fibre Channel Protocol: Formal Specification and Verification", V. Nagasamy, S. Rajan, and P. R. Panda, Silicon Valley Networking Conference, San Jose, CA, April 1995.

11. "Estimating the Complexity of Synthesized Designs from FSM Specifications", B. Mitra, P. R. Panda, and P. P. Chaudhary, In *Proceedings of the 5th International Conference on VLSI Design*, pages 175–180, Bangalore, India, January 1992.

12. "A Flexible Scheme for State Assignment Based on Characteristics of the FSM", B. Mitra, P. R. Panda, and P. P. Chaudhary, In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 226–229, Santa Clara, CA, November 1991.

Abstract of the Dissertation

Memory Optimizations and Exploration for Embedded Systems

by

Preeti Ranjan Panda

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 1998

Professor Nikil D. Dutt, Chair

The increase in level of abstraction of modern VLSI designs is accompanied by a corresponding increase in complexity of the building blocks that constitute the design library available to the tools used to automatically synthesise these designs from behavioral specifications. Modern design libraries frequently consist of pre-designed mega-cells such as embedded microprocessor cores and memories. The thesis aims at efficient utilization of memory-based modules such as Data Cache, Scratch-Pad Memory, and DRAM during System Synthesis. During System Synthesis, a behavioral specification could be realized in either hardware or software. In the case of hardware realization, CDFG restructuring techniques are proposed for exploiting efficient access modes of modern DRAMs in High-Level Synthesis.

The case of a software realization by mapping a behavior into an embedded processor is characterized by several unique optimization opportunities that have not been addressed by general purpose compiler and architecture research. Applications involving large arrays typically store such data in off-chip DRAMs, with a small amount of on-chip memory, in the form of data cache and Scratch-Pad memory. We describe techniques for memory data organization in order to improve cache performance, followed by a technique for partitioning data between on-chip Scratch-Pad memory and off-chip memory with the objective of minimizing off-chip memory accesses. Finally, we outline a strategy for architectural exploration of on-chip data memory, consisting of appropriate sizing and parameterization of data cache and Scratch-Pad memory, that involves a rapid estimation of the expected performance for various memory configurations. This estimation can serve as an important feedback to the system designer, who can then select an appropriate memory structure for the synthesized behavior.

Chapter 1

Introduction

1.1 Embedded System Synthesis

Modern digital circuits can be broadly classified into two categories: General-purpose systems, which can be used in a wide range of applications, e.g., microprocessors, microcontrollers, and memories; and Application-specific systems or Embedded systems, which perform the function of a single application, e.g., chips designed for fax/modem and automotive engine control. Time-to-market considerations have driven the need for automation in the embedded systems design industry. Thus, although a completely manual approach relying on a designer's intuition, helps reduce chip area and improve performance and power metrics, the time spent by a human designer to optimize the complex chips designed today is prohibitively large, and the manual approach is impractical from economic considerations. Thus, most application-specific systems rely on the use of Computer Aided Design (CAD) tools that generate the electronic circuitry from a high-level description of the chip. The process of automatic transformation of a more abstract behavioral description into a less abstract (hence, more detailed) level of specification is called *Synthesis*.

The existence of a multitude of implementation possibilities for a single behavior makes the search for an optimal design a challenging task for synthesis tools, which operate at various levels of abstraction. *Layout Synthesis* is the task of generating mask-level layouts from logic gate-level descriptions. *Logic Synthesis* is the task of generating optimized logic circuits from register-transfer-level descriptions. *Behavioral Synthesis* or *High-level Synthesis* is the task of generating a register-transfer-level description from a behavioral description[GDLW92]. Finally, *System Synthesis* refers to the task of automatic generation of more complex systems which could have a hybrid implementation, consisting of both hardware and software components.

The increase in level of abstraction is accompanied by a corresponding increase in complexity of the building blocks that constitute the design library available to the synthesis tools. For example, logic synthesis tools construct an implementation consisting of a network of logic gates. Behavioral synthesis tools realize an implementation in terms of register-transfer components such as adders, multipliers, and multiplexers. In the case of system synthesis, the design library frequently consists of complex blocks such as microprocessor cores, memories, numeric co-processors, and modules implementing standardized functions such as JPEG[Wal91]. For example, the CW33000 processor core from LSI Logic, the TMS320 series of DSP processors from Texas Instruments, the StrongARM processor from Advanced RISC Machines are available in the form of *embedded processor cores*, in addition to the standard packaged form. Memory-based mega-modules are widely used in the industry today in the form of cache, *Scratch-Pad SRAM*[LSI92] and embedded DRAM[Wil97, Mar97].

Embedded processor-based system design is characterized by various optimization opportunities that have not been addressed by traditional compilation and architecture research for general purpose processors. Compiler research has focussed on techniques for efficient generation of code for a given architecture. While embedded systems do benefit from these techniques, they also present an additional degree of freedom arising from the flexibility of the underlying architecture itself. This leads to various novel research directions involving both compilation and architectural issues, such as application-specific instruction set design, and application-specific memory architectures. Such issues have not been addressed in traditional architecture research, because the designer of a general-purpose microprocessor has to optimize the architecture to execute a large variety of applications efficiently, and consequently, usually selects the features that lead to the best average performance over a set of benchmark applications.

In this thesis, we focus on the topic of memory-related optimizations for embedded systems. We present an exploration strategy for determining an efficient on-chip data memory architecture based on an analysis of a given application.

1.2 Embedded System Architecture

Figure 1.1 shows a sample flow and architectural model in embedded system design. A behavioral specification is first analyzed in a Hardware/Software partitioning step[GVNG94] to determine what part is synthesized into hardware and what part is implemented as software. The portion of the specification mapped into hardware is translated into the *Synthesized HW* block shown in Figure 1.1,

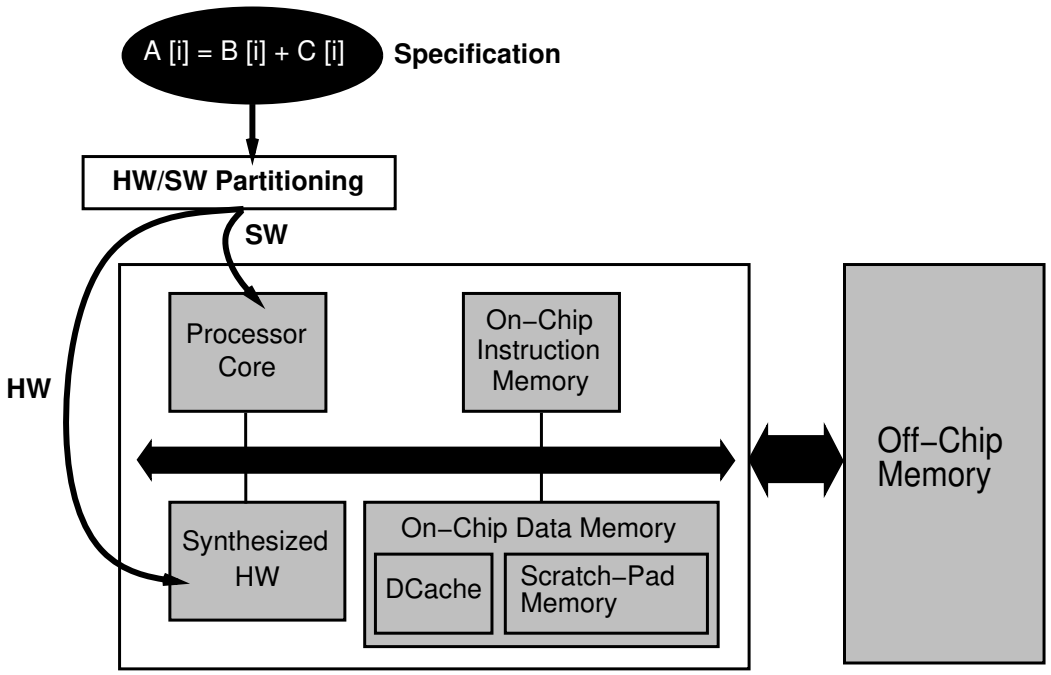


Figure 1.1: Embedded Processor-based Synthesis Flow

and the portion mapped into software is executed on the *Processor Core* block. In addition, the architecture has to support memory devices for storing the code executed by the processor core and for storing the data used during computation.

The on-chip instruction memory consists of instruction ROM and/or instruction cache. The on-chip data memory consists of data cache and Scratch-Pad memory. Scratch-Pad memory is on-chip data memory, to which the assignment of data is compiler-controlled – a portion of the total data memory space is stored in on-chip SRAM, instead of being accessed through the cache. The Scratch-Pad memory is used to store critical data, so that it is guaranteed to be present on-chip during every access, and is never subject to cache misses. In some modern embedded systems, the Scratch-Pad memory assumes the form of embedded DRAM[Wil97, Mar97]. Although we associate Scratch-Pad memory with only data

in this thesis, it could also be useful in the context of instructions. Usually, the code executed by the embedded processor is small enough to reside on-chip, but if the code is too large and has to be stored off-chip, Scratch-Pad memory could be utilized to store frequently executed instruction segments.

The synthesized hardware block uses the same bus to interface with the memory, as the processor core. However, while the processor core requires a memory controller (not shown in diagram) to interface with off-chip memory, the synthesized hardware could integrate the function of the memory controller (or parts of it) within itself.

In addition to the blocks shown in Figure 1.1, an embedded system may contain other modules such as co-processors. One important aspect of the embedded system architecture is that it is highly parameterizable. The processor core (and co-processor, if any) can be assumed to be fixed for our purpose, although researchers are also studying the impact of application-specific tailoring of the processor core itself[LMP94]. The synthesized hardware block is the result of high-level synthesis of the portion of the specification mapped into hardware. The on-chip instruction and data memory can be tailored for the specific application; clearly, the optimal on-chip memory configuration is a function of the characteristics of the application being synthesized. In this context, the interface of both the processor core and synthesized hardware block with on-chip and off-chip memory provides a rich set of interesting optimization opportunities with respect to performance and power, that form the core of this thesis.

1.3 Thesis Contributions

In this thesis, we address optimization and exploration issues that arise in different subsets of the architecture shown in Figure 1.1. Behavioral memory accesses could be realized in either software or hardware – we assume that this decision has been taken manually by the designer. In the case of memory accesses mapped into hardware (i.e., high-level synthesis), we address the interface between the synthesized hardware block and off-chip memory. In the case of memory accesses mapped into software, we address a subset of the architecture in Figure 1.1, with the processor core, the on-chip data memory blocks, and off-chip memory. The processor core/instruction cache interface has been addressed by other researchers[TY96a, TY96b, KLPMS97]. The problem of an integrated optimization and exploration for the entire architecture shown in Figure 1.1 is a challenging research topic; we address several aspects of this problem in this thesis. Specifically, we make the following contributions:

- We present models for incorporating off-chip memory access modes in behavioral synthesis, and optimization techniques for exploiting efficient memory access modes.
- We present techniques for storing data in memory with the objective of enhancing performance and power characteristics of embedded systems.
- We present a fast exploration strategy for evaluating the effect of different on-chip memory architectures on system performance. This exploration provides valuable feedback to the system designer, who can then select an optimal memory architecture for his application. We focus on data memory exploration.

1.4 Thesis Outline

The rest of the thesis is organized as follows:

Chapter 2: Off-chip Memory Access Optimizations. We describe the organization of off-chip memories, identifying optimization possibilities for performance and power. We then proceed to present techniques for incorporating off-chip memory organization during synthesis to enhance performance, and storing application data in off-chip memories to minimize power consumption.

Chapter 3: Data Organization: The Processor Core/Cache Interface. We present techniques for aligning data in off-chip memory targeting improvement in cache performance, and hence, reduction in memory traffic.

Chapter 4: On-chip Vs. Off-chip Memory: Utilizing Scratch-Pad Memory. We present an algorithm to efficiently partition application data between on-chip Scratch-Pad memory and off-chip memory so as to maximize performance.

Chapter 5: Memory Architecture Exploration. We present an analytical scheme for rapidly evaluating the impact of different on-chip memory configurations on the expected performance of an application.

Chapter 6: Case Study: MPEG Decoder. We present a case study of our data memory architecture exploration strategy on the MPEG decoder application.

Chapter 7: Summary and Future Work. We present our conclusions and identify possible directions of future research arising from this work.

Chapter 2

Off-chip Memory Access Optimizations

Due to the widening gap between processor and memory speeds, the memory interface becomes a serious performance bottleneck because off-chip memory access is a relatively slow operation, incurring 10-20 times the delay of a typical datapath operation [PH94]. The consideration of memory issues is critical because memory storage dominates chip area and power in signal processing applications – typical on-chip memory area is reported to be almost 10 times the typical area of a complex datapath[BCM95]. Thus, it becomes imperative to consider the effects of memory accesses during synthesis, in order to realize efficient system designs. In this chapter, we address performance and power optimization issues in the interface between the synthesized hardware and off-chip memory blocks of the embedded system architecture, which are highlighted in Figure 2.1.

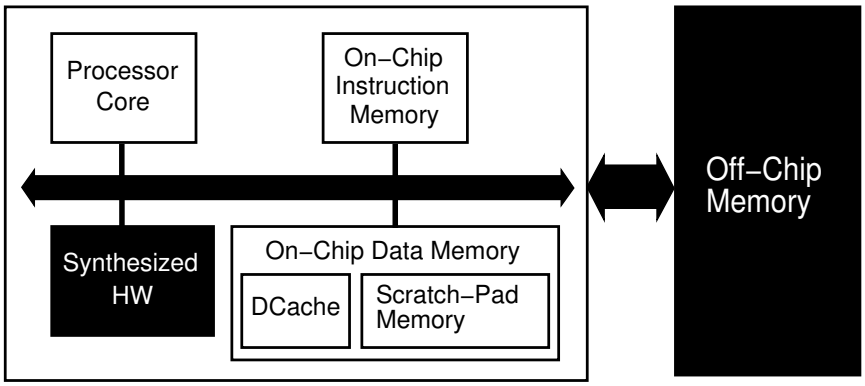


Figure 2.1: Shaded blocks represent the synthesized hardware/off-chip memory interface

2.1 Incorporating Off-chip Memory Accesses into Synthesis

Arrays in behavioral specifications are typically assigned to memories during synthesis. If these arrays are small enough, they may be mapped into on-chip SRAM. However, many applications involve large arrays, which need to be stored in off-chip memories, such as DRAM. Consequently, it is essential to employ a reasonably accurate model for memory operations during synthesis. Modern memories have efficient access modes (such as *page mode* and *read-modify-write*), that are known to improve the access bandwidth[Fly95]. Since the newer generation of memories (Extended Data Out DRAMs, Synchronous DRAMs, etc.) all incorporate these access modes, it is important to develop a methodology for incorporating realistic memory library interface protocols into high-level synthesis, so that scheduling, allocation and binding algorithms can exploit these features to generate faster and more efficient designs.

Several memory-related issues, such as memory *allocation* [BCM94, LvM+93], *packing* [KR94, JD97], *estimation* [BCM95], and *selection* [BG95] have been addressed in previous research on high-level synthesis. Most of them target register files (also referred in literature as *foreground memory*) and on-chip RAMs and ROMs (sometimes called *background memory*) that share space on the same chip as an ASIC. The problem of clustering of behavioral variables into single- and multi-ported register files has been addressed in [SJ92, KL93, RGC94]. However, the above works have not addressed the issues involved in using off-chip memory modules such as DRAM.

In [LKMM95], a scheduling algorithm using *behavioral templates* is described. A behavioral template represents a complex operation in a Control Data Flow Graph (CDFG) by enclosing several individual CDFG nodes, and fixing their relative schedule. Templates allow a better representation of memory access operations than single multicycled CDFG nodes, and lead to better schedules. However, since different memory operations are treated independently, typical features of realistic memory modules (e.g., page mode for DRAM) cannot be exploited, because the scheduler has to assume the worst-case delay for every memory operation, resulting in relatively longer schedules.

Our work, to the best of our knowledge, is the first to handle realistic off-chip memory modes during high-level synthesis. We make the following contributions towards incorporation of off-chip memory accesses in high-level synthesis:

1. We present models for the well-known operation modes of off-chip memories (DRAMs), which can be effectively incorporated into HLS tools.

2. We present algorithms for inferring the applicability of the memory access modes to memory references in the input behavior.

3. We outline techniques for transforming the Control-Data Flow Graph (CDFG) for the input behavior to incorporate the memory access modes, so as to obtain an efficient schedule.

Our proposed technique serves as a pre-pass step to scheduling during high-level synthesis, so that a typical scheduling algorithm, such as list scheduling, can be invoked on the transformed CDFG.

2.1.1 Memory Access Optimizations in DRAMs

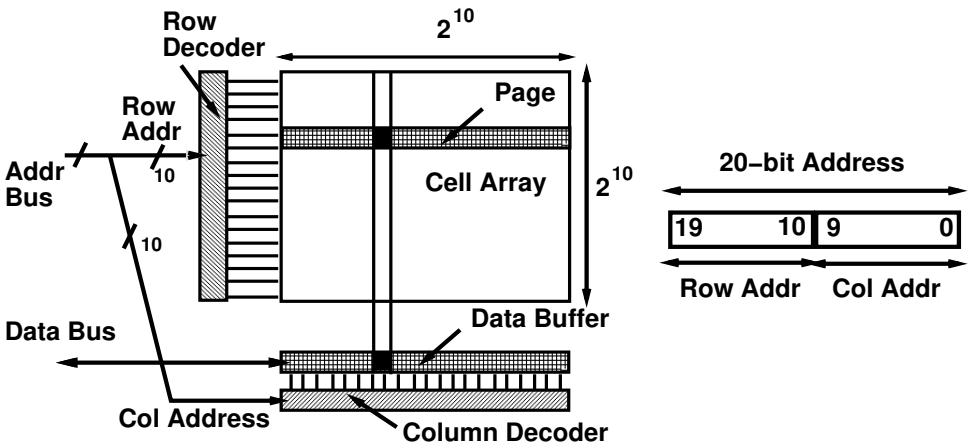


Figure 2.2: Typical DRAM Organization

Figure 2.2 shows a simplified view of the typical organization of a DRAM with 2^{20} words (= 1 M words), with the core storage array consisting of a $2^{10} \times 2^{10}$ square. A 20-bit logical address for accessing a data word in the DRAM is split into a 10-bit *Row Address* – consisting of the most significant 10-bits (bits $19 \ldots 10$); and a 10-bit *Column Address* – consisting of the least significant 10 bits (bits

9 . . . 0). The Row Decoder selects one of 2^{10} rows (or DRAM *pages*[1]) using the row address, and the *Column Decoder* uses the column address to select the addressed word from the selected page. Since the row and column addresses are used over disjoint time intervals, the address bus is time-multiplexed between row and column addresses in order to save pin count. The following feature, called *page mode access*, is an important organizational attribute that results in significant access time reduction: *the row-decoding step physically copies an entire page into a data buffer (Figure 2.2), anticipating spatial locality, i.e., expecting future references to be from the same page. If the next memory access is to a word in the same page, the row-decoding phase can be omitted, and the data fetched directly from the data buffer, leading to a significant performance gain.*

Modern DRAMs commonly utilize the following six memory access modes:

Read Mode – single word read, involving both row-decode and column-decode.

Write Mode – single word write, involving both row-decode and column-decode.

Read-Modify-Write (R-M-W) Mode – single word update, involving read from an address, followed by write *to the same address.* This mode involves one row-decode and column-decode stages each, and is faster than two separate Read and Write accesses.

Page Mode Read – successive reads to multiple words in the same page.

Page Mode Write – successive writes to multiple words in the same page.

Page Mode Read-Modify-Write – successive R-M-W updates to multiple words in the same page.

[1]For simplicity of explanation, we assume that one page = one row in this work.

In order to motivate the need for explicitly modeling and exploiting these modes, we demonstrate the effect on design performance in HLS between scheduling with normal read operation, versus scheduling that exploits the page mode read of a DRAM. The sample library memory module used in this paper is the IBM11T1645LP Extended Data Out (EDO) DRAM[IBM96], and the input behavior is the *FindAverage* routine in Figure 2.4(a), where the scalar variable av is mapped to an on-chip register, and the array $b[0 \ldots 3]$ is stored in off-chip memory.

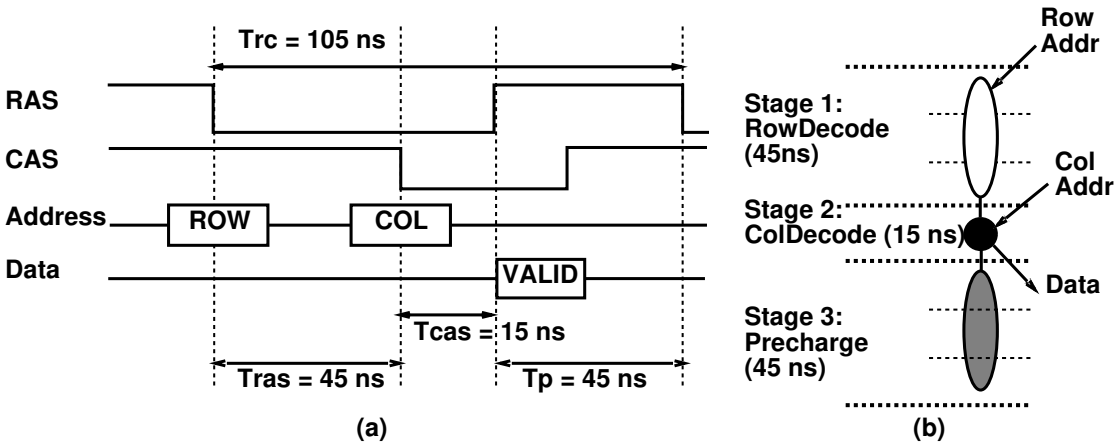


Figure 2.3: (a) Timing diagram for Memory Read cycle (b) Model for Memory Read operation

Figure 2.3(a) shows a simplified timing diagram of the *read cycle* of the $1M \times 64$-bit EDO DRAM. The Memory Read cycle is initiated by the falling edge of the RAS (Row Address Strobe) signal, at which time the row address is latched from the address bus. The column address is latched at the falling edge of CAS (Column Address Strobe) signal, which should occur at least $T_{ras} = 45$ ns later. Following this, the data is available on the data bus after $T_{cas} = 15$ ns. Finally, the RAS signal is held high for at least $T_p = 45$ ns to allow for *bit-line precharge*, which is necessary before the next memory cycle can be initiated.

From the above timing characteristics, we can derive a CDFG node cluster for the memory read operation, which consists of 3 stages (Figure 2.3(b)): (1) row decode; (2) column decode; and (3) precharge. The row and column addresses are available at the first and second stages respectively, and the output data is available at the beginning of the third stage. Techniques for formally deriving the node clusters from interface timing diagrams have been studied in the interface synthesis works such as [COB95], and can be applied in this context.

Assuming a clock cycle of 15 ns, and a 1-cycle delay for the addition and shift operations, we derive the schedule shown in Figure 2.4(b) for the code in Figure 2.4(a), using the memory read model in Figure 2.3(b). Since the four accesses to array b are treated as four independent memory reads, each of these incurs the entire read cycle delay of $T_{rc} = 105$ ns, i.e., 7 cycles, requiring a total of $7 \times 4 = 28$ cycles.

However, DRAM features such as page mode read can be efficiently exploited to generate a tighter schedule for behaviors such as the *FindAverage* example, which access data in the same page in succession. Figure 2.4(c) shows the timing diagram for the page mode read cycle, and Figure 2.4(d) shows the schedule for the *FindAverage* routine using the page mode read feature. Note that the page mode does not incur the long row decode and precharge times between successive accesses, thereby eliminating a significant amount of delay from the schedule. In this case, the column decode time is followed by a *minimum pulse width* duration for the CAS signal, which is also 15 ns in our example. Thus, the effective cycle time between successive memory accesses has been greatly reduced, resulting in an overall reduction of 50% in the total schedule length.

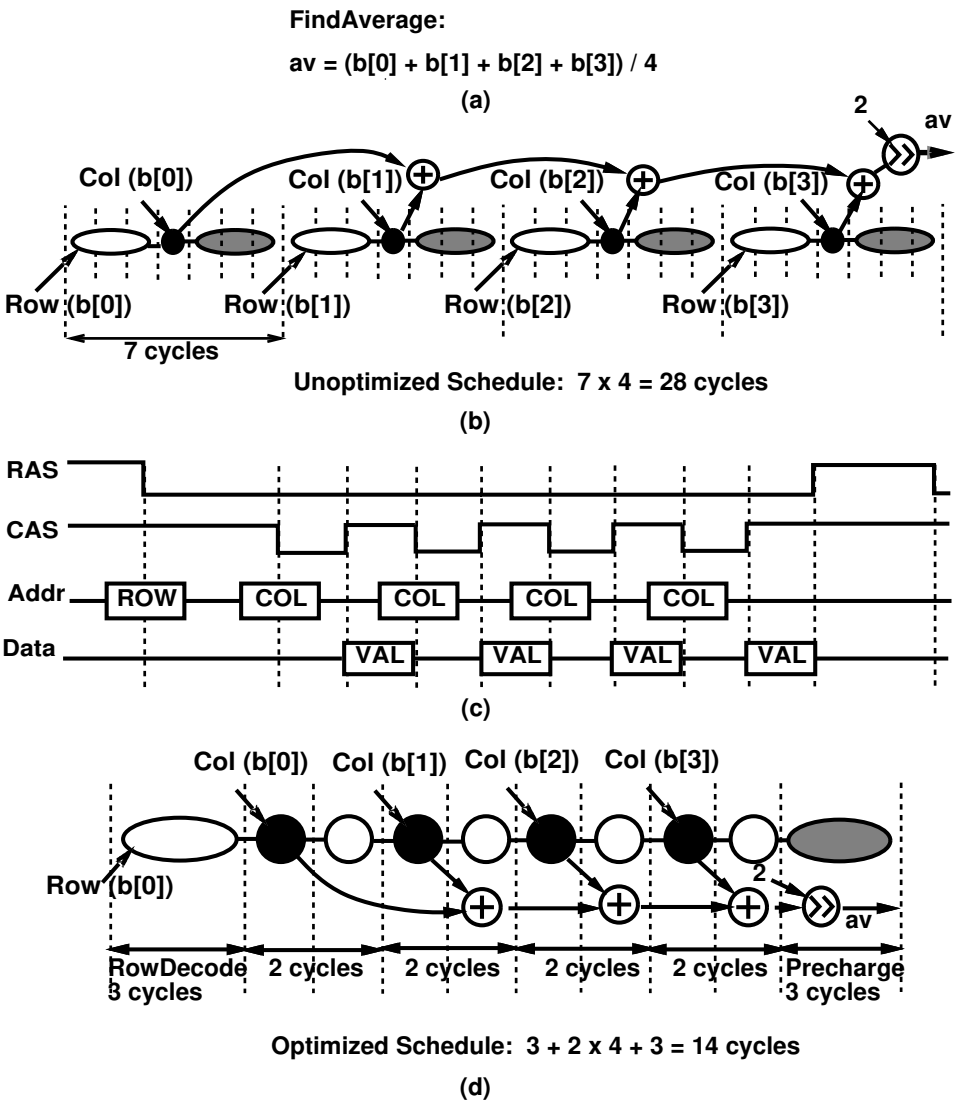


Figure 2.4: (a) Code for *FindAverage* (b) Treating the memory accesses as independent Reads (c) Timing diagram of *page mode read* cycle (d) Treating the memory accesses as one page mode read cycle

The key feature in the dramatic reduction of the schedule length in the example above is the recognition that the input behavior is characterized by memory access patterns that are amenable to the page mode feature, and the incorporation of this observation in the scheduling phase. In the following sections, we first provide synthesis models for the various access modes available in modern off-chip memories, and then describe a technique that incorporates the models into HLS by transforming the input behavior accordingly.

2.1.2 Representing Memory Accesses for HLS

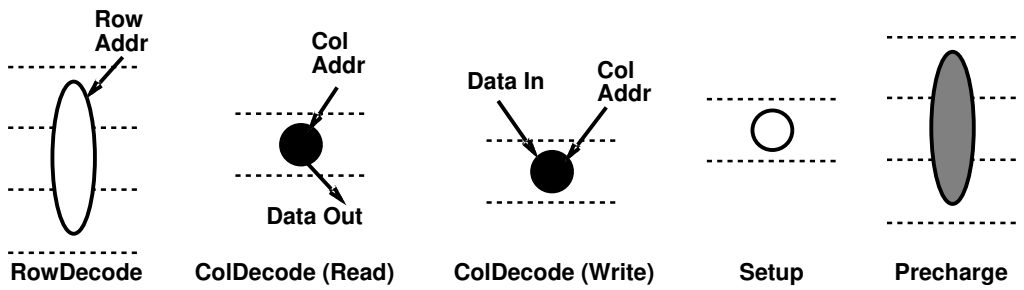


Figure 2.5: CDFG nodes used for memory operations

Instead of a single memory access node in the CDFG, we model memory operations corresponding to the different access modes using finer-grained memory access nodes (Figure 2.5). The **RowDecode** node is a multicycle operator that marks the beginning of any memory operation. The node has one input – the row address of the location, which should be ready before this node is scheduled. The **ColDecode (Read)** and **ColDecode (Write)** nodes form the second stage of the memory read and write operations. The **ColDecode (Read)** node has one input – the column address, and one output – the data read. The **ColDecode (Write)** node has two inputs – the column address, and the data to be written.

The **Setup** node serves as a "delay" node in order to implement minimum delay constraints between successive stages of a memory operation. The **Precharge** node is a multicycle node that marks the last stage of a memory cycle. Physically, it signifies the restoration of the memory bit-lines to the initial state, so that the next operation can be initiated after the completion of this stage.

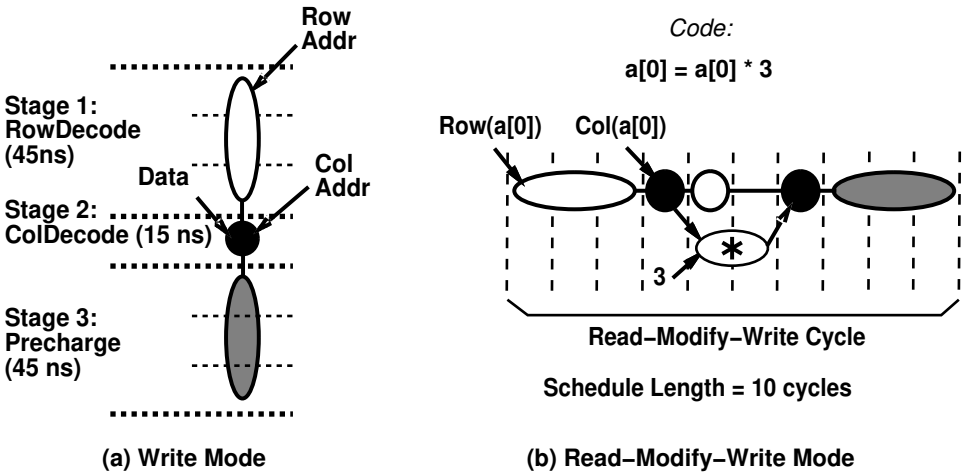


Figure 2.6: CDFG models of memory operations

For each memory access mode, we build composite memory access nodes in the CDFG, based on the access protocol for that mode.

Read Mode: This mode was described in Section 2.1.1.

Write Mode: The memory write operation has similar timing characteristics to the read operation. The CDFG node cluster for the write cycle is shown in Figure 2.6(a). The ColDecode node used in the second stage is the **ColDecode (Write)** node identified in Figure 2.5.

Read-Modify-Write Mode: The Read-Modify-Write (R-M-W) mode is illustrated in Figure 2.6(b) with a simple behavioral statement: $a[0] = a[0] * 3$,

which involves the reading and writing of memory address for $a[0]$. The schedule requires 10 cycles. Note that, an extra control step is introduced between the *setup* and *write* stages, because the '$*$'-operation requires 2 cycles. If the operation were to be modeled as separate read and write cycles, we would require: $2 \times 7 = 14$ cycles.

Page Mode Read: This mode was described in Section 2.1.1. The **Page Mode Write** and **Page Mode R-M-W** modes are constructed similarly.

2.1.3 Incorporation of Memory Models in HLS

The memory operations described in Section 2.1.2 were designed to exploit typical behavioral memory access patterns. This requires analysis of the CDFG to identify behavioral patterns that can be optimized by the various efficient memory access modes, and transformation of the CDFG to incorporate the optimizations. We briefly describe the actions performed. The detailed procedures are explained in [PDN97c].

Clustering of Scalars

An optimization opportunity that can be exploited in the case of scalar variables in a behavior is that, consecutive accesses to two different scalar variables can be implemented as a single page mode operation if both are located in the same memory page. Suppose the off-chip memory has a page size of P words. The following problem needs to be solved: *Group the scalar variables into clusters of size P (to reside in the same memory page), such that the number of consecutive*

accesses to the same memory page is maximized. The technique we use is similar to the solution of an analogous problem in the context of cache memory, where scalar variables are grouped into clusters of size L (where L is the size of a *cache line*), so as to minimize the number of cache misses (Chapter 3).

Reordering of Memory Accesses

The correct ordering of memory accesses is critical for exploiting efficient memory access modes such as R-M-W. For example, in the code: "$a[i] = b[i] + a[i]$", the sequence of accesses: *"Read* $b[i] \rightarrow$ *Read* $a[i] \rightarrow$ *Write* $a[i]$*"* allows the utilization of the R-M-W mode for the address $a[i]$, while the sequence *"Read* $a[i] \rightarrow$ *Read* $b[i] \rightarrow$ *Write* $a[i]$*"* does not allow the mode, because of the intervening *"Read* $b[i]$*"* operation.

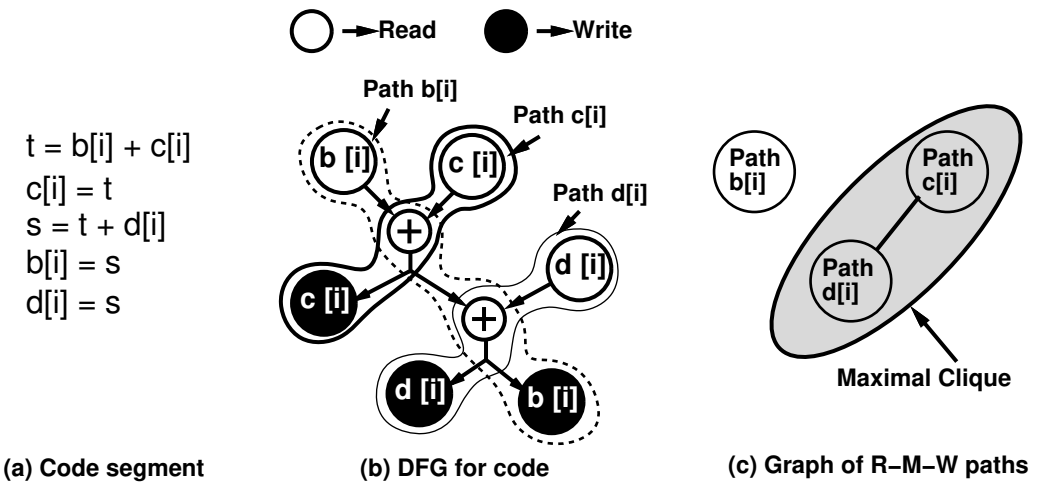


Figure 2.7: Ordering of memory operations determines possibility of exploiting R-M-W mode

Figure 2.7(a) shows an example basic block of code for which the corresponding DFG is shown in Figure 2.7(b). Figure 2.7(b) also shows three possible R-M-W

paths, for addresses $b[i], c[i]$, and $d[i]$. Note that it is not possible to implement the memory accesses to all the three locations as R-M-W operations. For example, if we implement the $c[i]$ path as an R-M-W operation, then the $b[i]$ path can no longer be implemented as R-M-W, because *"Read* $b[i]$*"* has to occur before *"Write* $c[i]$*"*, thus forcing the $b[i]$ path to be split into separate *"Read* $b[i]$*"* and *"Write* $b[i]$*"* operations.

In general, only one of a pair of intersecting paths can be implemented as R-M-W. In order to minimize the schedule length for a DFG, we need to maximize the number of R-M-W paths. The problem can be shown to be NP-complete by building a graph G, in which the nodes represent the DFG paths, and an edge exists between two nodes if the corresponding paths are non-intersecting (Figure 2.7(c)). Determining the maximal number of R-M-W paths in the DFG is now the problem of finding the *maximal clique* in G, which is known to be NP-complete. The greedy heuristic described in [PDN97d] can be used to solve this problem. However, since the number of such paths are very small in typical behaviors, an exhaustive solution can also be used.

Hoisting

Due to the time-multiplexing of the memory address bus between row and column addresses, a scheduling optimization is possible when two addresses in the same memory page are accessed from different paths of a conditional branch. Consider a behavior in which one of array elements $a[0]$ or $a[1]$ is fetched from memory, depending on the result of the conditional evaluation. The knowledge that $a[0]$ and $a[1]$ reside in the same memory page allows us to infer that both

have the same row address, and hence, the read cycle could be initiated before the comparison operation, resulting in a shorter schedule.

Loop Transformations

In order to utilize the page mode operations, the CDFG for the behavior has to be transformed to reflect the page mode operation. Since most of the computation occurs in the innermost loops of nested loop structures, we concentrate on the memory accesses in the innermost loops.

Loops accessing single page per iteration. If a loop accesses locations from only a single memory page per iteration, e.g., there is only one read operation (*Read* $a[i]$) per iteration, the page mode operations can be applied directly by restructuring the loop so that it iterates over the array in blocks of P iterations (where P is the page size, in words), so that we have one page mode read for every P iterations. Figure 2.8(a) shows a section of the CDFG of an example loop with a single memory access ($a[i]$) in one iteration. We introduce an inner loop in which upto P elements from the same page are accessed. The transformed CDFG is shown in Figure 2.8(b). Note that the RowDecode and Precharge nodes enclose the inner CDFG loop, forming one complete page mode operation.

Loops accessing multiple pages per iteration. If more than one array is accessed in one loop iteration, the transformation shown in Figure 2.8 cannot be directly applied, because different arrays usually lie in different memory pages. In such cases, we can use a well known transformation, *loop unrolling*[GDLW92], to create the opportunity for utilizing page mode operations.

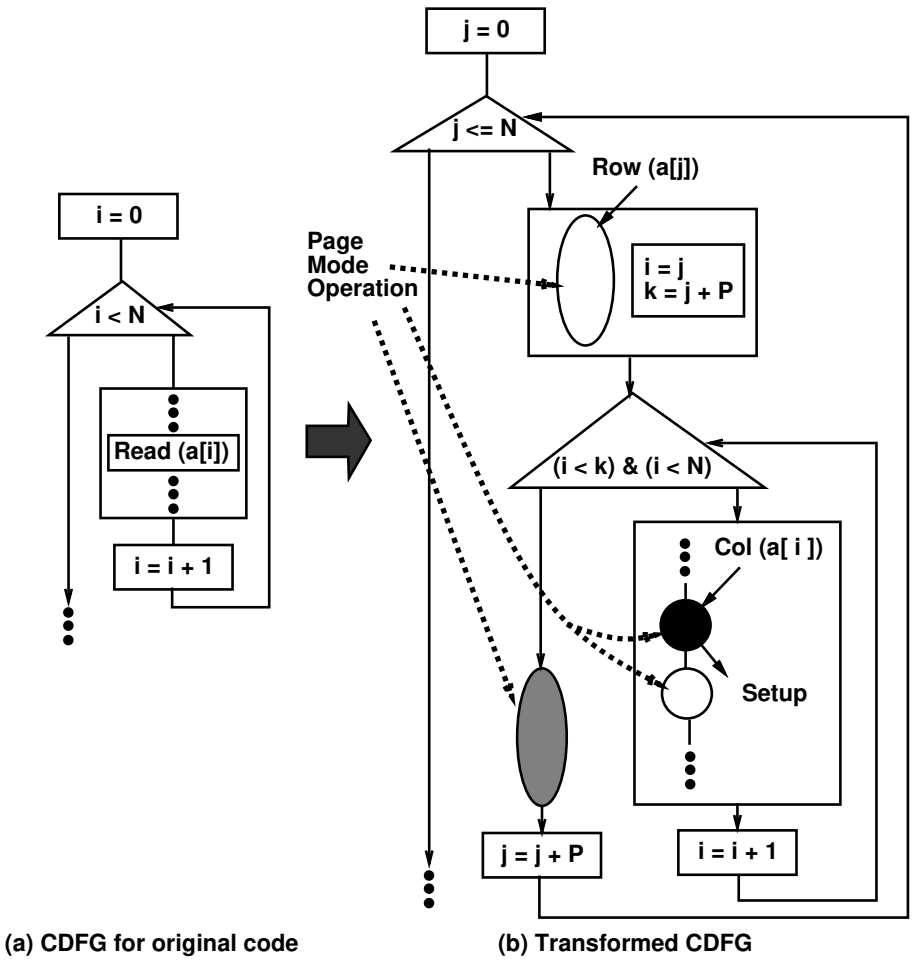


Figure 2.8: CDFG Transformation for page mode operation

Loops with disjoint subgraphs in body. A loop body is said to consist of disjoint subgraphs if the DFG representing the body (basic block) can be divided into two or more subgraphs with no data dependence across their memory operations (i.e., no data dependence edges from one subgraph to the other). In such a case, each subgraph with at least one memory access in it, can be split into a different loop in order to better utilize page mode memory operation.

In order to incorporate the memory access optimizations into synthesis, we transform the CDFG of the behavior into an optimized form, so that a scheduling algorithm, such as list scheduling, can be invoked on the transformed CDFG. After performing the scalar variable clustering step, we perform the Reordering, Hoisting and Loop Splitting transformations in sequence, followed by restructuring of the resulting loops, if applicable. The complexity of the algorithm is that of the dominant loop restructuring (with multiple pages per iteration) step — $O(Bn^2)$, where n is the number of CDFG nodes and B is the number of loop nests in the behavior.

Experiments

We tested our proposed optimizations for utilizing efficient memory access features, on benchmark examples from the digital signal processing and scientific computing domains, all of which share the common characteristic that they process large data arrays. We compared the execution time due the schedules generated by 3 techniques:

Coarse-grain – the traditional HLS approach, where each memory access operation is treated as a multicycled operation.

Fine-grain – a more refined memory access model (e.g., the template strategy of [LKMM95]), with each memory access operation being treated as an independent 3-stage operation.

Optimized – our proposed CDFG transformation-based approach, followed by the application of list scheduling.

The detailed experimental results can be found in [PDN97c]. In summary, the Optimized technique achieves a performance improvement of 45.2% over Coarse-grain, and an improvement of 40.8% over the Fine-grain technique. This is the consequence of efficient utilization of the memory features such as read-modify-write and page mode. The performance improvement for the Optimized technique above could be at the expense of a slight increase in controller area due to increased number of control states as compared to the controllers required for the Coarse-grain and Fine-grain schedules when the page mode operation is used. We observed that the controllers generated from the Optimized schedules have an average area only 14.9% larger than those generated from Coarse-grain, and 10.3% larger than those generated from Fine-grain, for the benchmark examples. Given the significant improvement in performance with our optimizations, we believe the control overhead is justifiable. Our technique for incorporating the synthesis models for memories into HLS is independent of the actual HLS tasks of scheduling, etc., and can be easily incorporated as a pre-processing step into existing HLS design flows.

2.2 Power Optimizations in Memory Accesses

In Section 2.1, we proposed optimization techniques for exploiting locality in memory accesses, which led to significant performance improvements. Locality of reference also plays a major role in reducing power consumption. As explained in Section 2.1.1, consecutive memory accesses from the same page lead to a performance gain because the precharging and row decoding phases can be omitted for the second access. This also results in a corresponding decrease in power consumption, because omission of the precharging and row decoding phases represents reduced switching activity in the circuit.

It is a well known fact that signal transitions, as represented by switchings in logic-level bit values, are the primary source of power consumption in CMOS circuits. Thus, most attempts at low power design of CMOS at the logic or behavioral levels usually include an effort at minimizing the number of signal transitions. We exploit the regularity in memory accesses by deriving address access patterns from the behavior and use these to reduce the number of transitions on the memory address bus.

The reduction of transition counts on the memory address bus has two important implications. First, it leads to a reduction in the switched capacitive load on the off-chip drivers of the address bus, which results in significant power savings, since off-chip capacitances are three orders of magnitude larger than typical on-chip capacitances [Bak88]. Second, reduced activity in the address bus also leads to reduced activity in the memory address buffers and decoding circuitry. Studies have shown that power dissipation in the address decoder and address buffers of

typical memory chips constitute a significant portion of the power consumed (upto 50%) in the memory chip [WCF+94]. This implies that design techniques leading to decrease in power dissipation in this part of the memory will make a significant impact on the overall power dissipation of the application.

In this section, we focus on the reduction of power dissipation in off-chip drivers and in the memory's decoding logic by reducing the number of transitions on the memory address bus. Our technique is an extension of the compiler techniques for reducing the number of memory accesses – we exploit regularity and spatial locality in the memory accesses and determine the mapping of behavioral array references to physical memory locations to minimize address bus transitions. Our technique can be applied in conjunction with other approaches ([WCF+94, SB95]) to obtain further power minimization at the behavioral level.

We have studied the address bus power reduction problem for three different memory configurations: (1) single memory (single-port), (2) multiple memories (single-port), and (3) single multiport memory.

2.2.1 Mapping Arrays into a Single Memory

Problem Definition

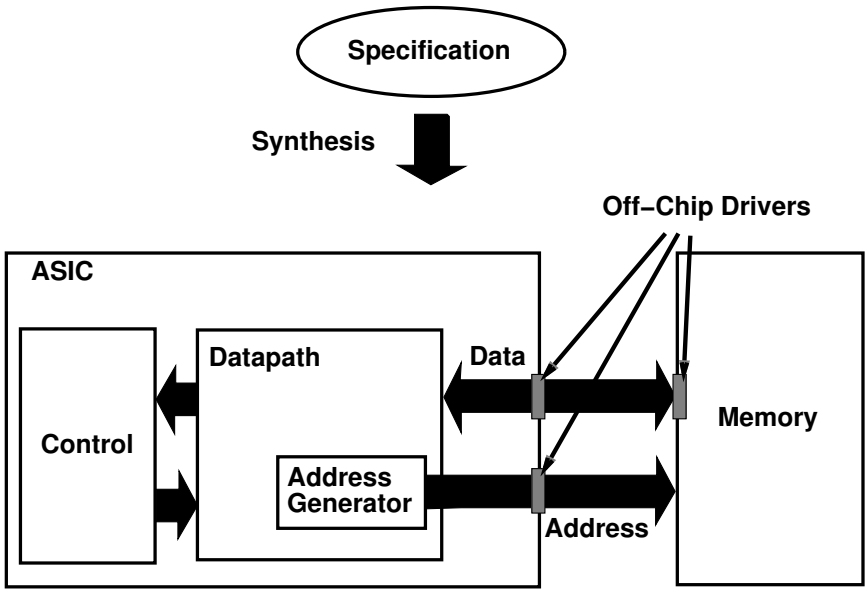


Figure 2.9: Synthesis model of a memory-intensive system

Figure 2.9 shows a more detailed view of the highlighted blocks in Figure 2.1 for a typical memory-intensive system, synthesized into an ASIC (consisting of datapath and control blocks) and an off-chip memory. Memory accesses result in power dissipation in both the ASIC and memory. Power dissipation in the ASIC is the sum of power dissipation in its constituents – the datapath, the control, and the off-chip drivers. A reduction in the transition count of the address bus also entails power savings in the off-chip drivers of the ASIC.

Power consumption during memory accesses within a memory can be roughly classified into two categories – *data-related* and *address-related*. *Data-related* power is the fraction of memory power dissipated during the transfer of the data between

the data bus and the memory location where the data is stored, and the dissipation in the refresh circuitry (if any) of the memory. *Address-related* power represents the fraction contributed by the address buffers and decoders. Since data-related power is dependent on the specific input data to the application, we have no control over minimizing power without knowledge of the data profile. However, we always have an opportunity to minimize address-related power, since the application's memory access sequence patterns are known at the specification stage. Our approach exploits these memory access sequences to guarantee a reduction in the memory address bus transitions, thereby leading to guaranteed power reduction during memory accesses.

Our technique takes as input the behavioral specification that is to be synthesized; the output is the assignment of arrays in the specification to physical addresses in memory. For simplification of our discussion, we have assumed that every array element occupies one memory word, although this assumption is not necessary in the analysis.

Array Mapping Strategy

We first assign the arrays in the specification to locations in a *logical memory*, and employ a Gray Code Converter (GCC) in the *address generator* (Figure 2.9) to map logical memory addresses to physical memory locations. The GCC converts a logical address into *Gray code* [Koh78] thereby ensuring that access of consecutive logical memory addresses results in the transition of exactly one bit on the memory address bus. The GCC helps bring down the address bus transition count

to a minimum, but our memory mapping strategies are valid techniques for ensuring reduction in the transition count even in the absence of the GCC from the architecture.

We concentrate on memory mapping strategies for two dimensional arrays. We consider three strategies when we attempt to find an effective mapping of arrays into memory:

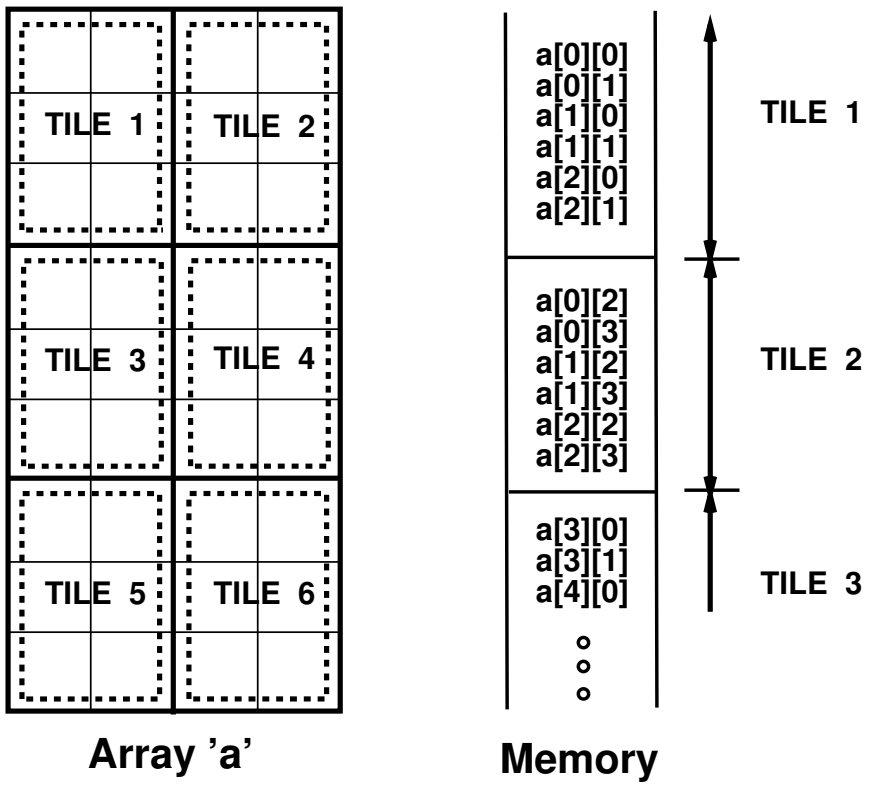


Figure 2.10: Tile-based mapping of a 4×9 array – with tiles of size 2×3

Row-Major. A simple way of mapping a logical array to physical memory (unless the order is imposed by the language) is to store the elements in row-major form [ASU93], that is, the elements of the first row are placed in consecutive memory locations in order of increasing column index. This is followed by the elements of the second row in the same order, and so on.

Column-Major. In column-major mapping [ASU93], the elements of the two dimensional array are stored column-by-column.

Tile-Based. This mapping style first partitions the behavioral array into tiles of smaller rectangles. Here, the elements of a row (or column) of tiles are stored in consecutive memory locations, with the tiles themselves being stored in either row or column major format (Fig. 2.10).

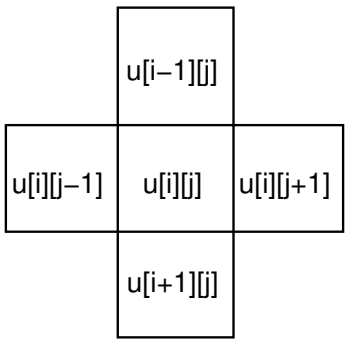
Similar ideas have been used in the context of compilers, where cache reuse is improved by dividing a loop's iteration space into tiles and transforming the loop nesting structure to iterate over the tiles [DFBS94]. Block decomposition of arrays in a multiprocessor environment [DFBS94] is also based on a similar concept.

Figure 2.11(a) shows a simplified version of the code kernel for a Successive Over Relaxation (SOR) algorithm [PTVF92], often used in the domain of Image Processing applications. The plus-shaped contour in Figure 2.11(b) shows the basic access pattern of the elements of array u in the inner loop. Figure 2.11(c) shows that this shape of accesses moves across by two columns to the right as we iterate through the inner loop. The second iteration of the outer loop causes the first pattern of the previous loop to move vertically down by one row. The remaining accesses of the inner loop are as before.

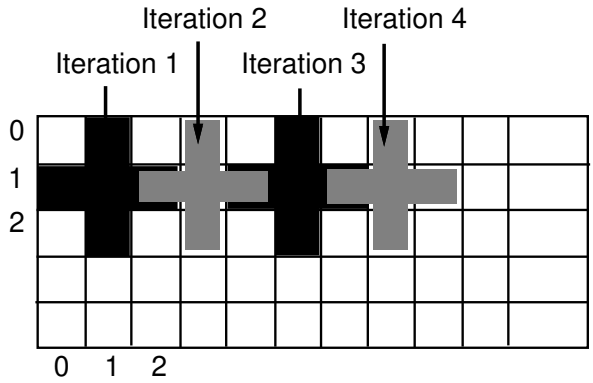
The one element that is common to successive patterns can be stored in a register instead of being accessed from memory again in the next iteration [DJKD94], hence the effective access pattern for this example is as shown in Figure 2.11(d). This pattern determines the dimensions of the tile to be used in the tile-based

```
for  (i = 1;  i < N–1; i++)
      for  ( j = 1; j < N–1; j = j + 2) {
            res = a[i][j] * u[i+1][j] + b[i][j] * u[i–1][j] +
                  c [i][j] * u[i][j+1] + d[i][j] * u[i][j–1] +
                  e[i][j] *   u[i][j]   – f[i][j];
            u[i][j]   =   u[i][j]   –    (K * res) / e[i][j];
      }
```

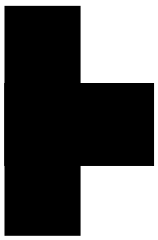
(a) Behavior



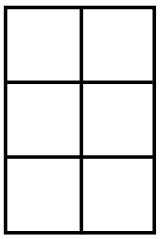
(b) Access Pattern Contour



(c) Inner Loop Execution Trace



(d) New Elements in each Iteration



(e) Tile

Figure 2.11: Example (SOR) – Access Pattern of Array 'u'

mapping scheme. The smallest rectangle that encloses this repeating shape forms the tile we use (Figure 2.11(e)).

We now examine the effect on the number of address bit transitions in the three mapping styles for the SOR example of Figure 2.11. This analysis helps determine the mapping style that minimizes the total transition count.

We use the observation that, in general, there is a low Hamming distance between elements in the same tile or adjacent tiles, and a possibly high Hamming distance between elements in distant tiles. We define a *maximal transition* as occurring when two logical addresses with a large difference are accessed in succession. This is because two numbers with a large difference are, on an average, likely to have a large Hamming Distance. A *minimal transition* occurs when this difference is small. In this case, *large* means comparable to the dimension of the array. For example, we treat all consecutive accesses to elements in the same tile (or adjacent tiles) as minimal transitions.

The above classification of maximal and minimal transitions is related to the the DRAM organization (Section 2.1.1). If the arrays involved are large, and are stored in different memory pages, a maximal transition refers to memory accesses from two different pages, and a minimal transition refers to consecutive memory accesses from the same page. If the arrays are small, and fit into the same DRAM page, the transitions refer to the switching activity on the column address (since the row address is now identical for both accesses).

In [PD95b] we show that if column-major mapping is used when the tile has n columns, there will be at least n maximal transitions in every inner loop iteration

(assuming at least one element from every column is accessed). Similarly, row-major mapping would entail n maximal transitions in an n-row tile. In comparison, tile-based mapping leads to a maximum of 2 maximal transitions, independent of the tile size, provided sufficient number of registers are available. [2] We thus choose to use tile-based mapping when the tile has both dimensions greater than 2. Otherwise, we use row- or column-major as appropriate. The above observation leads to the heuristic *SelectMapping* in Figure 2.12 for selecting a mapping scheme for a 2-level nested loop, involving a two-dimensional array.

Heuristic *SelectMapping*
Input: Memory References to Array in Loop
Output: Storage Scheme for Array

```
    Let the inner loop index be i and outer loop index be j
    Extract basic repeating shape from the access patterns
    Determine the enclosing rectangle R.
    Let i-dimension of R be L_i and j-dimension be L_j
    if j-increment in outer loop = L_j or min(L_i, L_j) > 2 then        (1)
        use tile-based mapping
    else
        if L_j > L_i                                                    (2)
            then use column-major mapping
        else
            use row-major mapping
        end if
    end if
end Heuristic
```

Figure 2.12: Selection of Memory Mapping Strategy

The reason for the condition "*j-increment in outer loop* = L_j" is that, if the increment of the outer loop is the same as the tile dimension, then every iteration of the inner loop will access elements from a single tile. This obviously makes

[2] If registers are insufficient, there will be more than 2 maximal transitions, but the sequence can always be ordered to perform as well, or better than the row-major case (in row-major, a maximal transition occurs every time the row changes).

tile-based mapping a preferable style, since all the transitions will be minimal in every iteration. If condition (1) fails, the extracted tile has dimensions N $\times$ 2, N $\times$ 1, 2 $\times$ N or 1 $\times$ N. Condition (2) ensures that for the first two cases, column-major mapping is selected, and row-major mapping is selected for the other two. If $L_j = L_i$, we arbitrarily select row-major mapping.

Implementation Overhead

In a typical implementation of the address generator, we have the following *index expressions* for generating the logical address for an arbitrary element $u[i][j]$:

Row-major – $A + (i \times MAX) + j$

Column-major – $A + (j \times MAX) + i$

Tile-based – $A + (i \times MAX) + (N \times j) - (i \bmod N) \times (MAX - 1)$, [3]

where the tile dimensions are $M \times N$ and A is the start address of array u of dimensions $MAX \times MAX$ (i and j range from 0 to $MAX - 1$).

For the address generation, we can use techniques similar to *strength reduction* and *induction variable elimination* [ASU93] to minimize the actual number of operations. A study of the implementation of the address generator circuitry [PD96] shows that the area and performance overhead for tile-based mapping is negligible – arising only from the usage of a 4×1 MUX, as opposed to 2×1 MUX for row and column major mapping, for the SOR example.

[3] This expression is independent of M – we have assumed that the elements within a tile are stored column-wise, while the tiles themselves are stored row-wise; in other (symmetrical) cases, it will be independent of N.

Experiments

We conducted experiments for testing the efficacy of various memory mapping schemes on several examples taken from the Image Processing applications domain [PD95a]. We use the *cumulative count of the bit transitions* on the memory address bus during the execution of the algorithm as a power consumption metric for each memory mapping technique. We varied the dimension of the arrays and studied the effects of the mapping schemes on transition counts.

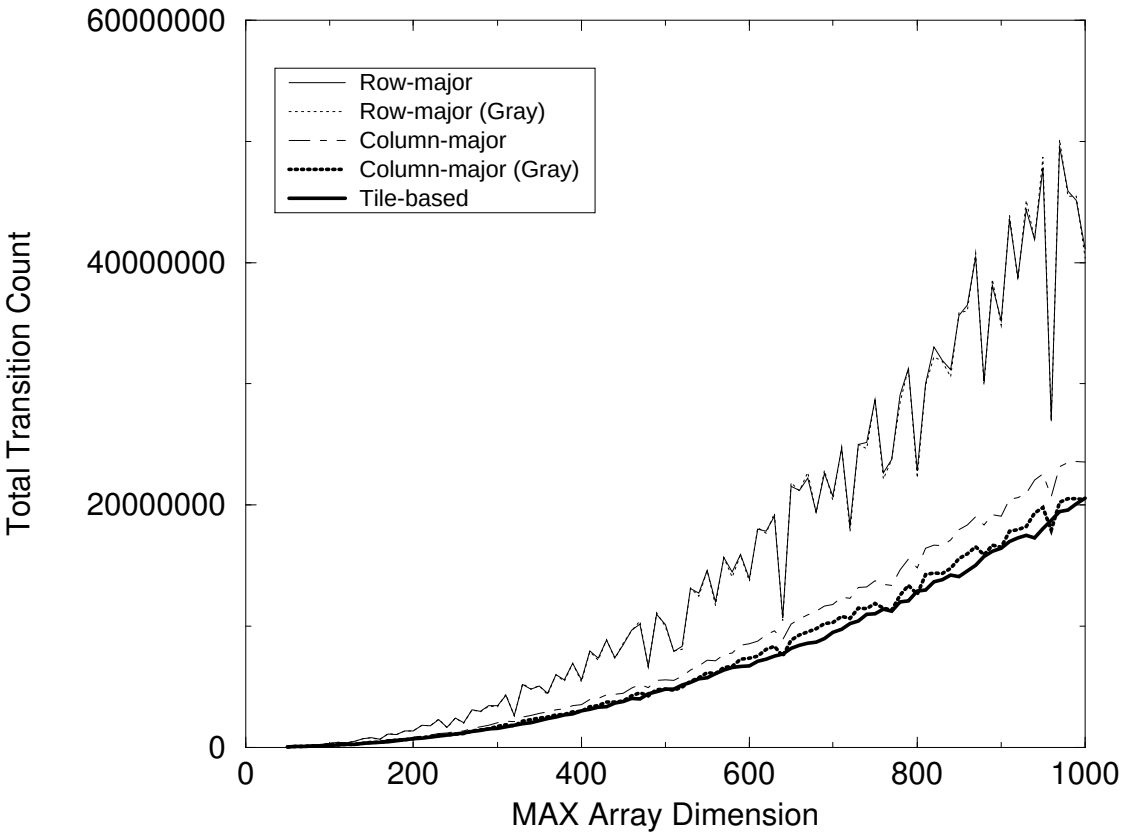


Figure 2.13: Transition Counts for the different mapping techniques for example SOR

Figure 2.13 shows the variation of the total transition count with respect to the dimension of the arrays for the SOR example. All arrays are of the form MAX $\times$ MAX, so MAX was varied along the x-axis from 50 to 1000 in steps of 10. The curves marked *Row-major* and *Row-major (Gray)* represent the total transition count for row-major mapping with and without the Gray code converter. Likewise for the *Column-major* and *Column-major (Gray)* curves. This illustrates the difference in transition counts due to the GCC alone. The curve marked *Tile-based* shows the transitions for the tile-based mapping.

For the SOR example, column-major mapping with the GCC does roughly as well as the tile-based mapping. In this case, the column-major mapping is still preferable, since it results in a simpler address generator compared to tile-based mapping, while performing almost equally well in terms of transition count.

	I/O Transition Count		% Red	Avg
Example	row-major	best	(1000)	% Red
Compress	11397910	6518370	42.8%	47.5%
GSR	28589669	28589669	0%	0%
Laplace	31739717	19653293	38.1%	41.3%
Lowpass	22413625	13218670	41.0%	41.2%
SOR	41153565	20438794	50.3%	51.9%

Table 2.1: Transition Count Reduction for Image Processing Examples

Table 2.1 summarizes our experimental results by comparing the total transition counts for the row-major to the best mapping out of the ones we have considered, for seven examples from the Image Processing domain. The second and third columns show the total I/O transition count for two mapping schemes – row-major and the best mapping predicted by Heuristic *SelectMapping*, with MAX = 1000. Column 4 shows the percentage reduction in I/O transition count for the

specific case where MAX = 1000. Column 5 shows the average percentage reduction in transition count obtained by performing the same experiment for values of MAX ranging from 50 to 1000 in steps of 10. We observe a significant decrease in the memory address bus transition counts, indicating guaranteed power reduction for these examples.

We used models of effective switched capacitances of the memory decoder, Gray code converter and Selector circuit (in address generator) to compute the reduction in transition count effected by tile-based mapping in comparison to row-major mapping, for the SOR example (with MAX=1000). Using average values of 20 fF and 10 pF for *on-chip* and *off-chip* capacitances [Bak88], we determined that the reduction in transition count (equivalent on-chip transitions) was 1772 for the ASIC *per memory access*, after considering the overhead introduced by the GCC and Selector. For the same execution, there was a 43% reduction in transition count in the memory decoder [PD95b].

2.2.2 Mapping Arrays into Multiple Memories

The technique presented in Section 2.2.1 can be extended to an architecture that consists of more than one single-ported memory modules of different sizes, as shown in Figure 2.14. We have a pair of data and address buses connecting the ASIC to each memory. This configuration is common in performance-critical systems where the required data access rate exceeds the maximum access rate possible when only a single memory can be accessed at any instant of time (i.e., multiple memories connected to a single data bus). In the past, works such as [KR94, JD97, ST97] have addressed the problem of utilizing multiple memory

modules during synthesis, but they have not addressed the issue of power dissipation in this context.

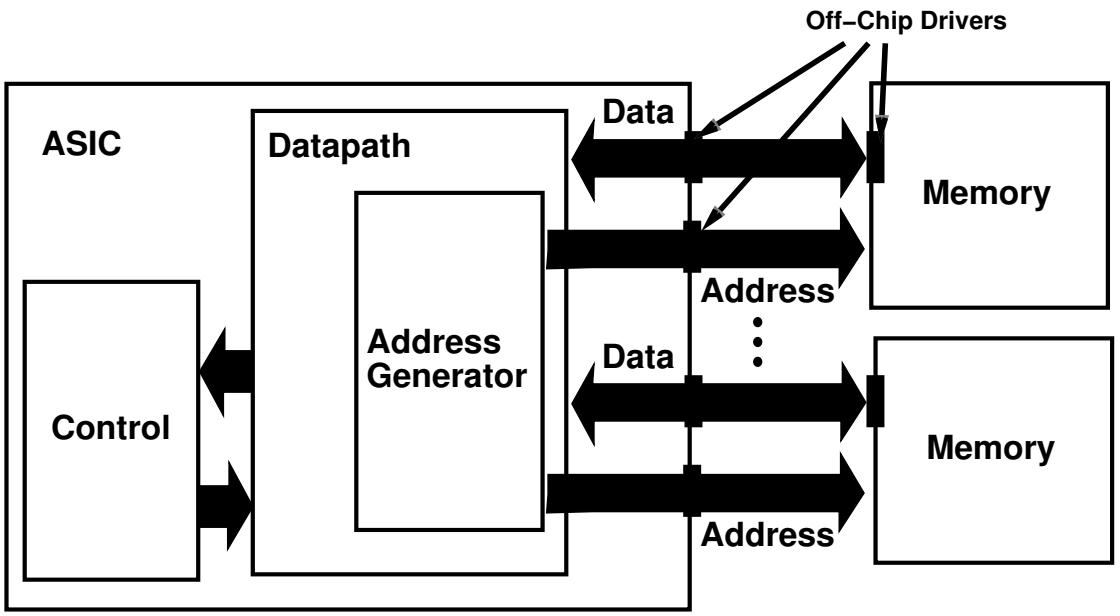


Figure 2.14: Architecture involving multiple memory modules

The approach we take is summarized in Figure 2.15. We first analyze the access patterns in the specification to determine the optimum partitioning of the arrays into multiple *logical array partitions*. For example, in Figure 2.15, array A_1 is split into logical array partitions K_1 and K_2. We then regroup the logical array partitions into *logical memories* based on criteria such as the possibility of interleaving (page 41). In Figure 2.15, logical array partitions K_1 and K_3 are merged into logical memory L_1. Finally, we map the logical memories into the available *physical memories*. The criterion here is minimization of the transition count overhead arising from mapping multiple logical memories to the same physical memory module. In Figure 2.15, logical memories L_1 and L_3 are assigned to physical memory P_1, whereas L_2 is assigned to P_2.

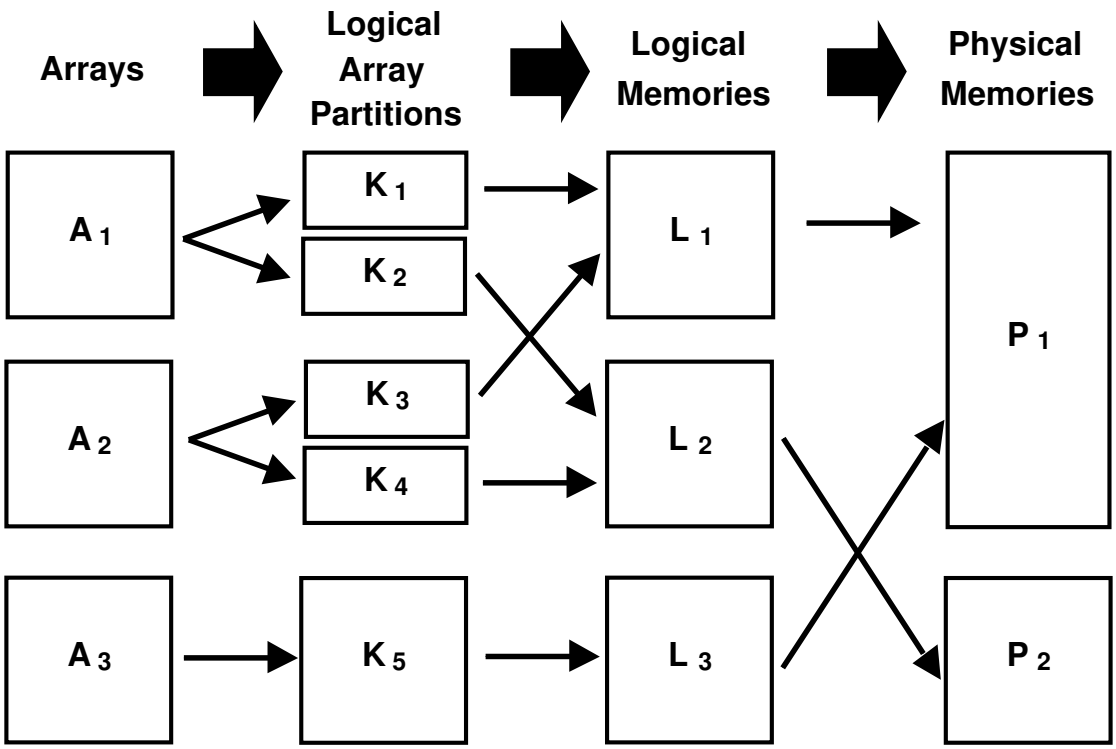


Figure 2.15: Mapping of Arrays to Multiple Physical Memories

Splitting into Logical Array Partitions

Regularity in behavioral access patterns, which is a common feature in most memory-intensive applications, especially those in the DSP/Image processing domain, allows us to extract the dimensions of the tiles (Section 2.2.1) for arrays accessed in loops. Figure 2.16 shows an array u, of dimensions 6×4, organized into four tiles. In the multiple memories scenario, for each array, we use the tile thus derived as the starting point and split it into as many logical array partitions as number of array elements in the tile.

Figure 2.16 illustrates the splitting of array u whose tile contour has already been established, into multiple array partitions. We have six array partitions corresponding to each element of the tile. There are as many elements in each array partition, as the number of tiles in array u. The rationale for this division is that if each array partition is mapped to a different logical memory, this ensures optimality in terms of bit-transitions on the address bus. For example, if $u[0][0]$ (in TILE1) is accessed in one loop iteration, we have $u[0][2]$, $u[3][0]$, and $u[3][2]$ being accessed in subsequent iterations (because the access pattern remains constant). The partitioning in Figure 2.16 results in all these four elements being mapped into the same logical partition, thereby ensuring that consecutive elements of the partition are accessed in each iteration. If the memory address is converted into Gray code, the address bus for each memory would have just one bit transitioning between consecutive iterations of the inner loop of the specification.

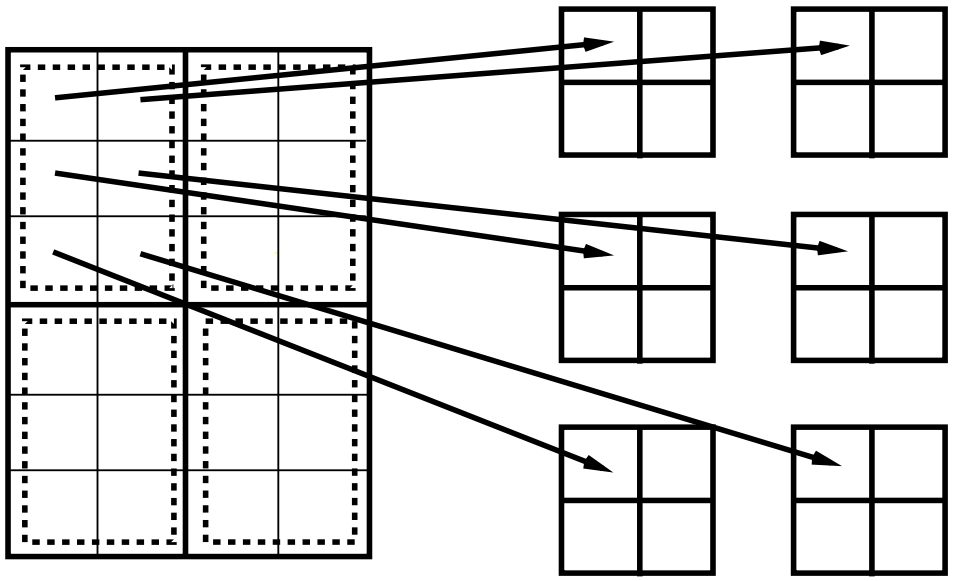


Figure 2.16: Partitioning of an array into multiple array partitions

Merging Logical Array Partitions

After partitioning the arrays in the specification to multiple logical array partitions, the next task is to group several array partitions into larger logical memories. This is because it might be prohibitively expensive to have a separate physical memory for each array partition. We consider the following properties while merging the array partitions:

Interleaving. It may be possible to interleave multiple arrays in the same physical memory without suffering any penalty in bit transitions.

Independence. If two arrays are not accessed in the same loop, they can be stored in the same logical memory.

Non-overlapping Lifetimes. If two arrays have non-overlapping lifetimes they can be stored in the same memory locations, since the same memory space can be reused for the two arrays, i.e., we need only one logical array partition for the two arrays.

Based on the above properties, we construct a *compatibility graph* G in which each vertex represents an array partition, and the presence of an edge indicates that the two partitions can be placed in the same physical memory with no transition count overhead (i.e., they are *compatible*). If two arrays are either independent, or can be interleaved, then we create an edge between the corresponding vertices.

After constructing the compatibility graph G, we apply a *clique partitioning* algorithm to divide the graphs into sub-graphs of array partitions. A clique is a fully connected subgraph of the original graph G. The significance of a clique is that all partitions in the clique can be placed in the same physical memory. An exact solution of the clique partitioning is known to be NP-complete[GJ79], so we employ an existing approximation algorithm [TS86] for this purpose. Each subgraph resulting from the clique partitioning corresponds to a logical memory.

Figure 2.17(a) shows an example behavior and Figure 2.17(c) shows the corresponding graph for the array partitions. The tile corresponding to arrays a and d have three elements each, since 3 elements of each array are accessed in the inner loop (Fig. 2.17(b)). This leads to three array partitions for the two arrays. The array p has edges to all other array partitions, since it is accessed in a different loop, thus the independence property allows it to be placed in the same logical memory with any of the other array partitions. One possible partitioning of this

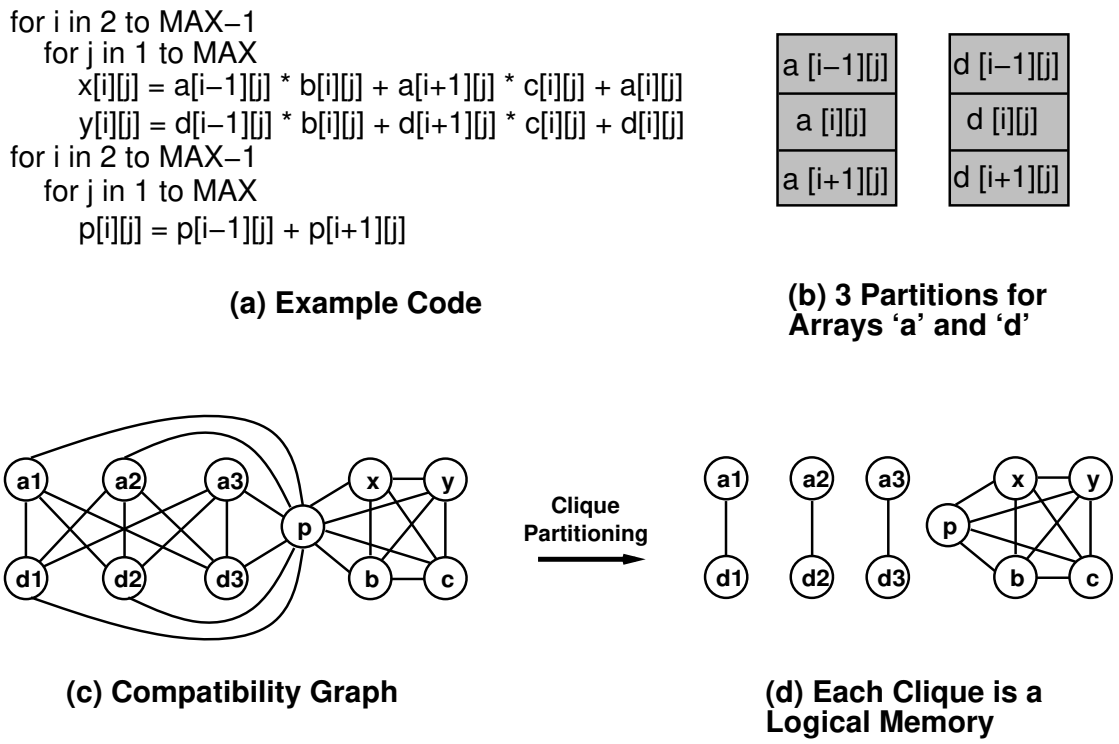


Figure 2.17: Deriving logical memories from behavior

graph into cliques is shown in Figure 2.17(d). This leads to four logical memories, each consisting of clusters of array partitions.

Mapping into Physical Memory

The next step is to map the logical memories to available physical memory modules. There exists a trade-off between memory utilization and address bus transition count: mapping each logical memory into a separate physical memory might be prohibitively expensive, forcing multiple logical memories to be mapped into the same physical memory. However, this mapping leads to inefficiency in terms of transition count, because some components of two different cliques in the compatibility graph would be incompatible (otherwise they would be in the same larger clique). We introduce an additional user-supplied constraint, a *memory packing factor f*, which allows the user to control this trade-off. The constraint *f*, which represents the minimum fraction of each physical memory that needs to be filled, can, of course, be set to 1, indicating that all the physical memories should be full. On the other hand, if the value of *f* is relaxed to less than 1, then a better packing in terms of transition count might be achieved. If *f* is set to 0, then the resulting mapping is optimum with respect to transition count, but could be area-expensive, as memory space would be wasted.

The algorithm we use for mapping the logical memories into physical memories is a variant of the *Best-Fit Decreasing* heuristic for the *bin-packing* problem [CL91]. The general strategy we adopt is to consider the logical memories one at a time, largest first and assign it to a physical memory module based on its own size and the sizes of the available memory modules. We start with the largest logical

memory (L), since this is a candidate that possibly accounts for large transition counts. If L is larger than all available physical memories, we implement it with (multiple copies of) the largest available physical memory module. If there is a remainder of lesser size, we add it to the list of logical memories and continue the mapping process. If there is at least one physical memory module of greater size than that of the logical memory under consideration, we map the logical memory into the largest physical memory module that satisfies the memory packing factor f. If the constraint f cannot be satisfied, the mapping is too expensive, and we need to compromise on transition counts by mapping more than one logical memory to the same physical memory. The details of the algorithm can be found in [PD95b].

Experiments

Experiment 1 - In our first experiment to determine the impact of multiple memories on transition count, we used a configuration of multiple memories in the examples, where there is one physical memory available for each logical memory. In other words, this is the best improvement possible over the single memory case, since the partitioning into logical memories represents the ideal mapping, according to our technique. We recall, from Section 2.2.2, that we try to avoid, as far as possible, mapping of multiple logical memories to the same physical memory.

Table 2.2 shows a comparison of transition counts for five examples, all involving 2-dimensional arrays of dimensions (MAX $\times$ MAX), of the best mapping in the case of a single physical memory, and multiple physical memories. Columns 2 and 3 show transition counts for the different examples, with the value of MAX as 1000. Column 4 shows the number of times by which transition counts decreased

Example	Transition Counts		Red. (1000)	Avg. Red.
	Single Memory	Multiple Memories		
Compress	6518370	999002	6.5 X	6.6 X
GSR	28589669	8897227	3.2 X	2.9 X
Laplace	19653293	3997735	4.9 X	4.4 X
Lowpass	13218670	4993734	2.6 X	2.7 X
SOR	20438794	5997502	3.4 X	3.2 X

Table 2.2: Transition Count Reduction due to Multiple Memories

as a result of using multiple memories. Column 5 shows the average reduction for all the different sizes of the arrays that were considered for each example (MAX was varied from 50 to 1000 in steps of 10). We observe that transition count could be reduced by a factor of between 2.7 and 6.6 times if multiple memories are considered.

Experiment 2 - Experiment 1 demonstrates the possible transition count reduction in the hypothetical case of physical memories of the same size as the logical memories being available. In practice, however, the designer may be constrained by a specific library of physical memory modules. In the second experiment, we used a specific library of memory modules of sizes 128 KBytes, 256 KBytes, 512 KBytes, 1 MByte and 2 MBytes, with the assumption that each array element in the arrays of the examples occupies one byte (data bus is 8 bits wide). Table 2.3 shows the transition counts obtained for the same five examples when the mapping is performed with the above library.

In Table 2.3, Column 2 shows the transition counts obtained for the case when MAX = 1000 using a simple algorithm that maps the arrays in the specification (in order of decreasing size) to the smallest memory that accommodates

Example	Transition Counts (MAX = 1000)		Red.
	Simple Mapping	Our Mapping	
Compress	6701381	999002	6.7 X
GSR	19877346	11010165	1.8 X
Laplace	12884093	3997735	3.2 X
Lowpass	14060806	4993734	2.8 X
SOR	14118103	9145590	1.5 X

Table 2.3: Transition Count Reduction for a Specific Library

them. Column 3 shows the corresponding transition counts using our approach. Comparing Column 3 in Table 2.2 and Table 2.3, we note that for three of the examples (*Compress, Laplace* and *Lowpass*), the transition count in Experiment 2 was the same as the best case (one physical memory for each logical memory) transition count in Experiment 1 (i.e., our algorithm performed the optimal mapping). The transition count reduction factors shown in Column 4 indicate a significant reduction resulting from our approach, ranging from a factor of 1.5 to 6.7.

2.2.3 Mapping into Multiport Memories

The array mapping techniques discussed above can also be targeted at minimizing power consumption in architectures employing multiport memories. In this case, the goal is to assign arrays in a behavioral specification to memory locations, and ports of multiport memories, so as to minimize the transition count on the memory address buses when these arrays are accessed. We have a pair of data and address buses connecting the ASIC to each memory port. The datapath is connected to an m-port memory, with ports $P_0..P_{m-1}$, consisting of m data buses $D_0..D_{m-1}$ and m address buses $A_0..A_{m-1}$. The address generator generates the

m addresses for the address buses $A_0..A_{m-1}$, so that upto m data words can be accessed in parallel.

The problem of mapping into a multiport memory requires the solution of two sub-problems: (i) determine the mapping style to use for each array, and (ii) assign specific array elements accessed within behavioral loops to memory ports, such that the total transition count on all the address buses is minimized. The details are described in [PD97].

2.3 Summary and Status

In this chapter, we presented optimization techniques for improving performance and power characteristics of synthesized designs for memory-intensive applications. The performance improvement is possible by using a refined model of memory access modes to pre-process the input specification. The power reduction is obtained through a judicious memory mapping strategy for behavioral arrays. Such techniques, which take the actual organization of modern memories, are necessary in order to incorporate memories in an automatic synthesis environment.

The performance optimizations for the DRAM access modes were measured by manually modeling the memory accesses in the application examples. The Finite State Machines derived from the models were synthesized using the *misII* synthesis package[BRSV87]. The power optimization experiments were performed on loop kernels for which tile sizes were derived manually. The transition count measurements were then taken by generating a trace of the memory addresses accessed in each example.

Chapter 3

Data Organization: The Processor Core/Cache Interface

3.1 Embedded Processor-based Systems

Modern embedded systems increasingly rely on processors due to the advantages they offer in terms of flexibility, reduction in design time and full-custom layout quality [MG95]. The software part of the partitioned specification of Figure 1.1 is executed on these embedded processors, which are often available in the form of *cores*, which can be instantiated as part of a larger system on a chip. This is feasible in current technology due to the relatively small area occupied by the processor cores, making the rest of the on-chip die area available for RAM, ROM, coprocessors, and other modules. Apart from the processors in the Digital Signal Processing domain (such as the TMS320 series from Texas Instruments), we also find microprocessors with relatively general purpose architectures available as embedded processors. An example of such a general purpose embedded processor is LSI Logic's CW4001 [ACG+95], which is based on the MIPS family of processors.

An important determinant of performance in embedded systems is the interaction between the processor and external memory. General purpose embedded processors such as the CW4001 are equipped with on-chip instruction and data caches, which interface with larger off-chip memories. Since off-chip memory accesses usually stall the CPU execution for significant durations (each access could take 10-20 processor cycles, depending on the relative processor and memory access speeds), it is important to design the interface between cache and main memory carefully. Figure 3.1 highlights the components in the embedded system architecture that are addressed in this chapter.

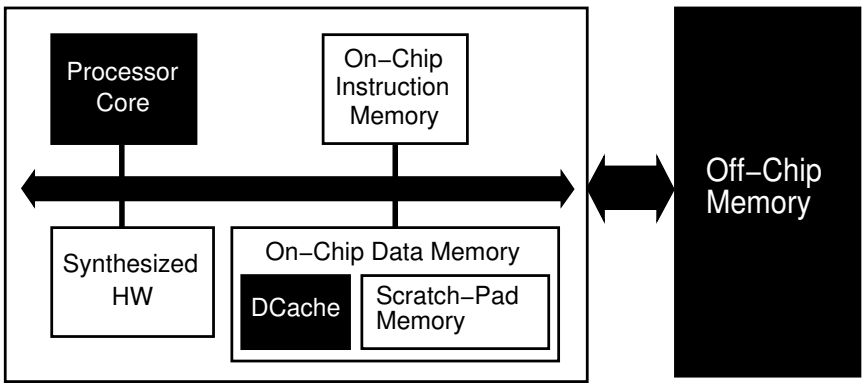


Figure 3.1: Shaded blocks represent the Processor Core/Data Cache/Off-chip Memory subsystem

Cache misses can be classified into three categories[PH94]: (1) *compulsory misses* – caused when a memory word is accessed for the first time, (2) *capacity misses* – caused when cache data that would be needed in the future is displaced due to the working data set being larger in size than the cache, and (3) *conflict misses* – caused when cache data is replaced by other data, in spite of the presence of usable cache space. Conflict misses can be avoided by using a fully associative cache, but due to access time constraints, all practical cache memories employ

a limited-associativity architecture. In fact, studies show that the simple direct-mapped cache architecture is indeed the best choice for most applications [Hil88]. For example, recent microprocessors including the HP 735, DEC AXP 3000, SGI Indigo2 and SUN Sparc5 are all equipped with direct-mapped data caches. In the computer architecture and compiler domains, many techniques for achieving cache miss reduction involve additional hardware assistance [Jou90, YJH+95, CKP91], which can often be expensive in terms of additional on-chip area.

Embedded system design is characterized by certain features that traditional compilers typically do not consider in their optimizations. For example, traditional compilers seldom take into account the specific cache parameters such as cache line size in their optimizations, because fast compilation speed requirements preclude the complex analysis procedures. However, in embedded systems, code generation can be tuned to the specific cache configuration to be used (or the specific configuration that is being currently explored) at the expense of longer compilation times. In this chapter, we outline techniques for exploiting this situation to organize data in memory in order to minimize data cache misses.

Code placement methods based on program traces for improvement of instruction cache performance have been reported [TY96a]. Rawat [Raw93] and Austin [Aus96] have addressed the problem of variable placement for improving cache performance. However, [Raw93] addresses only scalar variables, and [Aus96] treats scalar and array variables alike, thereby missing array placement opportunities based on their index expressions. In comparison, we outline a strategy for first placing scalar variables, and present an extension for analysis of array reference indices to determine the placement of arrays. We then show how a data alignment

strategy can be incorporated into *blocking*, a well known compiler optimization technique that combines strip mining and loop permutation to maximize temporal locality of reused data [Wol89].

3.2 The Data Organization Strategy

3.2.1 Problem Description

Consider a direct-mapped cache of size C ($C = 2^m$) words, with a cache line size L words, i.e., L consecutive words are fetched from memory on a cache read miss. In our formulation, we assume a *write-back* cache with a *fetch-on-miss* policy [Jou93], though the technique remains identical for other write policies, and is equally effective.

```
int   a[N], b[N], c[N]
. . .
for i in 0 to N–1
    c[i] = a[i] + b[i]
end for
```

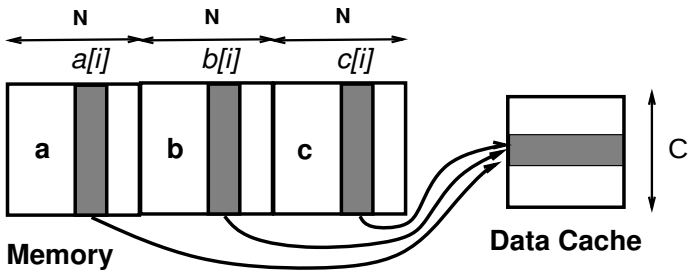


Figure 3.2: $a[i], b[i]$ and $c[i]$ map into the same cache line

We use a small example to illustrate the problem and our approach. Suppose the code fragment in Figure 3.2 is executed on a processor with the above cache configuration, where N is an exact power of 2, and $N > C$. Assuming one array

element per memory word, let array a begin at memory location 0, b at N, and c at $2N$. Let $f(x)$ denote the cache line to which the program variable x is mapped. In a direct-mapped cache, the cache line containing a word located at memory address M, is given by: $(M \bmod C)/L$. In the above example, array element $a[i]$ is located at memory address: i. Similarly, we have $b[i] : N + i$ and $c[i] : 2N + i$. We find that the corresponding cache lines to which each of them will be mapped are:

$f(a[i]) = (i \bmod C)/L$

$f(b[i]) = ((N + i) \bmod C)/L = (i \bmod C)/L$ (*since N is divisible by C*)

$f(c[i]) = ((2N + i) \bmod C)/L = (i \bmod C)/L$

In other words, $a[i], b[i]$, and $c[i]$ are mapped onto the same cache line (Figure 3.2).

Thus, the sequence of events that take place in one loop iteration is: a cache miss occurs while accessing $a[i]$; the line $f(a[i])$ is filled; accessing $b[i]$ now causes a miss, since, even if it was present in cache before, it was displaced by the last access to $a[i]$ (since $f(a[i]) = f(b[i])$); $f(a[i])$ is now filled by the line from b; the write to $c[i]$ also causes a miss, since $f(c[i]) = f(b[i])$; in a fetch-on-miss cache, this causes the same cache line to be displaced by elements of array c. The same cycle repeats in other iterations. In other words, every memory access results in a cache miss. Such memory access patterns are known to result in extremely inefficient cache utilization, especially because many applications deal with arrays whose dimensions are a perfect power of two [YJH+95]. In such situations, simply increasing the cache size does not present an efficient solution, because the cache misses are not caused due to lack of capacity. The conflict-misses can be avoided if the cache size C is made greater than $3N$, but this is often infeasible when N is large, and where feasible, there is an associated area and access time penalty

incurred when cache size is increased; similarly, access time penalties are also increased if the cache associativity is increased. Reorganization of the data in memory results in a more elegant solution, while keeping the cache size relatively small.

One way of preventing the thrashing caused by excessive cache conflicts for this simple example is to pad L dummy memory words between two consecutive arrays that are accessed in an identical pattern in the loops. For this example, if array a begins at 0, array b begins at location: $N + L$ (instead of N) and array c begins at: $2N + 2L$ (instead of $2N$). We have:

$f(a[i]) = (i \bmod C)/L$

$f(b[i]) = ((N + L + i) \bmod C)/L = (i \bmod C)/L + 1$

$f(c[i]) = ((2N + 2L + i) \bmod C)/L = (i \bmod C)/L + 2$

This ensures that $a[i], b[i]$, and $c[i]$ are always mapped into different cache lines, and their accesses do not interfere with each other in the data cache. We extend this basic idea to organize scalars (Section 3.2.2) and arrays (Section 3.2.3).

3.2.2 Memory Organization of Scalar Variables

We assume that the scheduling and register allocation of the code has already been performed, and the sequence of accesses to variables is fixed.

Constructing the Closeness Graph

We first generate an Access Sequence, which is a graph representing memory references (loads and stores are treated alike) in the code. Figure 3.3(a) shows an example Access Sequence. The label 3 on edge $e \rightarrow a$ represents a loop with bound = 3. We then construct a *Closeness Graph* of the variables, which represents the degree of desirability for keeping sets of variables in the same vicinity in memory. E.g., if L words accessed successively from memory are placed in consecutive locations, a single memory access could fetch them all into cache, thereby reducing upto $L-1$ extra memory accesses caused due to *compulsory* misses.

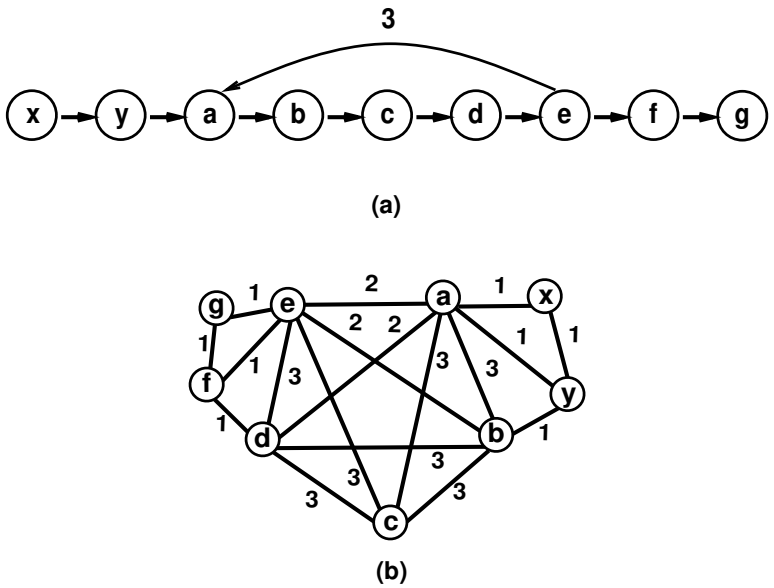


Figure 3.3: (a) Access Sequence (b) Closeness Graph for $M = 3$

We define the *distance* between two nodes u and v in the access sequence as: *distance*(u, v) = number of *distinct* variable nodes encountered on a path from u to v, or v to u (including u and v). The Closeness Graph *CG(V,E)* is constructed from the Access Sequence by first creating a node $v \in V$ for every variable in A, and

initializing all edge weights $e(u, v) = 0$. For all occurrences u' of variable u during traversal of the Access Sequence, we examine a window of width M preceding and following u'. For all node instances v' (of variable v) in this window (where $distance(u', v') \leq M$), we update the edge weight $e(u, v) = e(u, v) + k$, where k is the number of times control is expected to flow between u' and v' [1] .

Figure 3.3(b) shows the Closeness Graph derived from the Access Sequence in Figure 3.3(a), with $M = 3$.

Grouping of Variables into Clusters

The next step is to group the variables into clusters of M words, where M is the number of words in a data cache line. Intuitively, a higher edge weight between two variables u and v in the Closeness Graph represents a reduction in the number of memory accesses, if the two variables are stored in the same cache line – we can reduce cache misses by clustering variables with higher edge weights into the same cache line. We formulate the problem of maximizing the sharing of the cache lines by closely correlated variables as follows: *Partition the nodes of the Closeness Graph* CG *(i.e., the set of n variables) into clusters of size M, so that the total weight of edges in all the clusters is maximized.* Since an exact solution to the above problem has a computational complexity of $O(n^M)$, we employ the greedy heuristic procedure *PerformClustering* (Figure 3.4), with complexity $O(Mn^2)$.

[1] The required values of k in case of conditionals and loops could be obtained by using profiling information. However, in this work, we use the often-used simplifying assumption that branch probability is 0.5 for an *if*-statement, and that the loop bounds are always known at compile time.

Procedure *PerformClustering*
Input: $CG(V, E)$ – Closeness Graph; M – Cache Line Size
Output: Set F – Set of clusters of size M
  for each vertex u in V
    Find the sum of incident edge weights $S(u) = \sum_{v \in V} e(u, v)$
  end for
  Let X = vertex set V and $Y = \phi$
  -- Y keeps track of variables that are already assigned to clusters.
  while $(X \neq \phi)$ **do**
    Let u = vertex $v \in X$ with maximum $S(v)$ (i)
    Create new cluster $C = \{u\}$
    while (size of cluster $C \neq M$) **and** $(X \neq \phi)$ **do**
      Let x be the variable $\in X$ with maximum value for T, (ii)
      where $T = \sum_{u \in C, v \in X - C} e(u, v)$ *-- i.e., x is the variable with maximum sum of edge weights with nodes already in C*
      $C = C \cup \{x\}$
      $X = X - \{x\}$
    end while
    Set $e(u, v) = 0 \forall (u \in C)$ or $(v \in C)$
    -- i.e., delete all edges connecting to vertices in cluster C just formed
    Update $S(v) \forall v \in X$
  end while
end Procedure

Figure 3.4: Procedure for clustering the variables of the Closeness Graph

When procedure *PerformClustering* is applied on the graph in Figure 3.3(b), node b is selected first (line i). Next, line (ii) causes nodes c and d to be selected into the first cluster C_1. When we have equal T values for multiple nodes, we select one at random. Nodes b, c, d, and all connecting edges are now deleted. From the resulting graph, a, e, and g form the next cluster C_2. The final clustering is: $C_1 : [b, c, d]; C_2 : [a, e, g]; C_3 : [x, y, f]$.

The Cluster Interference Graph

After grouping the variables into clusters of size M, we build an *Interference Graph (IG)* of the clusters, which represents the desirability to store clusters in memory, so that they do *not* map into the same cache line. Each node in the Interference Graph represents one cluster of variables obtained from procedure *PerformClustering*. A high edge weight between two nodes indicates a large number of conflict misses in the data cache, if the respective clusters were to map into the same cache line. We first convert the Variable Access Sequence A into a Cluster Access Sequence by renaming each node u in the sequence by the cluster C, where $u \in C$. We then construct the Cluster Interference Graph by first creating a node in IG for each cluster in F (the set of clusters), and then assigning edge weight $e(u, v)$ between nodes u and v to be the number of times the access to clusters u and v alternate along the execution path. For instance, for the variable sequence $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e, M = 2$, and the clustering $x : [a, d]; y : [b, c]$ and $z : [e]$, we have the Cluster Access Sequence $x \rightarrow y \rightarrow y \rightarrow x \rightarrow z$. This results in an Interference Graph IG, with edges $e(x, y) = 2, e(x, z) = 1$, and $e(y, z) = 1$. The pair of nodes x and y alternate twice in the execution path, due to the edges $x \rightarrow y$

and $y \rightarrow x$, causing $e(x,y) = 2$. The other pairs change orders only once. The composition rules to be followed for conditionals and loops are identical to those used for building the Variable Access Sequence (Section 3.2.2).

Memory Location Assignment

The final assignment of variables to memory locations should consider the clustering and conflict-penalty information in the Interference Graph. To minimize the conflict misses in the data cache during code execution, we need to ensure that cluster pairs with large edge weights do not map to the same cache line when we assign memory locations.

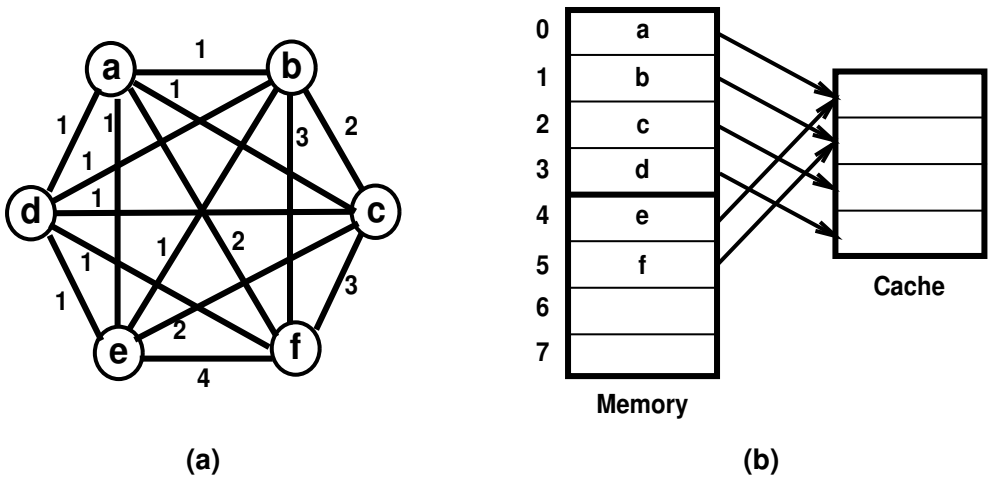


Figure 3.5: (a) Interference Graph (b) Memory Assignment

We define the *cost of a memory assignment* (C) as follows:

$$C = \sum_{x,y \in V(IG)} e(x,y) \times P(x,y) \tag{3.1}$$

where $e(x,y)$ is the edge weight, and $P(x,y) = 1$ or 0, depending on whether memory locations for x and y map into the same cache line or not. Figure 3.5(b)

shows the effect of a sample memory assignment for an IG with six clusters (Figure 3.5(a)), on a cache with four lines. We have $P(a,e) = P(b,f) = 1$ and $C = e(a,e) + e(b,f) = 1 + 3 = 4$. We solve the following problem: *Find an assignment of clusters in IG to memory locations, such that the assignment cost C is minimized.* This problem can be easily shown to be NP-hard, by using a reduction from the Graph Colouring problem. We present below procedure *AssignClusters*, which is a greedy $O(n^2)$ heuristic (where n is the number of clusters) to solve the Cluster Assignment problem for a cache of size k that is similar to the *PerformClustering* procedure.

We proceed to make the memory assignments *page* by *page*, where a memory page consists of k cache lines – the size of the data cache. Note that k consecutive clusters in memory will never conflict in cache. We define the cost of assigning cluster u to cache line i as $cost(u,i) = \sum_{v \in X} e(u,v)$, where X is the set of clusters that have already been assigned to cache line i. This cost is the sum of edge weights of u with all nodes that are already assigned to map into cache line i. The procedure is summarized in Figure 3.6.

For the example IG in Figure 3.5(a), the page size is $k = 4$ lines. When we apply procedure *AssignClusters* (Figure 3.6) on this example, we first sort the vertices in decreasing order of the sum of their incident edge weights: $f(13)$, $c(11)$, $e(9)$, $b(8)$, $a(6)$ and $d(5)$. Clusters f, c, e, and b are placed into the first page P_0. While attempting to assign a into the second page P_1, we find: $\text{cost}(a,0) = 2$ (since $e(a,f) = 2$), $\text{cost}(a,1) = 1$, $\text{cost}(a,2) = 1$, and $\text{cost}(a,3) = 1$. Thus, we choose a line within page P_1 that minimizes the cost, and assign a to line 1. Cluster d has:

```
Procedure AssignClusters
Input: CIG(V, E) – Cluster Interference Graph
Output: Assignment of Clusters to Memory Locations
   Sort the vertices of CIG in descending order of S(u)
   -- S(u) is the sum of edge weights incident on vertex u
   Let X be this sorted list of vertices
   while (X ≠ φ) do
      Create new page P in memory
      while (size of page P < k) and (X ≠ φ) do
         u = head of list X
         Assign u to line i of page P, where
         cost(u, i) is minimum over i = 0 ... k − 1
         Delete u from X
      end while
   end while
end Procedure
```

Figure 3.6: Procedure for assigning clusters to memory locations

$\text{cost}(d, i) = 1$ for all i, so we assign line 0 of P_1 to d. The final assignment is: P_0: $(0-f; 1-c; 2-e; 3-b)$ and P_1: $(0-d; 1-a)$.

For an n-way associative cache, we use the same definition of *cost*, except that the cost remains zero for the first n clusters assigned to the same cache line.

3.2.3 Memory Organization for Array Variables

We solve the memory organization problem for arrays by first constructing the *Interference Graph* among arrays in the code, and then assigning memory addresses to each array by minimizing the possibility of cache conflicts with other arrays in the code. The problem of clustering of variables to avoid compulsory misses is not relevant in the case of arrays, as most arrays are usually much larger than a cache line – often larger than the cache itself.

Constructing the Interference Graph

In the case of arrays, we note that if two arrays A and B are accessed repeatedly within a loop, then there is a possibility that the accesses to A and B might cause conflict misses in the data cache (Section 3.2.1). The Interference Graph (IG) of arrays represents the possibility of cache conflicts between the arrays in the code.

We first create a node for each array in the specification. Next, we determine the arrays that are repeatedly accessed in each loop, and add the loop bound B_l to the edge weights between each pair of arrays. This signifies that a total of B_l cache conflicts could possibly arise between each pair of arrays during execution of this loop. The resulting IG gives us a criterion to prioritize the order in which we assign memory addresses to arrays. The complexity of this procedure is $O(Ln^2)$, where L is the number of loops, and n is the number of arrays in the code.

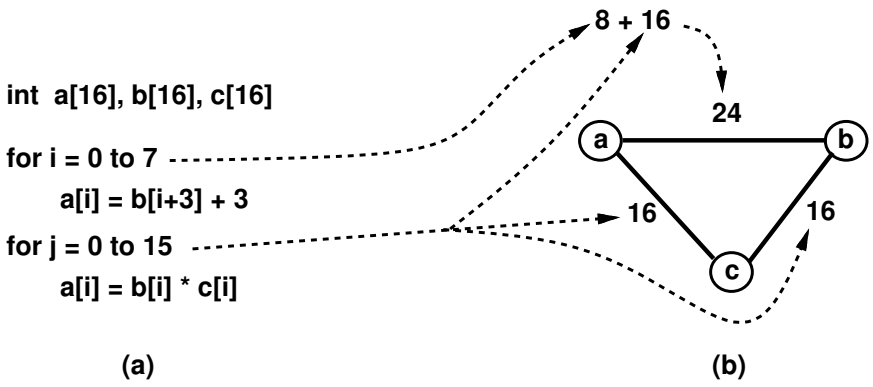


Figure 3.7: (a) Code showing arrays accessed in loops (b) Interference Graph

In Figure 3.7(b), we show the Interference Graph derived from the code shown in Figure 3.7(a). The first loop causes $e(a, b) = 8$. The second loop adds 16 to

$e(a,b), e(a,c)$, and $e(b,c)$. The IG helps identify the order in which the memory address assignment to arrays should be done.

Memory Assignment to Array Variables

In solving the problem of memory assignment of array variables, we assume that the loop bounds and array dimensions are known at compile time. We also assume that a unidimensional array of N elements is stored in N consecutive memory locations, and multidimensional arrays are stored in row-major format. (The issue of selection of a good storage technique for multi-dimensional arrays is addressed in Chapter 2 and in [CL95]). The memory assignment problem is NP-hard, because the degenerate case, when the array dimension = 1, itself happens to be NP-hard (Section 3.2.2).

From the Interference Graph, we use the $S(u)$ values for each node u (defined in Section 3.2.2) to determine the order of assignment of arrays. $S(u)$ signifies the relative importance of the nodes, because a higher $S(u)$ indicates that u could possibly be involved in many cache conflicts.

Central to the technique we use for memory assignment of arrays, is a computation of the *cost* of assigning an array (u) to begin at a specific memory address A. This cost is equal to the *expected number of cache conflicts with all arrays that have already been assigned*, if u were to begin at A. Note that if the first element of u is fixed at address A, all the other elements of u are automatically assigned their respective locations.

To determine whether two specific array accesses in the same loop will map into the same cache line (i.e, cause cache conflict miss), we perform a symbolic evaluation of the equality checking function. Two memory locations X and Y will map into the same cache line in a direct-mapped cache with k lines (M words per line), if the following condition holds:

$$\left(\left\lfloor\frac{X}{M}\right\rfloor - \left\lfloor\frac{Y}{M}\right\rfloor\right) \bmod k = 0$$

i.e., $(\lfloor X/M \rfloor - \lfloor Y/M \rfloor)$ is an integral multiple of k, which resolves to:

$$(nk - 1) < \frac{X - Y}{M} < (nk + 1) \tag{3.2}$$

where n is any integer. Clearly, the symbolically evaluated expression: $(X - Y)/M$, might not always reduce to a constant, because X and Y could be arbitrary functions of any variable in the code. If the expression does not resolve to a constant, then we conclude that the two arrays do not conflict.

We now formalize the strategy we presented in the example above into a procedure to perform the memory assignment of arrays. We first describe the cost function *AssignmentCost* (Figure 3.8) that returns the expected number of conflicts when an array is tentatively assigned a specific location. In practice, the inner **for**-loop is terminated if the assignment cost is zero for any assignment.

In line (i) of *AssignmentCost*, the number of times control alternates between v_l and u is determined by Condition (3.2). w represents the number of cache misses in the loop due to conflict between v_l and u – this is determined from the Access Sequence. Procedure *AssignArrayAddresses* (Figure 3.9) outlines the strategy for determining the addresses for each array.

Function *AssignmentCost*
Input: u – array under test; A – proposed start address; Access Sequence;
Array assignments already completed; IG – Interference Graph
Returns: Expected number of cache conflicts for this assignment
Initialize $cost = 0$
for all $v_l | e(v_l, u) \neq 0$, v_l already assigned
-- i.e., all assigned arrays that have an edge with u in IG
for each loop (bound L) in which accesses to v_l and u occur
w = no. of times control alternates between elements of v_l and u
that map into the same cache line, using Condition (1) (i)
$cost = cost + w \times L$ *-- $w = 0$ if there is no conflict*
end for
end for
return *cost*
end Function

Figure 3.8: Cost function for expected number of conflicts

Procedure *AssignArrayAddresses*
Input: IG – Interference Graph, k – no. of cache lines
Output: Assignment of addresses to all arrays (nodes in IG)
Address $A = 0$
Sort nodes in IG in decreasing order of $S(u)$ (sum of incident edge weights)
Let the list of nodes be: $v_0 \ldots v_{n-1}$
for $i = 0 \ldots n-1$
Initialize cost $c = \infty$
$min = 0$ *-- keeps track of cache line with minimum mapping cost*
for $j = 0 \ldots k-1$
if $AssignmentCost(v_i, A+j) < c$ **then**
c = $AssignmentCost(v_i, A+j)$
$min = j$
end if
end for
Assign address $(A + min)$ to first element of v_i
$A = A + min + size(v_i)$ *-- updating A for next iteration*
end for
end Procedure

Figure 3.9: Procedure for assigning addresses to arrays

Function *AssignmentCost* requires $O(P)$ time, where P is the total number of distinct array accesses in the code, since, in the worst case, $O(P)$ comparisons might be required. The worst case complexity of procedure *AssignArrayAddresses* could be $O(nkP)$, where n is the number of nodes (arrays), k is the number of cache lines. However, in real behaviors, we have observed that the assignment cost in loop $j = 0 \ldots k-1$ tends to converge to zero very soon (typically less than 2 or 3 iterations), because the number of different array elements accessed in inner loops of code is usually small.

This completes the memory address assignment of scalar and array variables in the behavior.

3.2.4 Experiments

In this section, we describe the experiments we performed on several benchmark examples to validate our memory data organization strategy. For our experiments, we used the CW4001 simulator kit from LSI Logic running on a SUNSparc5 workstation, which simulates code and reports performance statistics of code generated for execution on LSI Logic's CW4001 embedded processor. The specific performance statistics of interest for our experiments were the *data cache hit ratio* and the *total number of processor cycles* needed for execution of the code on the embedded processor.

For our experiments, the *read* and *write latencies* (the number of processor cycles required to access an external memory word) were assumed to be 5 processor cycles. These values are generally among the lowest possible (i.e., for the fastest

memories). Since the performance difference widens even more for higher penalties (higher access times), the improvements we have shown are the minimum possible (i.e., most conservative). A Write Buffer [PH94] of size 4 words was used in the data cache, which was configured to be a *write-through* cache employing *no-fetch-on-write, no-write-allocate* write miss policy, with a cache line of size 4 words. The details about the characteristics of the benchmark examples are given in [PDN97e].

Our first experiment was on the CW4001 processor core configured with a 1 KByte instruction cache and 256 Byte direct-mapped data cache with a 16-byte cache line. Figure 3.10 presents a comparison of the data cache hit ratios for the *Unoptimized* and *Optimized* memory data organizations, i.e., for the straightforward organization, and our technique respectively. In all the examples (except *Laplace*), we notice that the difference in the hit ratios is substantial. On an average, we observe a difference of 48 % in the data cache hit ratios.

Figure 3.11 shows the speed-up in execution time on the CW4001 embedded processor resulting from our data layout technique, which is defined as:

$$\text{Speedup} = \frac{\#\text{ Processor cycles for } \textit{Unoptimized} \text{ case}}{\#\text{ Processor cycles for } \textit{Optimized} \text{ case}}$$

The *Laplace* example, for which no speed-up is observed, involves only two arrays, *image1* and *image2*, with *image1* being read and *image2* written. Since the cache implements a *write-through, no-fetch-on-write* policy, the two arrays do not conflict in the cache. Hence, the cache performance is good in the unoptimized case and our technique does not lead to any improvement. In all the other examples, cache conflicts are reduced by our data layout technique, leading to reductions in execution time.

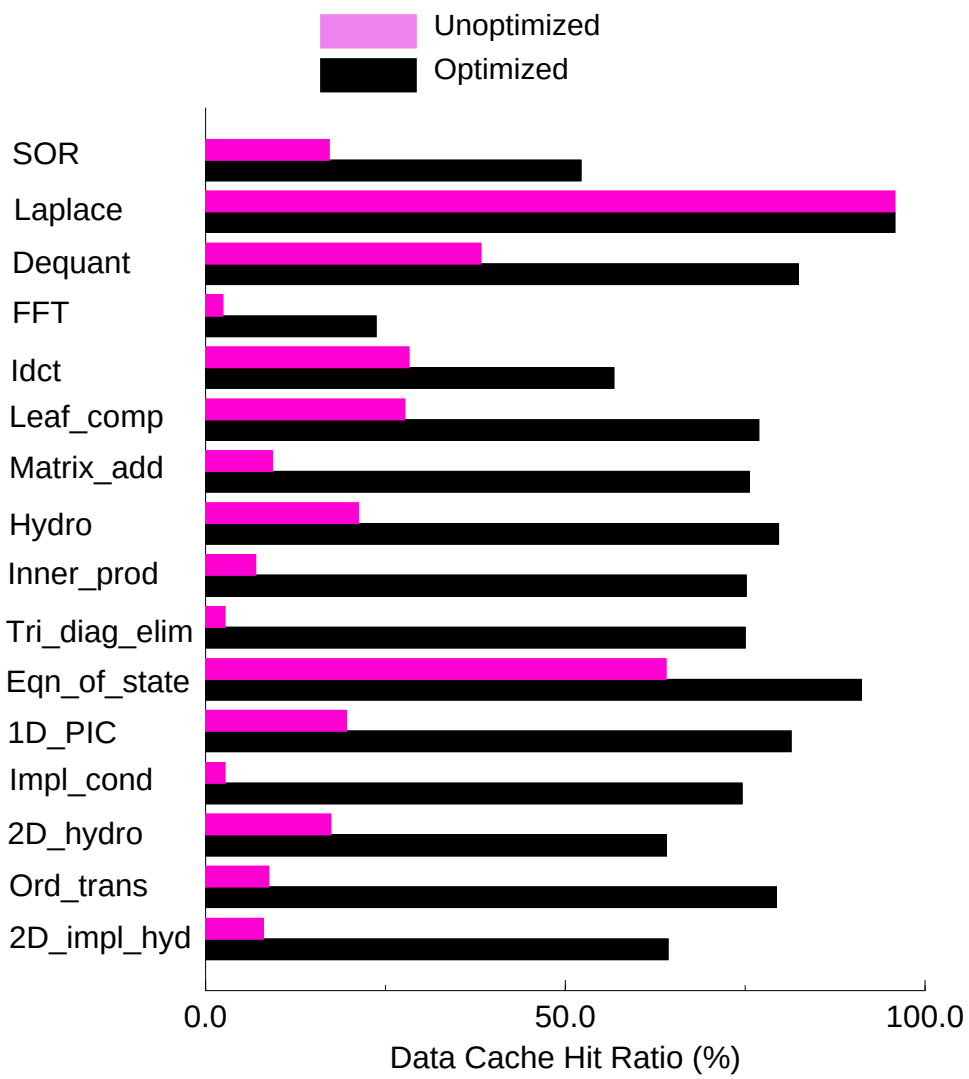


Figure 3.10: Comparison of cache hit ratios in direct-mapped cache

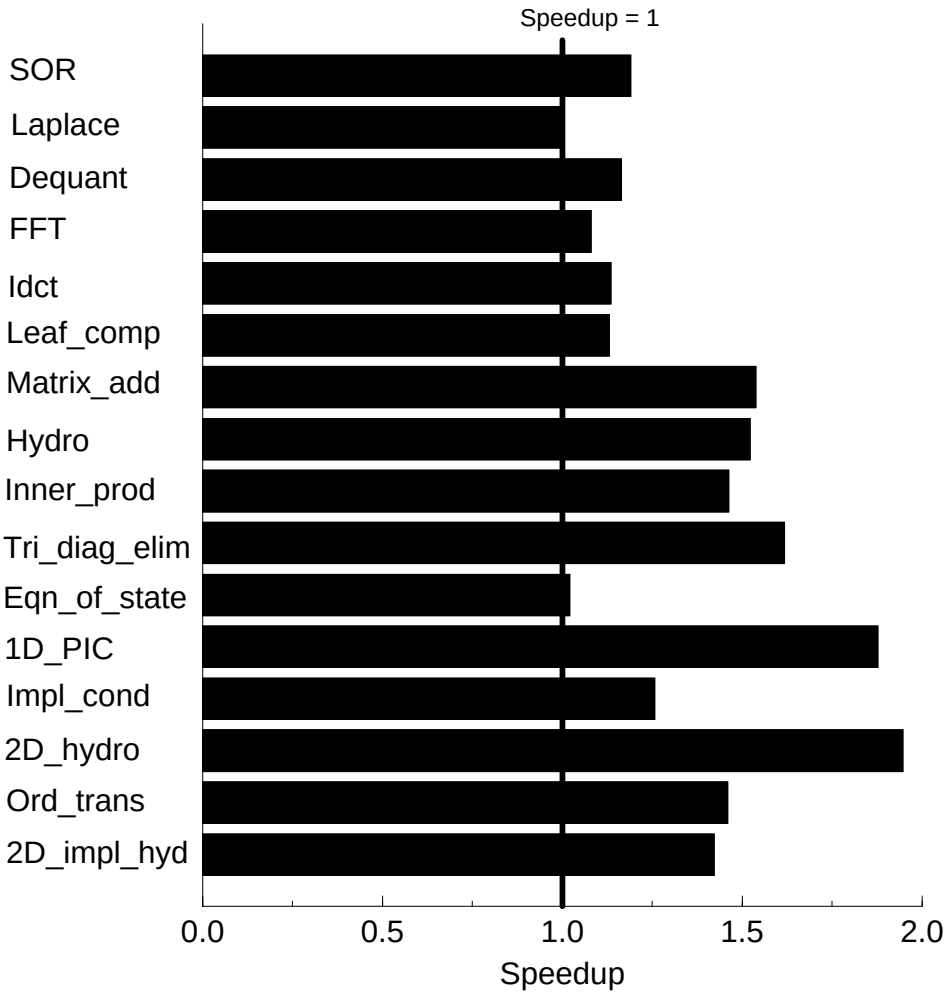


Figure 3.11: Comparison of speed-up in direct-mapped cache

There is a significant reduction in the total cycle time for most of the applications. The average reduction in the total cycle time over all the examples was 34%. The reason why the percentage difference tends to be lower than the difference data cache hit ratios is that, the total execution time also factors in all the other determinants of program execution speed, most notably, the instruction cache. The instruction cache has an identical performance in both the optimized and unoptimized cases, since in this work, we have concentrated on the performance of the data cache alone. Further, the speed-up is a function of the memory latency; it increases with memory latency ([PDN97e]). There is a negligible compilation time and memory space overhead due to our techniques. For the set of benchmark examples, we observed an average compilation time overhead of only 4.0%, and an average wasted memory space overhead of 2.2%.

Most of the performance improvement reported in Figures 3.10 and 3.11. arises from the placement of arrays. This is because the number of scalar variables present in the examples was relatively low. The miss ratio reductions as a result of our data placement strategy are significantly higher than those observed in [Aus96]. This is due to several reasons. Firstly, we have a scheme to analyze array indices and determine the relative placement of arrays in memory. Secondly, most of the benchmark examples we have used are code kernels exhibiting cache conflicts in the absence of data alignment. Thirdly, we have used a relatively smaller cache, thereby increasing the possibility of conflicts. The conflicts in the *Unoptimized* case will decrease when larger data caches are used. However, our experiments show that good cache performance can be obtained even for smaller caches by data alignment. Finally, the array sizes in the code kernels of our benchmarks were multiples of the cache size. It is possible that for other randomly chosen

array sizes, the *Unoptimized* case would incur fewer conflicts. However, the special case of array sizes being close to a multiple of the cache size occurs frequently – e.g., image processing applications frequently involve images of size 512×512 or 1024×1024, which are very likely to result in multiples of cache sizes, since cache sizes are typically a power of 2. Our technique helps prevent pathological cache conflicts irrespective of array sizes.

3.2.5 Mapping into Set-Associative Caches

We now consider the case of mapping of data into a *set-associative* cache. In an a-way associative cache with total capacity C elements and cache line size L, the number of lines $n = C/L$, and number of sets $s = n/a$. A word residing in block number M in memory would reside in set number: $(M \bmod s)$ in the set-associative cache, as opposed to line number: $(M \bmod n)$ in a direct-mapped cache of the same capacity.

Clearly, the clustering technique (Section 3.2.2) to minimize compulsory misses remains identical for the associative cache, since the clustering depends only on the cache line size. However, after the Cluster Interference Graph is obtained, the memory assignment phase differs from that used in the direct-mapped cache.

We show that the optimal memory assignment for direct-mapped caches is not necessarily optimal for set-associative caches, with an example in Figure 3.12. Figure 3.12(a) shows the distribution of six elements: a, b, c, d, e, and f located

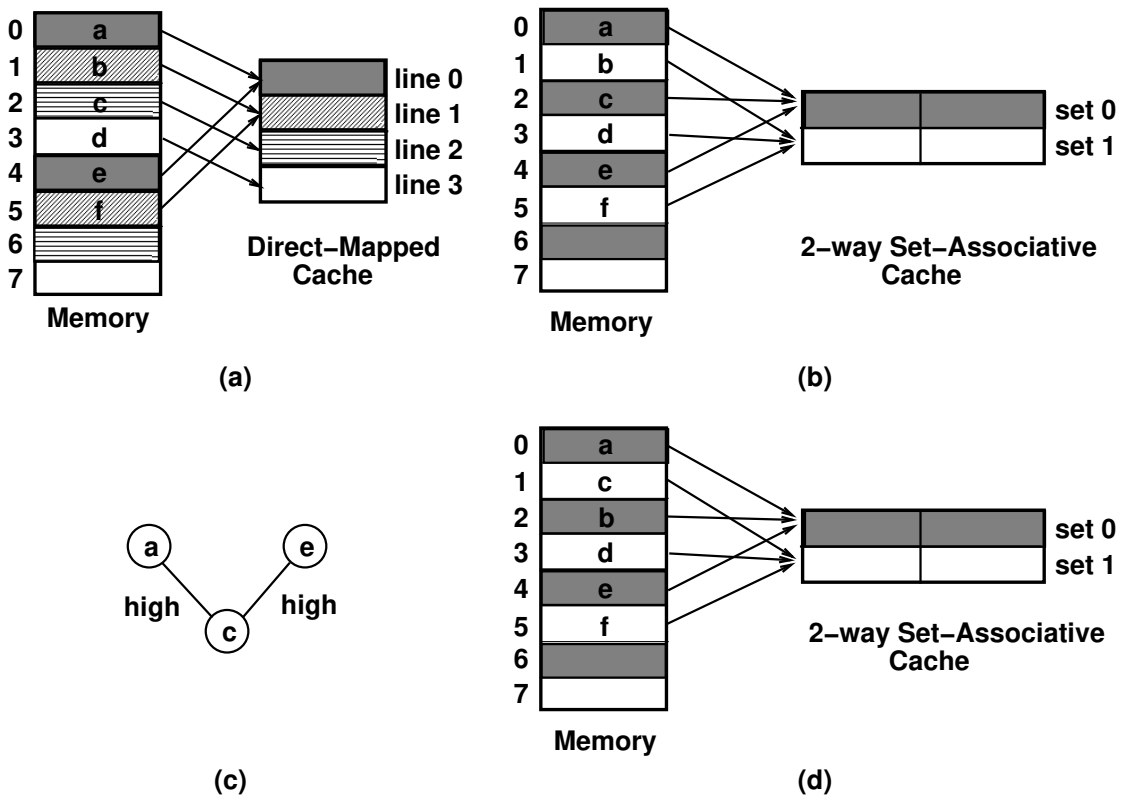


Figure 3.12: (a) Mapping into 4-element direct-mapped cache (b) Mapping into 4-element 2-way set-associative cache (c) Unfavourable Interference Graph for 2-way set-associative cache (d) Optimal data placement for 2-way set-associative cache

consecutively in memory into locations of a direct-mapped cache with size 4 elements, and cache line size = 1 element. We assume that this memory assignment is optimal for the given direct-mapped cache, as determined from Section 3.2.2. Figure 3.12(b) shows the distribution of the same memory elements in a 2-way associative cache of size 4 elements. We notice that elements a, c, and e map into the same set in the cache. This mapping could lead to undesirable conflicts, for example, when the interference graph is of the form shown in Figure 3.12(c). The weights of edges $e(a, c)$ and $e(c, e)$ are high, indicating expensive conflicts if the pairs compete for a cache location.[2] The conflicts are avoided in the direct-mapped cache (Figure 3.12(a)), because c is mapped into a different line than a and e.

The solution we propose, to obtain an optimal memory assignment for a set-associative cache with s-sets, is to consider the page size = s while doing the memory assignment in Section 3.2.2 (instead of page size = n in a direct-mapped cache). This decision is based on the observation that any s consecutive clusters in memory will always be mapped into distinct sets in the cache, and hence will not conflict. This argument is analogous to that used for direct-mapped caches – any n consecutive clusters in memory will always map into distinct cache lines. In the example shown in Figure 3.12, we have $n = 2$ and $s = 2$. To obtain the optimal memory assignment for the set-associative cache, we compute the memory assignment with page size = 2. Figure 3.12(d) shows an example solution. We observe that in this assignment, c is mapped into a different set from a and e, thereby avoiding the expensive conflicts indicated earlier.

[2] Figure 3.12(c) shows only the relevant portion of the interference graph. Other edges (with low edge weights) are not shown.

In summary, the data layout strategy for an a-way set-associative cache is identical to that for a direct-mapped cache of the same capacity (n), except that in the memory assignment phase, we use a page size of $s = n/a$, instead of n for the direct-mapped cache.

Our experiments, performed on the CW4001 processor configured with a 1 KByte instruction cache and 256 Byte, 2-way set-associative data cache with a 16-byte cache line, showed that our *Optimized* technique (Section 3.2.4) resulted in an average performance improvement of 21% over *Unoptimized*, incurring an average wasted memory space overhead of only 2.2%. [PDN97e].

3.3 Data Alignment Technique for Loop Blocking

3.3.1 Loop Blocking

Reduction of cache capacity misses, which occur when reusable data is evicted from the cache due to cache size limitations, is achieved by *loop blocking* (or *loop tiling*)[Wol89] – a well known compiler optimization that helps improve cache performance by dividing the loop iteration space into smaller *blocks* (or *tiles*). This also results in a logical division of arrays into tiles. The *working set* for a tile is the set of all elements accessed during the computation involving the tile. Reuse of array elements within each tile is maximized by ensuring that the working set for the tile fits into the data cache. Conflict misses, which occur when more than one

data element competes for the same cache location, are a consequence of limited-associativity caches. Cache conflicts in the context of tiled loops are categorized into two classes: (1) *self-interference*: cache conflicts among array elements in the same tile; and (2) *cross-interference*: cache conflicts among elements of different tiles (of the same array) or different arrays. Conflict misses can seriously degrade program performance; their reduction is generally addressed during the selection of tile sizes.

An important problem in the context of blocking is to determine the shape and size of the tile. Several techniques for determining the tile size have been proposed in the past[LRW91, Ess93, CM95]. The performance of the blocked code resulting from all the above tiling techniques show a large amount of sensitivity to the array sizes. In this section, we present DAT – a data alignment technique based on *padding*, which involves the insertion of dummy elements in a data structure for improving cache performance[PNDN97a]. Our technique ensures a more stable, and in most cases, better cache performance than existing approaches, in addition to maximizing cache utilization, eliminating self-interference, and minimizing cross-interference. Further, while all previous efforts are targeted at programs characterized by the reuse of a single array, we also address the issue of minimizing conflict misses when several tiled arrays are involved. As in previous work, we focus our attention on two-dimensional arrays.

Procedure *SelectTile*
Input:
TypeX, *TypeY*, *ShapeX*, *ShapeY*,
Cache Size (C), Associativity (A), Cache Line Size (L)
Output:
SizeX, *SizeY* /* No. of rows and columns of selected tile */

Case 1 ($ShapeX = ShapeY = 0$):
/* Common case: no application-specific info. available */
$SizeX = SizeY = \mathbf{max}\{k|(k = L \cdot t, t \in I) \wedge (ws(k,k) \leq C)\}$
Case 2 ($TypeX$ = CONSTANT **and** $TypeY$ = VARIABLE):
/* Fixed number of rows */
$SizeX = ShapeX$
$SizeY = \mathbf{max}\{k|(k = L \cdot t, t \in I) \wedge (ws(SizeX,k) \leq C)\}$
Case 3 ($TypeY$ = CONSTANT and $TypeX$ = VARIABLE):
/* Fixed number of columns */
$SizeY = ShapeY$
$SizeX = \mathbf{max}\{k|(k \in I) \wedge (ws(k,SizeY) \leq C)\}$
Case 4 ($TypeX$ = CONSTANT and $TypeY$ = CONSTANT):
/* Tile size already determined */
$SizeX = ShapeX$;$SizeY = ShapeY$
Case 5 ($TypeX$ = VARIABLE and $TypeY$ = VARIABLE):
/* $ShapeX/ShapeY$ gives ratio of #rows and #cols in tile */
$(SizeX, SizeY) = (ShapeX \cdot t, ShapeY \cdot t)$ such that
$t = \mathbf{max}\{k|(k \in I) \wedge (ws(ShapeX \cdot k, ShapeY \cdot k) \leq C)\}$
end Procedure

Figure 3.13: Procedure for selecting tile size

3.3.2 Data Alignment Strategy

Tile Size Computation

The tile size selection procedure can be summarized as follows: select the largest tile for which the working set fits into the cache. For computing the working set size, we use the formulation used in [CM95].

Figure 3.13 describes procedure *SelectTile* for computing the tile sizes. In addition to the common case where we select the square tile, we also allow the user to specify the *shape* of the tile in cases where additional information about the application is available. To achieve this, the procedure takes the parameters: *TypeX*, *TypeY*, *ShapeX*, and *ShapeY*. In the common case (case 1: $ShapeX = ShapeY = 0$), we choose the largest square tile for which the working set fits into the cache. $ws(x, y)$ gives the size of the working set for a tile with x rows and y columns. We maintain the number of columns to be a multiple of the cache line size to ensure that unnecessary data is not brought into the cache. If *ShapeX* and *ShapeY* have a non-zero value, the user wishes to guide the tile-selection process. *TypeX* and *TypeY* can take the value CONSTANT or VARIABLE. If *TypeX* (*TypeY*) has value CONSTANT and *TypeY* (*TypeX*) has value VARIABLE, as in cases 2 and 3, the user has fixed the number of rows (columns) in the tile as *ShapeX* (*ShapeY*). Procedure *SelectTile* only determines the number of columns (rows). If both *TypeX* and *TypeY* are CONSTANT (case 4), the user has already optimized the tile sizes, and the procedure performs no action. If both *TypeX* and *TypeY* are VARIABLE (case 5), the user intends the ratio of rows and columns of the tile to be *ShapeX* : *ShapeY*. The procedure selects the largest tile with the rows and columns in the given ratio for which the working set $\leq C$.

Set-associative caches can be effectively utilized in our tiling procedure to reduce cross interference in a tiled loop. We first determine the *largest allowed working set size* (M) for the tile by considering the possibility of cross-interference. If there is no cross-interference (e.g., there is only one array involved), we set $M = C$ (the cache size). However, there are circumstances where cross-interferences cannot be completely eliminated. For example, in the matrix multiplication benchmark

implementing $P_1 \times P_2 = P_3$, where P_1, P_2, and P_3 are matrices, the tiling algorithm is designed to exploit reuse in matrix P_2 – the other matrices contribute unavoidable cross-interferences with the tile from P_2 in the cache. In this case, if the cache associativity $A > 1$, we set $M = \frac{A-1}{A} \times S$. In other words, in an A-way cache, in any cache entry (with A lines), the *primary* tile (tile of the blocked array) occupies only $(A-1)$ lines, leaving the remaining one line to be occupied by elements outside the reused tile. This ensures that elements of the primary tile are seldom evicted from the cache. After determining M, we use procedure *SelectTile* (Figure 3.13), replacing the condition $ws(x, y) \leq C$ by $ws(x, y) \leq M$.

Pad Size Computation

The computation of tile size above ignores the possibility of self-interferences within the tile. We now formulate the procedure for an appropriate padding of the rows of the array with dummy elements in order to avoid this self-interference. Consider the mapping of a 30 rows $\times$ 30 columns tile in a 256 $\times$ 256 array into a 1024-element cache. Figure 3.14(a) shows that rows 1 and 5 of the tile cause self-interference because they map into the same cache locations. To overcome this conflict, we can pad each row of the array with 8 dummy elements so that the 5th tile row now occupies the space adjacent to row 1 in the cache (Figure 3.14(b)). The regularity of the cache mapping ensures that no self-interference occurs among the elements of all the tile rows.

Algorithm *ComputePad* (Figure 3.15) outlines the pad size computation procedure. The initial assignment to *InitPad* ensures that the rows of the padded array are aligned to the cache line size. For different multiples of *LineSize*, from a

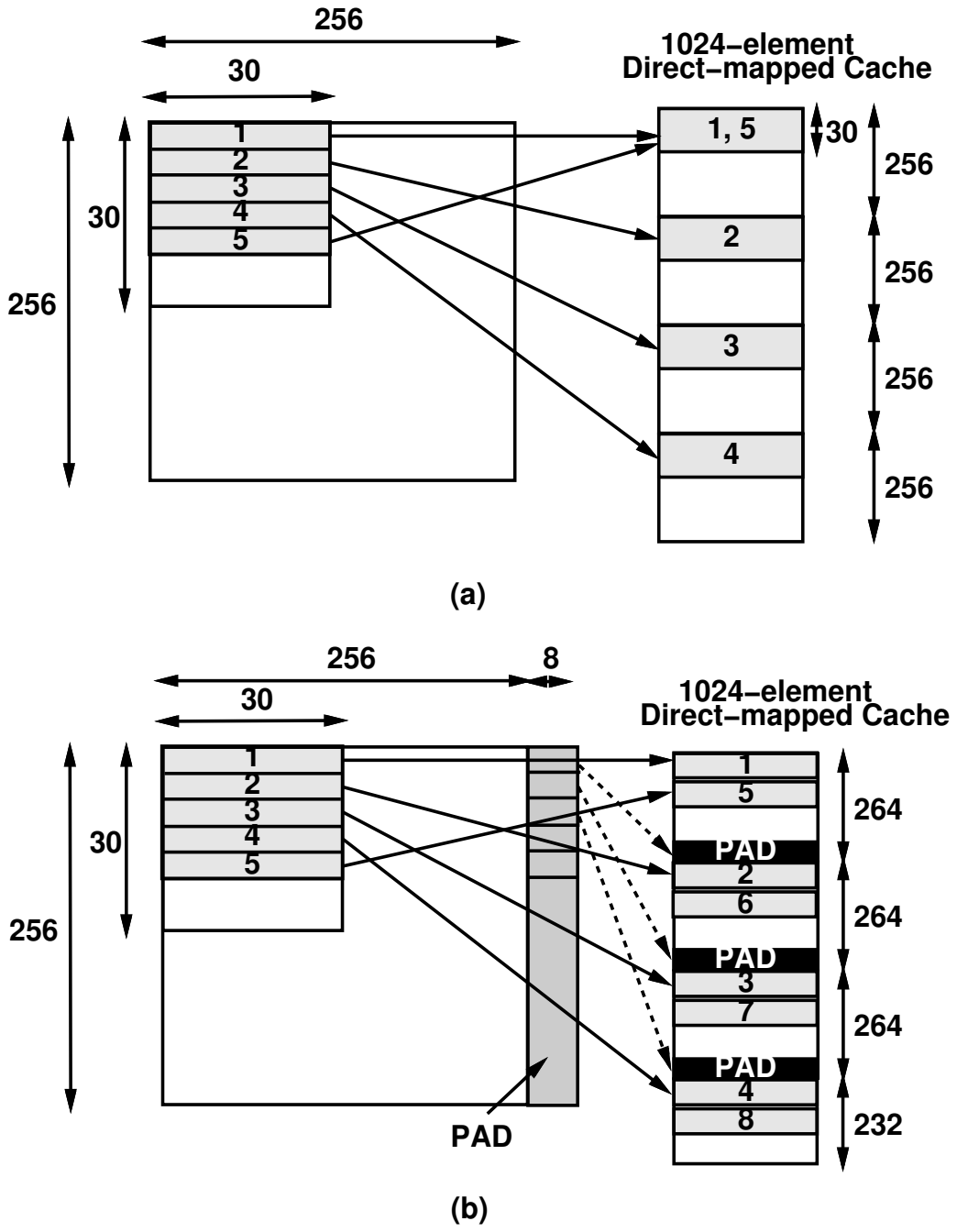


Figure 3.14: (a) Self-interference in a 30×30 tile: Rows 1 and 5 of tile interfere in cache (b) Avoiding self-interference by padding

Algorithm *ComputePad*
Input:
Array Dimensions: $N' \times N$; /* Row Size = N */
Tile Dimensions: $Rows \times Cols$
Cache Size (in words): $CacheSize$
Cache Line Size (in words): $LineSize$
Output:
Pad Size for Array: $PadSize$

if ($N \bmod LineSize == 0$) **then** $InitPad = 0$
else $InitPad = LineSize - N \bmod LineSize$
for $PadSize = InitPad$ **to** $CacheSize$ **step** $LineSize$
$Status$ = OK
Initialize $LinesArray[i] = 0$ for all i
$(0 \leq i \leq \lceil CacheSize/LineSize \rceil)$
for $i = 0$ **to** $Rows$
for $j = 0$ **to** $Cols$ **step** $Linesize$
$k = (i \times (N + PadSize) + j) \bmod CacheSize$
if ($LinesArray[k/$LineSize$] == 1$)
then $Status$ = CONFLICT
else $LinesArray[k/$LineSize$] = 1$
end for /* loop j */
end for /* loop i *
if ($Status ==$ OK) **then return** $PadSize$
end for /* loop $PadSize$ */
end Algorithm

Figure 3.15: Algorithm for computing pad size for array

minimum of 0 to a maximum of *CacheSize*, we test if the resulting padded row of size ($N + PadSize$) causes self-interference conflicts for the given tile of dimension $Rows \times Cols$. This is done by iterating through all the elements of the tile in steps of *LineSize*, and checking if any two elements map into the same cache line.

Multiple Tiled Arrays

The padding technique described above lends itself to an elegant generalization when computing the tile sizes of an algorithm involving more than one tiled arrays. We assume that the arrays have identical sizes ($S \times R$). Since both arrays are accessed in the same tiled loop, they have identical tile sizes. We choose the tile sizes such that, as before, the working set (which involves elements accessed from all the tiled arrays) fits into the cache. To determine the padding size such that both self-interferences within each tile, as well as cross-interferences across tiles are avoided, we first construct a tile consisting of the smallest rectangular shape (T) enclosing all the tiles. In Figure 3.16, the new contour formed from the tiles in arrays A and B is shown. We now determine the pad size such that rectangle T in an $S \times R'$ array does not incur self-interference, using the method described in Section 3.3.2. This becomes the pad size of all the tiled arrays. However, the tile size of each array remains $X \times Y$. Finally, we adjust the distance between the starting points of the arrays, so that the tiles are laid out the way they appear in rectangle T, in the first iteration. For example, in Figure 3.16, arrays A and B (both are now $S \times R'$) are laid out such that $A[0][0]$ and $B[0][0]$ are a distance $((X \cdot R') \bmod C)$ apart in the cache.

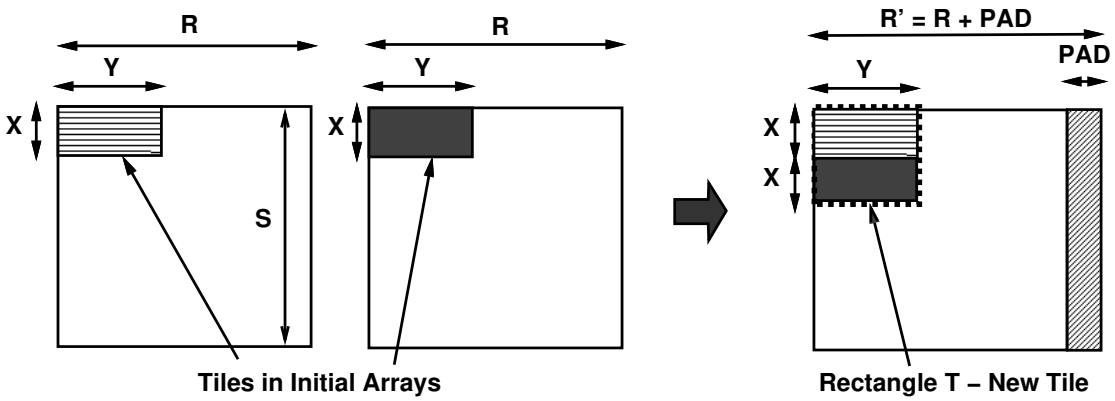


Figure 3.16: Multiple Tiled Arrays

3.3.3 Experiments

We performed experiments on two common machines widely used today – SUNSparc5 (SS5) and SUNSparc10 (SS10). The SS5 has a direct-mapped 8 KB data cache (1024 double precision elements) with a 16-byte cache line (2 elements per line). The SS10 has a 4-way associative 16 KB data cache (2048 elements) with a 32-byte cache line (4 double precision elements per line). Our experiments were performed both through actual measurement (MFLOPs) as well as simulations on the SHADE simulator from SUN Microsystems[Sun93].

The example programs on which we performed our experiments are: (1) Matrix Multiplication (MM) (with the standard *ijk*-loop nest permuted to *ikj*-order, as in [LRW91]); (2) Successive Over Relaxation (SOR)[CM95]; (3) L-U Decomposition without pivoting (LUD)[CM95]; and (4) Laplace[PTVF92].

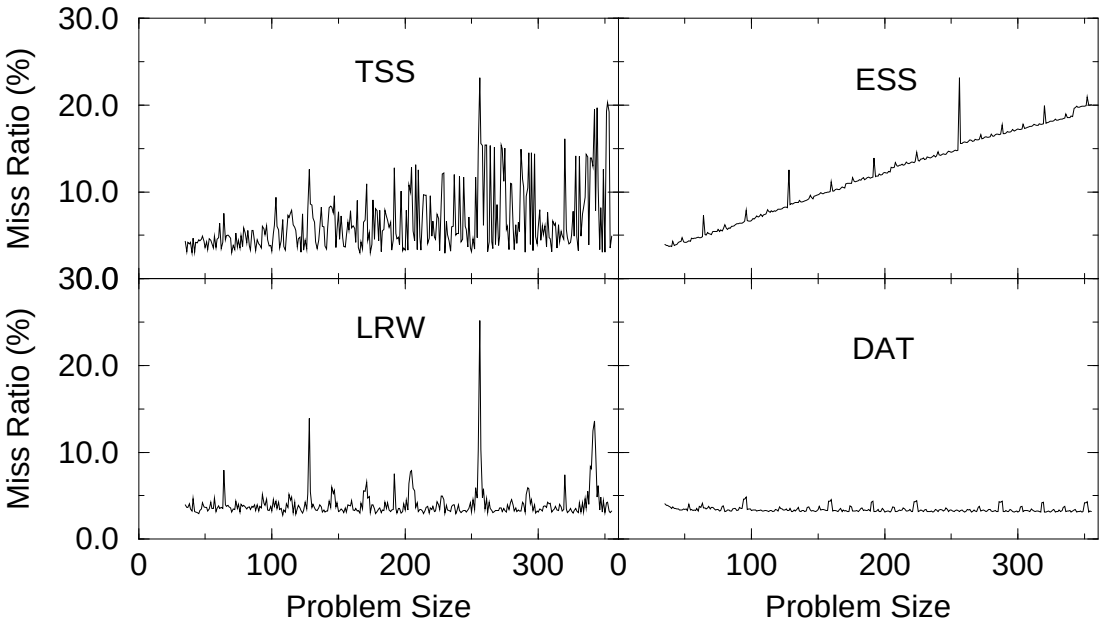


Figure 3.17: Matrix multiplication on SS-5: Variation of data cache miss ratio with array dimension

Uniformity of Cache Performance

Our first experiment was to verify our claim that the padding technique results in a uniformly good performance for a wide variety of array sizes. Figure 3.17 shows the variation of data cache miss ratios (on the SS5 cache configuration) of four algorithms (LRW ([LRW91]), ESS ([Ess93]), TSS ([CM95]), and DAT) on the matrix multiplication (MM) example for all integral array sizes between 35 and 350.[3] We observe that the miss ratio of DAT is consistently low, independent of problem size, whereas all other algorithms show some sensitivity to the size. This is attributed to the fact that DAT uses fixed tile dimensions (30×30) for the given

[3]For small array sizes, all the data fits into the cache, obviating the necessity for tiling; for larger sizes, the simulation times on the commercial simulator, SHADE, were too long to examine every integral data size. Hence the range 35-350. We expect similar performance for larger arrays.

cache parameters, independent of array size, whereas in the other algorithms, tile dimensions vary widely with array size.

Variation of Cache Performance with Problem Size

We present experiments below on the performance of each technique on the various examples, for several array sizes. We include array sizes of 256, 300, 301, and 550 to enable comparison with TSS [CM95] (which also presents data on these sizes). Array sizes 256 and 512 are chosen to illustrate the case with pathological cache interference, while 300 and 301 are chosen to demonstrate the widely different tile sizes for small changes in array size. Sizes 384, 640, and 768 illustrate the cases where the array size is a small multiple of a power of 2 ($384 = 128 \times 3; 640 = 128 \times 5; 768 = 256 \times 3$). We also include data for array sizes 700, 800, 900 and 1000.

Figure 3.18 shows the cache performance of LUD benchmark. An analysis of the array access patterns in LUD[PNDN97b] reveals the following relationship for the optimal tile shape:

$$\text{No. of Columns} = \text{No. of Rows} \times \text{Cache Line Size} \tag{3.3}$$

We observe that DAT and LRW have the lowest cache **miss ratios** (except at array sizes such as 256, 384, 512, etc., where LRW has considerably higher miss ratios). The lower miss ratio of LRW is, however, often at the expense of instruction count, since the relatively smaller tiles in LRW incur overheads introduced by loop tiling. TSS tends to incur higher miss ratios because it sometimes chooses a comparatively larger tile sizes, leading to the working set size being too close

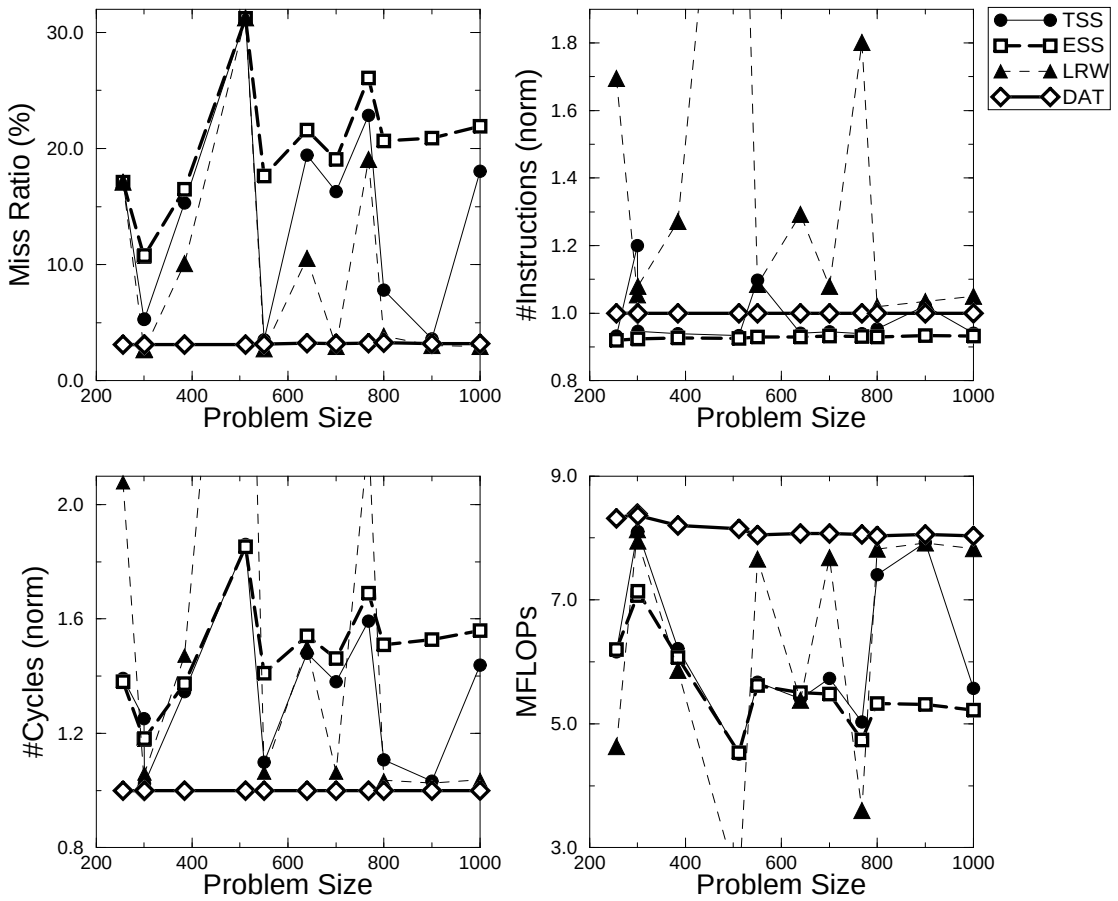


Figure 3.18: Performance of L-U Decomposition on SS-5: Variation of (1) Data Cache Miss Ratio, (2) Normalized Instruction Count, (3) Normalized Processor Cycles, and (4) MFLOPs with array size

to the cache size, which results in cross-interferences. ESS incurs cache conflicts because it does not account for cross-interferences.

A comparison of the **instruction count** (normalized to DAT) shows that, in general, ESS has a smaller number of executed instructions than DAT and LRW. This is a consequence of the larger tile sizes selected by ESS. The instruction count of TSS fluctuates because it selects widely varying tile dimensions.

The processor **cycle count** (with memory latency = 12 cycles; normalized to DAT) and **MFLOPs** measurements show that DAT has the best and most stable overall performance.

Table 3.1 summarizes the ratio of execution times of TSS, ESS, and LRW with that of DAT for the four benchmarks MM, LUD, SOR, and Laplace. In order to do a fair comparison, the table does not include array sizes such as 512, 640, etc., which would penalize all the other techniques. If these were taken into account, the improvement would be higher. On the SS5, we notice an average speedup of 24.2% over TSS, 38.0% over ESS, and 12.0% over LRW. On the SS10, we observe an average speedup of 7.2% over TSS, 13.8% over ESS, and 4.8% over LRW. The speedup on SS10 is smaller because the data cache has a higher capacity and associativity, which reduces the number of cache conflicts.

3.3.4 Discussion

The main advantage of DAT is its flexibility – decoupling tile selection from the padding phase allows the tile size to be independently optimized without regard to self-interference conflicts. This allows us to select larger tile sizes to maximize

Example	SUN Sparc 5			SUN Sparc 10		
	TSS/ DAT	ESS/ DAT	LRW/ DAT	TSS/ DAT	ESS/ DAT	LRW/ DAT
MM	1.30	1.64	1.14	1.15	1.30	1.06
LUD	1.22	1.40	1.13	1.08	1.13	1.09
SOR	1.23	1.26	1.07	1.02	1.05	1.00
Laplace	1.22	1.22	1.14	1.04	1.07	1.04

Table 3.1: Ratio of Execution Times: TSS/DAT, ESS/DAT, and LRW/DAT

the cache utilization. This also allows us to incorporate a user-supplied tile-shape which might be optimized for a specific application. For instance, one-dimensional tiling (e.g., in the SOR benchmark) can be easily handle by DAT, whereas it cannot be incorporated into TSS and LRW. For a given application and cache size, we choose a fixed tile size, independent of the array size (only the pad size varies with the array size), resulting in stability of the performance of the resulting tiled loop. Another advantage of our technique is that it is possible to avoid cross-interferences among several tiled arrays (e.g., in the Laplace benchmark). Finally, our padding technique can be easily extended to arrays with greater than two dimensions, by serializing the padding procedure over the different dimensions.

One consequence of the padding technique is that the structure of the data arrays is transformed, and references to the array in the rest of the code has to reflect the new structure. This, however, does not lead to any performance penalties. For example, when the row size is updated from R to R', the expression to compute the location for $a[i][j]$ changes from $(A + i \times R + j)$ to $(A + i \times R' + j)$, assuming the array begins at A. There could be a conflict in padding sizes if the same array is accessed in different tiled loops. In this case, we invoke the padding strategy on the more critical loop, and in the others, use one of the

existing methods (LRW or TSS) in the remaining loops, with updated row size for the array. Another consequence of our proposed data alignment technique is that it cannot be directly incorporated into a library routine, since the array sizes are not known. The optimization is most effective when it is integrated into the program, instead of compiled separately as a library routine. However, the incorporation into a library routine is possible under certain reasonable constraints – for example, if the interface to the tiled array is through file I/O, and the library routine constructs the data structure for the arrays in memory.

3.4 Summary and Status

Code generation for embedded processors reveals the scope for many optimizations that have been hitherto unaddressed in traditional compilers. An important set of features that can be exploited while generating code for embedded processors is the parameters of the data cache. In this chapter, we demonstrated how a careful data layout strategy for scalar and array variables that takes into account the parameters of the data cache, such as cache line size and cache size, could effect significant performance improvements in the execution of embedded code. We then presented an extension of the data alignment strategy to solve the tile size selection problem in the tiling/blocking compiler optimization. Data alignment through an appropriate padding of two-dimensional arrays leads to consistently good cache performance, independent of the array size. The blocking optimization is useful not only for scientific code, but also in the context of embedded applications. For example, image processing algorithms that compute the

value of a pixed using its value and that of its neighbours, have identical memory access patterns to the SOR algorithm discussed in Section 3.3.3.

The data organization strategy of Section 3.2 is implemented, and as reported in Section 3.2.4, incurs only a 4.0% compilation time overhead. For the tile size selection problem, we first derived the ideal shapes manually, and then used algorithms *SelectTile* and *ComputePad* to derive the actual tile sizes and pad sizes[PNDN97b].

Chapter 4

On-chip Vs. Off-chip Memory: Utilizing Scratch-Pad Memory

4.1 Scratch-Pad Memory

In addition to a data cache that interfaces with slower off-chip memory, a fast on-chip SRAM is often used in several applications, so that critical data can be stored there with a guaranteed fast access time. The on-chip SRAM, termed **Scratch-Pad memory**, refers to data memory residing on-chip, that is mapped into an address space disjoint from the off-chip memory, but connected to the same address and data buses. Both the cache and Scratch-Pad SRAM allow fast access to their residing data, whereas an access to the off-chip memory (usually DRAM) requires relatively longer access times. The main difference between the Scratch-Pad SRAM and data cache is that, the SRAM guarantees a single-cycle access time, whereas an access to the cache is subject to compulsory, capacity, and conflict misses.

The concept of Scratch-Pad memory is an important architectural consideration in modern embedded systems, where advances in fabrication technology have

made it possible to combine DRAM and ordinary logic on the same chip. As a result, it is possible to incorporate embedded DRAMs along with a processor core in the same chip [Mar97, Wil97]. Since data stored in embedded DRAM can be accessed much faster than that in off-chip DRAM, a related optimization problem that arises in this context is how to identify critical data in an application, for storage in on-chip memory. We use the terminology *Scratch-Pad SRAM* to include the embedded DRAM configuration.

We present a technique for minimizing the total execution time of an embedded application by a careful partitioning of scalar and array variables used in the application into off-chip DRAM (accessed through data cache) and Scratch-Pad SRAM. Figure 4.1 highlights the components of the embedded system architecture addressed by the techniques presented in this chapter.

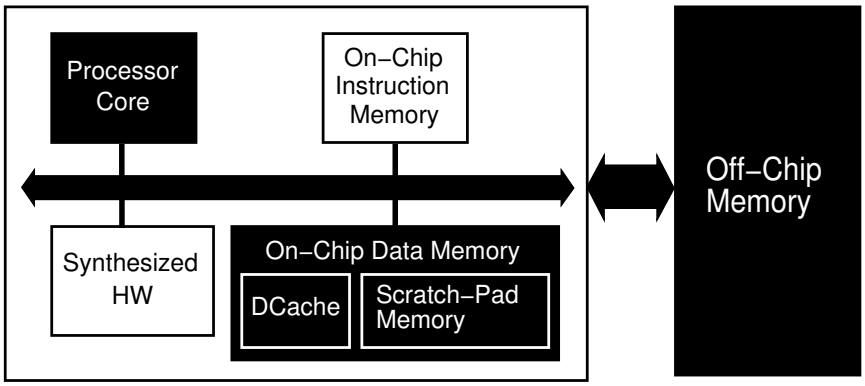


Figure 4.1: Shaded blocks are targetted by the optimization techniques presented in this chapter

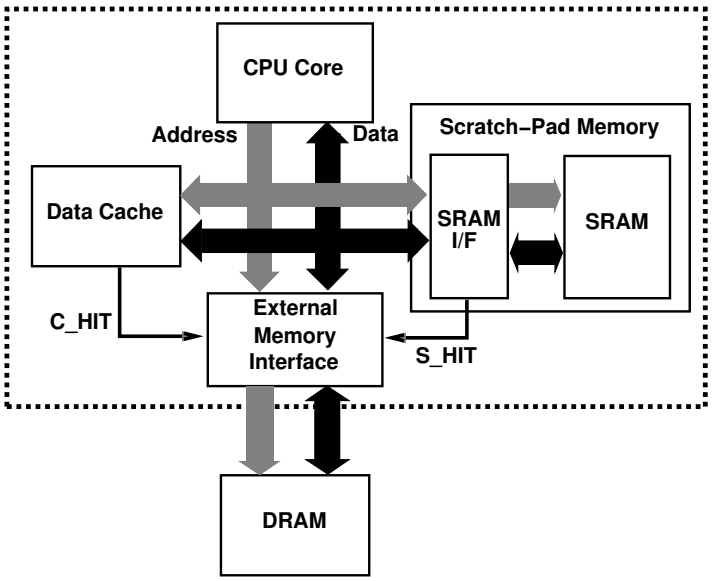


Figure 4.2: Block Diagram of Typical Embedded Processor Configuration

4.2 Problem Description

Figure 4.2 shows the architectural block diagram of an application employing an embedded core processor with Scratch-Pad memory. The address and data buses from the CPU core connect to the Data Cache, Scratch-Pad memory, and the *External Memory Interface* (EMI) blocks. On a memory access request from the CPU, the data cache indicates a cache hit to the EMI block through the **C_HIT** signal. Similarly, if the SRAM interface circuitry in the Scratch-Pad memory determines that the referenced memory address maps into the on-chip SRAM, it assumes control of the data bus and indicates this status to the EMI through signal **S_HIT**. If both the cache and SRAM report misses, the EMI transfers a block of data of the appropriate size (equal to the cache line size) between the cache and the DRAM.

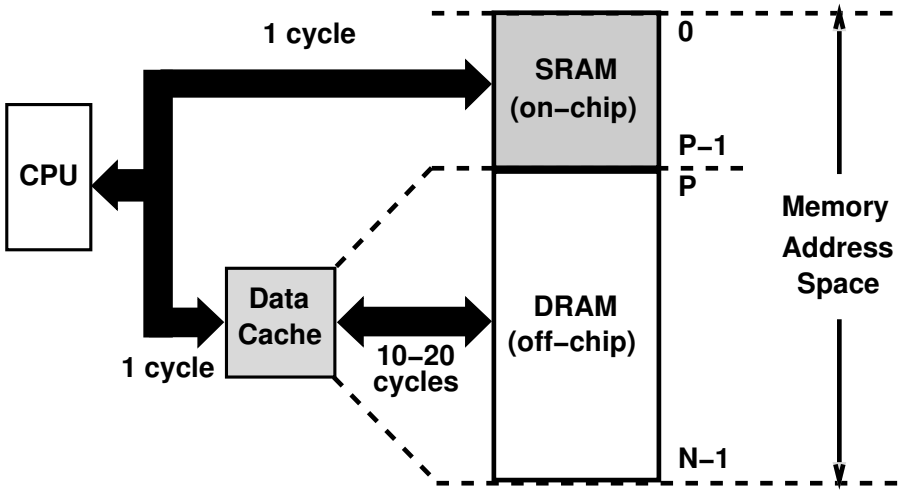


Figure 4.3: Division of Data Address Space between SRAM and DRAM

The data address space mapping is shown in Figure 4.3, for a sample addressable memory of size N data words. Memory addresses $0 \ldots P-1$ map into the on-chip Scratch-Pad memory, and have a single processor cycle access time. Thus, in Figure 4.2, S_HIT would be asserted whenever the processor attempts to access any address in the range $0 \ldots P-1$. Memory addresses $P \ldots N-1$ map into the off-chip DRAM, and are accessed by the CPU through the data cache. A cache hit for an address in the range $P \ldots N-1$ results in a single-cycle delay, whereas a cache miss, which leads to a block transfer between off-chip and cache memory, results in a delay of 10-20 processor cycles.

We assume that register allocation, the task that assigns frequently accessed program variables such as loop indices to processor registers, has already been performed. Given the embedded application code, our goal is to determine the mapping of each scalar and arrayed program variable into local Scratch-Pad SRAM or off-chip DRAM, while maximizing the application's overall memory access performance.

The sizes of the data cache and the Scratch-Pad SRAM are limited by the total area available on-chip, as well as by the single cycle access time constraint. Hence, we must first justify the need for both the data cache and SRAM. We motivate the need for both types of on-chip memory using the following example. Suppose the embedded core processor in Figure 4.2 can support a total of 2 KBytes for the data cache and the SRAM. We can analyze the pros and cons of four extreme configurations of the cache and SRAM:

(1) No local memory: In this case, we have the CPU accessing off-chip memory directly, and spending 10–20 processor cycles on every access. Data locality is not exploited and the performance is clearly inferior in most cases.

(2) Scratch-Pad memory of size 2K: In this case, we have an on-chip SRAM of larger size, but no cache. The CPU has an interface both to the SRAM and the off-chip memory. When large arrays that do not fit into the SRAM are present, the direct interface to external memory has to be used, thereby degrading performance.

(3) Data cache of size 2K: Here, we have a larger data cache, but no separate local SRAM. In many cases, having only a cache results in certain unavoidable cache misses that degrade performance due to stalled CPU cycles. We illustrate this with the example of a *Histogram Evaluation* code shown in Figure 4.4 from a typical Image Processing application, which builds a histogram of 256 brightness levels for the pixels of an $N \times N$ image.

The performance is degraded by the conflict misses in the cache between elements of the two arrays *Hist* and *BrightnessLevel*. Data layout techniques

```
char BrightnessLevel [512][512];
int Hist [256];      /* Elements initialized to 0 */
...
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        /* For each pixel (i, j) in image */
        level = BrightnessLevel [i][j];
        Hist [level] = Hist [level] + 1;
    end for
end for
```

Figure 4.4: Histogram Evaluation Example

(Chapter 3) are not effective in eliminating the above type of conflicts, because the accesses to *Hist* are data-dependent.

(4) 1K Data cache + 1K Scratch-Pad SRAM: The problem incurred in (3) above could be elegantly solved using the architecture of Figure 4.2, with a 1K data cache and 1K SRAM. Since the *Hist* array is relatively small, we can store it in the SRAM, so that it does not conflict with *BrightnessLevel* in the data cache. This storage assignment improves the performance of the *Histogram Evaluation* code significantly. The single-cycle access time guarantee for data stored in SRAM and the possibility of avoiding conflicts makes the architecture with a combination of cache and Scratch-Pad memory superior to cache alone.

From the above, it is clear that both the SRAM and data cache are desirable. Note that there could be applications where the Scratch-Pad SRAM offers no particular advantage over a single cache – for example, when all arrays are too big to fit into SRAM; or there is little temporal reuse among the arrays so that the cache conflicts do not cause any performance penalty. However, we noticed in our experiments that the SRAM improves performance in most typical applications.

We present a strategy for partitioning scalar and arrayed variables in an application code into Scratch-Pad memory and off-chip DRAM accessed through data cache, to maximize the performance by selectively mapping to the SRAM those variables that are estimated to cause the maximum number of conflicts in the data cache. We assume in this work that the array sizes and loop bounds are known, either statically, or through profiling. This is a reasonable assumption for many embedded applications.

4.3 The Partitioning Strategy

The overall approach in partitioning program variables into Scratch-Pad memory and DRAM is to minimize the interference between different variables in the data cache. We first outline the different features of the code affecting the partitioning, and then present a partitioning strategy based on these features.

4.3.1 Features Affecting Partitioning

The partitioning of variables is governed by the following code characteristics: (1) scalars variables and constants; (2) size of arrays; (3) life-times of variables; (4) access frequency of variables; and (5) conflicts in loops. We describe below each of the above factors and how our partitioning strategy address the features.

Scalar Variables and Constants

In order to prevent interference with arrays in the data cache, we map all scalar variables and constants to the Scratch-Pad memory. This assignment helps avoid the kind of conflicts mentioned in Section 4.2. If scalars are mapped to the DRAM, (and, consequently, accessed through the cache), it may be impossible to avoid cache conflicts with arrays, because arrays are assigned to contiguous blocks of memory, parts of which will map into the same cache line as the scalars, causing conflict misses.

Size of Arrays

We map arrays that are larger than the SRAM into off-chip memory, so that these arrays are accessed through the data cache. Mapping large arrays to the cache is the natural choice, as it simplifies the array addressing. If a part of the array were to map into the SRAM, the compiler would have to generate book-keeping code that keeps track of which region of the array is addressed, thereby making the code inefficient. Further, since most loops access array elements more or less uniformly, there is little or no motivation to map different parts of the same array to memories with different characteristics.

Life-times of Variables

The *life-time* of a variable, defined as the period between its *definition* and *last use* [ASU93], is an important metric affecting register allocation. Variables with disjoint lifetimes can be stored in the same processor register. The same

analysis, when applied to arrays, allows different arrays to share the same memory space.

The life-time information can also be used to avoid possible conflicts among arrays. To simplify the conflict analysis, we assume that an array variable accessed inside a loop is *alive* throughout the loop. Figure 4.5 shows an example lifetime distribution of four array variables, a, b, c, and d. Since b and c have disjoint life-times, the memory space allocated to array b can be reused for array c. Since arrays a and d have intersecting life-times, cache conflicts among them can be avoided by mapping one of them to the SRAM. However, since on-chip resources are scarce, we need to determine which array is more critical. The quantification of the criticality depends on the access frequency of the variables.

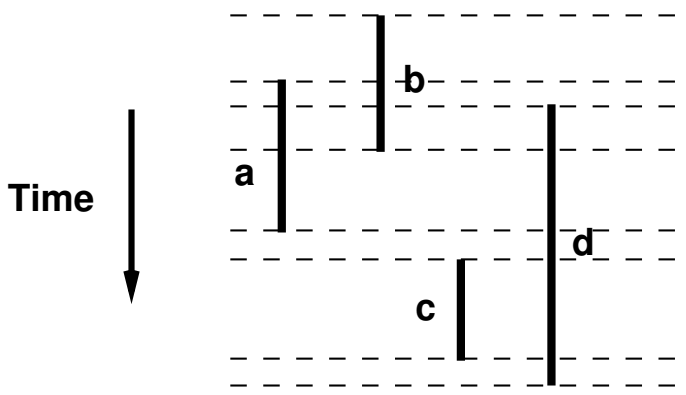


Figure 4.5: Example Life-Time Distribution

Access Frequency of Variables

To obtain an estimate of the extent of conflicts, we have to consider the frequency of accesses. For example, in Figure 4.5, note that array d could potentially conflict with all the other three arrays, so we could consider storing d in the SRAM.

However, if the number of accesses to d is relatively small, it is worth considering some other array (e.g., a) first for inclusion in SRAM, because d does not play a significant role in cache conflicts. For each variable u, we define the *Variable Access Count, VAC(u)*, to be the number of accesses to elements of u during its lifetime.

Similarly, the number of accesses to *other* variables during the lifetime of a variable, is an equally important determinant of cache conflicts. For each variable u, we define the *Interference Access Count, IAC*(u), to be the number of accesses to other variables during the lifetime of u.

We note that each of the factors discussed above, *VAC*(u) and *IAC*(u), taken individually, could give a misleading idea about the possible conflicts involving variable u. Clearly, the conflicts are determined jointly by the two factors considered together. A good indicator of the conflicts involving array u is given by the sum of the two metrics. We define the *Interference Factor, IF,* of a variable u as:

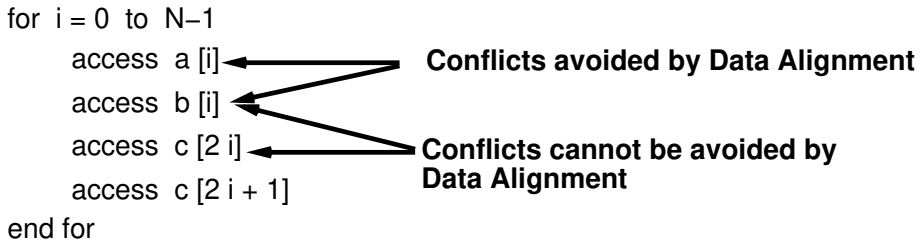
$$IF(u) = VAC(u) + IAC(u) \tag{4.1}$$

A high *IF* value for u indicates that u is likely to be involved in a large number of cache conflicts if mapped to DRAM. Hence, we choose to map variables with high *IF* values into the SRAM.

Conflicts in Loops

The *IF* factor defined in the previous section can be used to estimate conflicts in straight-line code and conditionals (i.e., non-loop code). In the case of arrays accessed in loops, it is possible to make a finer distinction based on the array

access patterns. Consider a section of a code in which three arrays a, b, and c are accessed, as shown in Figure 4.6(a).



(a)

LCF (a) = k (a) + (k (c)) = N + 2 N = 3N

LCF (b) = k (b) + (k (c)) = N + 2 N = 3N

LCF (c) = k (c) + (k (a) + k(b)) = 2N + N + N = 4 N

(b)

Figure 4.6: (a) Example Loop (b) Computation of LCF values

We notice that arrays a and b have an identical access pattern, which is different from that of c. Data alignment techniques (Chapter 3) can be used to avoid data cache conflicts between a and b – the arrays can be appropriately displaced in memory so that they never conflict in the cache. However, when the access patterns are different, cache conflicts are unavoidable (e.g., between b and c – the access patterns are different due to differing co-efficients of the loop variables in their index expressions). In such circumstances, conflicts can be minimized by mapping one of the conflicting arrays to the SRAM. For instance, conflicts can be eliminated in the example above, by mapping a and b to the DRAM/cache, and c to the Scratch-Pad memory.

To accomplish this, we define the *Loop Conflict Factor, LCF* for a variable u as:

$$LCF(u) = \sum_{i=1}^{p}(k(u) + \sum_{v} k(v)) \tag{4.2}$$

where the summation $\sum_{i=1}^{p}$ is over all loops $(1 \ldots p)$ in which u is accessed, and $\sum_v$ is over all arrays (other than u) that are also accessed in loop i, and for which it is not possible to use data placement techniques to completely eliminate cache conflicts with u. In the example above, where we have only one loop ($p = 1$), the LCF values shown in Figure 4.6(b) are generated. We have one access to a and two to c in one iteration of the loop. Total number of accesses to a and c combined is: $N + 2N = 3N$. Thus, we have $LCF(a) = 3N$, since cache conflicts between a and b can be completely eliminated by data placement techniques. Similarly, $LCF(b) = 3N$, and $LCF(c) = 4N$. The *LCF* value gives us a metric to compare the criticality of loop conflicts for all the arrays. In general, the higher the *LCF* number, the more conflicts are likely for an array, and hence, the more desirable it is to map the array to the Scratch-Pad memory.

4.3.2 Formulation of the Partitioning Problem

In Section 4.3.1, we defined two factors, Interference Factor (*IF*) and Loop Conflict Factor (*LCF*). We integrate these two factors and arrive at an estimate of the total number of accesses that could theoretically lead to conflicts due to an array. We define the *total conflict factor (TCF)* for an array u as:

$$TCF(u) = IF(u) + LCF(u) \tag{4.3}$$

where the $IF(u)$ value is computed over a lifetime that excludes regions of code where u is accessed inside loops. *TCF(u)* gives an indication of the total number of accesses that are exposed to cache conflicts involving array u, and hence, denotes the importance of mapping u to the SRAM.

We formulate the data partitioning problem as follows: **given a set of n arrays $A_1 \ldots A_n$, with TCF values $TCF_1 \ldots TCF_n$; sizes $S_1 \ldots S_n$; and an SRAM of size S, find an optimal subset $Q \subseteq \{1, 2, \ldots, n\}$ such that $\sum_{i \in Q} S_i \leq S$, and $\sum_{i \in Q} TCF_i$ is maximized.**

We note that the problem, as formulated above, is similar to the *Knapsack Problem* [GJ79]. However, there is an additional factor to consider in this case – *several arrays with non-intersecting lifetimes can share the same SRAM space.*

4.3.3 Solution of the Partitioning Problem

An exhaustive-search algorithm to solve the memory assignment problem would have to first generate clusters of all combinations of *compatible arrays* (arrays that can share the same SRAM space) and then generate all possible combinations of these clusters and pick the combination with total size fitting into the SRAM that maximizes the cumulative TCF value. This procedure requires $O(2^{2^n})$ time, which is unacceptably expensive, since the function $y = 2^{2^n}$ grows very rapidly, even for small values of n.

In our solution to the memory data partitioning problem, we first group arrays that could share SRAM space into clusters, and then use a variation of the *value-density* approximation algorithm [GJ79] for the Knapsack Problem to assign

clusters to the SRAM. The approximation algorithm first sorts all the items in terms of *value per weight* (i.e., the *value density*), and selects items in decreasing order of value-density until no more items can be accommodated. We define *Access Density (AD)* of a cluster c as:

$$AD(c) = \frac{\sum_{v \in c} TCF(v)}{\max\{\text{size}(v) | v \in c\}} \tag{4.4}$$

and use this factor to assign clusters of arrays into the SRAM. Note that the denominator is the size of the largest array in the cluster. This is because, at any time, only one of the arrays is *live*, and consequently, the arrays, which share the same memory space, need to be assigned only as much memory as the largest in the cluster. The details of the assignment algorithm (called *MemoryAssign*) are described in [PDN96].

4.4 Context Switching

Although an embedded system based on a processor core is likely to execute only a single program most of the time, some applications might require a context switch among different programs. In the architecture of Figure 4.2, if the entire SRAM is allotted to one program, the rest of the programs would not benefit from the presence of the SRAM in the architecture. One logical solution is to partition the SRAM space and allocate different partitions to different programs (Figure 4.7). This partitioning of the SRAM space is acceptable, because the number of programs involved is usually small in an embedded system which executes only a single application. Note that, unlike the register file, it is usually prohibitively expensive to save the contents of the entire Scratch-Pad SRAM in

external DRAM on a context switch, because it would add a considerable delay to the context switching time.

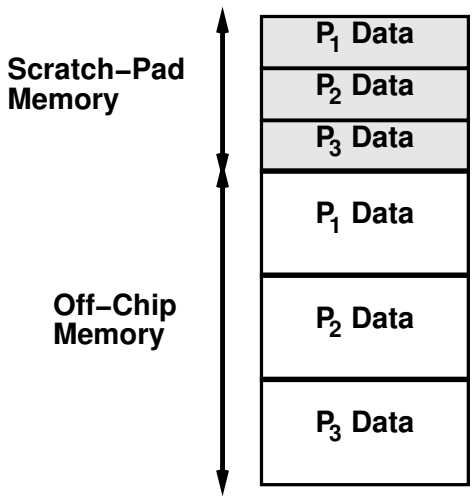


Figure 4.7: Memory space partitioned between data from different programs

To allocate the SRAM space efficiently among arrays of different programs, we use the same *TCF* metric identified in Equation 4.3. The partitioning algorithm, called *SRAMPartition*, is similar to algorithm *MemoryAssign* (Section 4.3.3). We now include arrays of all programs in the memory assignment phase. Since we have no prior knowledge of the order in which the different programs will be executed, we assume that arrays from different programs cannot share SRAM space. The SRAM space needs to be partitioned among variables in different programs only if the programs are expected to be executed in a threaded fashion. If the execution order of the programs is known to be sequential, then the entire SRAM is available to every program, and algorithm *MemoryAssign* can be used for memory assignment by considering each program separately.

4.4.1 Programs with Priorities

Some embedded systems are characterized by programs with varying priorities, which might arise from frequency of execution or the relative criticality of the programs. In such a case, it is necessary to modify algorithm *SRAMPartition* to take the relative priorities of programs $P_1 \ldots P_g$ into account. The approach we adopt in this case is to weight the TCF values for each array in a program by the priority of the respective program. That is, if the priorities of programs $P_1 \ldots P_g$ are given by $r_1 \ldots r_g$, we update $TCF(u) = TCF(u) \times r(i)$ for every array u in program P_i. The rest of the partitioning procedure remains the same as before.

4.5 Generalized Memory Hierarchy

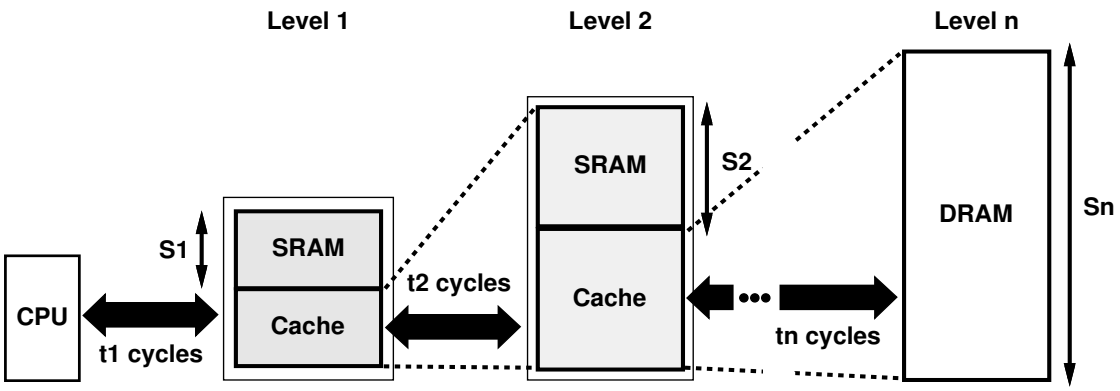


Figure 4.8: Generalized Memory Hierarchy

The architecture shown in Figure 4.2 can be generalized into an n-level hierarchy shown in Figure 4.8, with the data cache at each level acting as the interface between the levels above and below it. The total Scratch-Pad memory at level i

is S_i and the time required for the processor to access data at that level is t_i cycles. Such an architecture becomes realisitic with the advent of embedded DRAM technology, where large amounts of memory can now be integrated on-chip.

The memory assignment algorithms *MemoryAssign* and *SRAMPartition* can be generalized in a straightforward manner to effect memory assignment to program data in the hierarchical memory architecture shown in Figure 4.8. Since the memory access times of the different levels follow the ordering: $t_1 < t_2 < \ldots < t_n$, we store the most critical data in level 1, followed by level 2, etc. Thus, in the first iteration of the memory data partitioning, we invoke algorithm *MemoryAssign* with the SRAM size = S_1 and off-chip memory size = $S_2 + S_3 + \ldots S_n$. After the assignment to the first level Scratch-Pad memory, we remove the variables assigned to the first level, and continue the assignment for the remaining variables, now with SRAM size = S_2 and off-chip memory size = $S_3 + S_4 + \ldots S_n$. In other words, at every level i, the most critical data, as determined by the metric in Section 4.3.3, is mapped into the Scratch-Pad memory of size S_i.

The above greedy strategy is suitable when there is a relatively large number of arrays. However, an exhaustive search strategy could also be adopted for the memory assignment if the number of arrays involved is small.

4.6 Experiments

We performed simulation experiments on several benchmark examples that frequently occur as code kernels in embedded applications, to evaluate the efficacy

of our SRAM/DRAM data partitioning algorithm. We used an example Scratch-Pad SRAM and a *direct-mapped, write-back* data cache size of 1 KByte each. In order to demonstrate the soundness of our technique, we compared the performance, measured in total number of processor cycles required to access the data during execution of the example, of the following architecture and algorithm configurations: (**A**) *Data cache of size 2K*: in this case, there is no SRAM in the architecture; (**B**) *Scratch-Pad memory of size 2K*: in this case, there is no data cache in the architecture, and we use a simple algorithm that maps all scalars, and as many arrays as will fit into the SRAM, and the rest to the off-chip memory; (**C**) *Random Partitioning*: in this case, we used a 1K SRAM and 1K Data cache, and a random data partitioning technique (variables were considered in the order they appeared in the code, and mapped into SRAM if there was sufficient space); and (**D**) *Our Technique*: here we used a 1K SRAM and 1K data cache, and algorithm *MemoryAssign* for data partitioning. The size 2K was chosen in **A** and **B**, because the area occupied by the SRAM/cache would be roughly the same as that occupied by 1K SRAM + 1K cache, to a degree of approximation, ignoring the control circuitry. We use a direct-mapped data cache with line size = 4 words, and the following access times:

- Time to access one word from Scratch-Pad SRAM = 1 cycle.
- Time to access one word from off-chip memory (when there is no cache) = 10 cycles.
- Time to access a word from data cache, on cache hit = 1 cycle.
- Time to access a block of memory from off-chip DRAM into cache = 10 cycles for initialization + 1 $\times$ Cache Line Size = 10 + 1 $\times$ 4 = 14 cycles.

This is the time required to access the first word + time to access the remaining (contiguous) words. This is a popular model for cache/off-chip memory traffic[PH94].

Figure 4.9 presents a comparison of the performance for the four configurations **A, B, C**, and **D** mentioned earlier, on code kernels extracted from seven benchmark embedded applications. The characteristics of the applications are described in [PDN96]. The number of cycles for each application is normalized to 100. In the *Dequant* example, **A** slightly outperforms **D**, because all the data used in the application fits into the 2K SRAM used in **A**, so that an off-chip access is never necessary, resulting in the fastest possible performance. However, the data size is bigger than the 1K SRAM used in **D**, where the compulsory cache misses cause a slight degradation of performance. The results of *FFT* and *MatrixMult*, both highly computation-intensive applications, show that **A** is an inferior configuration for highly compute-oriented applications amenable to exploitation of locality of reference. Cache conflicts degrade performance of **B** and **C** in *SOR* and *DHRC*, causing worse performance than **A** (where there is no cache), and **D** (where conflicts are minimized by algorithm *AssignMemoryType*). Our technique resulted in an average improvement of 31.4% over **A**, 30.0% over **B**, and 33.1% over **C**.

In summary, our experiments on code kernels from typical embedded system applications show the usefulness of on-chip Scratch-Pad memory in addition to a data cache, as well the effectiveness of our data partitioning strategy. Our experiments on the data partitioning techniques in the presence of context switching

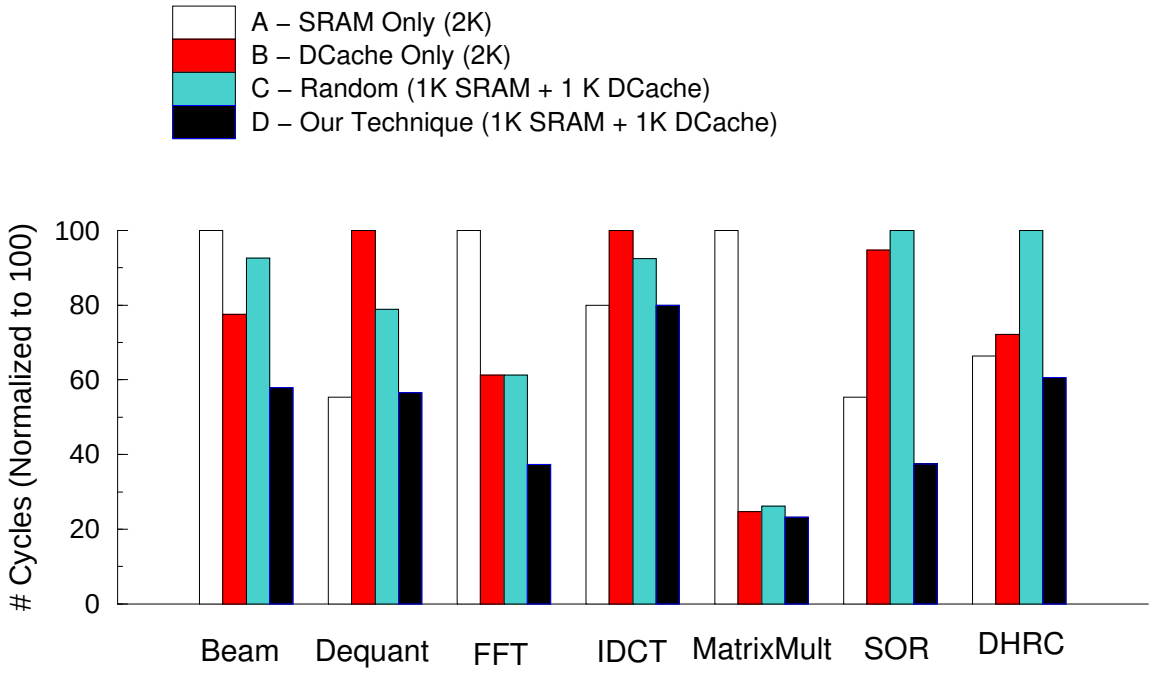


Figure 4.9: Performance Comparison of Configurations A, B, C, and D

show a similar performance improvement. Overall, we observed a 37% reduction in the total processor cycles due the memory accesses[PDN96].

4.7 Summary and Status

In order to effectively use on-chip memory in embedded systems, we need to leverage the advantages of both data cache (simple addressing) and on-chip Scratch-Pad SRAM (guaranteed low access time) by including both types of memory modules in the same chip. In this chapter, we presented a strategy for partitioning scalar and array variables in embedded code into Scratch-Pad SRAM

and data cache, that attempts to minimize off-chip memory traffic. We also presented extensions of our technique to handle context switching between different application programs, as well as a generalized hierarchical memory architecture.

The data partitioning algorithms presented in this chapter have been implemented for the case of a single application program. In the case of context switching, the SRAM/DRAM partitioning of variables was manually performed. The generated memory traces were analyzed with a Data Cache/SRAM simulator we developed for the purpose of the measurements.

Chapter 5

Memory Architecture Exploration

5.1 Motivation

An important feature of embedded processor-based design is that the CPU core is decoupled from the on-chip memory, and it is the system designer's responsibility to instantiate an appropriate on-chip memory configuration in his design. Traditionally, cache configuration for commercial microprocessors has been determined by conducting experiments on benchmark examples. Since general-purpose microprocessors in a PC or workstation environment have to execute a large variety of application software, a usual and effective means to improve performance has been to increase the cache size to the extent allowable by silicon area constraints. This strategy is acceptable because, in the absence of advance knowledge of the application programs that will execute on the processor, it is fair to assume that an increase in cache size will, in most cases, lead to improvement in performance, because more instruction and data can be held in local memory using larger caches, leading to possibly greater degree of reuse. However, in the embedded processor

domain, there is often only a single application program, and it is possible to conduct a more thorough analysis of the application to determine the best memory configuration. When coupled with an aggressive compiler that exploits the knowledge of this architecture, the impact on the overall design is significant. For instance, if an analysis of the application reveals that the data cache hit ratio is not likely to improve for cache sizes larger than 1 KByte, the information can be utilized to allocate expensive on-chip silicon area to other hardware resources, instead of an unnecessarily large cache.

An analytical prediction of the expected performance for various memory architectures has several advantages over a simulation-based approach. First, it is orders of magnitude faster, and is independent of the data size, unlike the simulation. Moreover, such an analysis is extremely useful in the initial stages of system design, when decisions on hardware/software partitioning are made. Although embedded systems afford longer program compilation times, the initial exploration phase of hardware/software trade-offs is, nevertheless, time consuming. The faster prediction made possible by an analytical technique allows a much larger design space to be explored, as opposed to a simulation-based technique.

In this chapter, we present an exploration strategy for determining an efficient on-chip data memory architecture – characterized by Scratch-Pad memory size and cache parameters – based on an analysis of a given application, so as to reduce off-chip memory traffic. The components in the generic embedded system architecture addressed in the exploration are the same as those highlighted in Figure 4.1.

```
# define N 128
# define M 4
# define NORM 16
int source[N][N], dest [N][N];
int mask [M][M];
int acc, i, j, x, y;
⋮
for (x = 0; x < N − M; x++)
  for (y = 0; y < N − M; y++) {
    acc = 0;
    for (i = 0; i < M; i++)
      for (j = 0; j < M; j++)
        acc = acc + source[x+i][y+j] * mask[i][j];
    dest[x+M/2][y+M/2] = acc/NORM;
  }
```

Figure 5.1: Procedure CONV

5.2 Illustrative Example

We illustrate our strategy for architectural exploration and optimization of on-chip memory on CONV, a convolution program frequently used in Image Processing tasks such as edge detection, regularization, and morphological operations [BMMZ95]. The code for CONV is shown in Figure 5.1.

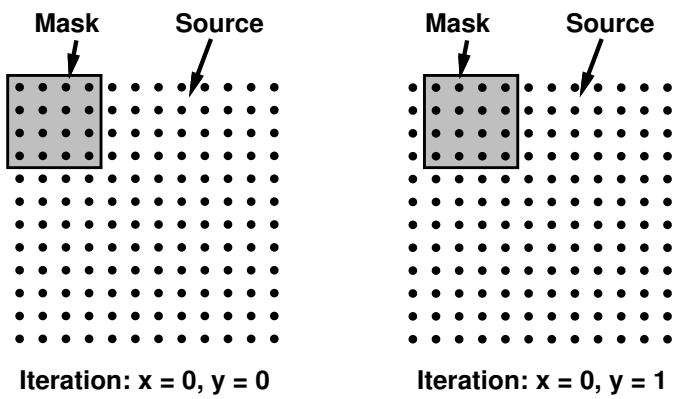


Figure 5.2: Memory access pattern in CONV example

A small (4×4) matrix of coefficients, *mask*, slides over the input image, *source*, covering a different 4×4 region in each iteration of y, as shown in Figure 5.2. In each iteration, the coefficients of *mask* are combined with the region of the image currently covered, to obtain a weighted average, and the result, *acc*, is assigned to the pixel of the output array, *dest*, in the center of the covered region.

We illustrate a typical memory exploration situation through the following problem: given a maximum amount, say 4 KB of on-chip memory space, find an efficient utilization of the space, in terms of data cache size, cache line size, and Scratch-Pad memory size, with the objective of minimizing the number of processor cycles required to access the arrays from memory. We assume that scalar variables are assigned to registers; the data cache is direct-mapped and write-through [PH94]; and the largest allowed cache line size is 128 Bytes.

Since we use a write-through cache, in which memory writes do not interfere with the cache contents in case of cache misses, the access to *dest* does not cause cache conflicts. However, if the two arrays *source* and *mask* were to be accessed through the data cache, the performance would be affected by cache conflicts. Further, an associative cache, by itself, will not eliminate the problem in general, because most practical caches have a limited associativity, and the general situation might require an associativity as large as the number of conflicting arrays.

The conflict problem can be solved by storing the small *mask* array in the Scratch-Pad memory. This assignment eliminates all conflicts in the data cache – the data cache is now used for memory accesses to *source*, which are very regular. Also, since $M(= 4)$ rows of the *source* array are *active* at any point in time, the

data cache need be only as large as $M(=4)$ rows of the source array. [1] Thus, we select a data cache of size $M \times N = 4 \times 128 = 512$ words = 2 KB. Since the accesses to *source* have good spatial locality, we select the largest allowed cache line, i.e., 128 bytes. The Scratch-Pad memory size is 4×4 words = 64 Bytes.

5.3 Exploration Strategy

In our formulation, a local (on-chip) memory architecture for an application is defined as a combination of:

- The total size of on-chip memory used for data storage.
- The partitioning of this on-chip memory into:
  - Scratch-Pad SRAM, characterized by its size; and
  - Data cache, characterized by
    - the cache size; and
    - the cache line size

The basic algorithm for memory architecture exploration is summarized in Figure 5.3.

For each candidate on-chip memory size T (loop L_1), we consider different divisions of T (loop L_2) into cache (size C) and Scratch-Pad SRAM (size $S = T - C$), selecting only powers of 2 for C. Procedure *DataPartition* essentially invokes algorithm *MemoryAssign* (Section 4.3.3) for partitioning program variables

[1] As mentioned before, the *dest* array is not allocated any storage in a write-through cache. If a *write-back* cache were used instead, we would allocate an equal cache space to *dest*, along with an appropriate memory assignment so that *source* and *dest* do not conflict in the cache [PDN97b].

```
Algorithm MemExplore
  L1: for on-chip memory size T (in powers of 2)
    L2: for cache size C (in powers of 2, < T)
      SRAM Size S = T − C
      DataPartition(S)
      L3: for line size L (in powers of 2, < C, < MaxLine)
        Estimate Memory Performance
      end for
    end for
    Select (C, L) that maximizes performance
  end for
end Algorithm
```

Figure 5.3: Overall algorithm for memory architecture exploration

into Scratch-Pad memory and cache. In the next section, we describe the memory performance estimation step. For each T, we select the (C, L) pair that is estimated to maximize performance. Finally, we perform the memory address assignment of the variables using the cache and SRAM parameters selected using the algorithms in Sections 3.2.2, 3.2.3 and 4.3.3.

5.4 Memory Performance Estimation

There is a trade-off in sizing the cache line. If the memory accesses are very regular and consecutive, i.e, exhibit good spatial locality, a longer cache line is desirable, since it minimizes the number of off-chip accesses and exploits the locality by pre-fetching elements that will be needed in the immediate future. On the other hand, if the memory accesses are irregular, or have large strides, a shorter cache line is desirable, as this reduces off-chip memory traffic by not bringing unnecessary data into the cache. The maximum size of a cache line is

the DRAM page size, which is usually a power of two. Thus, for example, for a DRAM with line size 1 KB = 256 words, there are a maximum of 9 (2^0 through 2^8) alternatives for the cache line size.

Suppose there are N scalar variables stored in off-chip memory, accessed M times in the program. We store all scalar variables in consecutive locations in memory. Since accesses to scalars invariably constitute a small fraction of the total accesses, we make the simplifying assumption that there is only one cache miss (a compulsory miss that occurs when the variable is first accessed into the cache) for every cache line containing scalars. Although this assumption looks too optimistic, it is actually reasonable when combined with our partitioning strategy for scalar and array variables – most of the scalars get mapped to the register file and Scratch-Pad memory, and not to the cache. Since there are N scalars, we require $\left\lceil \frac{N}{L} \right\rceil$ cache lines for them for a cache line size of L, i.e., there are $\left\lceil \frac{N}{L} \right\rceil$ cache misses, and consequently, $M - \left\lceil \frac{N}{L} \right\rceil$ cache hits for the M accesses. A memory access resulting in a cache hit requires one cycle, while a cache miss entails a delay of $(K + L)$ processor cycles, where K is a constant (usually 10–20 in modern processors [PH94]). Thus, the total number of processor cycles required for the M accesses to scalar variables is:

$$\text{Cycles (Scalars)} = (K + L) \left\lceil \frac{N}{L} \right\rceil + M - \left\lceil \frac{N}{L} \right\rceil \tag{5.1}$$

We determine an estimate of the processor cycles required to access the array elements by first dividing the application program into loop nests. Straight line code is treated as a singly-nested loop with an iteration count of one. Multi-dimensional arrays are assumed to be stored in row-major format.

for $i = 1$ **to** $M - 1$ **step** 1
  for $j = 1$ **to** $M - 1$ **step** 1
    $A[i][j] = A[i][j] + A[i-1][j] + A[i+1][j] +$
      $A[i][j-1] + A[i][j+1] + B[i] + C[j][i]$

(a)

L_1 : **for** $i_1 = l_1$ **to** h_1 **step** s_1
  L_2 : **for** $i_2 = l_2$ **to** h_2 **step** s_2
    ...
    L_m : **for** $i_m = l_m$ **to** h_m **step** s_m
      Read $a[i_1][i_2] \ldots [i_m]$
      ...
      L_n : **for** $i_n = l_n$ **to** h_n **step** s_n
        Read $b[i_1 + k_1][i_2 + k_2] \ldots [i_n + k_n]$
        Read $b[i_1 + k'_1][i_2 + k'_2] \ldots [i_n + k'_n]$

(b)

Figure 5.4: (a) Example loop (b) General n-level loop nest

Consider the example loop shown in Figure 5.4(a). $B[i]$ is reused in different j-iterations, so it is moved up into the i-loop. $A[i][j], A[i][j-1]$, and $A[i][j+1]$ exhibit *group-spatial* reuse [WL91] – the cache line accessed by one reference will usually also contain the data for the others. $A[i-1][j]$ and $A[i+1][j]$ have *self-spatial* reuse [WL91] – each can reuse the cache line it accessed in the previous j-iteration. Finally, reference $C[j][i]$ has no reuse. The memory references are grouped into *reuse equivalence classes* – each class consists of memory references that exhibit self-spatial or group-spatial reuse. This classification is used in [WL91] to aid in loop transformation procedures, such as skewing, reordering, etc., by enabling a comparison of the reuse properties of different candidate transformations. We propose a refinement of the above procedure in order to utilize the reuse analysis for our line size selection problem.

5.4.1 Refinement to self-spatial locality

In Figure 5.4(a) reference $B[i]$, after being moved to the i-loop, was assumed to have spatial reuse. However, this reuse can occur only if the cache line corresponding to $B[i]$ is still present in the cache when the next i-iteration begins, and has not been flushed out by the intervening accesses.

We generalize the above reuse condition for the example n-level nested loop of Figure 5.4(b). We assume that the loop bounds, and branch probabilities for conditionals are statically known, as is the usual case in many embedded applications. If such information is not known, we have to rely on profiling statistics. We use the following locality criterion: spatial locality for the reference $a[i_1][i_2]\ldots[i_m]$ in the level-m loop (L_m) can be exploited if the total number of memory accesses in inner loops (i.e., loops $L_{m+1}\ldots L_n$) is less than the cache size. Let the number of memory references at loop level j be c_j. For example, in Figure 5.4(b), $c_m = 1$ (for the single memory read), and $c_n = 2$ (for the two memory reads). The number of iterations (r_j) of loop level L_j is given by: $r_j = \left\lceil \frac{h_j - l_j + 1}{s_j} \right\rceil$. Thus, the sufficient condition for utilizing locality for the reference at nesting level m is:

$$\sum_{i=m+1}^{n} c_i \left(\prod_{j=m+1}^{i} r_j \right) \leq CacheSize \tag{5.2}$$

Note that the condition above, which involves the *number* of elements accessed, is an approximation of the cache behavior, for it assumes a fully associative cache with a perfect replacement policy. We incorporate the effect of cache conflicts that occur in limited associativity caches in Section 5.4.3.

5.4.2 Refinement to group-spatial locality

In Figure 5.4(a), reference $A[i-1][j]$ and $A[i][j]$ are assumed to have no spatial or temporal locality because they are in different rows. However, this is a pessimistic assumption. If one complete row of A fits in the cache, then the data read by $A[i][j]$ in one iteration can be reused by the $A[i-1][j]$ reference in the next i-iteration.

We formalize the above observation into a general condition for predicting reuse for the two b-references in Figure 5.4(b). We first determine a feasibility condition that needs to be met if the two references are to access the same cache line in different iterations. The sharing can occur only if all the index expressions in the higher dimensions $(1 \ldots n-1)$ match exactly for different values of the loop index, and the reuse-dimension (lowest dimension) resolves to expressions that differ in less than the cache line size. For example, we require that $(i_1 + k_1)$ in one iteration, say $i_1 = I$, be equal to $(i_1 + k_1')$ in some other iteration, say $i_1 = I + s_1 \cdot t$, for an integer t, since the two iteration numbers are separated by a multiple of the loop stride, s_1. We need to have: $I + k_1 + s_1 \cdot t = I + k_1'$, i.e., $k_1' - k_1 = s_1 \cdot t$, i.e., $(k_1 - k_1') \bmod s_1 = 0$.

Generalizing for dimensions $[1 \ldots n-1]$, we have:

$$\forall j \in [1 \ldots n-1], (k_j - k_j') \bmod s_j = 0 \tag{5.3}$$

For the reuse dimension (lowest dimension, indexed by i_n), we do not need an exact match, but only need that the expressions differ in less than the cache

line size L. Thus, for two iterations: $i_n = I$ and $i_n = I + s_n \cdot t$, we need to have:

$$(I + k_n + s_n \cdot t) - (I + k'_n) \leq L \tag{5.4}$$

i.e.,

$$k_n - k'_n - L \leq s_n \cdot t \tag{5.5}$$

To be applied as a feasibility test, this can be rephrased as:

$$\exists l \in [1 \ldots L], \text{ such that } (k_n - k'_n - l) \bmod s_n = 0 \tag{5.6}$$

Further, we need to ensure that the number of elements brought into the cache between the two accesses to b in Figure 5.4(b), is less than the cache size. For the two index expressions $[i_1 + k_1]$ and $[i_1 + k'_1]$, the number of iterations of loop L_1 that elapse between the two expressions resolving to the same value is: $\left\lfloor \frac{|k_1 - k'_1|}{s_1} \right\rfloor$. Since c_1 elements are accessed at the first loop level, the number of elements from loop L_1 brought into the cache in the $\left\lfloor \frac{|k_1 - k'_1|}{s_1} \right\rfloor$ iterations is: $c_1 \cdot \left\lfloor \frac{|k_1 - k'_1|}{s_1} \right\rfloor$. The number of elements in inner loops accessed in each iteration of loop L_1 is: $\sum_{i=2}^{n} c_i (\prod_{j=2}^{i} r_j)$ (LHS of Equation 5.2, with $m = 1$). Thus, the total number of elements (f_1) brought into the cache before $[i_1 + k_1]$ and $[i_1 + k'_1]$ resolve to the same value, is given by:

$$f_1 = c_1 \cdot \left\lfloor \frac{|k_1 - k'_1|}{s_1} \right\rfloor \cdot \sum_{i=2}^{n} c_i \left(\prod_{j=2}^{i} r_j \right) \tag{5.7}$$

Summing over all the n dimensions, we have the sufficient condition to enable group-spatial reuse as:

$$\sum_{t=1}^{n} f_t \leq CacheSize \tag{5.8}$$

i.e.,

$$\sum_{t=1}^{n} \left(c_t \cdot \left\lfloor \frac{|k_t - k'_t|}{s_t} \right\rfloor \cdot \sum_{i=t+1}^{n} c_i \left(\prod_{j=t+1}^{i} r_j \right) \right) \leq CacheSize \tag{5.9}$$

We conclude that the two b-references in Figure 5.4(b) exhibit spatial locality (i.e., fall into the same reuse equivalence class) if they satisfy the conditions 5.3, 5.6 and 5.9. The equations can be generalized to the case where the array index expressions are, of a more general form: $[a_j i_j + k_j]$ (where a_j and k_j are constants), as is the case with array references in many multimedia applications. The generalization is addressed in [PDN97a].

5.4.3 Refinement incorporating cache conflicts

Note that Equations 5.2 and 5.9 are approximations for cache reuse, for they ignore the possibility of cache conflicts. In this section, we outline the procedure for incorporating cache conflicts into the performance estimation.

Compatibility of Access Patterns

We call two array access patterns *compatible* if they satisfy the property $f_a - f_b = constant$ (independent of loop variables). This property holds for a large variety of typical array access patterns. For example, accesses to $A[i]$ and $A[i+2]$ in the same loop satisfy this property. The property also holds for two array accesses of the form $A[2i]$ and $B[2i+3]$.

If all accesses in a loop are compatible, then we can use a suitable data layout in memory to avoid cache conflicts completely. Consider the code fragment in Figure 5.5(a), to be mapped into a cache with line size = 4 words. The regions of the arrays that are referenced in one iteration are shown shaded in Figure 5.5(b). A memory assignment of arrays A and B that avoids conflicts should ensure that

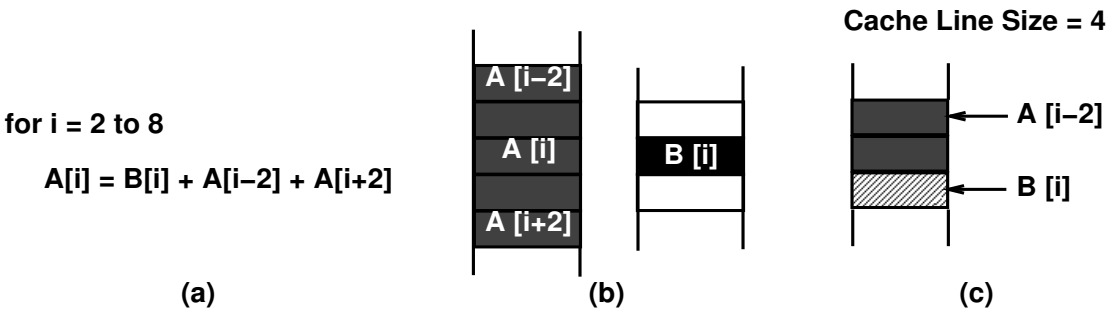


Figure 5.5: (a) Example code (b) Shaded region involved in one iteration (c) Cache mapping

the shaded regions of A and B never map into the same cache line. To do this, we note that the region of A involved in the array access in one iteration includes 5 words. We include the words such as $A[i-1]$ and $A[i+1]$, which are not used in the current iteration, but will be used in subsequent iterations, since they are automatically fetched into the cache line.

We note that the five consecutive words in A can occupy, in the worst case, a maximum of two lines in the cache with line size = 4 words [2] . Similarly, the one word from B can occupy one line. Thus, if we adjust the distance between $A[i-2]$ (the earliest A-word accessed in iteration i) and $B[i]$ (the earliest B-word accessed in this iteration), so that they are two lines apart when mapped into cache, we can avoid cache conflicts during loop execution. Using this strategy, when we avoid conflicts in one iteration, we also avoid conflicts in all future iterations because of the regularity (*affineness*) of array accesses (we accomplish this by selecting the maximum of cache lines that can span one iteration). In subsequent iterations, the entire pattern moves down the cache, eventually wrapping around to the top.

[2]If the region had six words, it could span three lines. E.g., we could have $A[i]$ mapping to line j; $A[i+1] \ldots A[i+4]$ mapping to line $(j+1)$; and $A[i+5]$ to line $(j+2)$.

To facilitate our analysis of conflicts, we group the memory references in loops into *Compatibility classes*. Compatibility classes are related to the reuse equivalence classes defined earlier in that, these classes are supersets of the reuse classes. Reuse classes of more than one array with compatible access patterns are clustered into the same compatibility class.

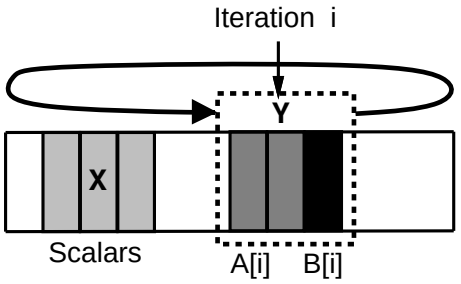


Figure 5.6: Cache conflict between scalars and arrays

Arrays Conflicting with Scalars

Scalar variables accessed in loops are typically stored in the register file, but if the number of scalars exceeds the available registers, they have to be stored in main memory, and consequently, accessed through the cache. In such a scenario, conflicts between arrays and scalars in memory are inevitable, since scalars are mapped to a fixed memory location (hence, a fixed cache location), whereas the accessed array elements map to different cache locations in different iterations. Figure 5.6 shows the cache memory where scalars in a loop map to region X of the cache, whereas elements accessed from arrays A and B map to different parts of the cache in different iterations. Let M_{sc} be the number of cache lines occupied by scalars accessed in the loop, and M_i be the number of cache lines occupied by array elements accessed in one loop iteration. For the example in Figure 5.6,

$M_A = 2$ cache lines; $M_B = 1; M_{sc}$ (the no. of lines occupied by scalars) $= 3$. In this case, conflicts between scalars and arrays occur whenever region X (which is fixed) and Y (which is moving) intersect in the cache.

During execution of the loop, we have cache conflicts whenever an accessed array element maps to one of the M_{sc} lines occupied by the scalars. Thus, if there are n_a array accesses in the loop for cache size C, each of them might map to the scalar region with a probability M_{sc}/C, assuming the accessed region moves uniformly over the loop iterations. Hence the expected total number of conflict misses per access $= (M_{sc}/C)n_a$. Similarly, the probability of an array interfering with one of the n_{sc} scalar accesses is (M/C) (where M is defined as $M = \sum_i M_i$), so the expected number of scalar misses per access $= (M/C)n_{sc}$. The expected conflict miss ratio is, then, $((M_{sc}/C)n_a + (M/C)n_{sc})/(n_{sc} + n_a)$.

Conflicts among Arrays in Different Equivalence Classes

When multiple arrays are accessed with different access patterns in a loop (i.e., they belong to different compatibility classes), it is difficult to formalize a strategy to avoid conflicts. To compute an estimate of the hit ratio in this case, we assume a uniform distribution of the memory accesses in the cache. This assumption usually holds for data sizes that are much larger than the cache.

We first divide the arrays into compatibility classes $S_1 \ldots S_g$. Each class covers a total of M_{S_j} lines in cache, where M_{S_j} is the sum of the cache lines occupied by the constituent arrays in class S_j. The probability of the arrays interfering with any of the n_{sc} scalars in cache is M/C where $M = \sum_i M_{S_j}$. Hence, the expected

number of scalar misses is: $(M/C)n_{sc}$,. Similarly, the probability that an access to any element in class S_j results in a miss, is the probability that it conflicts with any scalar, or any of the other classs. Thus, it could conflict if it were to map to any of the $M_{sc} + (M - M_{S_j})$ locations occupied by the scalars and remaining arrays. In other words, the probability of a conflict miss for class $S_j = (M_{sc} + M - M_{S_j})/C$.

If the miss ratio in loop L_j is predicted to be h_j, the expected conflict count is given by $h_j \cdot \left(\prod_{i=1}^{j-1} r_i\right)$. The estimated number of processor cycles wasted due to conflicts is given by:

$$EstConflict(j) = h_j \cdot \left(\prod_{i=1}^{j-1} r_i\right) \cdot (K + L) \tag{5.10}$$

where K and L are as defined in Equation 5.1.

5.4.4 Computation of Processor Cycles

Based on the analysis in the previous sections, we determine the number of processor cycles, T_{ji}, due to each reuse equivalence class R_{ji} in loop level L_j, as in the loop nest of Figure 5.4(b). Let the number of memory references in R_{ji} be $|R_{ji}|$. For the r_j iterations of loop L_j, the total number of memory accesses for class R_{ji} is: $r_j \cdot |R_{ji}|$, of which $\frac{r_j \cdot |R_{ji}|}{L}$ are cache misses, and the rest $(r_j \cdot |R_{ji}| - \frac{r_j \cdot |R_{ji}|}{L})$ are hits. Since a cache hit costs 1 cycle and a miss $(K + L)$ cycles, we have:

$$\begin{aligned} T_{ji} &= \frac{r_j \cdot |R_{ji}|}{L}(K + L) + r_j \cdot |R_{ji}| - \frac{r_j \cdot |R_{ji}|}{L} \\ &= r_j \cdot |R_{ji}| \cdot \left(\frac{K - 1}{L} + 2\right) \end{aligned} \tag{5.11}$$

For the set E of all the equivalence classes at level j, we have, the total cycles, T_j given by: $T_j = \sum_i T_{ij}$.

The number of accesses with no reuse at loop level L_j is given by: $c_j - \sum_i |R_{ji}|$. The number of cycles, Q_j, spent in accessing elements with no reuse is given by:

$$Q_j = (K + L) \cdot \left(c_j - \sum_i |R_{ji}| \right) \tag{5.12}$$

Finally, the number of processor cycles, P_j, wasted due to cache conflicts for accesses in loop L_j is given by:

$$P_j = r_j \cdot EstConflict(j) \tag{5.13}$$

Thus, the total time, spent accessing memory data at level L_j is given by: $(T_j + Q_j + P_j)$. For the entire loop nest, we multiply the total cycles at each loop level by the number of times the loop L_j is executed. Thus, the total cycles, C for the loop nest under consideration is given by:

$$C = \sum_{j=1}^{n} \left((T_j + Q_j + P_j) \cdot \prod_{i=1}^{j-1} r_j \right) \tag{5.14}$$

To determine the number of cycles required for all memory accesses in the program, we take the cumulative estimate over all loop nests. We use this estimate in the memory performance estimation step in algorithm *MemExplore* (Figure 5.3).

5.5 Experiments

We present some exploration results on sample application routines from the image processing and digital signal processing domain. The benchmark examples are: *Histogram* – a histogram evaluation routine, commonly used in image enhancement algorithms [GW87]; *Lowpass* – another image processing algorithm

used for accentuating low frequencies in an image, so that the resulting image has lower changes between neighboring pixel values [PD95a]; and *Beamformer* – a radar application, involving the summation of digitized signals from an antenna array [PD95a]. We present below a comparison of the estimated memory cycles for the benchmarks with the actual memory cycles. The estimated cycles are determined by the computations in Section 5.3. The actual cycles are measured by our cache/SRAM simulator, which takes as input a stream of memory addresses generated during execution of a benchmark, and reports the number of accesses to off-chip memory. For our experiments, we used a penalty of 10 processor cycles for off-chip memory accesses (i.e., $K = 10$ in Equations 3.2.2 and 5.12).

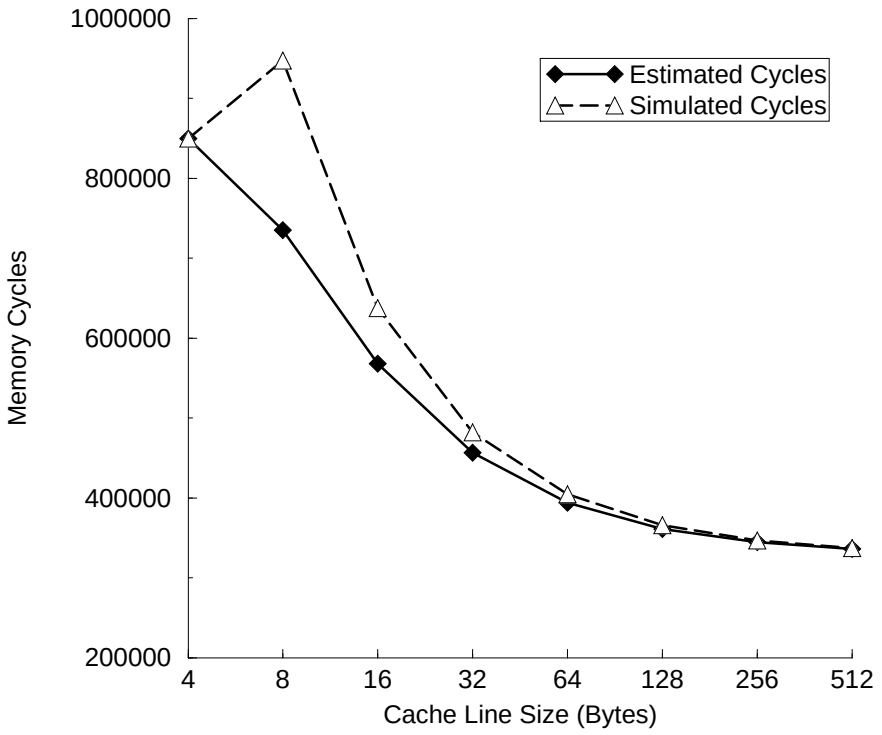


Figure 5.7: Variation of memory performance with line size for fixed cache size of 1 KB (*Histogram* Benchmark)

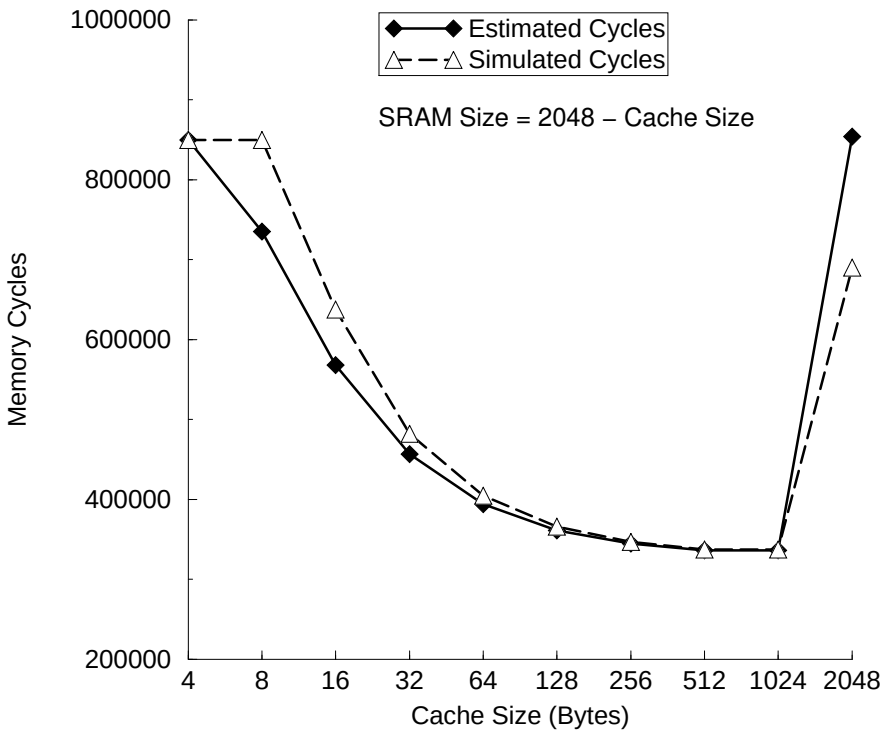


Figure 5.8: Variation of memory performance with different mixes of cache and Scratch-Pad memory, for total on-chip memory of 2 KB (*Histogram* Benchmark)

Our first experiment focusses on loop L_3 of the *MemExplore* exploration algorithm (Figure 5.3), where we study variation of the memory performance with the cache line size, for a given cache size. Figure 5.7 shows a comparison between the actual simulation result and the estimated number of processor cycles for the memory accesses in the *Histogram* routine, for a fixed cache size of 1 KB. We note that the estimated performance very closely follows the actual performance, except for one line size (8 bytes). The best cache line size selected by our algorithm (512 bytes), is also the best line size parameter determined from the simulation.

Figure 5.8 illustrates another slice of the exploration space on the same benchmark, where the estimated memory performance for different divisions of a fixed total on-chip memory space of 2 KB into data cache and Scratch-Pad memory, is compared against the simulated performance. The memory cycles plotted correspond to one iteration of the outer L_2 loop of *MemExplore*, where the best cache line size (from loop L_3) is selected for each candidate cache size. For each selected cache size, the corresponding Scratch-Pad SRAM size is given by:

$$\text{Scratch-Pad SRAM size} = 2 \text{ KB} - \text{cache size} \tag{5.15}$$

The points on the left and right extremes represent divisions incurring severe cache conflicts (for cache size = 2048 Bytes, the SRAM size is 0, causing unavoidable conflicts in the cache). The estimation process gives the best division of the 2 KB space as: 1 KB cache + 1 KB Scratch-Pad memory – this selection is validated with the actual simulation results, as shown in Figure 5.8.

Figure 5.9 shows the variation of the memory performance with the total on-chip memory space for the *Histogram* example. The y-axis shows the best performance obtained by any architecture (i.e., division into Scratch-Pad memory and

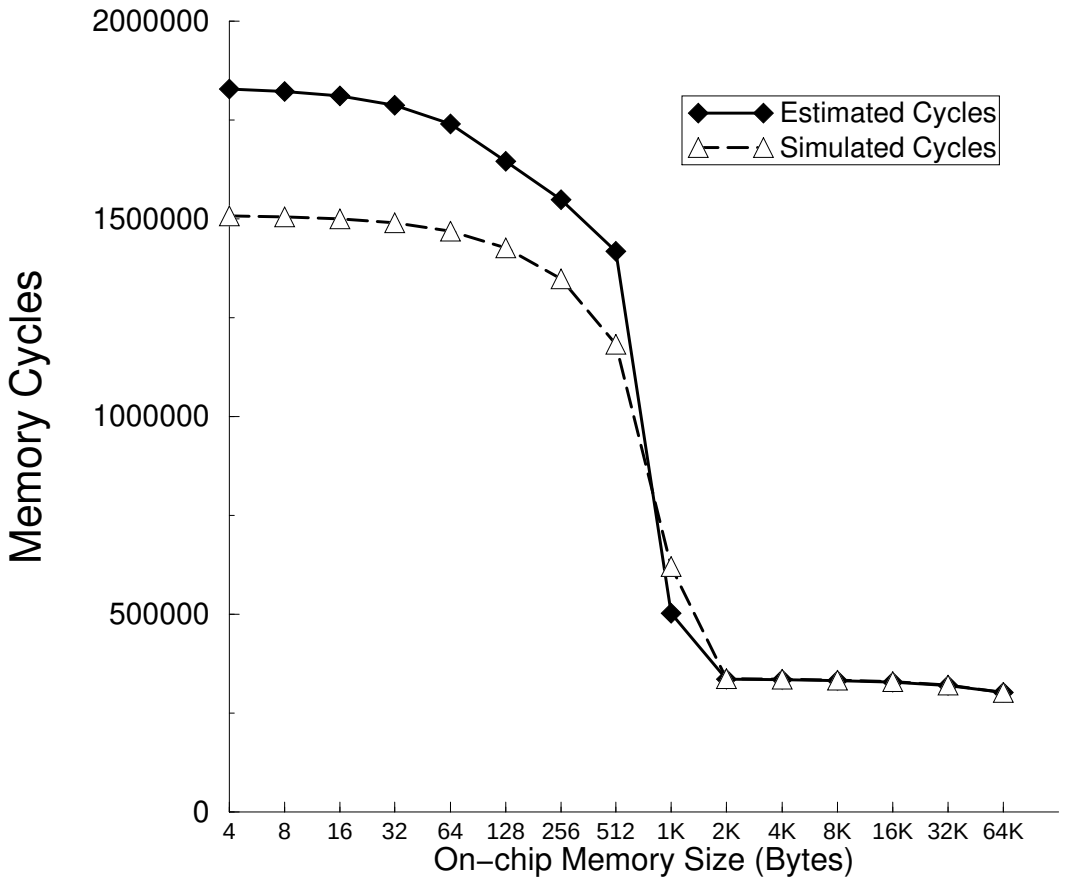


Figure 5.9: Variation of memory performance with total on-chip memory space (*Histogram* Benchmark)

cache, as well as selection of cache line size) for a given total on-chip memory space, i.e., one iteration of the outer loop L_1 in *MemExplore*. For a given application, the variation of the memory performance with the total on-chip memory is generated as feedback to the designer. The designer can then select an appropriate total on-chip memory size, based on the value beyond which no significant improvement is predicted. In the example of Figure 5.9, the total size of 2 KB is a good selection, as we observe very little improvement in cycle time beyond this cache size. Figures 5.10 and 5.11 show the results of the exploration for the *Lowpass* and *Beamformer* examples. We observe that the estimated performance curve follows the actual performance very closely. This curve can help the designer select the optimal on-chip memory size. Once the designer chooses an appropriate total memory size based on the estimation curve, the best memory architecture (consisting of the division of this space into Scratch-Pad memory and cache, as well as cache line size) is automatically chosen. Note that the each point in the exploration spaces of Figures 5.9, 5.10 and 5.11 corresponds to the best architecture found for the given total on-chip memory space.

The most important advantage of our analytical technique for exploring the memory performance of embedded applications is that candidate architectures can be rapidly evaluated for their memory performance. The estimation-based exploration for our experiments above required only a few seconds, which was about 1000 times faster than the simulation of the memory performance for the same set of architectures explored for the data sizes we considered. For larger data sizes, the simulation times increase further, whereas the estimation time remains constant. This rapid estimation capability is very important in the initial stages of system

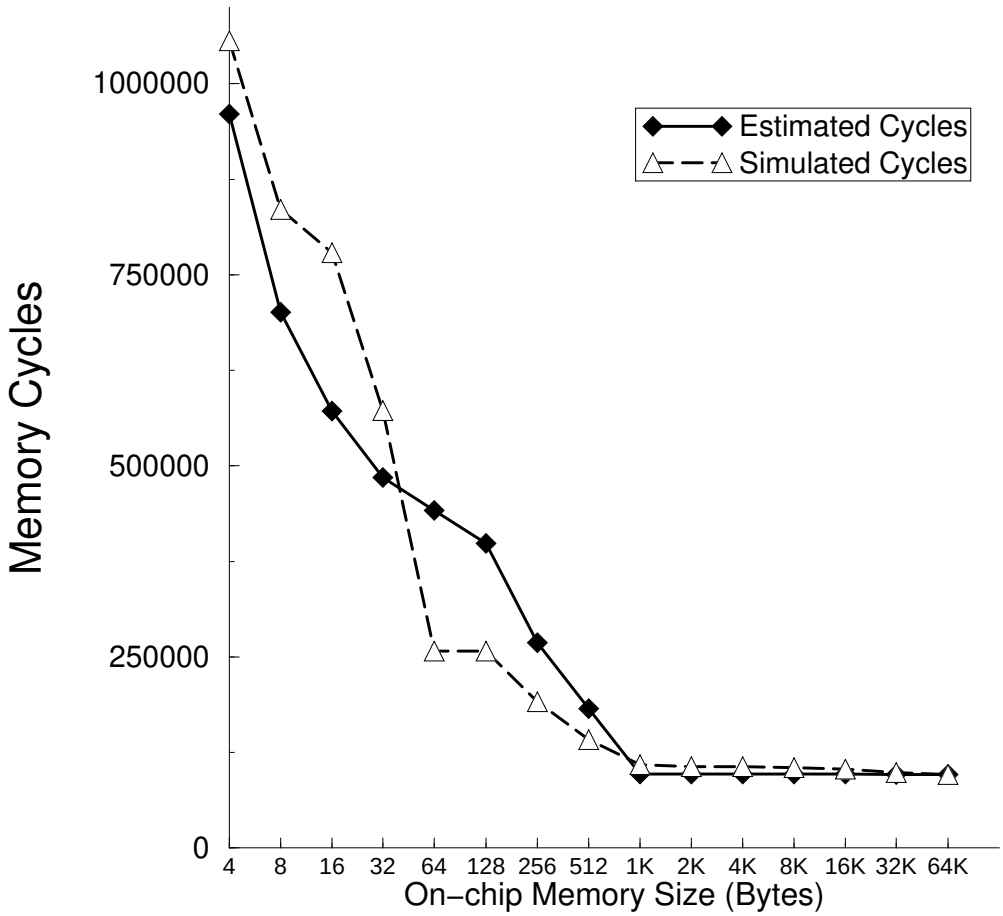


Figure 5.10: Memory exploration for *Lowpass* benchmark

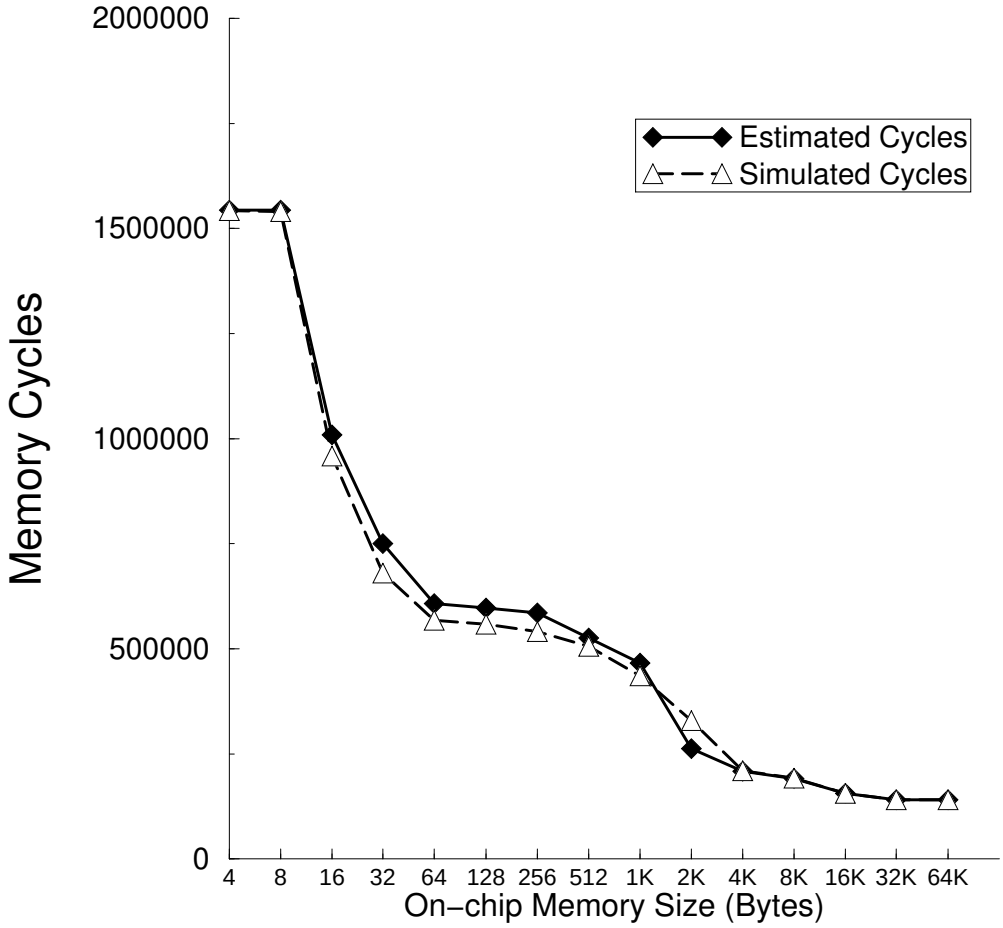


Figure 5.11: Memory exploration for *Beamformer* benchmark

design, where the number of possible architectures is too many, and a simulation of each architecture is prohibitively expensive.

Our proposed analytical approach can be combined with one or more final simulations to verify the predicted performance. The purpose of the analytical technique is to drastically reduce the design space so that the final design decision can be taken by the designer after a few simulations of design points around the predicted optimal result.

5.6 Summary and Status

The management of local memory space is an important problem in embedded systems involving large behavioral arrays, all of which cannot be stored on-chip. The ideal local memory architecture, consisting of a judicious division of the memory into Scratch-Pad SRAM (software-controlled) and data cache (hardware-controlled), as well as the cache line size, depends on the characteristics of the specific application. We presented a strategy for exploration of on-chip memory architecture for embedded applications, based on an estimation of the memory performance. Thus, the exploration technique is very useful during the early stages of system design, when a large number of different possible memory architectures need to be evaluated for their performance.

The exploration technique described in this chapter has been implemented. The experimental results were obtained by executing the analytical performance estimator on an internal data structure representing the memory accesses of the

various benchmark examples. These were compared against the cache/SRAM performance simulation of memory traces for the applications.

Chapter 6

Case Study: MPEG Decoder

In Chapter 5, we presented an exploration strategy for predicting the performance of various on-chip memory configurations for embedded systems. We validated our exploration strategy by performing simulation experiments on small benchmark loop kernels (Section 5.5). In this chapter, we present a case study of our exploration technique on a relatively large example: the MPEG decoder. The MPEG is a digital video compression standard which is widely used in the telecommunication, computer, and consumer electronics industry today to encode and decode video images[Gal91]. We perform the memory exploration on a software implementation of the MPEG decoder. The MPEG decoder forms an interesting case study for our technique because it is a memory-intensive application which forms a crucial component in various graphics-related applications such as digital TV and video conferencing. The MPEG behavior is characterized by a number of interesting array access patterns, which makes it a good candidate on which to apply our memory exploration technique. The case study demonstrates that our exploration technique is effective not just for individual loop kernels, but also more complex applications, since different loops in the MPEG decoder access the same arrays in dissimilar ways.

6.1 Overview of MPEG Decoder

The MPEG encoding algorithm exploits spatial and temporal redundancy of consecutive video images to achieve compression. The compression algorithm uses block-based *motion compensation* for reduction of temporal redundancy and Discrete Cosine Transform (DCT)-based compression for reduction of spatial redundancy [Gal91].

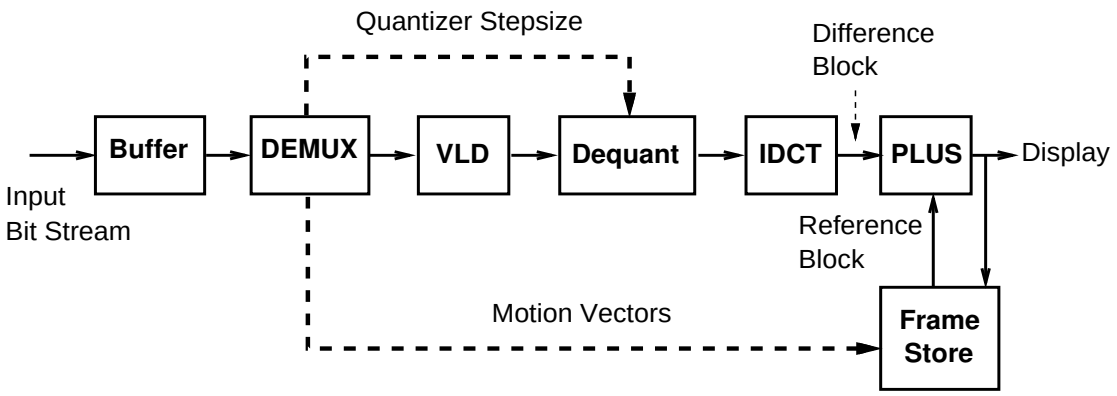


Figure 6.1: Overview of MPEG Decoder structure

Figure 6.1 shows an overview of the major functional blocks of the decoder algorithm[Tho95]. The *Buffer* stores the input bit stream so that the rest of the decoder can operate on it. The de-multiplexer (*DEMUX*) isolates the header information such as motion vectors, quantizer stepsize, and quantized DCT coefficients. The *VLD, Dequant,* and *IDCT* blocks help reconstruct the *difference-block*, which is a sort of increment to a *reference-block* stored in the *Frame Store* component. The two are then added in *PLUS* to generate the output image. This output image is stored into *Frame Store*, as well as displayed.

In the remaining sections, we first examine the memory characteristics of the individual *Dequant* and *PLUS* blocks of Figure 6.1, and finally, study the entire decoder application as a whole. We present a comparison of the estimated and simulated memory performance.

6.2 Memory Exploration of Sample Routines

6.2.1 Dequant Routine

The *Dequant* routine performs the de-quantization operation on the input quantized DCT coefficients received from the *DEMUX*[Gal91]. The code for the *Dequant* routine, which is a translation of a SpecCharts[GVNG94] model used in [Tho95], into C language, is shown in Figure 6.2. For simplicity of addressing, we convert 8-bit and 10-bit data types into the integer data type of C.

The code consists of a *for*-loop in which four arrays are accessed: *scan, curr_block, intra_quant,* and *dct_zz*. We observe that, while three of the arrays, *scan, curr_block,* and *intra_quant* are accessed by the simple loop index i, the element of *dct_zz* accessed in each iteration is data dependent. The access pattern is similar to the *Histogram* example of Section 4.2. If the *scan, curr_block,* and *intra_quant* arrays are mapped into off-chip memory and accessed through the cache, they can be aligned to prevent any cache conflicts among them. However, if *dct_zz* is also accessed through the cache, data alignment does not solve the conflict problem since the accessed addresses from *dct_zz* are statically unknown. In this case, we use the technique shown in Section 5.4.3 to predict the memory performance.

```
int curr_block [64];
int scan [64];
int intra_quant [64];
int dct_zz [64];
int zigzag, zz, val;
⋮
for (i = 0; i < 64; i++) {
  zigzag = scan [i];
  zz = dct_zz [zigzag];
  val = (2 * zz * quantizer_scale * intra_quant [i]) / 16;
  if (msb(val) == 0)
    val = val - sign (val);
  if (val > 2047)
    val = 2047;
  else
    if (val < -2047)
      val = -2047;
  curr_block [i] = val;
}
```

Figure 6.2: Code for *Dequant* Routine

Further, the on-chip Scratch-Pad memory is useful in such a situation, because all cache conflicts can be avoided if the *dct_zz* array is stored in the Scratch-Pad memory.

Cache Line Size

Figure 6.3 shows a comparison of the estimated and actual memory performance of the *Dequant* routine, when the cache line size is varied, keeping the total cache size fixed at 1 KB.

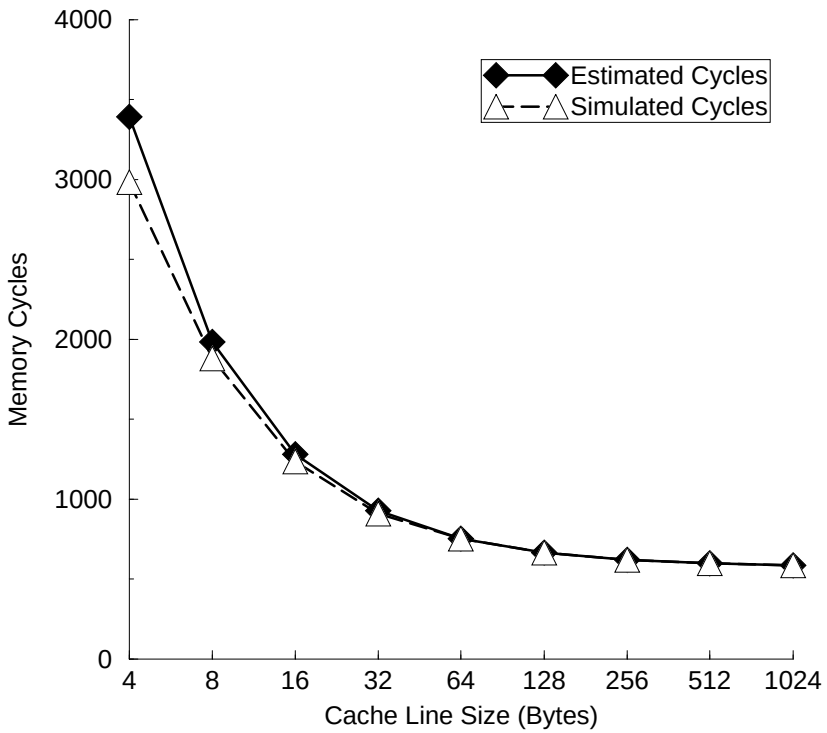


Figure 6.3: Variation of memory performance with line size for fixed cache size of 1 KB (*Dequant* Routine). No Scratch-Pad memory.

The performance improves with increasing cache line size because the combined size of all the four arrays is less than the cache size of 1 KB. In this case, there is no cache conflict, and the accesses to off-chip memory occur only due to compulsory misses when the data is first brought into the cache. Since compulsory misses are reduced by longer cache lines, we observe a monotonic performance improvement with increasing cache line size. Figure 6.3 shows that the estimated performance tracks the actual performance very well.

Cache and Scratch-Pad Memory Ratio

Figure 6.4 shows the variation of the memory performance with varying ratios of cache and Scratch-Pad memory for a fixed total on-chip memory size of 1 KB. Each point on the graph corresponds to the most favourable cache line size for the selected cache size. Thus, the points on the extreme right, corresponding to cache size = 1024 Bytes (i.e., Scratch-Pad memory size = 0) represents the best cache line size configuration for a 1024 Byte cache, i.e., the lowest (extreme right) point on the curve in Figure 6.3.

Since, as we observed before, the combined area of the arrays is less than 1 KB, the best performance is obtained when the arrays are all stored in on-chip Scratch-Pad memory, and the cache is not utilized at all. This is confirmed by Figure 6.4, where the best performance is predicted (and verified by simulation) for the smallest cache sizes. The performance begins to degrade when the cache size is 64 Bytes or larger. This point corresponds the the Scratch-Pad memory being too small to contain all the arrays. We observe from Figure 6.4 that the

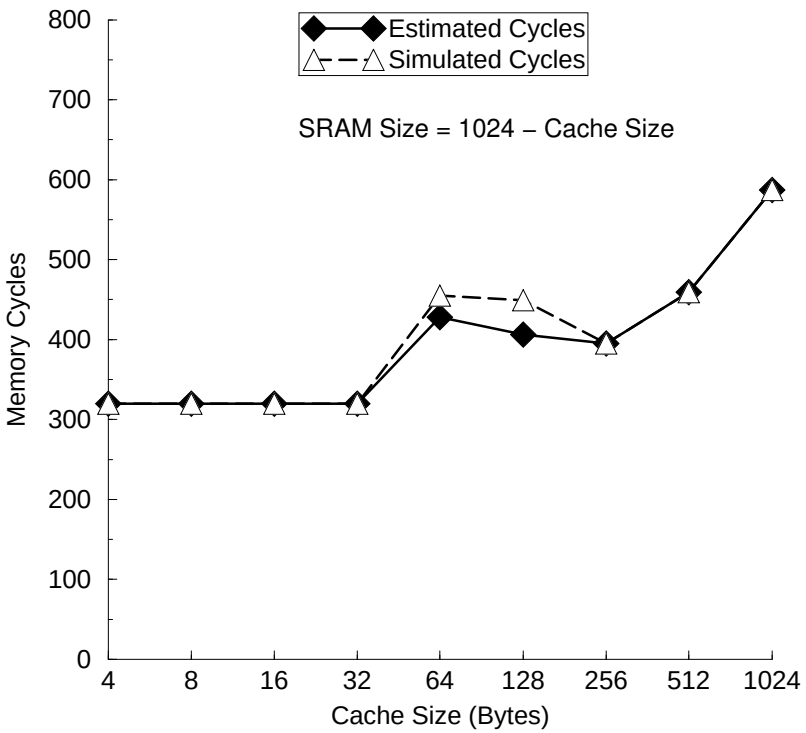


Figure 6.4: Variation of memory performance with different mixes of cache and Scratch-Pad memory, for total on-chip memory of 1 KB (*Dequant* Routine)

```
int curr_block [64];
int pred_block [64];
int val;
:
for (i = 0; i < 64; i++) {
  val = pred_block [i] + curr_block [i];
  if (val > 255)
    val = 255;
  else
    if (val < 0)
      val = 0;
  curr_block [i] = val;
}
```

Figure 6.5: Code for *PLUS* Routine

estimated memory performance is a very good indicator of the actual measured performance via simulation.

6.2.2 PLUS Routine

In the *PLUS* module, the difference-block (stored in array `curr_block`) is added to the reference-block (stored in array `pred_block`), to generate the output image (also stored in array `curr_block`). The code is shown in Figure 6.5.

Since the array accesses in the loop are all *compatible* (Section 5.4.3), we can align them in memory to eliminate cache conflicts altogether. Further, the memory performance prediction now becomes very accurate, since the array access patterns are easily analyzable.

Cache Line Size

Figure 6.6 shows the estimated and simulated memory performance as a function of cache line size, for a fixed cache size of 1 KB. We observe that the prediction is exact. The reason for the performance improvement with increasing cache line size is the same as that for the *Dequant* example: the combined sizes of the two arrays is smaller than 1 KB, hence the only misses that occur are compulsory misses, which decrease with increasing line size. Interestingly, the performance in this case worsens for the maximum cache line size (= 1024 Bytes, i.e., the cache contains only one line). This occurs because the two arrays together occupy only 512 Bytes; hence, when the line size is increased beyond 512 Bytes, unnecessary data is brought into the cache, thus increasing the memory traffic. Thus, the best cache line size for this routine is 512 Bytes.

Cache and Scratch-Pad Memory Ratio

The array access pattern in Figure 6.5 also makes it possible to make accurate prediction for the performance of various configurations with varying ratios of Scratch-Pad memory and data cache. Figure 6.7 shows that the estimated memory performance is accurate for different divisions of 1 KB on-chip memory into Scratch-Pad memory and data cache. In this case, the best performance is obtained as long as there is at least 512 Bytes of Scratch-Pad memory (which is the combined size of the two arrays), i.e., as long as the cache size is $\leq (1024 - 512 = 512)$ Bytes. When the cache is 1024 Bytes, i.e., SRAM Size = 0, then the arrays have

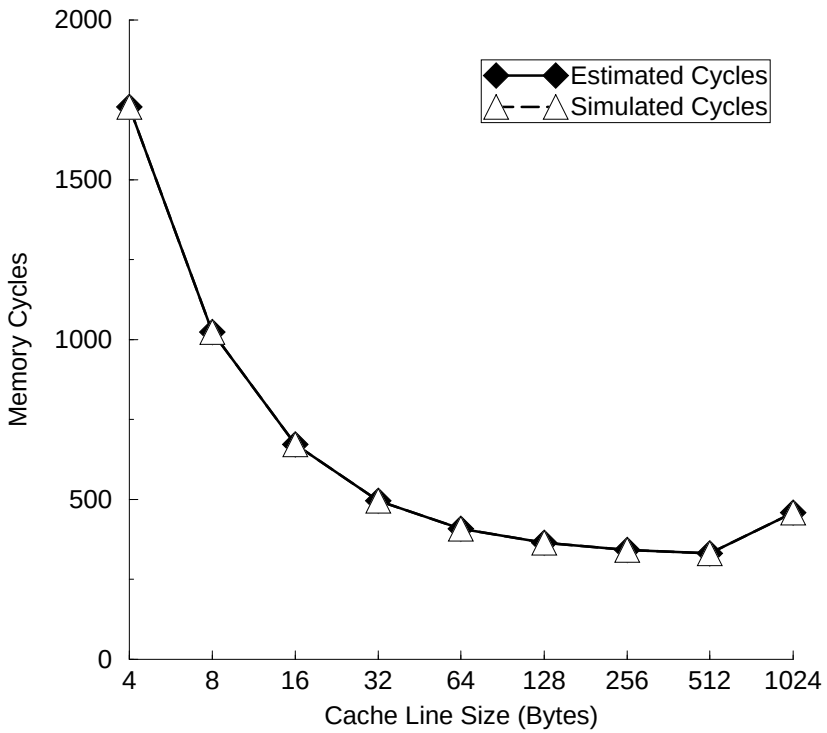


Figure 6.6: Variation of memory performance with line size for fixed cache size of 1 KB (*PLUS* Routine). No Scratch-Pad memory.

to be stored off-chip, and the performance degrades slightly because of compulsory misses.

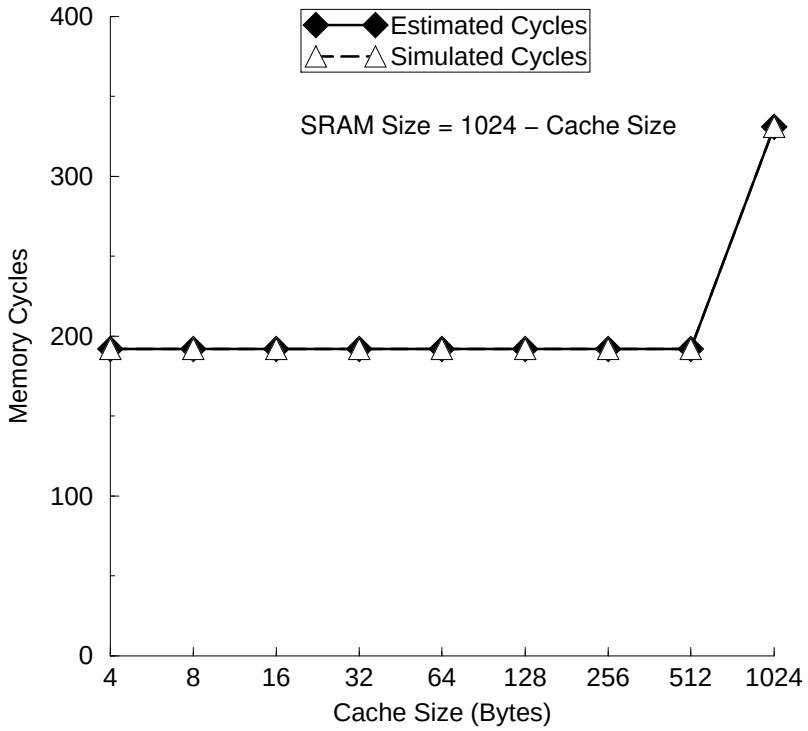


Figure 6.7: Variation of memory performance with different mixes of cache and Scratch-Pad memory, for total on-chip memory of 1 KB (*PLUS* Routine)

6.3 Memory Exploration of MPEG Decoder

The estimated and actual memory performance of routines such as *Dequant* and *PLUS* above, will be different when they are analyzed as a part of a larger application, rather than individually. This is because a global analysis, in which the same array is accessed in different loops with possibly different access patterns, leads to a different memory assignment of arrays. This global analysis is necessary,

since in its absence, different routines in the application will have different optimal memory configuration requirements. In this section, we study the memory performance of the entire MPEG decoder application, of which *Dequant* and *PLUS* are constituent routines. We conclude the section with a brief comparison of our approach with a simulation-based study of the MPEG-2 decoder[SL97].

We start with the SpecCharts[GVNG94] description of the MPEG decoder described in [Tho95], where each routine in Table 6.1 is modeled as a *behavior*. In the current implementation, all the routines in the MPEG decoder are flattened out for our analysis; thus, the application is treated as a series of loop nests. We assume that all array sizes, loop bounds, and number of calls to functions are statically known; this is true for the MPEG algorithm. Since our exploration is limited to the memory performance alone, we ignore all other computation in the loops, and focus only on the memory accesses; each array access is interpreted as a memory operation.

Table 6.1 shows some statistical information about the MPEG decoder application. In summary, there are seven routines including *Dequant* and *PLUS*, which include 12 loop nests. Column 1 gives the names of the routines; Column 2 gives the number of loop nests in each routine; Column 3 shows the number of arrays accessed in the routine; and Column 4 gives the total number of memory accesses in the routine. Of the 12 loops, four had incompatible array access patterns (Section 5.4.3), and the remaining 8 had compatible accesses. In some routines, such as *vld_control* and *prediction*, which involved conditional statements, we incorporated a user-provided hint as to which path was more likely to be taken. This information could be obtained by profiling. Another possible solution is to accept

probabilities of each path being taken, and incorporate them into the estimation by weighting the total estimated memory cycles by the probability of the memory access statements being executed.

Name of Routine	Number of Loop Nests	Number of Arrays Accessed	Total No. of Memory Accesses
Vld_control	1	2	100
Dequant	1	4	320
Idct	4	3	2176
Prediction	3	3	512
PLUS	1	2	192
Store	1	2	128
Display	1	1	64

Table 6.1: Characteristics of Routines in the MPEG Decoder

Since most of the scalar variables were used in local/temporary computations, we focussed our analysis on the array variables in the behavior. Table 6.2 gives details of the data involved in the decoder. Column 1 gives the array names; Column 2 gives the sizes of each array; and Column 3 shows the number of different loop nests that involve accesses to the array. Note that five of the arrays are accessed in multiple loops; for each of these arrays, the access patterns in several loops are considered when performing the memory assignment.

We ran our memory exploration algorithm of Section 5.3 on the MPEG decoder application; for each total on-chip memory size, we estimated and simulated the performance for data cache sizes ranging from one word to the total memory size. For each candidate cache, we explored cache line sizes ranging from one word to 1 KB; in general, the cache line size is limited by the page size of the off-chip DRAM. To perform our simulations, we first executed our memory address assignment algorithms for the candidate Scratch-Pad memory and cache parameters,

Array Name	Array Size (Bytes)	Number of Loops
curr_block	256	5
temp_block	256	2
pred_block	256	4
dct_zz	256	1
scan	256	1
intra_quant	256	1
idct_coeff	256	2
storage	12288	3
vld_length	200	1
vld_bits	200	1

Table 6.2: Array Accesses in the MPEG Decoder

and used the resulting array addresses to generate a trace of the data memory addresses. We then fed this trace to the cache/SRAM simulator to obtain the simulated performance result.

Cache Line Size

Figure 6.8 shows the variation of the estimated and actual performance with varying cache line size for a fixed cache size of 1 KB. Although the estimates are now less accurate, the estimation tracks the relative performance for different line sizes very well. The best performance is predicted to occur for a line size of 32 Bytes, which matches the simulation result, as shown in Figure 6.8. The estimation has very high *fidelity*, which is defined as the percentage of correctly predicted comparisons between points in the design space [GVNG94]. Fidelity is a very useful measure in design space exploration; an estimation with good fidelity can be effectively used to compare different different design points and arrive at the best alternative. Note that in Figure 6.8, the slopes of the line connecting two successive points in the two curves, always has the same sign, indicating that a correct decision

of the best cache line size can be made by comparing the estimations of the relative merit of two neighbouring points.

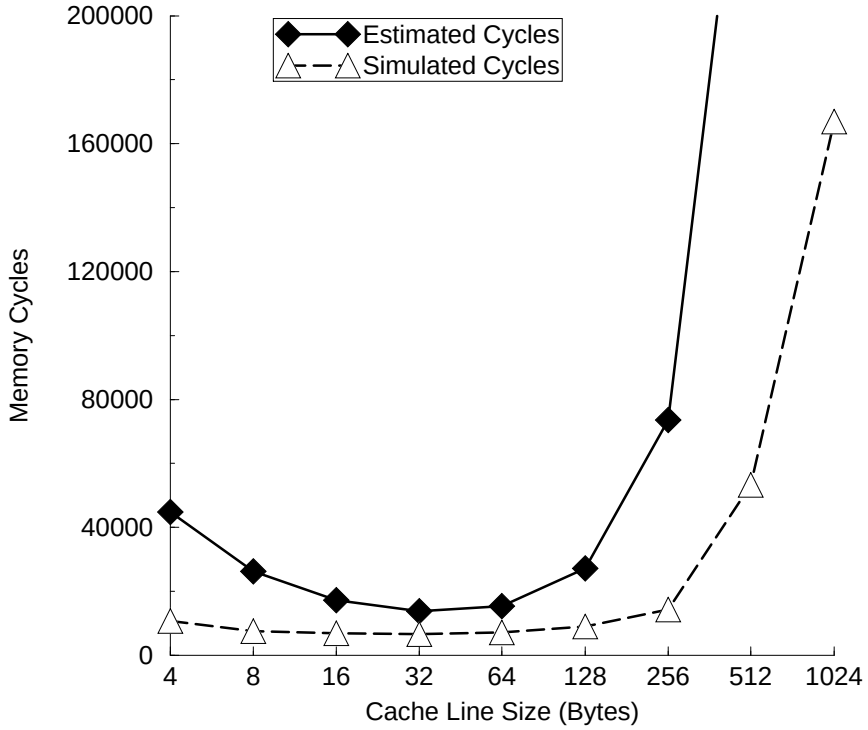


Figure 6.8: Variation of memory performance with line size for fixed cache size of 1 KB (*MPEG Decoder*). No Scratch-Pad memory.

In general, the estimated performance for the decoder is worse than the actual performance. This is because of the cache reuse occurring across different loops, which is not addressed in our estimation. For example, if an array accessed in one loop nest, is also accessed in the next loop, compulsory misses can be avoided if the array elements are still present in the cache when the second loop commences execution. However, in our estimation (Section 5.4.4), we assume that compulsory misses occur afresh in each loop. This topic requires further investigation: a

heuristic that makes an intelligent guess regarding compulsory misses will help enhance the accuracy of the prediction.

Cache and Scratch-Pad Memory Ratio

Figure 6.9 shows the variation of the estimated and measured memory performance for different ratios of cache and Scratch-Pad memory, for a fixed total on-chip memory size of 1 KB. The estimate follows the shape of the simulation curve (i.e., has good fidelity), except for the two cache sizes: 64 Bytes and 512 Bytes. In the estimation, 64 Bytes is predicted to perform better than 32 Bytes, whereas in the simulation, 32 Bytes is better. The estimate predicts the best cache sizes as 256 Bytes, whereas simulation results indicate 512 Bytes as the best cache size (i.e., Cache : Scratch-Pad memory ratio of 1 : 1). The relatively poor estimation for 512 Bytes cache size is due to the issue of compulsory misses discussed earlier. Data reuse across different loops becomes more significant for larger caches, leading to wider variations between estimated and simulated performance.

Thus, as mentioned in Section 5.5, our recommendation is that the analytical exploration phase be augmented by a final simulation of the predicted best configuration, as well as a few neighbouring points on the estimation curve, especially points for which the estimated performance is very close, e.g., cache sizes 256 and 512 Bytes in Figure 6.9. Note that this is still significantly faster than an exhaustive simulation, because the latter case entails not only a full simulation of all cache sizes (and all line sizes for each cache size), but also an exponential number of combinations of array variables assigned to the Scratch-Pad memory for each configuration.

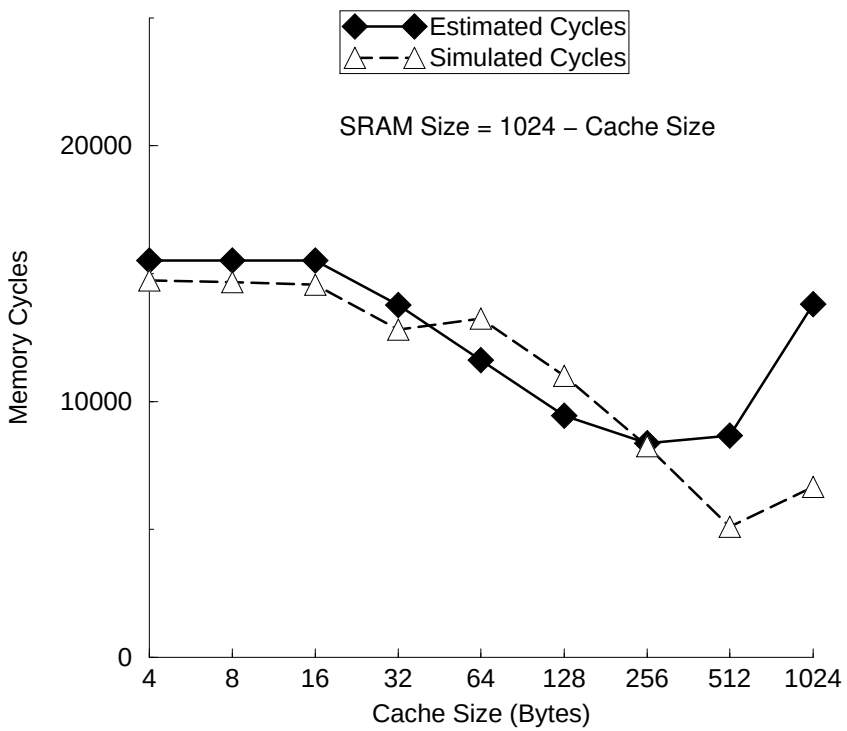


Figure 6.9: Variation of memory performance with different mixes of cache and Scratch-Pad memory, for total on-chip memory of 1 KB (*MPEG Decoder*)

Figure 6.9 clearly demonstrates the utility of the Scratch-Pad memory in the embedded system architecture. The simulation confirms that the memory configuration corresponding to cache size = 1024 Bytes (i.e., no Scratch-Pad memory: the points on the extreme right in Figure 6.9) does not lead to the best performance. Instead, the best performance is obtained in this case when the 1024 Bytes of on-chip data memory is split into equal halves of cache and Scratch-Pad memory so that critical data can be stored in the Scratch-Pad memory.

Total On-chip Memory Size

Figure 6.10 shows the estimated and simulated memory performance for the decoder, for various total on-chip memory sizes varying from 4 Bytes to 64 KB. Each point on the curve corresponds to the best division of the corresponding total size into cache and Scratch-Pad memory.

For very small on-chip memory sizes, the performance is expectedly poor because of consistently occurring cache conflicts. Similarly, for large memory sizes, the performance is the best because all the data fits on-chip. Note that if all the data fits on-chip, then the best configuration is to store all of it in the Scratch-Pad memory, and not use a cache at all. For the memory sizes in between, the estimated performance follows the simulation results very closely. Thus, the estimation curve provides very good feedback to a designer, and helps him perform a sound cost/performance trade-off. Once an appropriate on-chip memory size that is predicted to meet the performance requirements is selected, the final simulations can be performed and the exact configuration determined.

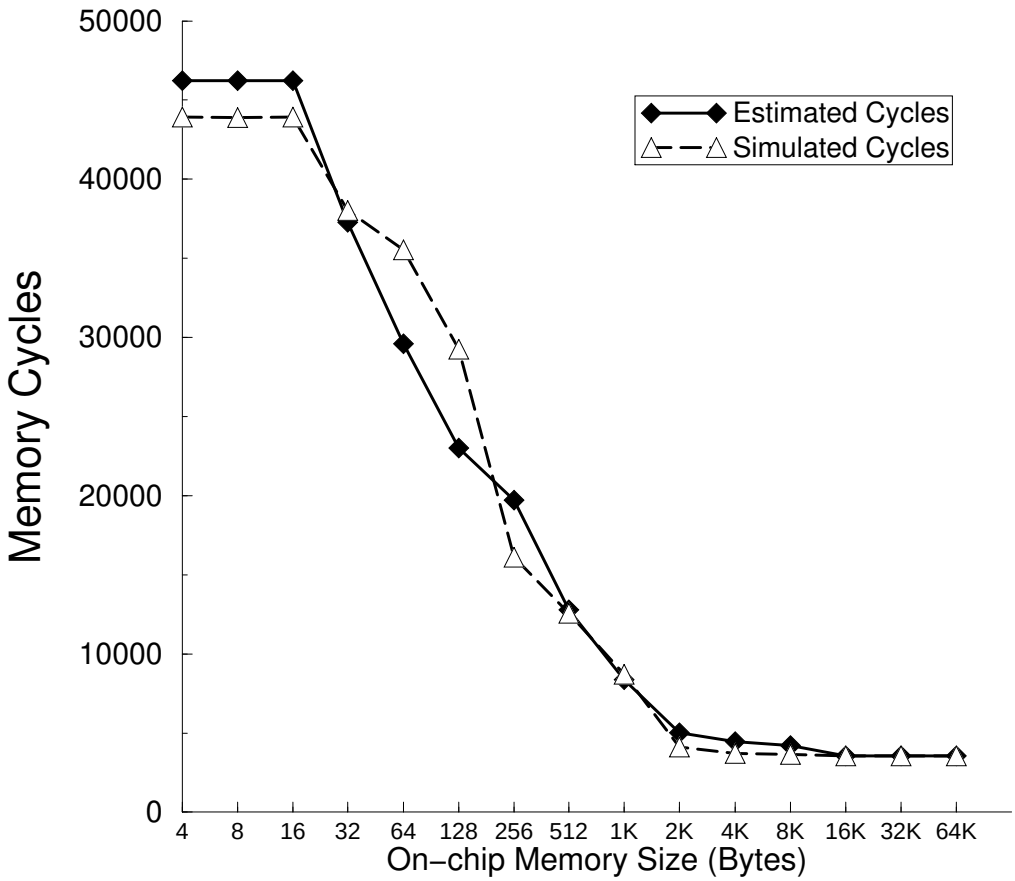


Figure 6.10: Variation of memory performance with total on-chip memory space (*MPEG Decoder*)

The results of our case study are in accordance with another study reported in [SL97] on the cache behavior of the MPEG-2 decoder. The MPEG-2 is an extension of the original MPEG algorithm. They report that the performance can possibly be improved with a *Scratch memory* by selectively storing critical data on-chip. This notion is exactly the same as the Scratch-Pad memory concept we have presented. Our technique and experimental results show that the performance is indeed enhanced by exploiting this architectural innovation. It should be noted that the cache performance for larger caches is different in our study than that in [SL97] because the original SpecCharts model from which we extracted the memory references, used a single picture block instead of the entire picture.

6.4 Summary and Status

We presented a detailed case study of the MPEG decoder to evaluate the efficacy of our analytical memory exploration technique. Our experiments show that our estimation provides a very good approximation to the actual memory performance. Thus, the memory exploration can provide very useful feedback to a system designer in the early phases of system design. The estimation capability could be improved by incorporating the effects of data reuse across multiple loops.

As mentioned earlier, the input to the exploration process is a series of loop structures with memory accesses, and the final output is a graph showing the estimated performance for different data memory sizes. Each array access in a loop is represented by a matrix of numbers representing the coefficients of the loop indices. For example, the index expression in a reference to $a[i + 5]$ in a

singly-nested loop with index i is represented as $\begin{bmatrix} 1 & 5 \end{bmatrix}$, and the expression in $a[i+j][i-j-2]$ in a doubly-nested loop with outer and inner loop indices i and j respectively, is represented as: $\begin{bmatrix} 1 & 1 & 0 \\ 1 & -1 & -2 \end{bmatrix}$. This scheme can be used to represent array index expressions that are linear functions of the loop indices. The exploration requires, for each loop nest, the bound and stride for each loop level, and the coefficient matrices of the arrays accessed in the loop. The matrices representing the array accesses were generated manually for our experiments.

Currently, the generation of the memory trace for the purpose of simulation is a semi-automatic process. We first run our exploration algorithm to obtain the memory address assignments for the data variables. We then replace each array access in the code by a function call that generates the appropriate memory address to be referenced. Each generated address is fed to our cache/SRAM simulator, and the memory performance statistics are generated by executing this modified code.

Chapter 7

Conclusions

7.1 Summary of Contributions

The trend towards increasing design complexity in recent times is accompanied by a corresponding increase in the complexity of the building blocks that constitute the design library available to synthesis tools. Modern design libraries frequently consist of pre-designed mega-cells such as processor cores and memories. Several new architectural and optimization issues arise in the process of incorporation of these complex components in application-specific embedded systems. In this thesis, we studied techniques for incorporating memory architectures in embedded systems.

We first showed how the organization of modern DRAMs can be exploited to yield significant performance improvements during the automatic synthesis of behavioral-level descriptions. We presented models for representation of memory accesses, and transformation techniques for incorporating page mode, and other efficient access features during synthesis. Page mode accesses can also be exploited to effect power optimizations during memory accesses. We presented a strategy for

minimizing memory access related power by suitably mapping behavioral arrays to memory.

Storage of behavioral scalar and array variables in memory has important implications in determining the performance of behaviors that are synthesized to execute on embedded processors. Due to the relatively longer compilation time available for embedded processor-based systems, it is possible to perform a more detailed analysis of the behavior. We outlined techniques for clustering scalar variables and assigning addresses to scalar and array variables for enhancing the cache performance.

The application-specific nature of embedded systems permit the incorporation of relatively unconventional components, compared to the general purpose processing domain. One such component is Scratch-Pad memory – on-chip memory to which the assignment of data is compiler controlled. We showed how the array sizes, access frequency, and other characteristics can be factored into a data partitioning strategy between on-chip Scratch-Pad SRAM and off-chip DRAM.

An embedded processor-based system allows for customization of on-chip memory configuration according to the requirements of the specific application. Since individual applications have widely varying memory characteristics, it is essential to generate an efficient memory organization for a given application. We presented a fast memory exploration strategy that combines the memory assignment and SRAM/DRAM data partitioning techniques with a cache performance estimation scheme to evaluate the expected performance of an application for different on-chip memory architectures. The speed advantage of such an analytical approach makes it an attractive tool for early system design exploration.

7.2 Future Directions

The work presented in this thesis can be extended in different directions in the future.

Memory optimization possibilities for DRAMs will continue to evolve in future generations of memory technology. The recently released Synchronous DRAM (SDRAM), while encompassing all the efficient modes described in Chapter 2, has additional architectural innovations, such as dual-bank storage, which opens up the possibility for more performance and power optimization techniques. Similarly, the proposed RAMBUS DRAM, which is likely to become a popular memory device in the future, has a more complex protocol that would require an advanced modeling strategy. The scenario of an off-chip DRAM augmented by on-chip cache forms an interesting enhancement to our proposed memory access model.

In the memory exploration phase, we have currently considered the cache line size and cache size parameters. However, other cache features such as write policy, associativity, and replacement policy could also be incorporated to refine the exploration. As the example in Section 5.2 illustrates, the best cache size for an application is different, depending on whether a write-through or write-back policy is adopted. Further, we have assumed in this work that the access delay for the data cache and Scratch-Pad memory does not change with the size. This assumption is valid as long as the delay is less than the cycle time of the processor core. Although this handles a large number of memory configurations, for larger memory sizes that result in long delays, the on-chip memory access time would

lie on the critical path, and the performance estimate should incorporate a more accurate cost model.

If the code executed in an embedded processor-based system is small enough, it can be stored in on-chip ROM. However, if the code is large, an instruction cache has to be used. In this case, interesting new issues arise with respect to the sizing of the instruction cache. The memory exploration strategy should, ideally, also consider the impact of instruction cache sizing on the code performance. For example, memory size trade-off studies for a given total amount of on-chip memory should now include the relative sizes of on-chip data and instruction memory in addition to Scratch-Pad memory to find an ideal balance that maximizes performance.

Bibliography

[ACG+95] K. Au, P. Chang, C. Giles, E. Hadad, R. Huang, D. Jones, E. Kristenson, M. Kwong, D. Lin, M. Murzello, B. Singh, S. Wan B. Wang, and F. Worrel. "Minirisc(tm) cw4001 – a small, low-power mips cpu core,". In *Proceedings, Custom Integrated Circuits Conference*, May 1995.

[ASU93] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1993.

[Aus96] T. M. Austin. *Hardware and Software Mechanisms for Reducing Load Latency*. PhD thesis, University of Wisconsin - Madison, April 1996.

[Bak88] H.B. Bakoglu. *Circuits, Interconnections, and Packaging for VLSI*. Addison-Wesley, Reading, MA, 1988.

[BCM94] F. Balasa, F. Catthoor, and H. D. Man. "Dataflow-driven memory allocation for multi-dimensional signal processing systems,". In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*, November 1994.

[BCM95] F. Balasa, F. Catthoor, and H. D. Man. "Background memory area estimation for multidimensional signal processing systems,". *IEEE Transactions on VLSI Systems*, 3(2):157–172, June 1995.

[BG95] S. Bakshi and D. Gajski. "A memory selection algorithm for high-performance pipelines,". In *Proceedings of the European Design Automation Conference*, 1995.

[BMMZ95] P. Baglietto, M. Maresca, M. Migliardi, and N. Zingirian. "Image processing on high performance RISC systems,". Technical Report TR SM-IMP/DIST/08, University of Genoa, December 1995.

[BRSV87] R. K. Brayton, R. Rudell, and A. Sangiovanni-Vincentelli. "Mis: A multiple level logic optimization system,". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-6(6):1062–1082, November 1987.

[CKP91] D. Callahan, K. Kennedy, and A. Porterfield. "Software prefetching,". In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, Santa Clara, CA, April 1991.

[CL91] E. G. Coffman, Jr. and G. S. Lueker. *Probabilistic Analysis of Packing and Partitioning Algorithms.* Wiley-Interscience, New York, 1991.

[CL95] M. Cierniak and W. Li. "Unifying data and control transofrmations for distributed shared memory machines,". In *ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 205–217, La Jolla, CA, June 1995.

[CM95] S. Coleman and K. S. McKinley. "Tile size selection using cache organization and data layout,". In *ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 279–289, La Jolla, CA, June 1995.

[COB95] P. Chou, R. Ortega, and G. Borriello. "Interface co-synthesis techniques for embedded systems,". In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*, pages 280–287, November 1995.

[DFBS94] S. L. Graham D. F. Bacon and Oliver J. Sharp. "Compiler transformations for high-performance computing,". *ACM Computing Surveys*, 26(4), December 1994.

[DJKD94] A. Nicolau D. J. Kolson and N. D. Dutt. "Integrating program transformations in the memory-based synthesis of image and video algorithms,". In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*, pages 27–30, San Jose, CA, November 1994.

[Ess93] K. Esseghir. "Improving data locality for caches,". Master's thesis, Department of Computer Science, Rice University, 1993.

[Fly95] M. J. Flynn. *Computer architecture - pipelined and parallel processor design.* Jones and Bartlett Publishers, 1995.

[Gal91] D. Le Gall. "MPEG: A video compression standard for multimedia applications,". *Communications of the ACM*, 34(4):46–58, April 1991.

[GDLW92] D. Gajski, N. Dutt, S. Lin, and A. Wu. *High Level Synthesis: Introduction to Chip and System Design.* Kluwer Academic Publishers, 1992.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractibility – A Guide to the Theory of NP-Completeness.* W.H. Freeman, 1979.

[GVNG94] D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and Design of Embedded Systems.* Prentice Hall, 1994.

[GW87] R. C. Gonzalez and P. Wintz. *Digital Image Processing.* Addison-Wesley, 1987.

[Hil88] M. D. Hill. "The case for direct-mapped caches,". *IEEE Computer*, 21(12):25–41, December 1988.

[IBM96] IBM Corporation. *http://www.chips.ibm.com/products/memory/50H7629/*, 1996.

[JD97] P. K. Jha and N. Dutt. "Library mapping for memories,". In *European Design and Test Conference*, pages 288–292, March 1997.

[Jou90] N. P. Jouppi. "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers,". In *International Symposium on Computer Architecture*, pages 364–373, Seattle, WA, May 1990.

[Jou93] N. P. Jouppi. "Cache write policies and performance,". In *International Symposium on Computer Architecture*, pages 191–201, San Diego, CA, May 1993.

[KL93] T. Kim and C. L. Liu. "Utilization of multiport memories in data path synthesis,". In *Design Automation Conference*, pages 298–302, June 1993.

[KLPMS97] D. Kirovski, C. Lee, M. Potkonjak, and W. Mangione-Smith. "Application-driven synthesis of core-based systems,". In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*, pages 104–107, San Jose, CA, November 1997.

[Koh78] Z. Kohavi. *Switching and Finite Automata Theory.* McGraw-Hill, New York, 1978.

[KR94] D. Karchmer and J. Rose. "Definition and solution of the memory packing problem for field-programmable systems,". In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*, pages 20–26, November 1994.

[LKMM95] T. Ly, D. Knapp, R. Miller, and D. MacMillen. "Scheduling using behavioral templates,". In *ACM/IEEE Design Automation Conference*, pages 599–604, June 1995.

[LMP94] C. Liem, T. May, and P. Paulin. "Instruction-set matching and selection for dsp and asip code generation,". In *Proceedings, European Design and Test Conference*, pages 31–37, Paris, France, February 1994.

[LRW91] M. Lam, E. Rothberg, and M. E. Wolf. "The cache performance and optimizations of blocked algorithms,". In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, April 1991.

[LSI92] LSI Logic Corporation, Milpitas. *CW33000 MIPS Embedded Processor User's Manual*, 1992.

[LvM+93] P. E. R. Lippens, J. L. van Meerbergen, et al. "Allocation of multiport memories for hierarchical data streams,". In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 728–735, November 1993.

[Mar97] B. Margolin. "Embedded systems to benefit from advances in DRAM technology,". *Computer Design*, pages 76–86, March 1997.

[MG95] P. Marwedel and G. Goosens, editors. *Code Generation for Embedded Processors*. Kluwer Academic Publishers, Norwell, MA, 1995.

[PD95a] P. R. Panda and N. D. Dutt. "1995 High level synthesis design repository,". In *International Symposium on System Synthesis*, pages 170–174, Cannes, France, September 1995.

[PD95b] P. R. Panda and N. D. Dutt. "Low power memory mapping through reducing address bus activity,". Technical Report 95-32, University of California, Irvine, November 1995.

[PD96] P. R. Panda and N. D. Dutt. "Reducing address bus transitions for low power memory mapping,". In *Proceedings, European Design and Test Conference*, pages 63–67, Paris, France, March 1996.

[PD97] P. R. Panda and N. D. Dutt. "Behavioral array mapping into multiport memories targeting low power,". In *Proceedings of the 10th International Conference on VLSI Design*, pages 268–272, Hyderabad, India, January 1997.

[PDN96] P. R. Panda, N. D. Dutt, and A. Nicolau. "SRAM vs. data cache: The memory data partitioning problem in embedded systems,". Technical Report 96-42, University of California, Irvine, September 1996.

[PDN97a] P. R. Panda, N. D. Dutt, and A. Nicolau. "Architectural exploration and optimization of local memory in embedded systems,". Technical Report ICS-TR-97-31, University of California, Irvine, June 1997.

[PDN97b] P. R. Panda, N. D. Dutt, and A. Nicolau. "Data cache sizing for embedded processor applications,". Technical Report ICS-TR-97-30, University of California, Irvine, June 1997.

[PDN97c] P. R. Panda, N. D. Dutt, and A. Nicolau. "Exploiting off-chip memory access modes in high-level synthesis,". In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*, pages 333–340, San Jose, CA, November 1997.

[PDN97d] P. R. Panda, N. D. Dutt, and A. Nicolau. "Exploiting off-chip memory access modes in high-level synthesis,". Technical Report ICS-TR-97-32, University of California, Irvine, June 1997.

[PDN97e] P. R. Panda, N. D. Dutt, and A. Nicolau. "Memory data organization for improved cache performance in embedded processor applications,". *ACM Transactions on Design Automation of Electronic Systems*, 2(4), October 1997.

[PH94] D. A. Patterson and J. L. Hennessy. *Computer Organization & Design – The Hardware/Software Interface*. Morgan Kaufman, San Mateo, CA, 1994.

[PNDN97a] P. R. Panda, H. Nakamura, N. D. Dutt, and A. Nicolau. "A data alignment technique for improving cache performance,". In *Proceedings IEEE International Conference on Computer Design*, pages 587–592, October 1997.

[PNDN97b] P. R. Panda, H. Nakamura, N. D. Dutt, and A. Nicolau. "Improving cache performance through tiling and data alignment,". In G. Bilardi, A. Ferreira, R. Lüling, and J. Rolim, editors, *Solving Irregularly Structured Problems in Parallel*, volume 1253. Springer-Verlag, Heidelberg, Germany, 1997.

[PTVF92] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 1992.

[Raw93] J. Rawat. "Static analysis of cache performance for real-time programming,". Master's thesis, Iowa State University, May 1993.

[RGC94] L. Ramachandran, D. Gajski, and V. Chaiyakul. "An algorithm for array variable clustering,". In *European Design and Test Conference*, February 1994.

[SB95] M. R. Stan and W. P. Burleson. "Bus-invert coding for low power I/O,". *IEEE Transactions on VLSI Systems*, 3(1):49–58, March 1995.

[SJ92] L. Stok and J. A. G. Jess. "Foreground memory management in data path synthesis,". *International Journal of Circuit Theory and Applications*, 20(3):235–255, 1992.

[SL97] P. Soderquist and M. Leeser. "Memory traffic and data cache behavior of an mpeg-2 software decoder,". In *Proceedings IEEE International Conference on Computer Design*, pages 417–422, October 1997.

[ST97] H. Schmit and D. E. Thomas. "Synthesis of application-specific memory designs,". *IEEE Transactions on VLSI Systems*, 5(1), March 1997.

[Sun93] Sun Microsystems Laboratories Inc., Mountain View, CA. *Shade User's Manual*, 1993.

[Tho95] A. Thordarson. "Comparison of manual and automatic behavioral synthesis of mpeg algorithm,". Master's thesis, University of California, Irvine, 1995.

[TS86] C. Tseng and D. P. Siewiorek. "Automated synthesis of datapaths in digital systems,". *IEEE Transactions on Computer Aided Design*, 6(7):379–395, July 1986.

[TY96a] H. Tomiyama and H. Yasuura. "Optimal code placement of embedded software for instruction caches,". In *European Design and Test Conference*, pages 96–101, Paris, France, March 1996.

[TY96b] H. Tomiyama and H. Yasuura. "Size-constrained code placement for cache miss rate reduction,". In *International Symposium on System Synthesis*, pages 96–101, La Jolla, CA, November 1996.

[Wal91] G. K. Wallace. "The JPEG still picture compression standard,". *Communications of the ACM*, 34(4):30–44, April 1991.

[WCF+94] S. Wuytack, F. Catthoor, F. Franssen, L. Nachtergaele, and H.De Man. "Global communication and memory optimizing transformations for low power systems,". In *IEEE International Workshop on Low Power Design*, pages 203–208, April 1994.

[Wil97] R. Wilson. "Graphics IC vendors take a shot at embedded DRAM,". *Electronic Engineering Times*, (938):41,57, January 27 1997.

[WL91] M. E. Wolf and M. Lam. "A data locality optimizing algorithm,". In *Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Canada, June 1991.

[Wol89] M. J. Wolfe. "More iteration space tiling,". In *Proceedings of Supercomputing*, pages 655–664, Reno, NV, November 1989.

[YJH+95] Y. Yamada, T. L. Johnson, G. Haab, J. C. Gyllenhaal, and W. W. Hwu. "Reducing cache misses in numerical applications using data relocation and prefetching,". Technical Report CRHC-95-04, University of Illinois, Urbana, 1995.