



TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY
Julkaisu 641 • Publication 641

Eero Aho

Design and Implementation of Parallel Memory Architectures



Tampereen teknillinen yliopisto. Julkaisu 641
Tampere University of Technology. Publication 641

Eero Aho

Design and Implementation of Parallel Memory Architectures

Thesis for the degree of Doctor of Technology to be presented with due permission for public examination and criticism in Tietotalo Building, Auditorium TB109, at Tampere University of Technology, on the 15th of December 2006, at 12 noon.

Tampereen teknillinen yliopisto - Tampere University of Technology
Tampere 2006

ISBN 952-15-1689-5 (printed)
ISBN 952-15-1726-3 (PDF)
ISSN 1459-2045

Abstract

Parallel processing is continually concerned about how to supply all the processing nodes with data. Many of the applications favor special data patterns that could be accessed in parallel. This is utilized in parallel memories, where the idea is to increase memory bandwidth with several memory modules working in parallel and feed the processor with only necessary data.

Traditional parallel memories are application specific and support only fixed data access requirements. In this Thesis, memory flexibility is increased to give support for several algorithms by adding run-time configurability to parallel memories. Multitude of data access templates and module assignment functions can be used within a single hardware implementation, which has not been possible in prior embedded parallel memory systems. The design reusability of the memories is also improved since the same memory system is applicable in several separate implementations.

Three novel parallel memory architectures are presented in this Thesis: one traditional application specific type and two with run-time configurability. The results show that run-time configurability can be included in parallel memories with a reasonable cost. As a case study with four memory modules, the normalized complexity of the proposed configurable parallel memories is 63–80% less than the conventional type of parallel memory. Moreover, in configurable parallel memories, the complexity increase in permutation networks is expressed to become the most critical when increasing the memory module count. According to evaluations, up to 79% of the total parallel memory gate count is consumed by the permutation networks excluding memory cells.

The results of this Thesis can be used for designing a flexible memory system for real-time multimedia applications that demand high data throughput and data parallel computation.

Preface

This research work has been carried out during the years 2001–2006 in the Institute of Digital and Computer Systems at Tampere University of Technology, Tampere, Finland.

I would like to express my gratitude to my supervisor Professor *Timo Hämäläinen* for his support and guidance during the research. Grateful acknowledgements go also to the reviewers of my Thesis, Professor *Michael Gössel* and Docent *Juha Plosila*, for their constructive comments. In addition, I would like to thank *Kimmo Kuusilinna*, Dr. Tech., for his guidance and advice during the work as well as Docent *Jukka Saarinen* for the encouragement during the start of my academic career.

Special thanks to my closest colleague *Jarno Vanne*, M.Sc., who helped me several ways during the research. Warm thanks to all my colleagues in DACI research group for the relaxed atmosphere. Especially, *Ari Kulmala*, M.Sc., *Olli Lehtoranta*, M.Sc., *Timo Alho*, M.Sc., *Erno Salminen*, M.Sc., *Tero Kangas*, Dr. Tech., *Jouni Riihimäki*, M.Sc., *Panu Hämäläinen*, Dr. Tech., and *Mauri Kuorilehto*, M.Sc., deserve many thanks for cooperation and inspiring discussions. In addition, I would like to thank *Jarno Tanskanen*, Dr. Tech., *Tuomas Järvinen*, Dr. Tech., *Jari Nikara*, Dr. Tech., *Tero Sihvo*, M.Sc., *Perttu Salmela*, M.Sc., and *Jari Heikkinen*, M.Sc., for fruitful conversations related to parallel memories and other topics.

This Thesis was financially supported by the Graduate School in Electronics, Telecommunications, and Automation (GETA), Academy of Finland, Nokia Foundation, Ulla Tuominen Foundation, Heikki and Hilma Honkanen Foundation, and the Foundation of Advancement of Technology (TES), which are gratefully acknowledged.

My warmest thanks go to my wife *Marika* for her love, support, and understanding. I dedicate this Thesis to my daughter *Eerika* and son *Petrus* for cheering me up after work days.

Tampere, November 2006

Eero Aho

Table of Contents

ABSTRACT	I
PREFACE.....	III
TABLE OF CONTENTS	V
LIST OF PUBLICATIONS	IX
LIST OF FIGURES.....	XI
LIST OF TABLES.....	XIII
ABBREVIATIONS.....	XV
SYMBOLS	XIX
1. INTRODUCTION	1
1.1 OBJECTIVES AND SCOPE OF RESEARCH	2
1.2 SUMMARY OF PUBLICATIONS.....	3
1.2.1 <i>Author's contribution to published work</i>	4
1.3 OUTLINE OF THESIS.....	5
2. MOTIVATION.....	7
2.1 COMMERCIAL PROCESSORS WITH SIMD EXTENSION.....	7
2.1.1 <i>Superscalar processors</i>	8
2.1.2 <i>VLIW processors</i>	8
2.1.3 <i>Cell multiprocessor</i>	9
2.1.4 <i>Summary</i>	9
2.2 DATA ACCESS IMPROVEMENTS IN MULTIMEDIA APPLICATIONS	10
2.2.1 <i>Contemporary architectures</i>	10
2.2.2 <i>Summary</i>	12
3. PARALLEL MEMORIES.....	13
3.1 MEMORY SYSTEM CLASSIFICATION	13

Table of Contents

3.2	PARALLEL MEMORY PRINCIPLES.....	14
3.3	ACCESS FORMATS FOR A SELECTION OF APPLICATIONS.....	16
3.3.1	<i>Image processing</i>	17
3.3.2	<i>Video processing</i>	17
3.3.3	<i>Vector processing</i>	18
3.3.4	<i>Numerical analysis</i>	18
3.3.5	<i>Pattern analysis</i>	18
3.4	MODULE ASSIGNMENT FUNCTIONS	18
3.4.1	<i>Linear functions</i>	19
3.4.2	<i>XOR-schemes</i>	20
3.4.3	<i>Periodic and multiperiodic schemes</i>	20
3.4.4	<i>Relationship between functions</i>	20
3.5	RELATED WORK	21
3.5.1	<i>Parallel memory implementations</i>	22
3.5.2	<i>Parallel memory benefits and drawbacks</i>	23
4.	CONFIGURABLE PARALLEL MEMORY ARCHITECTURE	25
4.1	ARCHITECTURE	25
4.2	PAGE TABLE.....	26
4.2.1	<i>Row address</i>	26
4.2.2	<i>Length of coordinates</i>	26
4.2.3	<i>Access format function</i>	27
4.2.4	<i>Module assignment function</i>	28
4.2.5	<i>Address function</i>	29
4.2.6	<i>Page Table usage</i>	30
4.3	ADDRESS COMPUTATION.....	30
5.	PARALLEL MEMORY ARCHITECTURE FOR ARBITRARY STRIDE ACCESSES	33
5.1	APPLICATION EXAMPLE: IMAGE SCALING.....	33
5.1.1	<i>Image scaling principles</i>	34
5.1.2	<i>Scaling evenly divisible images</i>	34
5.2	BLOCK LEVEL PARALLEL ARCHITECTURE FOR IMAGE SCALING	36
5.3	PARALLEL MEMORY ARCHITECTURE	37
5.3.1	<i>Functionality</i>	37
5.3.2	<i>Address Computation</i>	38
6.	PARALLEL MEMORY ARCHITECTURE FOR IMAGE DOWNSCALER.....	41
6.1	DOWNSCALING WITH BILINEAR INTERPOLATION	41
6.2	DOWNSCALER ARCHITECTURE	42

Table of Contents

6.3	PARALLEL MEMORY ARCHITECTURE	43
6.3.1	<i>Functionality</i>	44
6.3.2	<i>Write Address Computation and Data Permutation</i>	45
6.3.3	<i>Read Address Computation and Data Permutation</i>	45
7.	ANALYSIS	47
7.1	PARALLEL MEMORY FEATURE ANALYSIS	47
7.2	PARALLEL MEMORY COMPLEXITY ANALYSIS	48
8.	CONCLUSIONS	53
8.1	MAIN RESULTS	53
	REFERENCES	55
	PUBLICATIONS	67

List of Publications

This thesis consists of an introductory part and a part containing the following publications. In the text, these publications are referred to as [P1], ..., [P10].

- [P1] E. Aho, J. Vanne, K. Kuusilinna, T. Hämäläinen, and J. Saarinen, “Configurable Address Computation in a Parallel Memory Architecture,” in *Advances in Signal Processing and Computer Technologies*, G. Antoniou, N. Mastorakis, and O. Panfilov, ed., World Scientific and Engineering Society Press, Athens, Greece, pp. 390–395, 2001.
- [P2] E. Aho, J. Vanne, K. Kuusilinna, and T. Hämäläinen, “Diamond Scheme Implementations in Configurable Parallel Memory,” in *Proceedings of the IEEE International Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, Brno, Czech Republic, pp. 211–218, April 2002.
- [P3] E. Aho, J. Vanne, K. Kuusilinna, and T. Hämäläinen, “Access Format Implementations in Configurable Parallel Memory,” in *Proceedings of the International Conference on Computer and Information Science (ICIS)*, Seoul, Korea, pp. 59–64, August 2002.
- [P4] E. Aho, J. Vanne, K. Kuusilinna, and T. Hämäläinen, “XOR-scheme Implementations in Configurable Parallel Memory,” in *System-on-Chip for Real-Time Applications*, W. Badawy and G. A. Jullien, ed., Kluwer Academic Publishers, Boston, MA, USA, pp. 249–261, 2003.
- [P5] E. Aho, J. Vanne, K. Kuusilinna, and T. D. Hämäläinen, “Address Computation in Configurable Parallel Memory Architecture,” *IEICE Transactions on Information and Systems*, vol. E87-D, no. 7, pp. 1674–1681, July 2004.
- [P6] E. Aho, J. Vanne, K. Kuusilinna, and T. D. Hämäläinen, “Comments on “Winscale: An Image-Scaling Algorithm Using an Area Pixel Model”,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 15, no. 3, pp. 454–455, March 2005.
- [P7] E. Aho, J. Vanne, T. D. Hämäläinen, and K. Kuusilinna, “Block-Level Parallel Processing for Scaling Evenly Divisible Images,” *IEEE Transactions on Circuits and Systems I*, vol. 52, no. 12, pp. 2717–2725, December 2005.
- [P8] E. Aho, J. Vanne, and T. D. Hämäläinen, “Parallel Memory Architecture for Arbitrary Stride Accesses,” in *Proceedings of the IEEE Workshop on Design and*

Diagnostics of Electronic Circuits and Systems (DDECS), Prague, Czech Republic, pp. 65–70, April 2006.

- [P9] E. Aho, J. Vanne, and T. D. Hämäläinen, “Parallel Memory Implementation for Arbitrary Stride Accesses,” in *Proceedings of the Embedded Computer Systems: Architectures, MOdeling, and Simulation Conference (IC-SAMOS)*, Samos, Greece, pp. 1–6, July 2006.
- [P10] E. Aho, J. Vanne, T. D. Hämäläinen, and K. Kuusilinna, “Configurable Implementation of Parallel Memory Based Real-Time Video Downscaler,” *Microprocessors and Microsystems*, accepted for publication.

List of Figures

FIGURE 2.1. SUBWORD SIMD ADDITION	8
FIGURE 3.1. MEMORY SYSTEM CLASSIFICATION: A) INTERLEAVED MEMORY, B) PARALLEL MEMORY.....	14
FIGURE 3.2. A GENERALIZED BLOCK DIAGRAM OF A PARALLEL MEMORY ARCHITECTURE.	14
FIGURE 3.3. TWO EXAMPLES OF A PARALLEL MEMORY ACCESS.....	15
FIGURE 3.4. MAPPING THE ADDRESSES OF THE EXAMPLE IN FIGURE 3.3 TO MEMORY MODULES.....	15
FIGURE 3.5. EXAMPLE OF A 1-D PARALLEL MEMORY ACCESS.	16
FIGURE 3.6. ACCESS FORMAT EXAMPLES.	17
FIGURE 3.7. STRAIGHT ACCESS FORMATS WITH A PRIME NUMBER OF MEMORY MODULES ($N = 5$).....	19
FIGURE 3.8. RELATIONSHIPS BETWEEN MODULE ASSIGNMENT FUNCTIONS [38].....	21
FIGURE 4.1. CONFIGURABLE PARALLEL MEMORY ARCHITECTURE.	26
FIGURE 4.2. EXAMPLES OF THE IMPLEMENTED ACCESS FORMATS.....	27
FIGURE 4.3. ADDRESS COMPUTATION UNIT.....	31
FIGURE 5.1. ROW-COLUMN DECOMPOSITION OF 2-D INTERPOLATION.....	35
FIGURE 5.2. EVENLY DIVISIBLE IMAGES SCALED DOWN.....	36
FIGURE 5.3. BLOCK-LEVEL PARALLEL ARCHITECTURE FOR IMAGE SCALING.	36
FIGURE 5.4. PARALLEL MEMORY ARCHITECTURE FOR ARBITRARY STRIDE ACCESSES.....	37
FIGURE 5.5. ADDRESS COMPUTATION UNIT.....	39
FIGURE 6.1. SCALING WITH BILINEAR INTERPOLATION.....	42
FIGURE 6.2. DOWNSCALER ARCHITECTURE.	43
FIGURE 6.3. PARALLEL MEMORY UNIT.....	43
FIGURE 6.4. USED MODULE ASSIGNMENT AND ADDRESS FUNCTIONS.....	44
FIGURE 6.5. READ ADDRESS COMPUTATION UNIT.	45
FIGURE 6.6. READ DATA PERMUTATION UNIT.	46

List of Tables

TABLE 4.1. PAGE TABLE STRUCTURE	26
TABLE 7.1. PARALLEL MEMORY COMPLEXITY COMPARISON IN GATE COUNT, $N = 4$	49
TABLE 7.2. NORMALIZED PARALLEL MEMORY COMPLEXITY, $N = 4$	50
TABLE 7.3. STRIDE PMA AND CPMA COMPLEXITY IN GATE COUNT	51
TABLE 7.4. THE COMPLEXITIES OF THE THREE PERMUTATION NETWORKS IN STRIDE PMA AND CPMA (% OF THE TOTAL GATE COUNT).	51

Abbreviations

16VGA	16 x Video Graphics Array, 2560 x 1920 pixels
1-D	One-dimensional
2-D	Two-dimensional
3-D	Three-dimensional
ALU	Arithmetic Logic Unit
AND	Logical AND operation
BDTI	Berkeley Design Technology, Inc.
BSP	Burroughs Scientific Processor
CMOS	Complementary Metal Oxide Semiconductor
CPMA	Configurable Parallel Memory Architecture
CSI	Complex Streamed Instruction, multimedia ISA extension
DCT	Discrete Cosine Transformation
DLP	Data-Level Parallelism
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processor
FFT	Fast Fourier Transform
FIFO	First In First Out
FPGA	Field Programmable Gate Array
gcd	Greatest common divisor
GPP	General Purpose Processors
HDL	Hardware Description Language
HDTV	High Definition TeleVision, e.g. 1920 x 1080 pixels
HiPAR-DSP	Highly Parallel DSP
IC	Integrated Circuit
IDCT	Inverse Discrete Cosine Transformation

IEEE	The Institute of Electrical and Electronics Engineers
IP	Intellectual Property
IRAM	Intelligent RAM
ISA	Instruction Set Architecture
LSB	Least Significant Bit
LUT	Look Up Table
MAC	Multiply Accumulate
MAX	Multimedia Acceleration eXtension, HP's multimedia ISA extension
MMX	Matrix Math eXtensions, Intel's multimedia ISA extension
mod	Modulo
MOM	Matrix Oriented Multimedia, multimedia ISA extension
MPEG	Motion Pictures Experts Group
MSB	Most Significant Bit
MVI	Motion Video Instructions, DEC's multimedia ISA extension
NP	Nondeterministic Polynomial
OBMC	Overlapped Block Motion Compensation
PMA	Parallel Memory Architecture
PPE	Power Processor Element
RAM	Random Access Memory
ROM	Read Only Memory
SAD	Sum of Absolute Differences
SIMD	Single Instruction Multiple Data
SMC	Stream Memory Controller
SoC	System-on-Chip
SOI	Silicon On Insulator
SOR	Successive Over Relaxation
SPE	Synergistic Processor Element
SRAM	Static Random Access Memory
SSE	Streaming SIMD Extension, Intel's multimedia ISA extension
TUT	Tampere University of Technology
VHDL	VHSIC Hardware Description Language

VHSIC	Very High Speed Integrated Circuit
VIRAM	Vector IRAM, processor with embedded DRAM
VIS	Visual Instruction Set, Sun's multimedia ISA extension
VLIW	Very Long Instruction Word
VLSI	Very Large Scale Integration
VMX	IBM's multimedia ISA extension, similar to AltiVec
XOR	Logical Exclusive-OR operation
YUV	A color space consisting of a luma component (Y) and two chroma color difference components (U and V)

Symbols

Δx	Horizontal sampling point distance in image scaling
Δy	Vertical sampling point distance in image scaling
π	Permutation
a	Address function
BC	Shape of access format (black chessboard)
CB	Shape of access format (chessboard)
CR	Shape of access format (crumbled rectangle)
F	Access format
G	Shape of access format (generate)
i	i -coordinate axis
j	j -coordinate axis
L_i	Length of coordinates in i -direction
L_j	Length of coordinates in j -direction
M	Number of accessed data elements
n	Power of two number of memory modules ($N = 2^n$)
N	Number of memory modules
PE	Processing element
r	Scanning point
R	Shape of access format (rectangle)
RC	Shape of access format (red chessboard)
s	Distance between scaled and original pixel location in image scaling or stride specific parameter in parallel memory
S	Module assignment function
S_k	The k th memory module
V	Shape of access format (vector)

1. Introduction

Modern VLSI technology allows high computing capacity in a single chip. Thousands of arithmetic logic units (ALUs) operating at multigigahertz rates can fit on a die of 1 cm^2 [56]. However, delivering instructions and data to the ALUs is prohibitively expensive. For example, only 6.5% of the Intel Itanium 2 die is consumed by the computation units and register files [94]. Communication, control, and storage consume the remaining die area.

Especially multimedia processing demands a lot of computation performance and memory bandwidth [119]. Fortunately, inherent data-level parallelism exists in almost all multimedia applications. For example, most of the operations and data accesses in video compression are highly regular [107]. In a consequence, parallel processing can be utilized to achieve the demanded performance with moderate clock frequencies and, therefore, with reasonable power consumption.

General purpose and digital signal processors include multimedia extensions which utilize the parallelism potential by computing narrow data types (for example 8-bit pixels or 16-bit audio samples) in a SIMD (Single Instruction Multiple Data) manner. Accessing data is straightforward if the narrow data elements locate next to each other, in a correct order, and according to the word boundaries. However, this is not always the case. Up to 41% of the SIMD instructions of the commercial processors are used for data rearrangement before the requested operation can be done in the SIMD fashion [110].

Contemporary architectures try to diminish such overhead by collecting data to dense groups in an additional buffer. Either several serial data accesses or a single extra-wide access is used and only the demanded data is extracted. However, delay is increased with several serial data accesses, and additional complexity is induced in wide access implementations.

Parallel memories can be used to access special data patterns and feed the processors with only useful data without any additional buffers. Moreover, data can also be permuted into correct order by a parallel memory system. Parallel memories have shown to give significant performance increase when compared to conventional wide memory system in video compression [128].

A problem with conventional parallel memory implementations is that they are application specific with fixed data access patterns supported in hardware. The maximum performance with minimum available complexity may be achieved at the cost of very limited flexibility.

Time-to-market, increasing complexity, and system performance requirements have necessitated the increasing utilization of reusable intellectual property (IP) blocks and programmable platforms in system-on-chip (SoC) designs. Designing a reusable IP block is estimated to cost about two to three times the cost of preparing the same block for a single use [58]. However, ten times productivity benefit for that part of the design can be achieved by reusing the IP block. Parallel memories can not typically be reused due to their application specific properties. For reusability, parallel memories should be run-time configurable.

1.1 Objectives and scope of research

The objective of this Thesis is to present novel parallel memory architectures that can be configured at run time.

The scope of this Thesis is on real-time streaming multimedia applications that demand high data throughput and data parallel computation. The goal is to maximize the data access performance and still ensure configurability.

Cache memories can increase average memory bandwidth in general purpose systems, but they are not typically used in embedded signal processing applications due to power consumption, latency, and real-time restrictions [49]. Cache may even become performance obstacle in high data throughput systems [66]. Consequently, cache memories are out of the scope of this Thesis. In addition, instruction memory access is not included.

Three different parallel memory architectures are presented to prove the concept of configurability. Firstly, a traditional type of an application specific parallel memory is tailored for image downscaling. Secondly, limited run-time configurability is added to a novel parallel memory architecture allowing all the constant stride accesses. This is the first parallel memory enabling arbitrary strides. Finally, Configurable Parallel Memory Architecture (CPMA) is presented to offer very wide configurability properties and can support multitude of data patterns demanded in several applications.

All the parallel memory architectures are theoretically analyzed, implemented in register transfer level HDL (Hardware Description Language) and verified with simulations. Figures of merit include delay and area but power consumption has not been studied.

The core of configurable parallel memory is formed by run-time configurable address generation functions. This Thesis presents novel generalized address generation functions that have previously been presented as fixed. The module assignment functions in the stride specific architecture are simplified from the implementations found in literature in terms of hardware resources.

The claim of this Thesis is that run-time configurability can be included in parallel memory architecture with a reasonable cost. For example, with four memory modules, the normalized complexity of the proposed configurable parallel memories is 63–80% less than the reference traditional type of parallel memory. In this study, complexity

normalization considers the number of available run-time configurations in parallel memories.

1.2 Summary of publications

Publication [P1] describes the first framework of the CPMA address computation and page table implementation. Some access format types, a general form for a linear module assignment function, and a general form for an address function are introduced. Detailed block diagrams and a simulation example are also given.

Publication [P2] introduces a general diamond scheme implementation for CPMA. Diamond schemes are analyzed and parameters are chosen to define a suitable configuration for CPMA. Moreover, appropriate address functions for the diamond schemes are designed. Implementation details of the diamond schemes and address functions are also given.

Publication [P3] increases the possible access format types in CPMA. Details of the access format parameters are given and the block diagrams of the implementations are introduced. Required hardware resources are also analyzed.

Publication [P4] introduces XOR-scheme implementations for CPMA. With the implementation, any XOR-scheme can be constructed. Parameters and implementation details as well as required resources are given. Moreover, an example of configuring an XOR-scheme with CPMA is described.

Publication [P5] gives an overview of the implemented CPMA, described in detail in [P1]–[P4]. Moreover, it contains a literature review of the access formats required in different applications and how different types of module assignment functions inherently support specific types of access formats. Resource counts in separated and shared configurations are presented for address computation. Furthermore, logic synthesis based timing and area estimates are given for a 0.25 µm CMOS process for a variable memory module count of 2–32.

Publication [P6] comments a previous publication [69] that proposes a new image scaling method called *winscale*. Well-known bilinear interpolation is evaluated using a similar area pixel model interpretation that is utilized with winscale. In addition, the similarities between the two scaling methods are discovered.

Publication [P7] introduces how two-dimensional image scaling can be accelerated with a new coarse-grained parallel processing method. This paper is an extended version of the previous publication [1]. The proposed method is based on evenly divisible image sizes which is, in practice, the case with most video and image standards. The complexity of the method is examined with two parallel architectures. Several scaling functions can be handled with these generic architectures and parallelism can be adjusted independent of the complexity of the computational units. The most promising architecture is implemented as a simulation model and the hardware resources as well as the performance

are evaluated. Software performance is also estimated when utilizing the proposed block-level scaling.

Publication [P8] presents a novel parallel memory architecture. It allows conflict free accesses with all the constant strides which has not been possible in prior application specific parallel memories. Moreover, the possible access locations are unrestricted and the data patterns have equal amount of accessed data elements as the number of memory modules. The image scaling system shown in [P7] is based on this architecture. The complexity is evaluated with counting the hardware resources.

Publication [P9] gives deeper implementation details of the memory system shown in [P8]. This paper analyzes the impact of pipelining and parallelism degree on performance and complexity. Timing and area estimates are given for Altera Stratix FPGA and 0.18 micrometer CMOS process with memory module counts 2, 4, 8, 16, and 32.

Publication [P10] presents an image downscaler implementation capable of real-time scaling of color video. The scaler can be configured to support nearly arbitrary scaling ratios and the scaling method is based on evenly divisible images introduced in [P7]. Bilinear interpolation is utilized as the scaling algorithm. Fine-grained parallel processing is utilized to increase the performance and an application specific parallel memory system is used to attain the required bandwidth. The downscaler is verified with an FPGA and the complexity as well as performance is compared with several reference implementations.

1.2.1 Author's contribution to published work

The Author has been the main author in all the included publications. He is also the main contributor in all the publications.

The idea of the CPMA concept was originally introduced by Kimmo Kuusilinna, Dr. Tech., in [72]. CPMA was designed and implemented by the Author and Jarno Vanne, M.Sc. The Author is responsible for the Address Computation and Page Table units that are described in [P1]–[P5]. Jarno Vanne, M.Sc., implemented other CPMA units [138], [139] as well as a processor framework for CPMA [137].

Publication [P1]. The design and implementation presented in this publication was done by the Author. The text was written by the Author assisted by Jarno Vanne, M.Sc., Kimmo Kuusilinna, Dr. Tech., Professor Timo Hämäläinen, and Docent Jukka Saarinen.

Publications [P2]–[P7] and [P10]. The writing, ideas, implementations and simulations shown in these publications were the work of the Author. Jarno Vanne, M.Sc., Kimmo Kuusilinna, Dr. Tech., and Professor Timo Hämäläinen gave valuable criticism and comments.

Publications [P8]–[P9]. The idea, design, and implementation of the work was resulted by the Author. The publications were written by the Author with support from Jarno Vanne, M.Sc., and Professor Timo Hämäläinen.

1.3 Outline of Thesis

The Thesis consists of an introductory part and ten publications. The publications embody the main results of the Thesis.

The introductory part of this Thesis is organized as follows. Chapter 2 describes motivation for the work. Multimedia extensions in commercial processors as well as research improvements to multimedia data accesses are analyzed. Chapter 3 introduces principles of parallel memories and introduces applications that benefit from proper data patterns. Moreover, related work on parallel memories is presented. The Configurable Parallel Memory Architecture is described in Chapter 4. Chapter 5 presents the implemented parallel memory architecture allowing all the stride accesses. Moreover, the image scaling application utilizing the proposed memory system is presented. In Chapter 6, an application specific parallel memory system for an image downscaler is described. Chapter 7 evaluates and compares the three proposed parallel memory implementations and, finally, Chapter 8 concludes the introductory part of the Thesis.

2. Motivation

Multimedia applications, especially image and video processing, can be divided into *low-level*, *medium-level*, and *high-level* tasks [107]. The low-level tasks are characterized by highly regular sequences of operations and data accesses. Respectively, the medium-level tasks may include data-dependent decisions and have lower regularity whereas the high-level tasks have a highly data-dependent computation flow.

Major part of multimedia computation workload is caused by low-level operations. As an example, low-level operations comprise up to 90% of the overall computation in frame based video compression [107]. Fortunately, the low-level operations offer high data-level parallelism (DLP) that can straightforwardly be utilized in SIMD (Single Instruction Multiple Data) processing. In SIMD computation, several processing units simultaneously execute the same instruction for different data.

2.1 Commercial processors with SIMD extension

In the past, SIMD processing was mostly utilized in special purpose machines with dedicated hardware [49]. Nowadays, SIMD is mainly appearing within general purpose processors (GPPs) and digital signal processors (DSPs) [30], [124].

Multimedia applications typically use narrow data types, for example, 8-bit pixels or 16-bit audio samples. However, the narrow data types are not well-suited for recent GPPs and DSPs that normally utilize 32 or 64-bit data width. Therefore, multimedia ISA (Instruction Set Architecture) extensions are added to GPPs and DSPs. These extensions utilize SIMD parallelism at subword level by operating concurrently on, for example, eight 8-bit or four 16-bit values packed in a 64-bit register. Figure 2.1 gives an example of packed addition with four 16-bit values [18]. Two 64-bit data is accessed from the registers, four simultaneous additions are operated, and the result is stored to a register.

Typically, high performance GPPs utilize superscalar architecture with high clock frequency whereas high performance DSPs utilize VLIW (Very Long Instruction Word) architecture and lower frequency [29]. However, some superscalar DSPs can also be found; an example is ZSP architecture from LSI Logic [125].

This Section describes the parallelism available in commercial processors with subword SIMD properties. Superscalar and VLIW processors are separated. Moreover, a very powerful contemporary single-chip multiprocessor is described in detail.

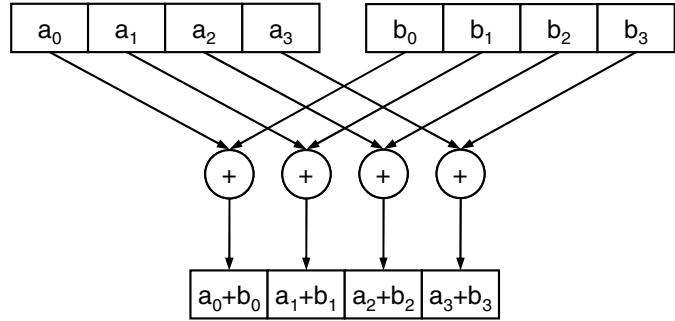


Figure 2.1. Subword SIMD addition.

2.1.1 Superscalar processors

Multimedia ISA extensions available in existing GPPs include Intel's Matrix Math eXtension (*MMX*) [104], [105], Streaming SIMD Extension (*SSE*, *SSE2*, *SSE3*) [133], [12], AMD's *3DNow!* [96], Sun UltraSparc Visual Instruction Set (*VIS*) [134], DEC Alpha Motion Video Instructions (*MVI*) [14], HP's Multimedia Acceleration eXtension (*MAX-1*, *MAX-2*) [83], [84], and Motorola PowerPC *AltiVec* [26]. *AltiVec* is also referred to as Apple *Velocity Engine* and IBM *VMX* with some minor changes.

Register widths of the ISA extensions are 32 bit (*MAX-1*), 64 bit (*MMX*, *3DNow!*, *VIS*, *MVI*, *MAX-2*), or 128 bit (*SSE*, *SSE2*, *SSE3*, *AltiVec*). The number of packed instructions varies significantly from 9 in *MAX-1* and 13 in *MVI* to 162 in *AltiVec*. Typical packed instructions include arithmetic (addition/subtraction, multiplication), logical, compare, load/stroe, data conversion, and data rearrangement operations.

2.1.2 VLIW processors

Texas Instruments *TMS320C62x* and *TMS320C64x* are VLIW DSPs which can operate up to eight fixed point operations with 32-bit data simultaneously [132]. With *TMS320C64x*, nearly all the operations can also be 8 or 16 bit wide subwords. Hence, a maximum of 28 arithmetic operations (24 8-bit and 4 16-bit) can be executed in a single clock cycle.

Analog Device *TigerSHARC* has two computation units and in each a 64-bit wide register bank with 4 read and 3 write ports [33], [34]. By utilizing subword and some other specific techniques, *TigerSHARC* can execute up to 24 16-bit fixed point arithmetic operations or six 40-bit floating point operations in a single cycle. However, at the maximum rate, some of the operands have to be same.

Tensilica supports modifications to its processor architecture and instructions [11], [40]. Basic version of Tensilica *Xtensa LX* processor with DSP-oriented *Vectra LX* extension has three slots with 160-bit wide registers. Simultaneously, four 40-bit MAC (Multiply Accumulate) and eight 20-bit basic ALU (Arithmetic Logic Unit) instructions can be executed whereas the third slot is used for memory access instructions.

Philips *TriMedia* and Equator *MAP-CA* are media processors with VLIW architecture. TriMedia TM3270 [144] has 5 issue slots for 32 bit wide data and it supports 1 x 32-bit, 2 x 16-bit, and 4 x 8-bit SIMD instructions. MAP-CA VLIW core has four execution units [9]. Two of the units operate with 32-bit registers and the rest two with 64 or 128-bit registers. Totally, the core allows 2 x 32-bit and 2 x (8 x 16-bit) operations in a clock cycle.

2.1.3 Cell multiprocessor

Cell multiprocessor is a novel architecture developed by Sony, IBM, and Toshiba [55], [106]. Target applications of Cell include image processing for high definition TV, image processing for medical usages, high performance computing, and gaming [120]. Sony's upcoming PlayStation 3 game console is one example that will contain a Cell processor. Cell contains a Power processor element (*PPE*) and eight synergistic processor elements (*SPE*). As an example of task allocation, PPE is responsible for running the operation system and coordinating the data processing threads through SPEs [31]. PPE is a multi-threaded 64-bit Power Architecture compliant core with 128-bit VMX multimedia extension. SPE is a new processor architecture designed to accelerate media workloads through high clock frequency, dual issue, and subword SIMD operations. For 90-nm CMOS SOI (Silicon On Insulator) process, SPE can achieve up to 5.6 GHz clock frequency with 23 pipeline stages. SPE issues and completes all instructions in program order. It uses a VLIW type dual issue feature allowing up to two instructions per cycle [31]. One issue slot supports logical, fixed point and floating point operations whereas the other provides loads/stores and byte permutation operations as well as branches. All instructions utilize 128-bit data with varying element width in SIMD fashion (2 x 64-bit, 4 x 32-bit, 8 x 16-bit, 16 x 8-bit, and 128 x 1-bit).

Memory latency and bandwidth are the most crucial for achieving the demanded performance with Cell processor [55]. High processor frequencies are hard to meet due to long DRAM (Dynamic Random Access Memory) latencies. SPE has 256 Kbyte embedded SRAM (Static Random Access Memory) as local memory (not a cache) [7], [55]. Direct memory access (DMA) is used to load/store data and instructions between local memory and the off-chip main memory. Local memory is the largest component in SPE and, therefore, single port memory is used to minimize area. The memory provides a special feature, narrow and wide access. In narrow access, 128 bits are transferred between the local memory and registers. Wide access (1024 bits) is used for instruction (pre)fetch and to transfer data from DRAM with DMA operations. Wide DMA access demands 16 processor cycles to place data on the 64-bit wide on-chip bus. Thus, much bandwidth is released for narrow accesses and instructions fetches. A 128-bit load operation from local memory takes six clock cycles whereas write demands four cycles. Pipelining the memory enables that 5.6 GHz clock frequency can be achieved [7].

2.1.4 Summary

As mentioned above, SIMD instructions perform the same operation in multiple data elements. As can be seen in Figure 2.1, the subwords have to be in *proper order* and

aligned correctly in registers. Moreover, the used subword bit-width is often too small for intermediate results during computation. For example, each of the four 16-bit additions produces a 17-bit result (Figure 2.1). Therefore, subwords are unpacked to *wider computational format* and again, repacked to a register when the final result is achieved. To achieve the proper subword order, some special data reordering instructions can be used.

There are several methods to implement correct subword data alignment. The older architectures allow only aligned data accesses, meaning that data can only be read from memory according to whole word boundaries (i.e. not according to subword boundaries). Some special instruction or combination of two shift operations and a bitwise OR operation are used to align data from two registers correctly into one register. Also unaligned accesses are supported in some processors where proper data alignment is achieved automatically during data access with hardware. However, a single unaligned access may be slower than an aligned access.

2.2 Data access improvements in multimedia applications

The above mentioned data reordering, alignment, and (un)packing operations mean a significant amount of overhead instructions. According to GPP extension evaluations done in [110], up to 41% of all the SIMD instructions are such data rearrangement overhead instructions. As another case study, Slingerland evaluates five different superscalar GPPs with multimedia extensions that execute several multimedia kernels [116]. As a result, 10-17% of the total instruction count was reported to be data rearrangement instructions. The same overhead has been recognized by other researchers as well [18], [75].

Slingerland proposed two methods to reduce data rearrangement overhead: *stride* memory access support and implicit unpacking/packing of data [116]. Stride is the distance that separates accessed data elements. This Section presents architectures diminishing that rearrangement overhead.

2.2.1 Contemporary architectures

Matrix Oriented Multimedia (MOM) extension contains vector SIMD instructions [21]. They operate on matrices and each matrix row corresponds to a single packed data type. MOM allows arbitrary stride between consecutive rows. However, the subwords in a row have to be adjacent.

In the *Complex Streamed Instruction (CSI)* multimedia architecture [17], [18], a single instruction can process long vector data streams with SIMD fashion. Contrary to MOM, strides can be arbitrary in two dimensions. Several control registers are used to control the stream. In a simple operating example shown in [18], 12 setup instructions are firstly executed to initialize the stream control registers. After that, a nested loop is substituted by a single instruction that may take several tens of clock cycles. Data is read from an on-chip

cache to registers 512 bits at a time. Wanted subwords are extracted, unpacked, operated in SIMD fashion, repacked, inserted (i.e. unextracted), and stored back to memory.

Impulse memory controller [148] uses a specific address translation hardware to remap data structures in main memory. The memory controller gathers the demanded data elements, packs them into a dense cache line, and sends this data to the cache. Among others, stride access is supported. Applications may access just the required data from main memory keeping the cache coherent at the same time. This data is also in the right order.

Stream Memory Controller (SMC) takes account of the properties of modern DRAMs [92], [93]. Some of such properties are fast page mode, nibble mode (bursting) as well as interleaved memory properties of Rambus DRAM [92]. Since access delay to random memory location is not constant, the order of request strongly affects the bandwidth. SMC combines a compile-time detection of data streams with execution-time selection of access order and issue. Stream accesses are typically generated in algorithm loops. In SMC, each stream is characterized with a starting location, stride, and data element count.

Imagine stream processor is specialized for long media streams of data [23], [59]. A stream in Imagine is a sequence of similar elements, for example, a row of image pixels. Imagine is suited for applications performing many operations on each stream element. It has 48 32-bit wide arithmetic units divided into eight clusters. Each cluster receives the same VLIW instruction in SIMD fashion. Also 8 and 16-bit subword SIMD instructions are supported. The processed prototype die contains 21 million transistors [56]. Similar to SMC, Imagine takes account of DRAM properties by reordering main memory accesses and supports strided addressing.

VIRAM is a vector IRAM (Intelligent RAM) [103] processor with embedded DRAM [67], [68]. It contains eight 64-bit data paths divided to four *lanes*. All the lanes receive identical control signals on each clock cycle. A single vector instruction operates with four data paths and can take several clock cycles when operating up to 32 64-bit elements. Also 16 and 32-bit subwords are supported. The chip contains 125 million transistors including 113-million transistor DRAM.

The VIRAM main memory is 13-MB on-chip embedded DRAM constructed by eight memory modules that are accessed in interleaved manner. Each memory module is 256 bits wide and has nonpipelined 5-cycle latency with 200 MHz clock frequency. VIRAM supports stride as well all *indexed* memory accesses. An indexed load uses elements in a vector register as pointers to access multiple memory locations. However, the bandwidth is lower for arbitrary stride and indexed access compared to unit stride (sequential) access.

Vector and stream processors utilize pretty similar concepts. However, vector processors like VIRAM execute a single arithmetic operation on each vector element. In contrast, Imagine stream processor performs all of the operations for one stream element before moving on to the next element [56].

2.2.2 Summary

All the previous architectures support stride and possibly some irregular accesses by collecting data to dense groups in an additional buffer. The size of the buffer varies. For example, in SMC prototype, there are four FIFO (First In First Out) buffers and each buffer size is adjustable from eight to 128 64-bit elements (total 2 KB to 32 KB) [93]. Impulse, SMC, Imagine and VIRAM access data to the buffer from DRAM whereas MOM and CSI perform accesses from cache. Data is either gathered with several serial memory accesses or a single wide access is used and merely the required data is extracted. Unfortunately, several serial memory accesses take multiple clock cycles whereas wide access demands additional data extraction logic as well as extra wide memory and buses. Moreover, with longer strides, wide memory requires several data accesses as well. Additional data accesses are a matter of power consumption in addition to increased delay.

3. Parallel Memories

Many applications favor special data patterns that could be accessed in parallel. This phenomenon is utilized in parallel memories, where the idea is to increase memory bandwidth with several memory modules working in parallel and to feed processors with only useful data. The basic concepts of parallel memories have been known for some time [13].

Parallel memory architecture can be utilized especially in SIMD and subword SIMD processing. The architectures shown in Section 2.2 collect data to some additional buffer whereas parallel memories can be used to access only the required data in parallel without extra buffering. Neither additional memory accesses nor extra wide memory accesses are required. Moreover, the data is also in correct order.

This Chapter concentrates mainly on two-dimensional array (also called raster) data structures in parallel memories. However, one-dimensional structures are considered as well. Three-dimensional array and tree data structures are also studied in parallel memory systems [8], [19], [35], [62] but they are not in the scope of this Thesis.

3.1 Memory system classification

Figure 3.1 depicts the memory system classification on interleaved and parallel memory architecture used in this Thesis. The both memory configurations utilize N memory modules S_0, S_1, \dots, S_{N-1} working in parallel. Moreover, M processing elements $PE_0, PE_1, \dots, PE_{M-1}$ are utilized with parallel memories ($M \leq N$).

Interleaved memories use *time-multiplexed* parallel memory modules that receive access requests serially one by one [113]. The memory access cycle takes several processor cycles. Typically, there are buffers in the inputs and outputs to queue the request [136]. Traditionally, vector computers utilize interleaved memories. Respectively, parallel memories are defined as *space-multiplexed* memories that are used, for example, in SIMD processing [113]. Parallel memories have wide address and data buses and the memory modules receive access requests in parallel. In some multivector supercomputers, a memory system with plenty of memory modules may be arranged in both space- and time-multiplexed manner [113]. This Thesis concentrates on parallel memories.

In the following, some terminology variations between the two memory architectures are clarified. In a *matched system*, the minimum number of memory modules can feed data to all the processing elements in each processor cycle. This means in interleaved memory that

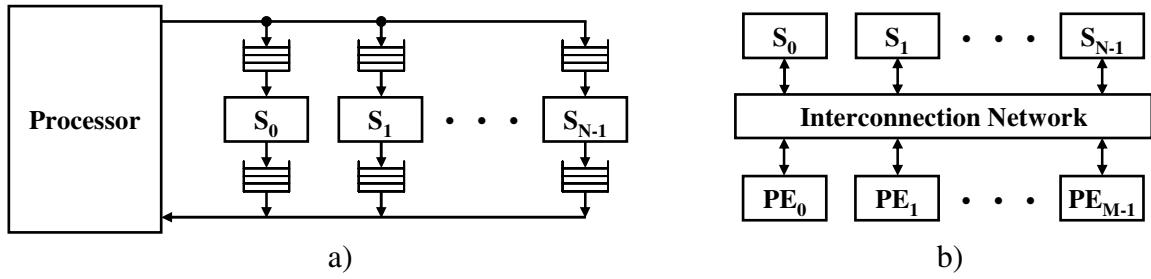


Figure 3.1. Memory system classification: a) interleaved memory, b) parallel memory.

the number of memory modules equals to the ratio between the memory cycle and the processor cycle [136]. Respectively, in matched parallel memory system, the number of memory modules equals to the number of processing elements ($N = M$) [122].

A *memory conflict* occurs when more than one data locations are simultaneously addressed in a single memory module. Multiport memories enable several simultaneous memory accesses, but they are expensive solutions especially when the number of ports is large. Memory conflict in interleaved memories may decrease the bandwidth, especially in matched system [136]. In parallel memories, it is normally supposed that M parallel data elements can be accessed simultaneously in each processor cycle. Therefore, usually no memory conflicts are allowed in parallel memory system.

3.2 Parallel memory principles

A generalized block diagram of a parallel memory architecture is depicted in Figure 3.2. The functional blocks in Figure 3.2 are an *Address Computation* unit, N *memory modules* S_0, S_1, \dots, S_{N-1} , and a *Data Permutation* unit [38]. Depending on the *access format* F and the location of the first element (*scanning point*) r , the Address Computation unit computes the addresses and directs them to the appropriate memory modules. The Data Permutation unit organizes the *data* into correct order specified by the access format and scanning point.

A binary raster picture is depicted in Figure 3.3. Grey squares illustrate pixels that have value of one. White squares have value of zero. Pixels can be read with an access format F in parallel. There are two access formats in this example. They are enclosed by a fat line.

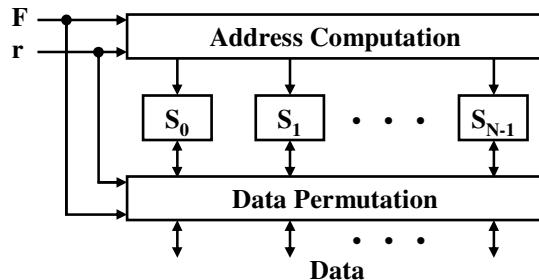


Figure 3.2. A generalized block diagram of a parallel memory architecture.

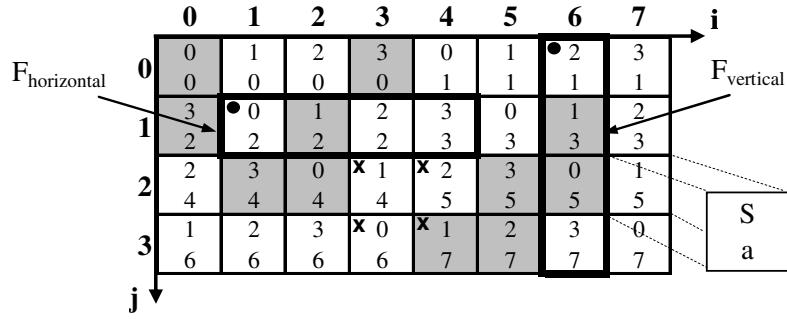


Figure 3.3. Two examples of a parallel memory access.

The first access format $F_{\text{horizontal}}$ is four pixels horizontally and the second F_{vertical} is four pixels vertically. Access is placed to a certain pixel location with a scanning point $r(i, j)$ that is in the left upper corner of the access format. The scanning points are marked with a dot ‘•’. Horizontal format is located in $r(i, j) = (1, 1)$ and vertical format in $r(i, j) = (6, 0)$. Upper number in a square is the *memory module number S* where the value of a pixel is located. There are four memory modules in the example. The lower number in a square is an *address a* in the memory module. A *scanning field* determines the addressable area. The i and j dimensions of the scanning field are determined by *lengths of coordinates* L_i and L_j . In the example, $L_i = 8$ and $L_j = 4$.

The raster picture is mapped in four memory modules (S_0 – S_3) as can be seen in Figure 3.4. When reading with the horizontal format $F_{\text{horizontal}}$, the values of the pixels are loaded from the memory modules S_0 , S_1 , S_2 , and S_3 under the address 2, 2, 2 and 3 respectively. On the other hand, vertical format F_{vertical} accesses the pixels from memory modules S_2 , S_1 , S_0 , and S_3 under the addresses 1, 3, 5, and 7 respectively. The physical addresses used with the vertical format are enclosed with a fat line in Figure 3.4. The addresses are sent from the Address Computation unit to the proper memory modules. The accessed data values are shown below the memory modules. The Data Permutation unit shuffles the data into correct order.

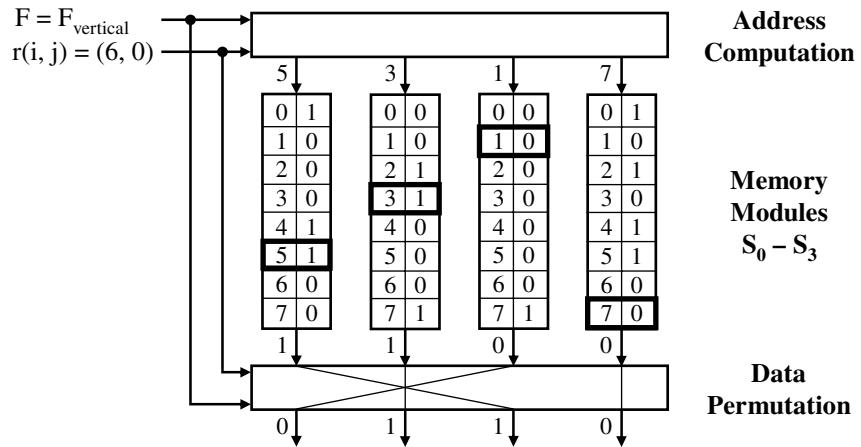


Figure 3.4. Mapping the addresses of the example in Figure 3.3 to memory modules.

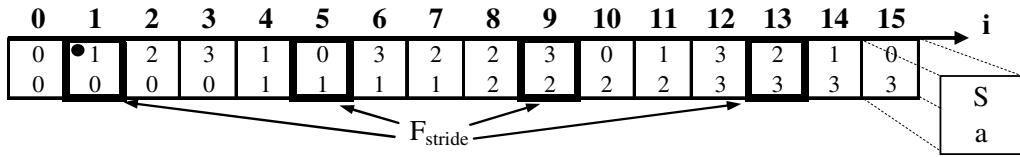


Figure 3.5. Example of a 1-D parallel memory access.

The assignment of a pixel (i, j) to the memory module is determined as

$$S(i, j) = (i + 3j) \bmod 4 , \quad (1)$$

and the address

$$a(i, j) = \left\lfloor \frac{i + 8j}{4} \right\rfloor . \quad (2)$$

S is called a *module assignment function* (skewing scheme) and a an *address function*. Traditionally, address computation is hardwired to use a specific module assignment and address function using, for example, ROM (Read Only Memory).

All the data addressed by an access function should be accessible in parallel, in other words, the access format should be *conflict free* within the associated module assignment function. That means that only one pixel can be accessed from one memory module at a time. The example shown is conflict free. However, it is not possible to read out the pixels marked by crosses ‘x’ in parallel since pixels (3, 2) and (4, 3) in Figure 3.3 are stored in the same memory module S_1 .

The above example utilizes two-dimensional (2-D) memory addressing. Also one-dimensional (1-D) addressing can be used. Figure 3.5 depicts an example skewing scheme with 1-D addressing. There are four memory modules and sixteen memory locations in the example. As an example, a data element at location $i = 7$ is stored in memory module $S(i) = 2$ under address $a(i) = 1$. The data elements referenced with an access format F_{stride} (stride = 4) are enclosed by a fat line. The access format F_{stride} is conflict free since all the four elements are located in different memory modules (S_1, S_0, S_3 , and S_2).

3.3 Access formats for a selection of applications

The address space in parallel memories cannot be assigned arbitrarily since the data has to be stored in memory using predetermined patterns, called access formats or templates. The data elements of an access format can be read from the memory modules in parallel when a conflict free access is performed.

Each application may utilize different access formats. Some examples of templates are depicted in Figure 3.6. Squares in an access format depict the relative positions of

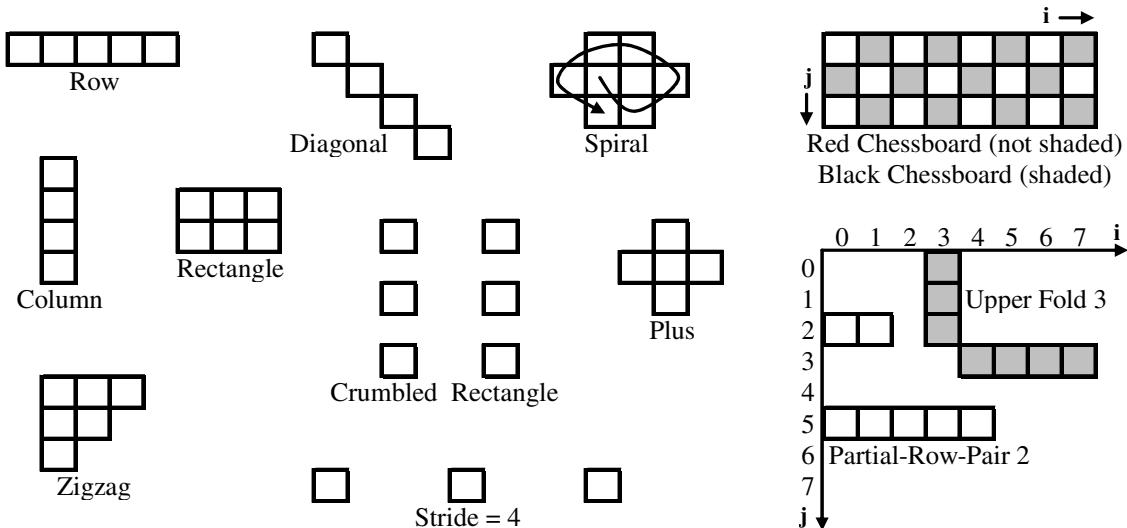


Figure 3.6. Access format examples.

accessed data elements. The following summarizes some applications that benefit from parallel memories.

3.3.1 Image processing

An image can be represented by a two-dimensional array. Images can be stored, viewed from a display, rotated, scaled, and compressed, to name a few image specific operations. Image processing can utilize access formats like *rows*, *columns*, *rectangles*, *crumpled rectangles* (also called *distributed blocks*), *chessboards*, *diagonals*, and *backward diagonals* [61], [63], [81], [98], [123], [143]. Moreover, three-dimensional (3-D) stereo vision benefits from *square* templates. One such FPGA (Field Programmable Gate Array) implementation utilizes a 3×3 window in parallel memory [85]. In image preprocessing, the Lee routing algorithm benefits from *spiral* type access formats [48].

3.3.2 Video processing

Some of the basic video coding operations, for example, in H.263, H.264, and MPEG-4 standards, are motion estimation, interpolation, motion compensation, discrete cosine transformation (DCT), quantization, inverse quantization, and inverse discrete cosine transformation (IDCT). Suitable access formats for these operations are *rows*, *columns*, *crumpled rows*, *crumpled columns*, *rectangles*, and *crumpled rectangles* [115], [126], [130], [140]. Rectangle format is also required in MPEG-4 shape encoding [73] as well as row and column formats in H.264 deblocking filter [87]. Moreover, the basic zigzag scan as well as the additional alternate-horizontal scan and alternate-vertical scan in MPEG-4 can utilize quite irregular templates [72], [111]. These formats are similar to the “Zigzag” format in Figure 3.6.

3.3.3 Vector processing

Vector computers are specialized in accessing addresses separated by a vector *stride*. In addition, *rectangle* formats are utilized. Many applications in the fields of digital signal processing and telecommunications benefit from the use of strides and *stride permutations*. Vector/matrix computation, fast Fourier transform (FFT) and Viterbi algorithms are some examples [44], [53], [123]. Stride permutation access is similar to stride access. However, in stride permutation access, the accessed data element locations exceeding the scanning field wrap around to the beginning of the scanning field [121], [123]. For example, accessing four data elements from eight elements ($d_0, d_1, d_2, d_3, d_4, d_5, d_6, d_7$) in stride-by-4 order produces data elements d_0, d_4, d_1 , and d_5 .

3.3.4 Numerical analysis

Useful access formats include *red* and *black chessboards* for red-black relaxation methods [32]. The red chessboard consists of data elements in the rectangle in such a way that the sum of their i and j coordinates is even. The black chessboard consists of the remaining elements in the rectangle. Bitonic sorting benefits from the use of *crumbled row* type accesses [5]. *Rectangle* templates help data accesses of convolution algorithms and a *plus* ('+') shaped access format is useful for successive over relaxation (SOR) [97].

3.3.5 Pattern analysis

Two-dimensional template matching and pattern recognition can utilize a *rectangle* access format [6], [61]. *Chessboard* and *crumbled rectangle* formats are useful in image sampling [91]. Hierarchical clustering, cluster validity, feature extraction, and classification can benefit from *upper folds*, *lower folds*, *partial-row-pairs*, and *partial-column-pairs* [89]. These formats are also desirable patterns for image processing. Those access formats are defined for an $N \times N$ matrix where N is the number of memory modules. Only $N - 1$ memory accesses are executed and, therefore, the number of accessed data elements is $M = N - 1$. Examples of an upper fold 3 and a partial-row-pair 2 are depicted in Figure 3.6. If the size of the matrix is 8×8 , the number of memory modules is $N = 8$, and the number of accessed data elements $M = N - 1 = 7$.

3.4 Module assignment functions

The distribution of data to the memory modules is called a module assignment function S and also known as a skewing scheme. The used module assignment function determines access formats that can be used conflict free. Different types of module assignment functions inherently support specific types of access formats. These differences are shortly introduced in the following.

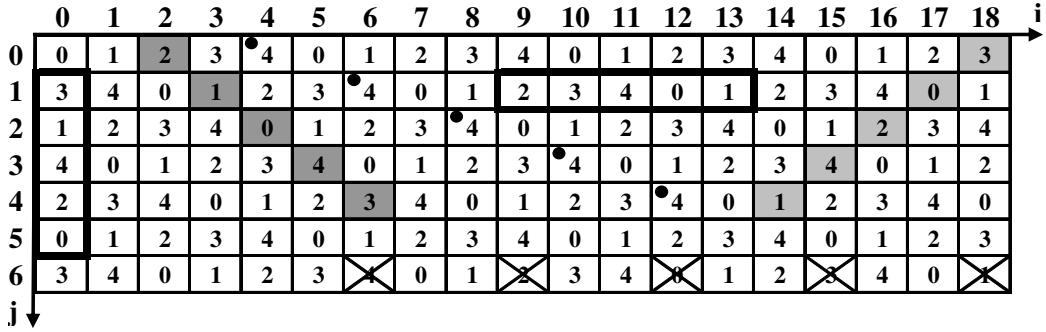


Figure 3.7. Straight access formats with a prime number of memory modules ($N = 5$).

3.4.1 Linear functions

Linear module assignment functions were first introduced in the 1970's by Budnik and Kuck [13]. A prime number of memory modules N enables access to many straight access formats (for example rows, columns, diagonals, constant stride accesses) [13]. Moreover, the number of accessed data elements M can be equal to the number of memory modules N . On the other hand, special properties of *Fibonacci numbers* (1, 1, 2, 3, 5, 8, 13, ...) can be utilized when a conflict free access to all axially oriented rectangles with a minimum number of memory modules is desired [37], [38].

A case with a prime number of memory modules $N = M = 5$ is illustrated in Figure 3.7. The used module assignment function is $S(i, j) = i + 3j \bmod 5$. Some conflict free straight access formats are shown: row, column, forward diagonal, backward diagonal, and forward jumps (stride) of three. The access format with jumps of $(i, j) = (2, 1)$ is not conflict free because the data elements are accessed from the same memory module 4. It is illustrated with dots ‘•’.

However, a fast hardware implementation of the arithmetic modulus of a prime number is expensive. If the number of memory modules is a power of two, the arithmetic operations are implemented using just shifting and masking. Unfortunately, restricting the number of memory modules restricts also the allowed access formats. Budnik and Kuck [13] as well as Lawrie [76] described linear skewing schemes that use more memory modules than there are accessed data elements to ease the implementation of the modulus operation. Unfortunately, this approach generates unused memory locations, *holes*.

If two elements are stored in the same memory module and their respective neighbors are also stored in the corresponding memory modules the module assignment function is called *isotropic* [38]. If an access format F is conflict free in *any* point with respect to an isotropic module assignment function S , then the access format F is conflict free in *every* point [38]. In other words, if an isotropic module assignment function is used, an access format can be placed conflict free everywhere in memory or not at all. In addition, the same is true for linear module assignment functions because they are a subset of isotropic functions.

3.4.2 XOR-schemes

XOR-schemes always utilize a power of two ($N = 2^n$) memory modules. All the required calculations are bitwise logical operations. [32]

As previously noted, if a linear module assignment function is used and an access format is conflict free at any point in memory then the access format is always conflict free. With XOR-schemes, this is not true. To show whether or not an access format is conflict free with respect to an XOR-scheme, methods of linear algebra can be applied [32].

Furthermore, no linear module assignment function exists which is conflict free with respect to rows, columns, and rectangles of size $p \times q$, when the number of accessed data elements $M = N = p \times q$ [143]. On the other hand, it is always possible to find an XOR-scheme that meets the requirements, but access formats are then restrictedly conflict free [36], [38].

Finding an XOR-scheme for a set of conflict free access formats is shown to be NP-complete (Nondeterministic Polynomial time complete) when the number of memory modules is arbitrary [2]. To solve the problem, a heuristic algorithm and automated heuristic methods have been presented [2], [4], [15].

3.4.3 Periodic and multiperiodic schemes

Periodic module assignment functions have certain periodicity in the access scheme [114], [146]. A periodic rectangle is called a *basic domain*. The basic domain of a periodic function may be any kind of rectangle, but this discussion concentrates on axially oriented rectangles because of their simplicity in hardware implementation.

Diamond schemes are a special class of multiperiodic functions [38], [131]. The basic domain in diamond schemes is formed as with periodic functions, but there are also two permutations π_1 and π_2 . The permutations change the order of the basic domain when i and j increase, respectively.

3.4.4 Relationship between functions

The relationships between different classes of module assignment functions are illustrated in Figure 3.8 [38]. For example, linear functions are isotropic, periodic, belong to diamond schemes, and general module assignment functions. On the other hand, a periodic function does not have to be a dyadic function but dyadic functions are periodic functions. XOR-schemes are a subset of dyadic functions and multiperiodic schemes are an extension of diamond schemes [38].

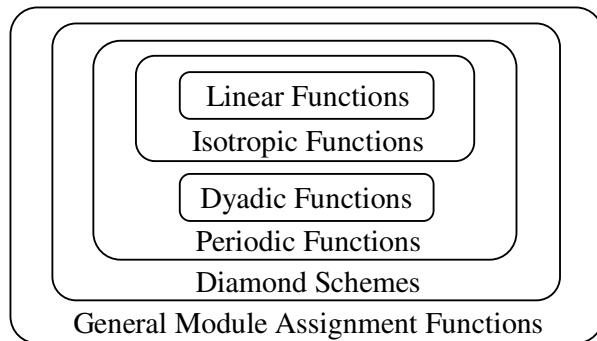


Figure 3.8. Relationships between module assignment functions [38].

One important parameter with a conflict free access is the diversity of available scanning points. Having parallel access with an access format at any memory position induces strong limits on the variety of shapes of accessible data templates [114]. However, restricting the positions of access format with a linear module assignment function does not obtain additional conflict free access formats. That is because linear functions are a subset of isotropic functions. On the other hand, using dyadic functions and other periodic and multiperiodic functions may increase the amount of available access formats that can restrictedly be placed on the scanning field.

Skewing schemes are intended to be equivalent if they differ only in the denotation of memory modules. In *isomorphic* schemes, one-to-one mapping exists between the skewing schemes [38]. For example, two simple schemes $\{0, 1, 2, 3\}$ and $\{0, 3, 2, 1\}$ are isomorphic since equal memory module numbers are received by exchanging 1 and 3 in either of the schemes.

3.5 Related work

In a typical published parallel memory research work, certain access formats are assumed as the basis of the research and a new conflict free skewing scheme is introduced [24], [47], [65], [99], [145]. A new solution offers more access formats in a scheme [25], [28], [91], [95], uses less memory modules for conflict free access formats [20], [147], or utilizes the memory space more efficiently [62], [142]. Moreover, a new scheme may allow simpler and faster implementation for address generation [3], [42], [78], [80] or data permutation [79], [82], [90], [141]. Sometimes, automatic methods are used to find suitable skewing schemes [4], [15], [16], [108]. In the ideal case, the number of data elements in an access format is equal to the number of memory modules and the memory space is fully utilized. In turn, compile-time techniques to detect and correct memory conflicts are presented in [39] and [57]. An extensive categorization of parallel memory publications is given in [127, Appendix B]. In the following, some parallel memory architectures are described.

3.5.1 Parallel memory implementations

In 60's, ILLIAC IV machine had 256 coupled computers. Skewed storage was used to allow columns in addition to rows to be accessed [70].

In 70's, STARAN computer architecture allowed data accesses in either word or bit-slice direction [10]. Bit-slice access contains bits of the same stage from all the words and each bit is sent to separate processing element. An XOR-scheme was used as module assignment function.

In 80's, the Burroughs Scientific Processor (BSP) was a SIMD machine with 16 parallel processors and a 17-module parallel memory system [71], [77]. The prime number of memory modules used in the system supported many conflict free straight access formats. Unused memory locations (holes) were generated to simplify the implementation.

A multiaccess frame buffer architecture for graphical display systems is introduced in [47] and [145]. The proposed parallel memory supports row, column, and rectangle access formats. Memory module count as well as the width of the scanning field is restricted to a power of two.

Highly parallel DSP (HiPAR-DSP) is a parallel processor for image and video processing [50], [112]. It consists of four or 16 parallel data paths controlled in a SIMD manner. Each of the data paths contains three-slot VLIW with 16-bit MAC, 32-bit ALU, and 40-bit shift & round units. Subword instructions are supported by the ALU unit. Parallel memory (called matrix memory in HiPAR-DSP) contains nine memory modules in 4-path HiPAR-DSP whereas the memory in 16-path processor includes 25 modules. Two-dimensional addressing is used and crumbled rectangle access formats are supported. Moreover, a single data element can be broadcasted to all the data paths. The manufactured 4-path and 16-path DSP chips contain 1.3 and 5.5 million transistors including 4.5 KB and 50 KB matrix memory, respectively [50], [64].

A byte and modulo addressable parallel memory architecture for video coding is proposed by Tanskanen [129]. Byte addressing means that a row access format can arbitrarily be located conflict free in the scanning field, i.e. unaligned access without delay overhead. Respectively, modulo addressable memory behaves like circular buffer. Area, delay, and power consumption results in CMOS are given for memory module count 2, 4, and 8.

Another Tanskanen's parallel memory architecture specialized for video coding is implemented in [128] and [130]. The evaluated three-slot VLIW DSP core allows two simultaneous data access operations to two data memories in addition to one arithmetic operation [128]. Subword SIMD instructions are supported. The two data memories are implemented with two different parallel memory systems. Complexity and performance is evaluated with 8-bit subwords and memory module counts of 2, 4, 8, and 16.

A H.264 deblocking filter with a two-dimensional parallel memory is shown in [87]. Linear skewing scheme with eight single port memory modules is used. The system is implemented with Verilog and the results are given in 0.35 μm CMOS technology.

Kuzmanov introduces a special hardware to accelerate MPEG-4 shape encoding in [73]. The architecture includes two parallel memory systems. In the evaluated configurations, each of the parallel memories contains 1, 2, 8, or 16 embedded memory modules in Altera FPGA. Two-dimensional memory addressing is utilized.

Kuzmanov also presents a parallel memory architecture for rectangle access formats in [74]. Two-stage memory hierarchy is considered. Dual port parallel memory modules receive data from main memory with one memory port and communicate with the processing elements with the other port. The complexity of the parallel memory system is evaluated for up to 256 memory modules with Xilinx Virtex II Pro FPGA.

A parallel memory architecture allowing stride permutation templates is shown in [53] and [123]. FFT and Viterbi algorithms are some of the target applications. Viterbi decoding computation is rescheduled in [53] so that only one data permutation network is required. Data reading can be done along hardwired input ports whereas interconnection network is required for write operation. XOR-schemes are utilized.

A SIMD computer with parallel memory system is introduced by Park [100], [101], [102]. The memory module count N is a prime number and greater than the processing element count M . Linear module assignment function is used and several access formats are supported including rows, columns, diagonals, rectangles as well as crumbled rows, columns, diagonals, and rectangles. A look up table (LUT) based address computation is utilized.

Motion stereo is used, for example, to get reliable three-dimensional information for intelligent robots that move autonomously in a dynamically changing environment [85]. An FPGA implementation comprising motion-stereo processor with parallel memories is shown in [85]. In motion stereo, large amount of computation is required to find corresponding pixels between 2-D images. As a similarity measure, a sum of absolute differences (SAD) is used in [85]. In the implementation, four FPGAs and nine memory modules in each are utilized. Similar architecture for VLSI (Very Large Scale Integration) implementation is shown in [41]. Four 16 x 16 computation units with 256 small capacity memory modules are used. Only restricted interconnection network between the computation units and memory modules is supported in both implementations [41] and [85].

3.5.2 Parallel memory benefits and drawbacks

Tanskanen evaluate five different MPEG-4 video encoding functions including full and half pixel motion estimation, interpolation, overlapped block motion compensation (OBMC), and IDCT in [128] and [130]. Performance and complexity of the proposed parallel memory systems ($N = 2, 4, 8, 16$) are compared to byte addressable and, especially, to word addressable memory. Word addressable memory can reference memory just according to word boundaries (only aligned accesses). All the results of logic units are

obtained from the generated CMOS process layouts. Memory module information is achieved with a silicon vendor simulation tool.

According to evaluations, a three-slot VLIW DSP core with conventional word addressable memory demands an average cycle count higher by a factor of 1.44–1.98 and an average instruction count higher by a factor of 1.22–1.62 than with the proposed parallel memory system [128]. These improvements are mainly resulted due to optimal data alignment and ordering in parallel memory system.

As drawbacks, the studied parallel memories require larger silicon area by a factor of 1.14–1.93 and longer total memory access delay by a factor of 1.22–2.36 than word addressable memory [128]. Moreover, the power consumption per a memory access is estimated to be 1.30–2.77 times higher with parallel memories. Nevertheless, the total power consumption is also affected with two other factors. Firstly, the memory access operations are reduced with parallel memories on average by a factor of 1.05–1.35 and secondly, power consumption of the DSP core is decreased due to the above mentioned total instruction count reduction with parallel memories.

As another performance evaluation, Kuzmanov compares linear memory to two-stage memory configuration containing linear main memory and parallel memory as local storage [74]. The linear memories have one to four bytes wide word length whereas parallel memories support 8 x 8 or 16 x 16 byte accesses. With rectangle shaped accesses, the two-stage memory achieves transfer speedup from 0.91 to 8.10. The speedup varies according to temporal reuse of data, memory module count, and access alignment according to word boundaries. With no temporal reuse, two-stage memory configuration is naturally worse.

A H.264 deblocking filter with parallel memory is compared to register based reference implementation in terms of gate count and clock cycles in [87]. The proposed system complexity is shown to be 45–49% of the reference implementation. Moreover, the performance is increased by 8–55% when utilized 0.35 μm CMOS technology compared to (newer) 0.25 μm technology used in the reference design.

A parallel memory system with eight memory modules is compared to conventional memory with a single memory module in [139]. Image scaling with a simple interpolation algorithm is evaluated. The speed-up factor of the parallel memory system over the conventional memory is from 6.5 to 8 in terms of clock cycles.

4. Configurable Parallel Memory Architecture

A typical problem with the previously presented parallel memory implementations is that they are application specific with fixed access formats supported in hardware. This ensures the fastest speed, but requires redesign if other formats need to be included. To address this issue, the Configurable Parallel Memory Architecture (CPMA) [72] can change access formats and the required module assignment and address functions at run time. This Chapter summarizes [P1]–[P5].

4.1 Architecture

A block diagram of the Configurable Parallel Memory Architecture is depicted in Figure 4.1. The inputs to CPMA are *Virtual Address* and *Command* (write, read). The Virtual Address includes a *Page Address*, which is a pointer to a *Page Table*. *Page Offset* directly indicates the *scanning point r* for the parallel memory. The *Row Address* and the Access Functions are fed from the Page Table to the Address Computation unit. In addition, memory banks and a Data Permutation unit are required for CPMA operation. The CPMA concept allows the memory banks to be any kind of RAM. In the following, single port DRAM is considered.

CPMA address computation is implemented using several configurable computation units in parallel. One unit is dedicated for each type of access formats, module assignment functions, and address functions that CPMA supports. The presented implementation aims at the fastest operation still maintaining sufficient configurability; with pipelining used to enhance the system throughput [138]. Moreover, a configurable Data Permutation unit was designed to conform to system specifications [139].

CPMA is designed to be configurable both at design time and at run time. In the hardware implementation, many parameters are VHDL generics, which means that the number of memory blocks, size of a memory module, width of the data bus, and width of input buses are adjustable. Except for the memory modules, all the required blocks, logic, and buses are implemented with logic synthesis from VHDL. In the following, the run-time configurability and some of the CPMA blocks are described in more detail.

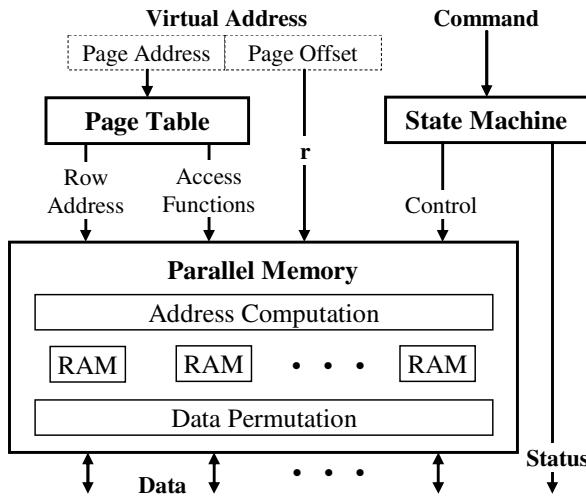


Figure 4.1. Configurable Parallel Memory Architecture.

4.2 Page Table

The application software can change the contents of the Page Table at run time. The Page Address is a pointer to a specific row in the Page Table which determines the CPMA functionality during that memory access. The contents of a single Page Table row are explained in the following. Table 4.1 depicts the Page Table with two example rows shown for image processing (pixel data).

4.2.1 Row address

Row address has a similar function as a page number in a normal memory architecture [49]. Separate pages can be utilized to construct several module assignment functions $S(i, j)$ allowing different conflict free access formats as illustrated in Table 4.1.

4.2.2 Length of coordinates

Length of coordinates L_i and L_j restrict the addressable two-dimensional memory area, the scanning field, in dimensions i and j . The values are scaled according to application.

Table 4.1. Page Table structure.

Page Address	Row Address	L_i	L_j	F	$S(i, j)$	$a(i, j)$
:	:	:	:	:	:	:
6	1	22	18	G(2, 1, 3)	1, 1, 3, 5	1, 0, 5
7	3	44	36	CR(3, 2, 3, 6)	1, 2, 1, 8	1, 0, 8
:	:	:	:	:	:	:

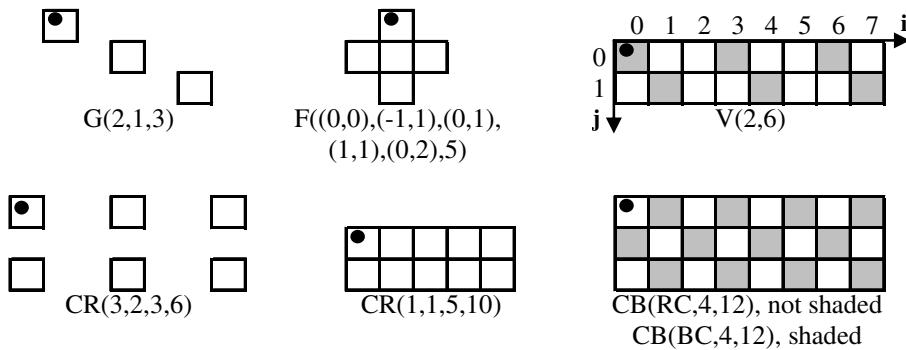


Figure 4.2. Examples of the implemented access formats.

4.2.3 Access format function

A Page Table row contains an access format type indicator and the parameters for constructing the requested access format (access format function F). All the templates introduced in Section 3.3 can be formed.

There are five types of access formats, of which some examples are given in Figure 4.2. Squares in an access format depict the relative positions of accessed data elements. A scanning point is marked with a dot “•”. The implemented access formats and the parameters required by a Page Table are described in the following [P3].

1. *Generate format* $G(a_i, a_j, p)$, where the parameters a_i and a_j are jumps in respective (i, j) directions and p gives the total number of accessed data elements. The parameter a_i may be any integer whereas a_j has to be a nonnegative integer. All the straight templates, for example rows, columns, and diagonals, can be formed using the *generate* format.
2. *Crumbled rectangle format* $CR(a_i, a_j, p_i, p)$, where the parameters a_i and a_j are jumps to i or j directions respectively, p_i shows the number of columns used, and p again corresponds to the number of data elements. All the rectangles and crumbled rectangles can be accessed using this format.
3. *Chessboard format* $CB(a, p_i, p)$, where the parameter $a \in \{RC, BC\}$. That is, a is either a red chessboard or a black chessboard. The parameter p_i is the number of accessed data elements in each row and p the total number of data elements.
4. *Vector format* $V(a, p)$, where the parameter a is the stride of the vector and p is used as in the previous formats.
5. *Free format* $F((i_0, j_0), (i_1, j_1), \dots, (i_{h-1}, j_{h-1}), p)$, where the parameter p is utilized as before and the other parameters are the coordinate offsets (i, j) from the scanning point r . The parameter i_k may be any integer whereas j_k has to be nonnegative. The maximum number of free format pixels h directly influences the system bus widths

and the Page Table size. The *free* format can access arbitrary memory patterns within the implementation limits. All the other access formats could be formed using this format, but, for the general case, it would be very expensive in terms of implementation area. This is due to the Page Table size requirements and resulting system bus widths.

When a coordinate value exceeds the scanning field, a *fit* operation has to be performed to return the coordinate to an acceptable value. The details of this operation are application specific. Vector access is quite similar to the generate format except that the fitting techniques are different. If necessary, the vector format wraps the access to the beginning of the *next row*, as depicted in Figure 4.2. The *generate* as well as all the other access formats wrap to the beginning of the *same row*. This difference is due to the fact that vector accesses are intended for one-dimensional memory addressing whereas the other accesses use two-dimensional memory space.

The data elements are accessed starting from the scanning point and continuing primarily from left to right and secondarily from top to down. However, exceptionally the order of the elements in the *generate* format depends on the jump direction whereas the *free* format element arrangement is based on the original order of the coordinate parameters. The same scanning point is used for the both chessboard formats, as shown in Figure 4.2. Thus, when using the black chessboard, the scanning point is exceptionally located outside of the accessed data elements. For all the access formats, the lengths of coordinates in i and j -directions, L_i and L_j , restrict the distance between the first and the last element. The ranges are $[-L_i, 2L_i-1]$ in i -direction and $[0, 2L_j-1]$ in j -direction.

4.2.4 Module assignment function

There are three kinds of module assignment functions $S(i, j)$ implemented in the system.

Linear module assignment functions are formed as [114]

$$S(i, j) = (a \cdot i + b \cdot j) \bmod N . \quad (3)$$

The parameters a and b are nonnegative integers whereas N is a positive integer [P1]. The number of memory modules N may also be a prime number that enables many useful straight access formats. In Table 4.1, a page address 6 utilizes linear function shown in Figure 3.7.

XOR-schemes always utilize a power of two ($N = 2^n$) memory modules. XOR-schemes are defined as [32]

$$S(i, j) = \mathbf{AI} \oplus \mathbf{BJ} , \quad (4)$$

where $I = (i \bmod 2^u)_2$ and $J = (j \bmod 2^v)_2$. \mathbf{A} and \mathbf{B} are $(n \times u)$ and $(n \times v)$ matrices containing binary numbers. The elements of \mathbf{A} and \mathbf{B} matrices are marked in an unordinary fashion. The top left element of matrix \mathbf{A} is $a_{n-1, u-1}$ and the bottom right is $a_{0, 0}$. The vector

addition and multiplication in (4) are achieved by bit-wise logical XOR and AND operations, respectively. The values of the binary matrices \mathbf{A} and \mathbf{B} are required to be stored in the Page Table. Additional discussion and examples can be found in [P4].

Diamond schemes are multiperiodic functions. Within diamond schemes, the construction of a basic domain and the two permutations π_1 and π_2 need to be determined. An axially oriented basic domain forms the initial periodicity with either linear function (5) or XOR-schemes (6).

$$S_b(i_b, j_b) = (a \cdot i_b + b \cdot j_b) \bmod N_b. \quad (5)$$

$$S_b(i_b, j_b) = \mathbf{A}I_b \oplus \mathbf{B}J_b. \quad (6)$$

The side lengths of a basic domain, N_1 and N_2 , are restricted to powers of two because of hardware simplicity. Due to the restriction, the memory module count in the basic domain (N_b in (5)) is also desirable to be a power of two. Naturally, N_b has to be a power of two number that is equal or less than the real memory module count in the implemented hardware. The variables (i_b, j_b) are the respective basic domain coordinates derived from the coordinates (i, j) as

$$\begin{cases} i_b = i \bmod N_1 \\ j_b = j \bmod N_2. \end{cases} \quad (7)$$

The permutations π_1 and π_2 change the order of the basic domain when i and j increase, respectively. The permutations π_1 and π_2 are formed using simple functions

$$\begin{cases} \pi_1(x) = (x + c_1) \bmod N_b \\ \pi_2(x) = (x + c_2) \bmod N_b. \end{cases} \quad (8)$$

Permutations in (8) change the order of the module numbers x relative to the periodic rectangle next to it. The parameters c_1 and c_2 are nonnegative integers and the module count is N_b as in (5) because all the periodic rectangles use as many memory modules as the basic domain. Totally, the values of N_1 , N_2 , N_b , c_1 , c_2 as well as either a and b or \mathbf{A} and \mathbf{B} are stored to the Page Table for constructing a proper diamond scheme. Further details as well as example schemes are presented in [P2].

4.2.5 Address function

The address function $a(i, j)$ has to be determined so that two data elements are not located under the same address of the same memory module. In other words, an address function has to be suitable for the used module assignment function. Two address functions have been implemented in the CPMA framework.

The address function mainly intended for linear module assignment functions and XOR-schemes is formed as

$$a(i, j) = b + \lfloor (i + j \cdot L_i)/N \rfloor, \quad (9)$$

where b is a nonnegative integer whereas L_i and N are positive integers [P1]. In the Page Table, L_i is determined in the L_i column. The scanning field can be divided to multiple parts with the parameter b . A page address 6 in Table 4.1 employs address function (9) with $b = 0$ and $N = 5$.

The address function developed for the diamond schemes is defined as

$$a(i, j) = b + \left\lfloor \frac{i}{c_1} \right\rfloor + d \cdot \left\lfloor \frac{j}{c_2} \right\rfloor. \quad (10)$$

Parameters b and d are nonnegative integers but c_1 and c_2 are restricted to powers of two in (10). As in (9), the scanning field can be separated to several regions with the parameter b . An example function and its intended usage with diamond schemes are given in [P2].

4.2.6 Page Table usage

The process for specifying an algorithm for CPMA includes defining the contents of the associated Page Table row. First, the required access formats for the algorithm are evaluated. A module assignment function for conflict free access is chosen and a suitable address function for the module assignment has to be found. Finally, the gathered information is stored in the Page Table.

4.3 Address Computation

A block diagram of the Address Computation unit is depicted in Figure 4.3. The outputs of the unit are the DRAM column addresses. The DRAM row address is passed straight to the memory modules.

Inside the unit, the *Format Control* unit [P3] specifies the coordinates of the data elements accessed in the scanning field. The *Module Assignment* unit [P1], [P2], [P4] computes the corresponding memory modules that contain the accessed data. The *Address Decode* unit [P1], [P2], in turn, computes the physical addresses for each memory module. Finally, the *Address Select* units [P1] direct the physical addresses to the proper memory modules.

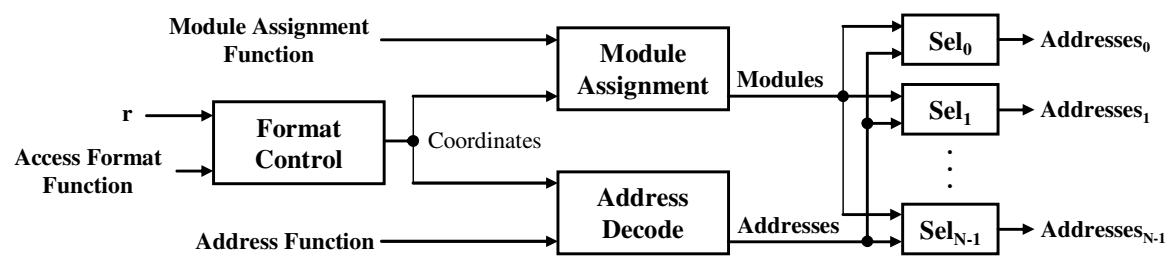


Figure 4.3. Address Computation unit.

5. Parallel Memory Architecture for Arbitrary Stride Accesses

Digital signal processing and telecommunications benefit from the use of strides. Some examples are vector/matrix computation, FFT, and Viterbi algorithm [44], [53], [123]. Due to applicability of stride accesses in several algorithms, Slingerland [116] as well as Hennessy and Patterson [49] recommend stride access support in multimedia SIMD extensions. MOM, CSI, Impulse, SMC, Imagine, and VIRAM allow stride accesses but none of them utilizes parallel memory architecture. In addition, these architectures have restrictions as well as complexity overhead as discussed in Section 2.2.2.

Traditionally, the research on parallel stride accesses has concentrated on module assignment functions (skewing schemes) that try to ensure conflict free parallel data accesses to a set or maximal amount of access strides [42], [44], [53], [95], [109], [113], [117], [135], [136]. Normally, the scheme can be changed only at design time. This Chapter considers merely constant stride accesses and data patterns having equal amount of accessed data elements as the number of memory modules. Moreover, access locations of the data patterns are supposed to be unrestricted. Unfortunately, no single skewing scheme can be found that allows unrestrictedly placed conflict free accesses for all the constant strides [P8].

Arbitrary stride access capability with interleaved memories is described in previous research [43], [45], [46] where the skewing scheme is changed at run time according to the currently used stride. This Chapter presents the improved schemes which are adapted to parallel memories. The proposed novel parallel memory architecture allows conflict free accesses with all the constant strides which has not been possible in prior application specific parallel memories. Moreover, the possible access locations are unrestricted and the data patterns have equal amount of accessed data elements as the number of memory modules.

This Chapter begins with a real-time image scaling application example which demands arbitrary stride accessibility with parallel memories [P7]. After that, the proposed novel parallel memory system is presented [P8], [P9].

5.1 Application example: image scaling

In resampling, a discrete image is transformed to a new set of coordinate points. Some resampling examples are image scaling, spatial distortion correction, and image rotation. Resampling can conceptually be divided into two sequential processes. At first, a discrete

image is interpolated to a continuous image. Then, the interpolated image is sampled. In practical implementations, the interpolation and sampling are often combined so that the image is interpolated at only those points that are sampled.

5.1.1 Image scaling principles

In the following image scaling theory, pixels are interpolated only at the sampling points. The interpolated image sampling point distances are horizontal distance Δx and vertical distance Δy . Normally, these distances are defined as

$$\begin{aligned}\Delta x &= \frac{\text{original_image_width}}{\text{scaled_image_width}}, \\ \Delta y &= \frac{\text{original_image_height}}{\text{scaled_image_height}}.\end{aligned}\tag{11}$$

As an example interpolation function, Cubic B-splines [51] use 4 x 4 original image pixels to filter one scaled pixel. One-dimensional cubic B-splines determines scaling coefficients as follows [51], [86]:

$$\text{Coeff}(s) = \begin{cases} (1/2)|s|^3 - |s|^2 + 2/3, & 0 \leq |s| < 1 \\ -(1/6)|s|^3 + |s|^2 - 2|s| + 4/3, & 1 \leq |s| \leq 2 \\ 0, & |s| > 2. \end{cases}\tag{12}$$

The distance between a scaled pixel location and an original pixel location is denoted by s . Coefficient is multiplied with the respective original image pixel.

Figure 5.1 depicts two-dimensional interpolation with 4 x 4 pixels by utilizing row-column decomposition. The sixteen black dots describe the original image pixels. The 2-D interpolation is computed by performing the one-dimensional interpolation over the rows first. Scaling coefficients are computed by utilizing the x -direction distances $s_{x0}, s_{x1}, s_{x2}, s_{x3}$ and some suitable interpolation function, for example (12). The four grey points in Figure 5.1 depict achieved values. These intermediate interpolation results and the y -direction distances $s_{y0}, s_{y1}, s_{y2}, s_{y3}$ are used for the final column-wise 1-D interpolation that produces the scaled pixel P in the middle of Figure 5.1.

5.1.2 Scaling evenly divisible images

Most 2-D image sizes are, in practice, divisible by some integer other than one [P7]. This knowledge of evenly divisible images can be utilized during the scaling.

In the following, the *original block* is always a part of the original image. The width and/or height of the original image are multiples of the original block. In addition, the original block can be scaled with the same scaling ratio as the original image. A *scaled block* is

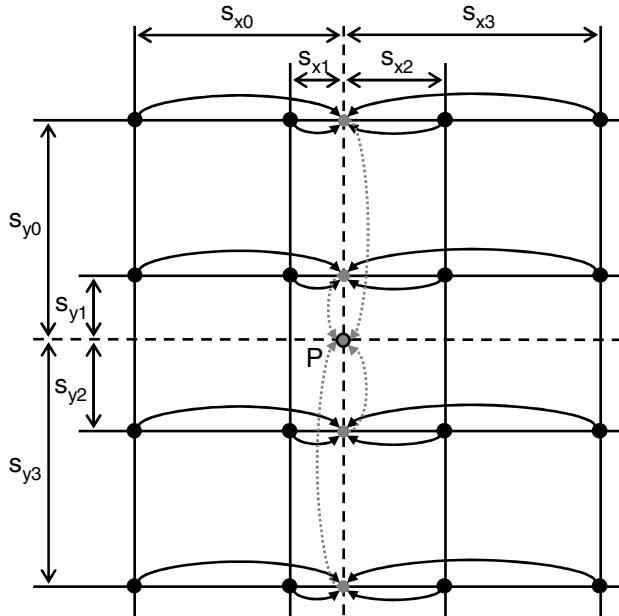


Figure 5.1. Row-column decomposition of 2-D interpolation.

scaled from the original block with the scaling ratio. Naturally, the width and height of the original and the scaled block must be positive integers.

Figure 5.2 depicts an original block size $c_1 \times d_1$ that is scaled to size $c_2 \times d_2$. The original image is scaled with the same ratios and, in this example, the width a_1 is three times and the height b_1 twice the size of the original block. Therefore, the sampling point distances Δx and Δy are the same in the both scaling examples. Hence, block-level parallelism can be utilized when scaling the original image. In Figure 5.2, if six units designed for block scaling are processing in parallel, each can scale a separate part of the original image.

In general, the same scaling can be done in $\text{gcd}(a_1, a_2)$ times in x -direction and $\text{gcd}(b_1, b_2)$ times in y -direction, where gcd is the greatest common divisor. This method is applicable for both upscaling and downscaling providing that the interpolated image sampling point distance is defined as shown in (11). All the interpolation methods shown in [86] fulfill the requirement and the algorithms can be parallelized in block-level. This is not the case with all the interpolation methods. For example, winscale overlap stamping method [69] exceptionally defines the distances as follows:

$$\begin{aligned}\Delta x &= \frac{\text{original_image_width} - 1}{\text{scaled_image_width} - 1}, \\ \Delta y &= \frac{\text{original_image_height} - 1}{\text{scaled_image_height} - 1}.\end{aligned}\tag{13}$$

This overlap stamping technique does not fulfill the requirement and cannot be parallelized with the proposed method.

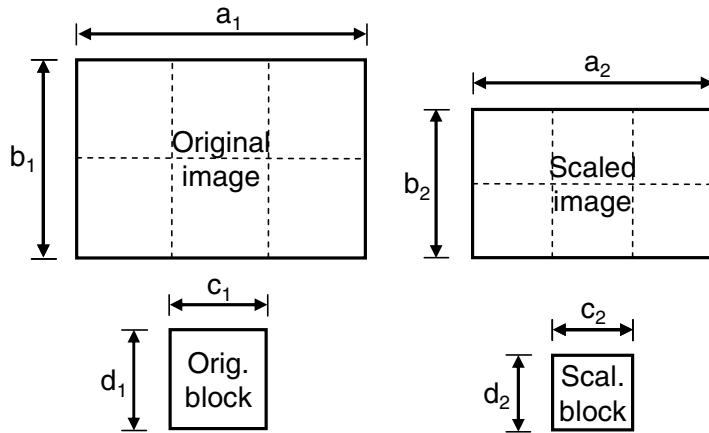


Figure 5.2. Evenly divisible images scaled down.

5.2 Block level parallel architecture for image scaling

The proposed block-level parallel architecture for image scaling is depicted in Figure 5.3 [P7]. The architecture has N Filter units, N being the parallelization factor. In addition, a *Coefficient Generator* unit and two *Parallel Memory* units are needed. Same coefficients are fed to all the Filter units. However, each Filter unit scales a separate image block. The Filter and Coefficient Generator units are implemented to comply with the interpolation algorithm. The simplest possible Filter unit can be implemented with a MAC unit when one pixel is accessed at a time to each Filter unit. However, more complex structures can also be used to further increase performance. Coefficients can be calculated in advance as done in [88] or on the fly using the Coefficient Generator unit.

The parallelization can be increased by adding more Filter units. However, the block count in an image restricts the maximum parallelism. This architecture is suitable for all the interpolation algorithms that fulfill the block-level parallelism restrictions mentioned previously. All the filters utilize the same memory address space. The entire original

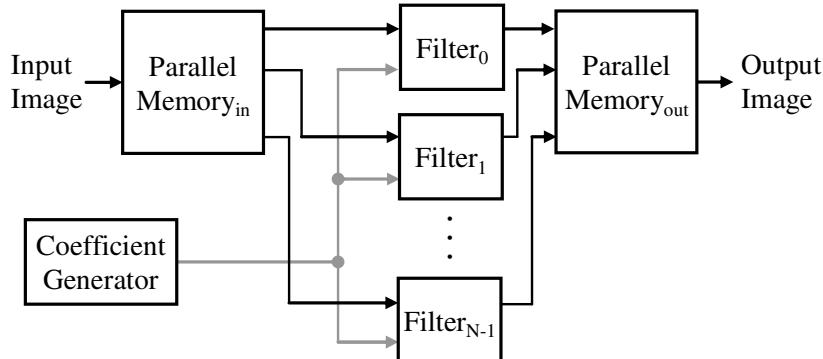


Figure 5.3. Block-level parallel architecture for image scaling.

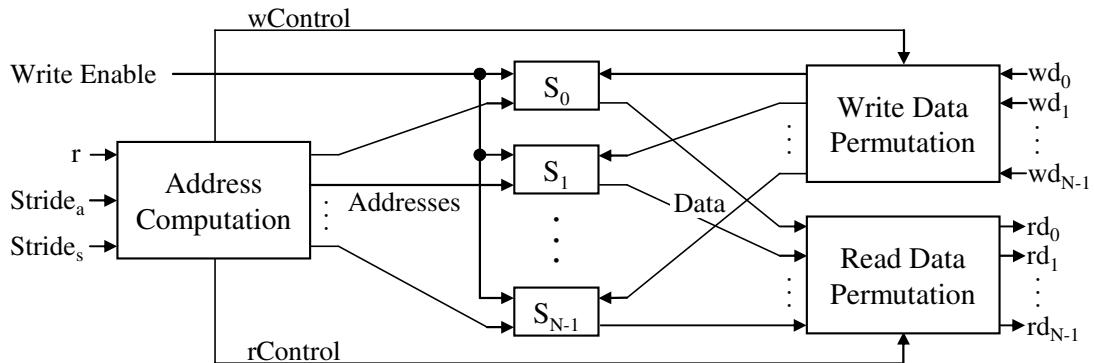


Figure 5.4. Parallel memory architecture for arbitrary stride accesses.

image is stored to the Parallel Memory_{in}, optionally in smaller pieces multiplexed in time to save memory.

In the following, an image scaling implementation utilizing one pixel at a time in each Filter unit is supposed. Consequently, for Parallel Memory_{in}, a row access format is required for writing and a constant stride access format for reading the original image. The formats for read and write are reversed for Parallel Memory_{out}. The stride length used for read accesses equals the used original block size and the write stride equals the scaled block size. The block sizes depend on the original and scaled image sizes and the filter count N . Since original and scaled image sizes can be arbitrary integers from an implementation specific range, the parallel memory needs control that allows conflict free accesses for all the strides.

5.3 Parallel memory architecture

A block diagram of the proposed parallel memory architecture is shown in Figure 5.4. The referenced data locations are defined with the *scanning point* r and the stride length $Stride_a$. The skewing scheme is formed according to the $Stride_s$ input. In write operation, *Write Enable* is asserted simultaneously with the data to be written $wd_0, wd_1, \dots, wd_{N-1}$. The two *Data Permutation* units permute the written or read data according to the control signals *wControl* and *rControl* from the *Address Computation* unit. [P8]

The proposed parallel memory system is implemented in VHDL. The system is configurable both at design time and at run time. Most design-time parameters like the memory module count ($N = 2^n$), size of a memory module, and width of buses are adjustable. The run-time configurability refers to the ability to change the skewing scheme at run time.

5.3.1 Functionality

Let strides be divided to groups by defining as

$$\text{stride} = \sigma \cdot 2^s , \quad (14)$$

where σ is an odd positive integer (i.e. $\sigma \in \{1, 3, 5, \dots\}$) and s a nonnegative integer (i.e. $s \in \{0, 1, 2, \dots\}$). All the strides can be formed by changing σ and s . Any skewing scheme that allows a conflict free access for a specific stride 2^s also provides a conflict free access for strides (14) with all σ [45]. Thus, each stride group has unique s and all σ values. Therefore, by finding conflict free storage schemes for each of the stride groups (i.e. single scheme for a single group) enables conflict free access for arbitrary strides.

The proposed module assignment for *even* strides ($s > 0$) is defined as [P8]

$$S(i) = i[n+s-1:s] \oplus i[n-1:0] , \quad (15)$$

where a data element location i is in binary form [MSB : LSB] and \oplus denotes bit-wise logical XOR operation. The implementation of (15) demands defining n and s . In a particular implementation, the number of memory modules $N = 2^n$ is a constant. Respectively, s is determined by a stride (14). Therefore, n is defined at design time and s at run time.

For *odd* strides ($s = 0$), the conventional *low-order interleaved scheme* is formed as [P8]

$$S(i) = 0 \oplus i[n-1:0] = i[n-1:0] . \quad (16)$$

All the constant strides are conflict free by using (15) and (16) with practical memory module count [P8], [136].

The proposed and a feasible address function for (15) and (16) is defined as

$$a(i) = \lfloor i/N \rfloor . \quad (17)$$

In (17), the address in a memory module $a(i)$ is received directly from the MSBs of the data element location i when $N = 2^n$ and no additional hardware is required.

5.3.2 Address Computation

Figure 5.5 depicts a block diagram of the Address Computation unit. The outputs of the unit are the memory module addresses for each of the memory module. Moreover, control signals (rControl, wControl) are formed for the Data Permutation units.

Inside the unit, the *Format Control* unit specifies the accessed data locations. Respectively, the *Scheme Determination* unit defines the used skewing scheme. The *Module Assignment* unit computes the specific memory modules that contain the accessed data and the *Address Decode* unit, in turn, computes the physical addresses for each memory module. The *Permutation Control* unit reorders the memory module numbers.

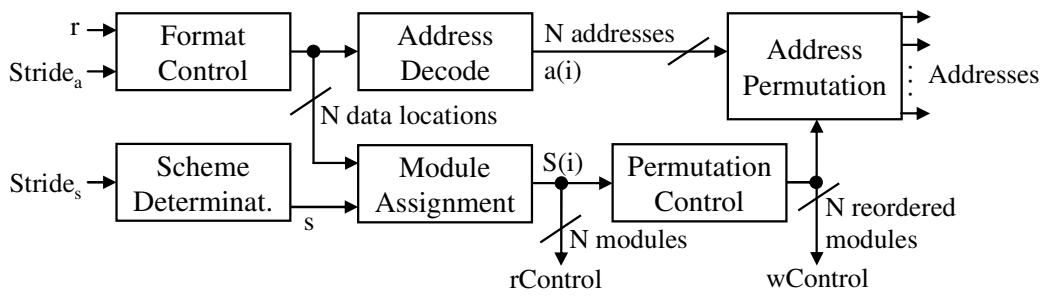


Figure 5.5. Address Computation unit.

According to the reordered module numbers, the *Address Permutation* unit directs the physical addresses to the proper memory modules.

6. Parallel Memory Architecture for Image Downscaler

Spatial scalability refers to the feature of image representation in different sizes or spatial resolutions. Real-time spatial scalable video coding with several frame resolutions demands lots of computation power for scaling. In addition, video resolutions used in typical applications tend to increase. Nowadays, consumer video cameras capturing frames with the size of 1440 x 1080 pixels in real time are available [118]. High performance downscaling is required when the post processed video resolution is smaller.

This Chapter presents a hardware implementation capable of downscaling large resolution YUV 4:2:0 color format video in real time. The original and scaled image sizes are fixed at design time. A new scaling ratio demands hardware reconfiguration and the ratio can be from a nearly arbitrary range. Bilinear interpolation is used in the proposed implementation. In the implementation, all the four original image pixels necessary in bilinear interpolation are processed in parallel [P10].

The proposed downscaler utilizes an application specific parallel memory system. The proposed parallel memory is designed to optimally support the downscaling hardware. This enables maximum performance with minimum complexity. However, the reusability in other designs may be difficult.

6.1 Downscaling with bilinear interpolation

Figure 6.1 illustrates the bilinear interpolation [P6]. At the sampling point, the scaled pixel value P can be filtered by the original image pixels $C0–C3$ as

$$P = W0 \cdot C0 + W1 \cdot C1 + W2 \cdot C2 + W3 \cdot C3. \quad (18)$$

Distances between the scaled pixel P and the original pixel $C0$ are depicted by Δw and Δh in x and y directions, respectively. Distance from $C0$ to $C2$ and from $C0$ to $C1$ equals 1.0. The coefficients $W0–W3$ are defined as follows

$$\begin{cases} W0 = (1 - \Delta w) \cdot (1 - \Delta h) \\ W1 = (1 - \Delta w) \cdot \Delta h \\ W2 = \Delta w \cdot (1 - \Delta h) \\ W3 = \Delta w \cdot \Delta h. \end{cases} \quad (19)$$

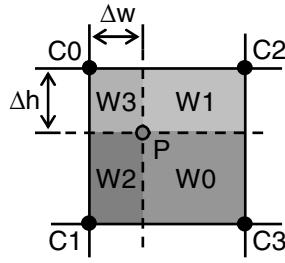


Figure 6.1. Scaling with bilinear interpolation.

The coefficient values W_0 – W_3 correspond to the areas shown in Figure 6.1.

Bilinear interpolation consists of four operation phases [69]. In *pixel moving*, the location of the scaled pixel P is calculated from the previous scaled pixel location utilizing the sampling point distances Δx and Δy (11). *Memory accesses* are needed to load the four original image pixels C_0 – C_3 around the scaled pixel location. *Weighting factor calculation* defines the coefficients W_0 – W_3 (19). Finally, *pure filter* operations are performed in order to calculate the scaled pixel value P (18).

6.2 Downscaler architecture

The original and scaled image sizes are fixed at design time to make the pixel moving and weighting factor calculation operations simpler. For example, some of the parameters can be calculated with accurate floating point arithmetic and stored in ROM beforehand.

Evenly divisible image property introduced in Section 5.1.2 can also be utilized in the proposed downscaler architecture. Each of the image blocks in evenly divisible images has the respective sampling point locations and the same weighting factors. Therefore, less memory is required in ROM based implementations. The same aspect was partly noticed with windowed sinc interpolation function in [88]. In the proposed implementation, a trade-off between ROM consumption and logic is taken into account in bilinear interpolation based scaling [P10].

A block diagram of the proposed downscaler architecture is shown in Figure 6.2. The original image pixels are received as input, four pixels at a time. The scaled image is given to the output pixel by pixel. Scaling operations are performed as follows. Pixel moving operations are done in the *Main Controller* unit and memory accesses are handled in the *Parallel Memory* unit. Weighting factor calculation is carried out in the *Coefficient Generator* unit. Finally, pure filter operations are performed in a *Filter* unit. Four original image pixels read from the Parallel Memory unit and four coefficients computed in the Coefficient Generator unit are used to filter one scaled pixel at a time.

The downscaler is designed to be design-time configurable. In the hardware implementation, the original and scaled image sizes as well as the weighting factor calculation bit widths are adjustable. According to these parameters, the ROM contents are automatically calculated at design time. All the required logic, ROM, and buses are

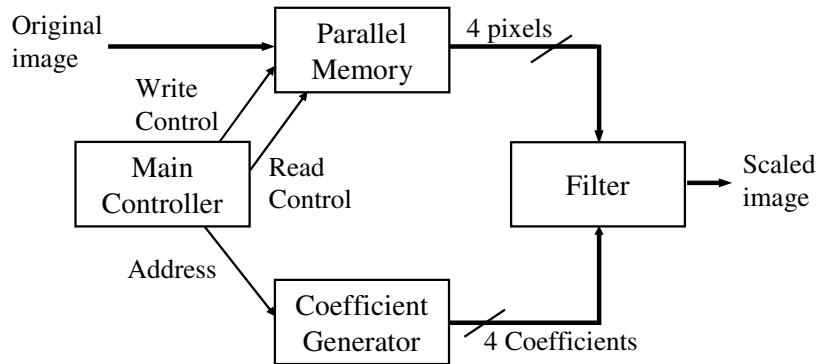


Figure 6.2. Downscaler architecture.

implemented with logic synthesis from VHDL. The ready-made memory modules in FPGA are used as data memory.

6.3 Parallel memory architecture

The block diagram of the Parallel Memory unit used in this implementation is shown in Figure 6.3. The unit includes two *Address Computation* and two *Data Permutation* units as well as four dual port memory modules S_0, S_1, S_2, S_3 . Each memory module is capable of simultaneous read and write operations. Depending on the *write scanning point* (w_i, w_j) or the *read scanning point* (r_i, r_j), the respective Address Computation unit computes the addresses and directs them to the appropriate memory modules. The Data Permutation units organize the data into correct order according to the control signals received from the Address Computation units. In the write operation, four data elements wd_0, wd_1, wd_2, wd_3 are permuted and stored in the memory modules. With reading, the data elements rd_0, rd_1, rd_2, rd_3 are outputs of the unit.

Each memory module is one byte (pixel) wide. Two original image rows are demanded for filtering and two rows for buffering. Therefore, memory capacity of the four original image rows is required. Circular/module memory addressing [129] is used in y -direction.

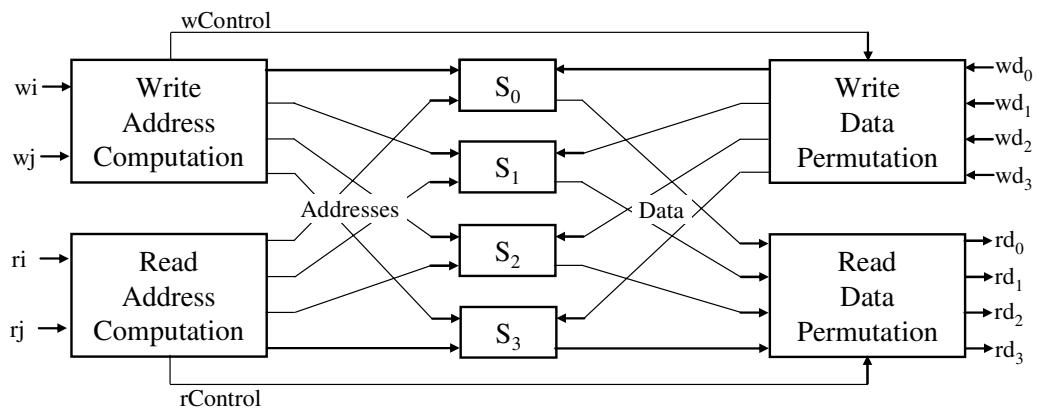


Figure 6.3. Parallel Memory unit.

	0	1	2	3	4	5	6	7	i
0	0	1	2	3	0	1	2	3	
1	2	3	0	1	2	3	0	1	
2	0	1	2	3	0	1	2	3	
3	2	3	0	1	2	3	0	1	j

	0	1	2	3	4	5	6	7	i
0	0	0	0	0	1	1	1	1	
1	2	2	2	2	3	3	3	3	
2	4	4	4	4	5	5	5	5	
3	6	6	6	6	7	7	7	7	j

Figure 6.4. Used module assignment and address functions.

6.3.1 Functionality

In the proposed implementation, four bytes wide row access format is needed for writing and a square (2×2) access format for reading. The unrestricted placement set is required for reading, meaning that the square access format can be placed anywhere in the scanning field. For writing, an unrestricted placement set is not required, but the row access format needs to cover the scanning field.

For the proposed four memory module implementation, a suitable module assignment function for the two access formats is

$$S(i, j) = (i + 2j) \bmod 4. \quad (20)$$

A feasible address function for the module assignment function is

$$a(i, j) = \lfloor (i + j \cdot L_i) / 4 \rfloor, \quad (21)$$

where L_i is the width of the scanning field. Four memory modules with capacity of four original image rows are occupied in this case. In addition, although the width of the original image can be arbitrary, the size of the memory module is normally power of two. Therefore, the width of the scanning field L_i can be restricted to power of two without extra memory consumption compared to situation when L_i equals to the original image width. Naturally, L_i has to be a power of two number that is equal or greater than the original image width.

The used module assignment and address functions are illustrated in Figure 6.4. The width of the scanning field L_i is defined as eight and 8×4 data elements are displayed. A single data element is stored in a memory module with a number shown in the corresponding table cell on the left-hand side of Figure 6.4. The respective address in the memory module is shown on the right-hand side. The row and square (2×2) access formats are enclosed by a fat line in Figure 6.4.

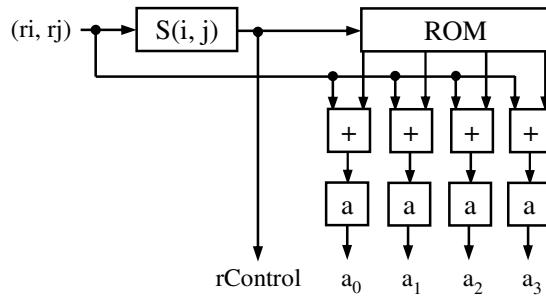


Figure 6.5. Read Address Computation unit.

6.3.2 Write Address Computation and Data Permutation

As mentioned above, the row access format used for writing needs only to cover the scanning field. Therefore, the accesses in the scanning field can be restricted to non-overlapped locations starting from the top left corner of the scanning field. Due to the address function (21), the same address is fed to each memory module. Only hardwired masking and shifting is used in the Write Address Computation unit since L_i is restricted to the power of two with (21).

There are two data permutation possibilities with the utilized module assignment function (20) and the non-overlapped covering placement set. For this reason, the logic in the Write Data Permutation unit includes four 2-input multiplexers with each of the input and output widths being eight bits.

6.3.3 Read Address Computation and Data Permutation

As stated before, the read operation demands a square access format with an unrestricted placement set. Therefore, the addresses have to be calculated separately for each memory module. Figure 6.5 depicts the Read Address Computation unit in more detail. The outputs of the unit are addresses for each memory module (a_0, a_1, a_2, a_3) and rControl signal for controlling the Read Data Permutation unit. At first, the module number of the scanning point is calculated with the module assignment function (20). The function can be refined as

$$S(i, j) = (i + 2j) \bmod 4 = ((i \bmod 4) + (2j \bmod 4)) \bmod 4 = ([i_1, i_0] + [j_0, 0]) \bmod 4, \quad (22)$$

where $[i_1, i_0]$ and $[j_0, 0]$ mean the two least significant bits. It should be noted that only a two bit adder is required. After calculation of the module number, four relative (i, j) coordinates are determined with a ROM (Figure 6.5) having a size of four 8-bit locations. These four i and four j values are added in parallel with the scanning point using eight (2×4) adders. Finally, addresses are calculated utilizing the same address function $a(i, j)$ (21) as with the Write Address Computation unit. Merely hardwired masking and shifting is utilized for address calculation.

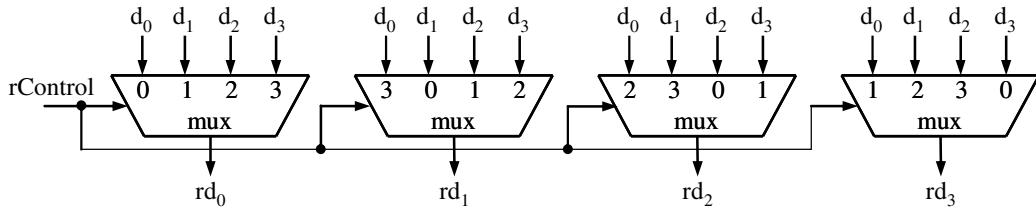


Figure 6.6. Read Data Permutation unit.

The *isotropic* property [38] of the used linear module assignment function (20) can be utilized for implementing the Read Data Permutation unit. At most N different permutations have to be carried out with an isotropic module assignment function when the number of accessed elements is N [38]. Therefore, the implementation supports four different permutations. The Read Data Permutation unit is depicted in more detail in Figure 6.6. With *rControl* signal the correct data is selected for each memory module. Four 4-input multiplexers are required for the implementation.

7. Analysis

The three parallel memory architectures differ from each other in many ways. Parallel memory architecture for the image downscaler (called here Downscaler PMA) is a case example of a traditional type of application specific parallel memory. Parallel memory architecture for arbitrary stride accesses (called Stride PMA) allows some run-time configurability for a skewing scheme. In turn, the Configurable Parallel Memory Architecture (CPMA) offers very wide run-time configurability properties.

This Chapter mainly analyses the differences between these three parallel memory architectures. For reference, also some other parallel memory architectures are shortly analyzed in a similar manner.

7.1 Parallel memory feature analysis

The provided run-time support for different module assignment and address functions varies between the proposed parallel memory architectures. Downscaler PMA utilizes a single skewing scheme and address function. This is the normal way in parallel memory implementations [53], [60], [74], [129]. In Stride PMA, a skewing scheme can be changed at run time according to the used stride. The same property is supported by Harper's interleaved memories [45], [46]. In [91], the used skewing scheme can be decided between four possibilities at run time. However, in all these three cases, the skewing schemes can only be changed in such a concise way that a single address function in each system is compatible for all the possible schemes. Therefore, the address function cannot be changed. CPMA is the only parallel memory architecture allowing both the module assignment and address function to be widely changed at run time.

When a module assignment or address function is changed, the data located already in the memory modules is naturally invalidated. Therefore, the functions can only be changed before writing new data to memory. This restriction can be avoided by dividing the address space into several regions that can utilize different skewing schemes and address functions. In Tanskanen's parallel memories [22], [130], two different memory regions with separate module assignment functions are used. The division is done by adding a constant offset to the used address function in the second address space. In CPMA, the address space division can be done by using different row addresses in the Page Table (Section 4.2.1). Moreover, arbitrary offsets can be added to address functions to divide a single page to multiple parts (Section 4.2.5).

The diversity of conflict free access formats also differs between the implementations. Downscaler PMA supports a separate template for read and write operations. Stride PMA and Harper's interleaved memory [45], [46] allow arbitrary stride. Kuzmanov supports only rectangle access patterns [74] whereas few different access formats are supported in Tanskanen's memories [22], [130]. Park [101] allows several (crumpled) straight and (crumpled) rectangle templates and HiPAR-DSP supports many crumpled rectangle formats [112]. CPMA allows several types of parametrizable templates [P3]. Each of the template types allows multiple access formats to be formed including all the access formats described in Section 3.3.

A scanning field determines the addressable memory area. In 2-D addressing, length of coordinates L_i and L_j restrict the scanning field. In Downscaler PMA, L_i and L_j are defined at design time and cannot be changed at run time. The same holds for HiPAR-DSP [112], as well as Tanskanen's [22], [130] and Kuzmanov's [74] memory systems. When 1-D addressing is used as in Stride PMA, [53], and [129], 2-D data rows can be stored in sequence. Therefore, those systems do not restrict the 2-D data size. However, the application using Stride PMA have to take account of the 2-D dimensions whereas Downscaler PMA takes care of the 2-D dimensions by itself. In CPMA, a 2-D scanning field can be run-time resized nearly arbitrarily with the Page Table parameters L_i and L_j (Section 4.2.2). Naturally, the total implemented address space restricts the possible size of the scanning field both in Stride PMA and CPMA.

One aspect is whether the parallel memory system allows an access format to reference over the scanning field or not. Stride PMA as well as Tanskanen's system [130] do not consider the possibility to exceed the scanning field. This seems to be the normal way because applications rarely benefit from the scanning field exceeding. However, some systems can utilize this possibility [P10], [123], [129]. Downscaler PMA simplifies application functionality by allowing circular/module memory addressing [129] in y -direction. CPMA allows circular addressing both in x and y -directions. Moreover, the exceeded part in x -direction may wrap around to the *next row* in CPMA stride access. This corresponds 1-D memory addressing.

The memory module count N is normally power of two and equals to the accessed element count M [P8], [P10], [53], [74], [129], [130]. However, Park [101] and HiPAR-DSP [112] utilize an odd number of N that is greater than M . CPMA allows arbitrary N and M .

7.2 Parallel memory complexity analysis

The complexities of the three proposed parallel memory implementations are evaluated in the following. The purpose is to estimate the complexity overhead paid due to achieved run-time configurability. The fairness of the comparison is confirmed with the following ways:

- The same person has implemented all the three parallel memory systems in synthesizable register transfer level VHDL. Naturally, the systems have been coded in different time periods, started with CPMA and followed by Downscaler and

Table 7.1. Parallel memory complexity comparison in gate count, $N = 4$.

	Address Computation	Read Data Permutation	Write Data Permutation	Total
Downscaler PMA	95 + 0	193	85	373
Stride PMA	703	203	203	1 109
CPMA	10 763	203	203	11 169

Stride PMA, respectively. Therefore, some coding skill development may be visible in the implementations.

- The results are given by the same person. This means that all the significant design-time parameters are equal and the same methods are used to define complexity.
- The used synthesis tool and process technology vendor are same in all the memory systems. However, Downscaler and Stride PMA are synthesized with 0.18 μm whereas CPMA with 0.25 μm CMOS process technology.

The complexity results are received by logic synthesis. The gate counts are based on 2-input NAND gate sizes of the respective processes (0.18 or 0.25 μm). All the other units but memory modules were synthesized. Pipelining is not taken into account. Memory capacity of a single memory module is constant 2048 bytes. Therefore, the total memory capacity increases linearly as a function of the number of memory modules. The data width of a single memory module is eight bits in this study.

For proper comparison, all the three memory systems are supposed to utilize single cycle SRAM memory modules. Thus, in this evaluation, CPMA does not include DRAMs with nonpipelined multicycle access time or DRAM controllers [139]. With single cycle memory modules, a specific architectural extension of CPMA called CPMA access instruction correlation recognition [138] gives no advantage and is also excluded. The Permutation Control and Address Permutation units in Stride PMA have the same functionality as the Address Select units in CPMA. Moreover, the Data Permutation networks in Stride PMA are suitable for CPMA operation. Therefore, for proper CPMA functionality, Stride PMA complexity results of the Permutation Control, Address Permutation, and Data Permutation units are also utilized with CPMA.

Downscaler PMA is designed as application specific with $N = 4$ and does not support other memory module counts. Table 7.1 tabulates the gate counts of the three proposed memory systems with the memory module count $N = 4$. The gate counts of the Address Computation and the two Data Permutation units are separated. Stride PMA and CPMA contains single port memory modules whereas Downscaler PMA allows simultaneous read and write. Therefore, Downscaler PMA demands two Address Computation units: the Read Address Computation unit requires 95 gates whereas the Write Address Computation unit contains no logic but only hardwired masking and shifting.

Address computation in Stride PMA is over seven times more complex than in Downscaler PMA. However, due to lower differences in data permutation, the total Stride

Table 7.2. Normalized parallel memory complexity, $N = 4$.

	Number of Schemes	Complexity / Scheme
Downscaler PMA	1	373
Stride PMA	8	139
CPMA	149	75

PMA gate count is only three times larger. Respectively, CPMA address computation consumes over 15 times more gates than respective functionality in Stride PMA whereas the whole CPMA is ten times more complex.

The results of Downscaler PMA and Stride PMA differ from those in [P10] and [P9]. In this study, memory capacity of a single memory module is 2048 bytes whereas 4096 and 512 bytes are used in [P10] and [P9], respectively. Moreover, some extra functionality required only for downscaling is excluded from the Downscaler PMA results.

The complexity results in Table 7.1 are refined in Table 7.2 by taking into account the run-time configurability properties. The results in Table 7.2 are normalized according to the number of available skewing schemes with each of the three parallel memory implementations.

With CPMA and $N = 4$, it is possible to form schemes utilizing one to four memory modules. However, skewing schemes with less than four memory modules are ignored in the results of Table 7.2. Moreover, only different schemes are considered by removing isomorphic skewing schemes described in Section 3.4.4. In Table 7.2, only one representative of the isomorphic schemes is included.

Downscaler PMA has a single fixed skewing scheme. In Stride PMA, implementation specific maximum stride defines the number of available skewing schemes. In this study, the maximum stride is restricted to 255 producing eight different skewing schemes (Table 7.2). The normalized complexity of Stride PMA is 63% less than Downscaler PMA complexity. CPMA allows multitude of schemes with three different parametrizable module assignment functions. However, several of those schemes are ignored in Table 7.2 due to isomorphism yielding still 149 different skewing schemes. Normalized complexity of CPMA is 80% and 46% less than that of Downscaler PMA and Stride PMA, respectively.

More extensive complexity evaluation can be done between Stride PMA and CPMA. However, CPMA allows arbitrary memory module count but Stride PMA only power of two. Therefore, complexities with $N = 2, 4, 8, 16$, and 32 are shown in Table 7.3. The relative complexity difference between the memory systems diminish when increasing the memory module count. With $N = 2$, CPMA is shown to be over 12 times more complex than Stride PMA whereas CPMA is only 3.6 times larger with $N = 32$.

The variation in relative complexity is mainly due to three permutation networks: one Address Permutation and two Data Permutation networks. The difference is clarified in

Table 7.3. Stride PMA and CPMA complexity in gate count.

	$N = 2$	$N = 4$	$N = 8$	$N = 16$	$N = 32$
Stride PMA	285	1 109	4 312	15 728	60 463
CPMA	3 494	11 169	29 952	79 639	217 575

Table 7.4 that tabulates the permutation network complexities in terms of proportional amount of the total gate count. As mentioned above, both Stride PMA as well as CPMA use the same permutation units. In the both memories, the permutation network complexities increase more rapidly than other logic as a function of the memory module count. The three networks consume from 50 to 79% of the total gate count in Stride PMA and 4–22% in CPMA.

The Data Permutation networks in Stride PMA and CPMA are full crossbars which are also used in [47], Tanskanen’s memory [130], and HiPAR [112]. The two Data Permutation units could be combined in the proposed systems [P9]. As a drawback, the delay of the combined Data Permutation unit would increase due to additional multiplexers. Moreover, write after read turn-around time of the parallel memory system would also increase. An additional possibility to decrease the demanded logic is to utilize multistage interconnection networks like Benes [27]. For example, HiPAR utilizes a two-stage crossbar data network.

However, the data permutation networks do not always have to be full crossbars. The required permutation network depends on the utilized skewing scheme and diversity of allowed scanning points. The write access format in the Downscaler PMA is conflict free in all the locations. However, the scanning points are restricted to allow easier address computation and data permutation logic. The Write Data Permutation demands only two permutation possibilities out of 16 available in full crossbar and utilizes 42% of the network complexity in Stride PMA or CPMA (Table 7.1).

The Read Data Permutation unit in Downscaler PMA demands four permutation possibilities and is implemented with a one-stage rotator. Tanskanen [129] as well as Kuzmanov [74] also utilize rotators as the data permutation network. Also algorithm modifications may be utilized to decrease data permutations. In [52] and [53], Viterbi decoding computation is rescheduled in such a way that only one data permutation network is demanded. Data reading can be done along hardwired input ports whereas interconnection network is required for write operation. As a drawback, address computation may be more complex than in the memory system with two data permutation networks [52].

Table 7.4. The complexities of the three permutation networks in Stride PMA and CPMA (% of the total gate count).

	$N = 2$	$N = 4$	$N = 8$	$N = 16$	$N = 32$
Stride PMA	50.4	60.9	67.2	73.2	78.5
CPMA	4.1	6.1	9.7	14.5	21.8

The third full crossbar permutation network in CPMA and Stride PMA is the Address Permutation unit. Park [101] utilizes a rotator in address permutation. In an application specific parallel memory system, the address is typically calculated straight to the correct memory module without any specific address permutation network. In such systems, the skewing scheme is implicitly embedded in the hardware. Some examples are Downscaler PMA, Tanskanen's memories [129], [130], Kuzmanov's system [74], as well as in [54].

8. Conclusions

The increasing number of transistors available for a single chip allows digital signal processing systems to become more and more complex. Reusability and programmability are some methods to increase design productivity. On the other hand, multimedia applications are mostly computation intensive but utilize regular and highly predictable tasks that can straightforwardly be parallelized.

Parallel memories can be used to meet high memory bandwidth demands in parallel processing. They access only the required data in correct order. Part of the algorithm execution can be performed with the memory system since no explicit rearrangement needs to be done by the processing elements.

This Thesis adds run-time configurability to parallel memories to allow support for several algorithms with the same hardware. The design reusability of the proposed memories is also improved, so the same memory system can be utilized with several different applications. The proposed architectures are mainly useful in SIMD as well as subword SIMD processing.

Area efficient implementation favors that operands are powers of two. For example, restricting the number of memory modules in CPMA to a power of two would greatly simplify the address computation hardware. However, the use of a prime number of memory modules would not be possible in that case, which would prevent the use of several straight access formats.

8.1 Main results

Some very concise run-time configurability properties have previously been presented for parallel memories. This Thesis presents the first parallel memory implementation which allows access formats, skewing schemes, and address functions to be widely changed at run-time [P1]–[P5]. The proposed CPMA can be configured to support multitude of different algorithms.

In CPMA, all the possible linear module assignment functions and XOR-schemes can be formed with the implementations shown in [P1] and [P4], respectively. Generalized function for diamond schemes including an appropriate generalized address function is defined in [P2]. Multitude of multiperiodic skewing schemes are available with the proposed parametrizable diamond schemes [P2]. Wide range of access formats can also be formed in CPMA [P3], for example, all the possible straight, rectangle, crumbled

rectangle, chessboard, and stride templates. Furthermore, several irregular data patterns are allowed. With literature review, access formats demanded in several applications are categorized as well as solved how different types of skewing schemes inherently support specific types of templates [P5]. Address computation dominates the CPMA implementation area and delay almost irrespectively from the implementation details. For the address computation, logic synthesis based estimates for a 0.25 μm CMOS process show moderate timing and area increase as a function of the memory module count [P5].

A novel coarse-grained parallelization method for 2-D image scaling is introduced in [P7]. With the proposed architecture, several interpolation methods can be utilized and parallelism level can widely be adjusted. By utilizing contemporary FPGAs, real-time scaling is achievable with large resolution 2-D images and a good quality interpolation. The proposed block-level scaling is also shown to increase software scaling performance over four times.

A novel parallel memory architecture with certain run-time configurability is introduced in [P8] and [P9]. Besides CPMA, this is the only parallel memory system allowing arbitrary constant stride accesses. The module assignment functions are simplified from the reference implementation in terms of hardware resources [P8]. The image scaling architecture shown in [P7] is one target for such memory. According to the parallel memory results on FPGA, relatively high clock frequency can be achieved with moderate complexity and low latencies [P9].

Previous research investigated a novel image scaling method called winscale that utilizes exceptional pixel model interpretation [69]. In [P6], similar pixel model is adapted to well-known bilinear interpolation. It is shown in [P6] that scaling down with the winscale concept gives exactly the same results as bilinear interpolation.

Image downscaling hardware utilizing bilinear interpolation is presented in [P10]. The concepts shown in [P7] are applied to fine-grained parallel processing. A parallel memory system is tailored for the downscaler to achieve the maximum performance with minimum resources. The results show that an FPGA implementation can downscale high resolution video (16VGA and HDTV) in real-time with a complexity of less than half of the reference implementations. The results obtained in [P6] are utilized with the complexity and performance comparison.

The three proposed parallel memory architectures are evaluated in Chapter 7. Run-time configurability properties are the most significant differences between these memory systems. Configurability benefits and drawbacks are analyzed. With the case studies, the normalized complexity of configurable parallel memories is shown to be noticeable less than with a traditional type of an application specific parallel memory. When increasing memory module count in configurable parallel memories, the complexity increase in permutation networks is expressed to become the most critical.

References

- [1] E. Aho, J. Vanne, T. D. Hämäläinen, and K. Kuusilinna, “Block-level parallel processing for scaling evenly divisible frames,” in *Proc. IEEE Int. Symp. Circuits Syst.*, Kobe, Japan, pp. 1134–1137, May 2005.
- [2] M. A. Al-Mouhamed and S. S. Seiden, “Minimization of memory and network contention for accessing arbitrary data patterns in SIMD systems,” *IEEE Trans. Computers*, vol. 45, no. 6, pp. 757–762, June 1996.
- [3] M. Al-Mouhamed, L. Bic, and H. Abu-Haimed, “A compiler address transformation for conflict-free access of memories and networks,” in *Proc. IEEE Symp. Parallel Distrib. Processing*, pp. 530–537, Oct. 1996.
- [4] M. A. Al-Mouhamed and S. S. Seiden, “A heuristic storage for minimizing access time of arbitrary data patterns,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 8, no. 4, pp. 441–447, Apr. 1997.
- [5] M. Al-Mouhamed, “Array organization in parallel memories,” *International Journal Parallel Programming*, vol. 32, no. 2, pp. 123–163, Apr. 2004.
- [6] K. H. Al-Saqabi and E. W. Davis, “Extending parallelism to memory hierarchies in massively parallel systems,” *IEE Proceedings – Computers and Digital Techniques*, vol. 138, no. 4, pp. 193–202, July 1991.
- [7] T. Asano, J. Silberman, S. H. Dhong, O. Takahashi, M. White, S. Cottier, T. Nakazato, A. Kawasumi, and H. Yoshihara, “Low-power design approach of 11FO4 256-Kbyte embedded SRAM for the synergistic processor of a Cell processor,” *IEEE Micro*, vol. 25, no. 5, pp. 30–38, Sept./Oct. 2005.
- [8] V. Auletta, S. K. Das, A. De Vivo, M. C. Pinotti, and V. Scarano, “Optimal tree access by elementary and composite templates in parallel memory systems,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 4, pp. 399–411, Apr. 2002.
- [9] C. Basoglu, W. Lee, and J. O’Donnell, “The Equator MAP-CATM DSP: An end-to-end broadband signal processorTM VLIW,” *IEEE Trans. Circuits Syst. Video Technol.*, vol. 12, no. 8, pp. 646–659, Aug. 2002.
- [10] K. E. Batcher, “The multidimensional access memory in STARAN,” *IEEE Trans. Computers*, pp. 174–177, Feb. 1977.
- [11] Berkeley Design Technology, Inc., “Tensilica Xtensa LX Processor with Vectra LX,” *BDTi Analysis*, Available online: http://www.bdti.com/articles/xtensa_lx_analysis.pdf, 2005.
- [12] D. Boggs, A. Baktha, J. Hawkins, D. T. Marr, J. A. Miller, P. Roussel, R. Singhal, B. Toll, and K. S. Venkatraman, “The microarchitecture of the Intel® Pentium® 4

- processor on 90nm technology,” *Intel Technology Journal*, vol. 8, no. 1, pp. 1–17, Feb. 2004.
- [13] P. Budnik and D. J. Kuck, “The organization and use of parallel memories,” *IEEE Trans. Comp.*, vol. C-20, no. 12, pp. 1566–1569, Dec. 1971.
 - [14] D. A. Carlson, R. W. Castelino, and R. O. Mueller, “Multimedia extensions for a 550-MHz RISC microprocessor,” *IEEE Journal Solid-State Circuits*, vol. 32, no. 11, pp. 1618–1624, Nov. 1997.
 - [15] S. Chen, A. Postula, and L. Jozwiak, “Synthesis of XOR storage schemes with different cost for minimization of memory contention,” in *Proc. Euromicro Conf.*, Milan, Italy, pp. 170–177, Sep. 1999.
 - [16] S. Chen and A. Postula, “Synthesis of custom interleaved memory systems,” *IEEE Trans. VLSI Syst.*, vol. 8, no. 1, pp. 74–83, Feb. 2000.
 - [17] D. Cheresiz, B. Juurlink, S. Vassiliadis, and H. A. G. Wijshoff, “Implementation of a streaming execution unit,” *Journal Systems Architecture*, vol. 49, no. 12–15, pp. 599–617, Dec. 2003.
 - [18] D. Cheresiz, B. Juurlink, S. Vassiliadis, and H. A. G. Wijshoff, “The CSI multimedia architecture,” *IEEE Trans. VLSI Syst.*, vol. 13, no. 1–13, Jan. 2005.
 - [19] D. Cohen-Or and A. Kaufman, “A 3D skewing and de-skewing scheme for conflict-free access to rays in volume rendering,” *IEEE Trans. Computers*, vol. 44, no. 5, pp. 707–710, May 1995.
 - [20] C. J. Colbourn and K. Heinrich, “Conflict-free access to parallel memories,” *Journal Parallel Distrib. Comput.*, vol. 14, no. 2, pp. 193–200, Feb. 1992.
 - [21] J. Corbal, M. Valero, and R. Espasa, “Exploiting a new level of DLP in multimedia applications,” in *Proc. Int. Symp. Microarchitecture*, Haifa, Israel, pp. 72–79, Nov. 1999.
 - [22] R. Creutzburg, J. Niittylahti, T. Sihvo, J. Takala, and J. Tanskanen, “Parallel memory architectures for video encoding systems, part II: Applications,” in *Proc. Int. Workshop Spectral Techniques and Logic Design for Future Digital Systems*, Tampere, Finland, pp. 567–594, June 2000.
 - [23] W. J. Dally, U. J. Kapasi, B. Khalany, J. H. Ahn, and A. Das, “Stream processors: Programmability with efficiency,” *ACM Queue*, vol. 2, no. 1, pp. 52–62, Mar. 2004.
 - [24] A. Deb, “Conflict-free access of arrays – A counter example,” *Information Processing Letters*, vol. 10, no. 1, pp. 20, Feb. 1980.
 - [25] A. Deb, “Multiskewering – A novel technique for optimal parallel memory access,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, no. 6, pp. 595–604, June 1996.
 - [26] K. Diefendorff, P. K. Dubey, R. Hochsprung, and H. Scale, “AltiVec extension to PowerPC accelerates media processing,” *IEEE Micro*, vol. 20, no. 2, pp. 85–95, Mar./Apr. 2000.

- [27] S. Dutta, W. Wolf, and A. Wolfe, “A methodology to evaluate memory architecture design tradeoffs for video signal processors,” *IEEE Trans. Circuits Syst. Video Technol.*, vol. 8, no. 1, pp. 36–53, Feb. 1998.
- [28] C. Erbas, M. M. Tanik, and V. S. S. Nair, “A circulant matrix based approach to storage schemes for parallel memory systems,” in *Proc. IEEE Symp. Parallel Distrib. Processing*, pp. 92–99, Dec. 1993.
- [29] J. Eyre and J. Bier, “The evolution of DSP processors,” *IEEE Signal Processing Magazine*, vol. 17, no. 2, pp. 43–51, Mar. 2000.
- [30] P. Faraboschi, G. Desoli, and J. A. Fisher, “The latest word in digital and media processing,” *IEEE Signal Processing Magazine*, vol. 15, no. 2, pp. 59–85, Mar. 1998.
- [31] B. Flachs, S. Asano, S. H. Dhong, H. P. Hofstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty, B. Michael, H.-J. Oh, S. M. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, N. Yano, D. A. Brokenshire, M. Peyravian, V. To, and E. Iwata, “The microarchitecture of the synergistic processor for a Cell processor,” *IEEE Journal Solid-State Circuits*, vol. 41, no. 1, pp. 63–70, Jan. 2006.
- [32] J. M. Frailong, W. Jalby, and J. Lenfant, “XOR-schemes: A flexible data organization in parallel memories,” in *Proc. Int. Conf. Parallel Processing*, St. Charles, IL, USA, pp. 276–283, Aug. 1985.
- [33] J. Fridman, Z. Greenfield, “The TigerSHARC DSP architecture,” *IEEE Micro*, vol. 20, no. 1, pp. 66–76, Jan./Feb. 2000.
- [34] J. Fridman, “Sub-word parallelism in digital signal processing,” *IEEE Signal Processing Mag.*, vol. 17, no. 2, pp. 27–35, Mar. 2000.
- [35] M. Gössel and B. Rebel, “Memories for parallel subtree-access,” in *Proc. Int. Workshop Parallel Algorithms and Architectures*, Suhl, Germany, pp. 122–130, May 1987.
- [36] M. Gössel, V. V. Kaversnev, and B. Rebel, “Parallel memories for straight line and rectangle access,” in *Proc. Int. Workshop Parallel Processing by Cellular Automata and Arrays*, Berlin, Germany, pp. 89–109, Oct. 1988.
- [37] M. Gössel and B. Rebel, “Parallel access to rectangles,” in *Recent Issues in Pattern Analysis and Recognition*, LNCS 399, V. Cantoni, R. Creutzburg, S. Levialdi, and G. Wolf, ed., Springer-Verlag, Berlin, Germany, pp. 201–213, 1989.
- [38] M. Gössel, B. Rebel, and R. Creutzburg, *Memory Architecture & Parallel Access*, Elsevier Science B.V, Amsterdam, The Netherlands, 1994.
- [39] R. Gupta and M. L. Soffa, “Compile-time techniques for improving scalar access performance in parallel memories,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, no. 2, pp. 138–148, Apr. 1991.
- [40] T. R. Halfhill, “Tensilica tackles bottlenecks – New Xtensa LX configurable processor shatters industry benchmarks,” *Microprocessor Report*, vol. 6, no. 1, pp. 1–7, May 2004.

- [41] M. Hariyama, S. Lee, and M. Kameyama, “Highly-parallel stereo vision VLSI processor based on an optimal parallel memory access scheme,” *IEICE Trans. Electron.*, vol. E84-C, no. 3, pp. 382–389, Mar. 2001.
- [42] D. T. Harper III and J. R. Jump, “Vector access performance in parallel memories using a skewed storage scheme,” *IEEE Trans. Computers*, vol. C-36, no. 12, pp. 1440–1449, Dec. 1987.
- [43] D. T. Harper III and D. A. Linebarger, “Dynamic address mapping for conflict-free vector access,” U.S. Patent 4 918 600, Apr. 17, 1990.
- [44] D. T. Harper III, “Block, multistride vector, and FFT accesses in parallel memory systems,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, no. 1, pp. 43–51, Jan. 1991.
- [45] D. T. Harper III and D. A. Linebarger, “Conflict-free vector access using a dynamic storage scheme,” *IEEE Trans. Comput.*, vol. 40, no. 3, pp. 276–283, Mar. 1991.
- [46] D. T. Harper III, “Increased memory performance during vector accesses through the use of linear address transformations,” *IEEE Trans. Comput.*, vol. 41, no. 2, pp. 227–230, Feb. 1992.
- [47] D. T. Harper III, “A multiaccess frame buffer architecture,” *IEEE Trans. Computers*, vol. 43, no. 5, pp. 618–622, May 1994.
- [48] R. W. Hartenstein, A. G. Hirschbiel, K. Schmidt, and M. Weber, “A novel paradigm of parallel computation and its use to implement simple high-performance hardware,” in *Proc. Int. Conf. Information Technology*, Tokyo, Japan, Oct. 1990.
- [49] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed., Morgan Kaufman Publishers, San Francisco, CA, USA, 2003.
- [50] W. Hinrichs, J. P. Wittenburg, H. Lieske, H. Kloos, M. Ohmacht, and P. Pirsch, “A 1.3-GOPS parallel DSP for high-performance image-processing applications,” *IEEE Journal Solid-State Circuits*, vol. 35, no. 7, pp. 946–952, July 2000.
- [51] H. S. Hou and H. C. Andrews, “Cubic splines for image interpolation and digital filtering,” *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-26, no. 6, pp. 508–517, Dec. 1978.
- [52] T. Järvinen, *Systematic Methods for Designing Stride Permutation Interconnections*, Dr. Tech. Thesis, Tampere University of Technology, Tampere, Finland, 2004.
- [53] T. Järvinen, P. Salmela, T. Sipilä, and J. Takala, “Systematic approach for path metric access in Viterbi decoders,” *IEEE Trans. Commun.*, vol. 53, no. 5, pp. 755–759, May 2005.
- [54] T. Järvinen, P. Salmela, and J. Takala, “Interconnection optimized path metric access scheme for k/n -rate viterbi decoders,” in *IEEE Workshop Signal Processing Systems Design and Implementation*, Athens, Greece, pp. 503–508, Nov. 2005.
- [55] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, “Introduction to the Cell multiprocessor,” *IBM Journal Research & Development*, vol. 49, no. 4/5, pp. 589–604, July/Sep. 2005.

- [56] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, J. D. Owens, “Programmable stream processors,” *IEEE Computer*, vol. 36, no. 8, pp. 54–62, Aug. 2003.
- [57] R. S. Katti, “Nonprime memory systems and error correction in address translation,” *IEEE Trans. Computers*, vol. 46, no. 1, pp. 75–79, Jan. 1997.
- [58] M. Keating and P. Bricaud, *Reuse Methodology Manual For System-on-a-Chip Designs*, 2nd ed., Kluwer Academic Publishers, Boston, MA, USA, 1999.
- [59] B. Khailany, W. J. Dally, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, A. Chang, and Scott Rixner, “Imagine: Media processing with streams,” *IEEE Micro*, vol. 21, no. 2, pp. 35–46, Mar./Apr. 2001.
- [60] G.-Y. Kim, Y. Baek, and H.-K. Lee, “Data distribution and alignment scheme for conflict-free memory access in parallel image processing system,” *IEICE Trans. Inf. Syst.*, vol. E81-D, no. 8, pp. 806–812, Aug. 1998.
- [61] K. Kim and V. K. Prasanna Kumar, “Parallel memory systems for image processing,” in *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, San Diego, CA, USA, pp. 654–659, June 1989.
- [62] K. Kim and V. K. Prasanna, “Latin squares for parallel array access,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 4, pp. 361–370, Apr. 1993.
- [63] W.-Y. Kim, P. T. Balsara, D. T. Harper III, and J. W. Park, “Hierarchy embedded differential image for progressive transmission using lossless compression,” *IEEE Trans. Circuits Syst. Video Technol.*, vol. 5, no. 1, pp. 1–13, Feb. 1995.
- [64] H. Kloos, J. P. Wittenburg, W. Hinrichs, H. Lieske, L. Friebe, C. Klar, and P. Pirsch, “HiPAR-DSP 16, a scalable highly parallel DSP core for system on a chip video and image processing applications,” in *Proc. Int. Conf. Acoustics, Speech and Signal Processing*, Orlando, FL, USA, pp. 3112–3115, May 2002.
- [65] P. M. Kogge, “Skewed matrix address generator,” U.S. Patent 4 370 732, Jan. 25, 1983.
- [66] C. E. Kozyrakis and D. A. Patterson, “A new direction for computer architecture research,” *IEEE Computer*, vol. 31, no. 11, pp. 24–32, Nov. 1998.
- [67] C. Kozyrakis, D. Judd, J. Gebis, S. Williams, D. Patterson, and K. Yellick, “Hardware/compiler codevelopment for an embedded media processor,” *Proc. IEEE*, vol. 89, no. 11, pp. 1694–1709, Nov. 2001.
- [68] C. E. Kozyrakis and D. A. Patterson, “Scalable vector processors for embedded systems,” *IEEE Micro*, vol. 23, no. 6, pp. 36–45, Nov./Dec. 2003.
- [69] C.-H. Kim, S.-M. Seong, J.-A. Lee, and L.-S. Kim, “Winscale: An image scaling algorithm using an area pixel model,” *IEEE Trans. Circuits Syst. Video Technol.*, vol. 13, no. 6, pp. 549–553, Jun. 2003.
- [70] D. J. Kuck, “ILLIAC IV software and application programming,” *IEEE Trans. Comput.*, vol. C-17, no. 8, pp. 758–770, Aug. 1968.
- [71] D. J. Kuck and R. A. Stokes, “The Burroughs Scientific Processor (BSP),” *IEEE Trans. Computers*, vol. C-31, no. 5, pp. 363–376, May 1982.

- [72] K. Kuusilinna, J. Tanskanen, T. Hämäläinen, J. Niittylahti, and J. Saarinen, “Configurable parallel memory architecture for multimedia computers,” *Journal Systems Architecture*, vol. 47, no. 14–15, pp. 1089–1115, Aug. 2002.
- [73] G. Kuzmanov, S. Vassiliadis, and J. van Eijndhoven, “A 2D addressing mode for multimedia applications,” in *Proc. Workshop Systems, Architectures, Modeling, and Simulation*, Samos, Greece, pp. 291–306 July 2001.
- [74] G. Kuzmanov, G. Gaydadjiev, and S. Vassiliadis, “Multimedia rectangularly addressable memory,” *IEEE Trans. Multimedia*, vol. 8, no. 2, pp. 315–322, Apr. 2006.
- [75] V. Lappalainen, T. D. Hämäläinen, and P. Liuha, “Overview of research efforts on media ISA extensions and their usage in video coding,” *IEEE Trans. Circuits Syst. Video Technol.*, vol. 12, no. 8, pp. 660–670, Aug. 2002.
- [76] D. H. Lawrie, “Access and alignment of data in an array processor,” *IEEE Trans. Computers*, vol. C-24, no. 12, pp. 1145–1155, Dec. 1975.
- [77] D. H. Lawrie and C. R. Vora, “The prime memory system for array access,” *IEEE Trans. Computers*, vol. C-31, no. 5, pp. 435–442, May 1982.
- [78] D.-L. Lee, “Scrambled storage for parallel memory systems,” in *Proc. Int. Symp. Computer Architecture*, pp. 232–239, May 1988.
- [79] D.-L. Lee, “On access and alignment of data in a parallel processor,” *Information Processing Letters*, vol. 33, no. 1, pp. 11–14, Oct. 1989.
- [80] D.-L. Lee, “Efficient address generation in a parallel processor,” *Information Processing Letters*, vol. 36, no. 3, pp. 111–116, Nov. 1990.
- [81] D.-L. Lee, “Design of an array processor for image processing,” *Journal Parallel Distrib. Comput.*, vol. 11, no. 2, pp. 163–169, Feb. 1991.
- [82] D.-L. Lee, “Architecture of an array processor using a nonlinear skewing scheme,” *IEEE Trans. Computers*, vol. C-41, no. 4, pp. 499–505, Apr. 1992.
- [83] R. B. Lee, “Accelerating multimedia with enhanced microprocessors,” *IEEE Micro*, vol. 15, no. 2, pp. 22–32, Apr. 1995.
- [84] R. B. Lee, “Subword parallelism with MAX-2,” *IEEE Micro*, vol. 16, no. 4, pp. 51–59, Aug. 1996.
- [85] S. Lee, M. Hariyama, and M. Kameyama, “An FPGA-oriented motion-stereo processor with a simple interconnection network for parallel memory access,” *IEICE Trans. Inf. Syst.*, vol. E83-D, no. 12, pp. 2122–2130, Dec. 2000.
- [86] T. M. Lehmann, C. Gönner, and K. Spitzer, “Survey: Interpolation methods in medical image processing,” *IEEE Trans. Med. Imag.*, vol. 18, no. 11, pp. 1049–1075, Nov. 1999.
- [87] L. Li, S. Goto, and T. Ikenaga, “An efficient deblocking filter architecture with 2-dimensional parallel memory for H.264/AVC,” in *Proc. Asia and South Pacific Design Automation Conf.*, Shanghai, China, pp. 623–626, Jan. 2005.

- [88] R. Li and S. Levi, "An arbitrary ratio resizer for MPEG applications," *IEEE Trans. Consumer Electron.*, vol. 46, no. 3, pp. 467–473, Aug. 2000.
- [89] X. Li, "Parallel algorithms for hierarchical clustering and cluster validity," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 12, no. 11, pp. 1088–1092, Nov. 1990.
- [90] Z. Liu and J.-H. You, "An implementation of a nonlinear skewing scheme," *Information Processing Letters*, vol. 42, no. 4, pp. 209–215, June 1992.
- [91] Z. Liu and X. Li, "XOR storage schemes for frequently used data patterns," *Journal Parallel Distrib. Comput.*, vol. 25, no. 2, pp. 162–173, Mar. 1995.
- [92] S. A. McKee, R. H. Klenke, K. L. Wright, W. A. Wulf, M. H. Salinas, J. H. Aylor, A. P. Batson, "Smarter memory: Improving bandwidth for streamed references," *IEEE Computer*, vol. 31, no. 7, pp. 54–63, July 1998.
- [93] S. A. McKee, W. A. Wulf, J. H. Aylor, R. H. Klenke, M. H. Salinas, S. I. Hong, and D. A. B. Weikle, "Dynamic access ordering for streamed computations," *IEEE Trans. Computers*, vol. 49, no. 11, pp. 1255–1271, Nov. 2000.
- [94] S. D. Naffziger, G. Colon-Bonet, T. Fischer, R. Riedlinger, T. J. Sullivan, and T. Grutkowski, "The implementation of the Itanium 2 microprocessor," *IEEE Journal Solid-State Circuits*, vol. 37, no. 11, pp. 1448–1460, Nov. 2002.
- [95] A. Norton and E. Melton, "A class of boolean linear transformations for conflict-free power-of-two stride access," in *Proc. Int. Conf. Parallel Processing*, University Park, PA, USA, pp. 247–254, Aug. 1987.
- [96] S. Oberman, G. Favor, and F. Weber, "AMD 3DNow! technology: Architecture and implementations," *IEEE Micro*, vol. 19, no. 2, pp. 37–48, Mar./Apr. 1999.
- [97] P. R. Panda, N. Dutt, and A. Nicolau, *Memory Issues in Embedded Systems-On-Chip: Optimizations and Exploration*, Kluwer Academic Publishers, London, Great Britain, 2002.
- [98] J. W. Park, "An efficient memory system for image processing," *IEEE Trans. Computers*, vol. C-35, no. 7, pp. 669–674, July 1986.
- [99] J. W. Park and D. T. Harper III, "An efficient memory system for the SIMD construction of a gaussian pyramid," *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, no. 8, pp. 855–860, Aug. 1996.
- [100] J. W. Park, "An efficient buffer memory system for subarray access," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 3, pp. 316–335, Mar. 2001.
- [101] J. W. Park, "Multiaccess memory system for attached SIMD computer," *IEEE Trans. Comput.*, vol. 53, no. 4, pp. 439–452, Apr. 2004.
- [102] J. W. Park, "Conflict-free memory system and method of address calculation and data routing by using the same," U.S. Patent 6 845 423, Jan. 18, 2005.
- [103] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yellick, "A case for Intelligent RAM," *IEEE Micro*, vol. 17, no. 2, pp. 34–44, Mar./Apr. 1997.

- [104] A. Peleg and U. Weiser, "MMX technology extension to the Intel architecture," *IEEE Micro*, vol. 16, no. 4, pp. 42–50, Aug. 1996.
- [105] A. Peleg, S. Wilkie, and U. Weiser, "Intel MMX for multimedia PCs," *Communications of the ACM*, vol. 40, no. 1, pp. 24–38, Jan. 1997.
- [106] D. C. Pham, T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, P. M. Harvey, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Pham, J. Pille, S. Posluszny, M. Riley, D. L. Stasiak, M. Suzuki, O. Takahashi, J. Warnock, S. Weitzel, D. Wendel, and K. Yazawa, "Overview of the architecture, circuit design, and physical implementation of a first-generation Cell processor," *IEEE Journal Solid-State Circuits*, vol. 41, no. 1, pp. 179–196, Jan. 2006.
- [107] P. Pirsch, C. Reuter, J. P. Wittenburg, M. B. Kulaczewski, and H.-J. Stolberg, "Architecture concepts for multimedia signal processing," *Journal VLSI Signal Processing*, vol. 29, no. 3, pp. 157–165, Nov. 2001.
- [108] A. Postula, S. Chen, L. Jozwiak, and D. Abramson, "Automated synthesis of interleaved memory systems for custom computing machines," in *Proc. Euromicro Conf.*, Västerås, Sweden, pp. 115–122, Aug. 1998.
- [109] R. Raghavan and J. P. Hayes, "On randomly interleaved memories," in *Proc. ACM/IEEE Conf. Supercomputing*, New York, NY, USA, pp. 49–58, Nov. 1990.
- [110] P. Ranganathan, S. Adve, and N. P. Jouppi, "Performance of image and video processing with general-purpose processors and media ISA extensions," in *Proc. Int. Symp. Computer Architecture*, Atlanta, GA, USA, pp. 124–135, May 1999.
- [111] I. E. G. Richardson, *H.264 and MPEG-4 Video Compression: Video Coding for Next-Generation Multimedia*, John Wiley & Sons, West Sussex, UK, 2003.
- [112] K. Rönner and J. Kneip, "Architecture and applications of the HiPAR video signal processor," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 6, no. 1, pp. 56–66, Feb. 1996.
- [113] A. Seznec and J. Lenfant, "Interleaved parallel schemes," *IEEE Trans. Parallel Distrib. Syst.*, vol. 5, no. 12, pp. 1329–1334, Dec. 1994.
- [114] H. D. Shapiro, "Theoretical limitations on the efficient use of parallel memories," *IEEE Trans. Computers*, vol. C-27, no. 5, pp. 421–428, May 1978.
- [115] Y. Q. Shi, X. M. Zhang, Z.-C. Ni, and N. Ansari, "Interleaving for combating burst of errors," *IEEE Circuits Syst. Mag.*, vol. 4, no. 1, pp. 29–42, Jan./Mar. 2004.
- [116] N. Slingerland and A. J. Smith, "Measuring the performance of multimedia instruction sets," *IEEE Trans. Computers*, vol. 51, no. 11, pp. 1317–1332, Nov. 2002.
- [117] G. S. Sohi, "High-bandwidth interleaved memories for vector processors – A simulation study," *IEEE Trans. Comput.*, vol. 42, no. 1, pp. 34–44, Jan. 1993.
- [118] Sony, HDR-FX1 HDV Features, Available online: <http://www.sony.com>.
- [119] H.-J. Stolberg, M. Berekovic, S. Moch, L. Friebe, M. B. Kulaczewski, S. Flügel, H. Klüßmann, A. Dehnhardt, and P. Pirsch, "HiBRID-SoC: A multi-core SoC

- architecture for multimedia," *Journal VLSI Signal Processing*, vol. 41, no. 1, pp. 9–20, Aug. 2005.
- [120] O. Takahashi, S. Cottier, S. H. Dhong, B. Flachs, and J. Silberman, "Power-conscious design of the Cell processor's synergistic processor element," *IEEE Micro*, vol. 25, no. 5, pp. 10–18, Sep./Oct. 2005.
 - [121] J. Takala, "Data processing method and device for parallel stride access," U.S. Patent 6 640 296, Oct. 28, 2003.
 - [122] J. H. Takala, T. S. Järvinen, and H. T. Sorokin, "Conflict-free parallel memory access scheme for FFT processors," in *Proc. IEEE Int. Symp. Circuits Syst.*, Bangkok, Thailand, pp. 524–527, May 2003.
 - [123] J. Takala and T. Järvinen, "Stride permutation access in interleaved memory systems," in *Domain-Specific Multiprocessors – Systems, Architectures, Modeling, and Simulation*, S. S. Bhattacharyya, E. F. Deprettere, and J. Teich, ed., Marcel Dekker, New York, NY, USA, pp. 63–84, 2004.
 - [124] D. Talla, L. K. John, V. Lapinskii, and B. L. Evans, "Evaluating signal processing and multimedia applications on SIMD, VLIW and superscalar architectures," in *Proc. Int. Conf. Computer Design*, Austin, TX, USA, pp. 163–172, Sep. 2000.
 - [125] E. J. Tan and W. B. Heinzelman, "DSP architectures: Past, present and future," *ACM SIGARCH Computer Architecture News*, vol. 31, no. 3, pp. 6–19, June 2003.
 - [126] J. Tanskanen, T. Sihvo, J. Niittylahti, J. Takala, and R. Creutzburg, "Parallel memory access schemes for H.263 encoder," in *Proc. IEEE Int. Symp. Circuits Syst.*, Geneva, Switzerland, pp. 691–694, May 2000.
 - [127] J. K. Tanskanen, *Parallel Memory Architectures for Video Coding*, Dr. Tech. Thesis, Tampere University of Technology, Tampere, Finland, 2004.
 - [128] J. K. Tanskanen and J. T. Niittylahti, "Scalable parallel memory architectures for video coding," *Journal VLSI Signal Processing*, vol. 38, no. 2, pp. 173–199, Sep. 2004.
 - [129] J. K. Tanskanen, T. Sihvo, and J. Niittylahti, "Byte and modulo addressable parallel memory architecture for video coding," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 14, no. 11, pp. 1270–1276, Nov. 2004.
 - [130] J. K. Tanskanen, R. Creutzburg, and J. T. Niittylahti, "On design of parallel memory access schemes for video coding," *Journal VLSI Signal Processing*, vol. 40, no. 2, pp. 215–237, June 2005.
 - [131] G. Tel and H. A. G. Wijshoff, "Hierarchical parallel memory systems and multiperiodic skewing schemes," *Journal Parallel Distrib. Comput.*, vol. 7, no. 2, pp. 355–367, Oct. 1989.
 - [132] Texas Instruments, Inc., TMS320C64x Technical Overview, Jan. 2001.
 - [133] S. Thakkar and T. Huff, "Internet streaming SIMD extensions," *IEEE Computer*, vol. 32, no. 12, pp. 26–34, Dec. 1999.
 - [134] M. Tremblay, J. M. O'Connor, V. Narayanan, and L. He, "VIS speeds new media processing," *IEEE Micro*, vol. 16, no. 4, pp. 10–20, Aug. 1996.

- [135] M. Valero, T. Lang, J. M. Llaceria, M. Peiron, E. Ayguade, and J. J. Navarro, “Increasing the number of strides for conflict-free vector access,” *Computer Architecture News*, vol. 20, no. 2, pp. 372–381, May 1992.
- [136] M. Valero, T. Lang, M. Peiron, and E. Ayguadé, “Conflict-free access for streams in multimodule memories,” *IEEE Trans. Computers*, vol. 44, no. 5, pp. 634–646, May 1995.
- [137] J. Vanne, E. Aho, K. Kuusilinna, and T. Hääläinen, “Co-simulation of configurable parallel memory architecture and processor,” in *Proc. IEEE Int. Workshop Design and Diagnostics of Electronic Circuits and Systems*, Brno, Czech Republic, pp. 310–313, Apr. 2002.
- [138] J. Vanne, E. Aho, K. Kuusilinna, and T. Hääläinen, “Enhanced configurable parallel memory architecture,” in *Proc. Euromicro Symp. Digital System Design*, Dortmund, Germany, pp. 28–35, Sep. 2002.
- [139] J. Vanne, E. Aho, K. Kuusilinna, and T. Hääläinen, “Configurable parallel memory implementation for system-on-chip designs,” in *System-on-Chip for Real-Time Applications*, W. Badawy, and G. A. Jullien, ed., Kluwer Academic Publishers, Boston, MA, USA, pp. 237–248, 2003.
- [140] J. Vanne, E. Aho, T. D. Hääläinen, and K. Kuusilinna, “A high-performance sum of absolute difference implementation for motion estimation,” *IEEE Trans. Circuits Syst. Video Technol.*, vol. 16, no. 7, pp. 876–883, July 2006.
- [141] C. Verdier, A. Demeure, and F. Jutand, “Addressing scheme for a parallel memory system,” in *Proc. Euromicro Workshop Parallel Distrib. Processing*, Gran Canaria, Spain, pp. 131–135, Jan. 1993.
- [142] C. Verdier, E. Boutillon, A. Lafage, and A. Demeure, “Access and alignment of arrays for a bidimensional parallel memory,” in *Proc. Int. Conf. Application Specific Array Processors*, San Francisco, CA, USA, pp. 346–356, Aug. 1994.
- [143] D. C. van Voorhis and T. H. Morrin, “Memory systems for image processing,” *IEEE Trans. Computers*, vol. C-27, no. 2, pp. 113–125, Feb. 1978.
- [144] J.-W. van de Waerdt, S. Vassiliadis, S. Das, S. Mirolo, C. Yen, B. Zhong, C. Basto, J.-P. van Itegem, D. Amirtharaj, K. Kalra, P. Rodriguez, and H. van Antwerpen, “The TM3270 media-processor,” in *Proc. IEEE/ACM Int. Symp. Microarchitecture*, Barcelona, Spain, pp. 331–342, Nov. 2005.
- [145] B. Wei, “Comments on “A multiaccess frame buffer architecture”,” *IEEE Trans. Computers*, vol. 45, no. 7, pp. 862, July 1996.
- [146] H. A. G. Wijshoff and J. van Leeuwen, “The structure of periodic storage schemes for parallel memories,” *IEEE Trans. Computers*, vol. C-34, no. 6, pp. 501–505, June 1985.
- [147] H. A. G. Wijshoff and J. van Leeuwen, “On linear skewing schemes and d -ordered vectors,” *IEEE Trans. Computers*, vol. C-36, no. 2, pp. 233–239, Feb. 1987.

- [148] L. Zhang, Z. Fang, M. Parker, B. K. Mathew, L. Schaelicke, J. B. Carter, W. C. Hsieh, and S. A. McKee, “The Impulse memory controller,” *IEEE Trans. Computers*, vol. 50, no. 11, pp. 1117–1132, Nov. 2001.

Publications

Publication 1

E. Aho, J. Vanne, K. Kuusilinna, T. Hämäläinen, and J. Saarinen, “Configurable Address Computation in a Parallel Memory Architecture,” in *Advances in Signal Processing and Computer Technologies*, G. Antoniou, N. Mastorakis, and O. Panfilov, ed., World Scientific and Engineering Society Press, Athens, Greece, pp. 390–395, 2001.

Copyright © World Scientific and Engineering Society Press.
Reprinted with permission.

Configurable Address Computation in a Parallel Memory Architecture

EERO AHO, JARNO VANNE, KIMMO KUUSILINNA, TIMO HÄMÄLÄINEN, and
JUKKA SAARINEN

Digital and Computer Systems Laboratory, Tampere University of Technology
Hermiankatu 3A, 33720 Tampere, FINLAND
eero.aho@tut.fi

Abstract: - Implementation of an address computation unit for a configurable parallel memory architecture is presented. The benefit of configurability is that a multitude of access formats and module assignment functions can be used with a single hardware implementation, which has not been possible in prior embedded parallel memory system. In addition to design and implementation, a simulation example is given for an image processing application. Results show significant reduction in the overall memory accesses, which is the key to solve the processor-memory bottleneck.

Key-Words: configurable parallel memory, address computation, conflict free access, module assignment function, page table

1 Introduction

For a number of years the processor versus memory performance gap has been widening. Advances in the DRAM technology are lagging behind the processor development, since the physical memory cell access time has not decreased very much although the memory sizes have grown rapidly. For these reasons, several methods and architectures to enhance the memory system performance have been presented [7].

One solution to the problem is to place a cache between the main memory and the processor. Widening the bus between the memory and cache makes it possible to refresh the cache in larger blocks of data. Unfortunately, not all of the data in such a refresh is necessary, which means that some data transfers take place in vain. It is also a matter of power consumption in addition to the time lost in such transfers.

Applications like video and graphics favors special data patterns that could be accessed in parallel. This is utilized in *parallel memories*, where the idea is to feed the processor with only necessary data [5].

Traditionally the parallel memory development has been concentrated on *module assignment functions (skewing schemes)* that try to ensure conflict free, truly parallel data accesses to as many *access formats (templates)*

as possible. *Linear* and *dyadic (xor)* functions are used frequently [1][2][4][6][8]. A skewing scheme where several different linear module assignment functions are used in one scheme is presented in [3]. When using a prime number of memory modules, it is possible to use much useful conflict free access formats, but arithmetic modulo of the prime number is required.

A general block diagram of a parallel memory architecture is shown in Fig. 1. There are an *address computation unit*, N *memory modules* and a *permutation unit* [5]. Depending on the access format (\mathbf{F}) and the address of the first element (\mathbf{r}), the address computation unit computes the right memory module(s) and addresses in the memory module. The permutation unit organizes the data in a correct order.

A problem with parallel memories has been that very often they are application specific with fixed access formats implemented in hardware. This ensures fastest speed, but requires redesign if other formats have to be included.

We have addressed the problem by designing a parallel memory, in which *address computation* is configurable. The address effectively includes the access format function, a module assignment function and *address function* [5], which can be changed at the run-time.

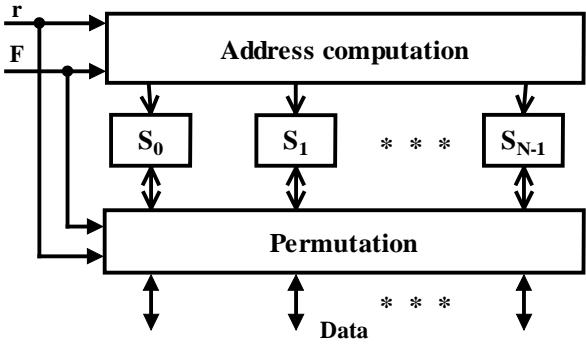


Fig. 1. General parallel memory architecture.

In this paper we present an implementation of the configurable address computation unit. We first introduce our configurable parallel memory architecture in Section 2, which is followed by the address computation implementation details in Section 3 and simulation results in Section 4. Section 5 concludes the paper.

2 Configurable Parallel Memory Architecture

A block diagram of our Configurable Parallel Memory Architecture (CPMA) is depicted in Fig. 2. The inputs to CPMA are the *virtual address* and the command (write, read). The virtual address includes a *page address*, which goes to a *page table*. Also included is a *page offset*, which directly gives the *scanning point r* for the parallel memory. Row address and access functions are given from the page table to the address computation unit. Other parts are DRAM banks and a permutation unit.

CPMA is designed to be configurable at design time: most implementation parameters are generic, which means that the number of DRAMs, size of a DRAM, width of the data bus and width of input buses are adjustable. All the needed blocks, logic and buses are implemented by logic synthesis from VHDL. In the following, we describe the CPMA blocks in more detail.

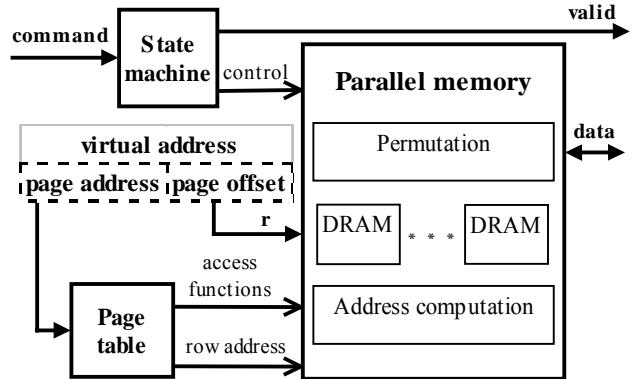


Fig. 2. Configurable parallel memory architecture

2.1 Page table

The page table is accessed by a page address, which determines the functionality in CPMA, briefly described in the following. Fig. 3 depicts the page table with two example rows highlighted for image processing (pixel data).

Row address in the page table is like a page number in a physical memory [7]. A single page should carry only one module assignment function ($S(i,j)$).

Length of coordinates L_i and L_j that gives a memory area (dimensions i and j) to be accessed by a scheme defined in this page entry. The values are scaled according to application.

Page	row	L_i	L_j	F	$S(i,j)$	$a(i,j)$
:						
18	6	4	4	G(1,1,5)	1,1,2,8	1,0,8
19	6	4	4	G(2,2,3)	1,1,2,8	1,0,8
:						

Fig. 3. An example of the page table.

Parameters for access format function (F). There are three kind of access formats, of which some examples are given in Fig. 4. The parameters are:

1. *Rectangle format* $R(a,b)$, where parameters a and b are the length of horizontal and vertical sides respectively.
2. *Generate format* $G(a_i,a_j,p)$, where parameters a_i and a_j are jumps in respective (i,j) directions and p gives how many elements are included. Parameter a_i may also be a negative integer.

3. *Free format* $F((i_0, j_0), (i_1, j_1), (i_2, j_2), \dots, (i_{p-1}, j_{p-1}))$, where the parameters are the exact coordinates (i, j) from the scanning point r . Parameter i_k may also be a negative integer. The maximum number of free format pixels p has to be determined before implementing the system so that the required bus widths can be implemented.

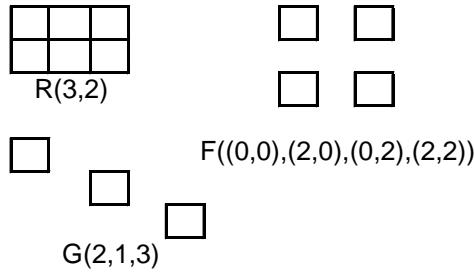


Fig. 4. Access format examples.

Parameters for a module assignment function $S(i, j)$. There are two kind of module assignment functions implemented in our system.

A *Linear function* is defined as

$$S(i, j) = (a \cdot i + b \cdot j) \bmod N,$$

where a, b and N are positive integers.

A *dyadic function* is formed as

$$S(i, j) = \pi(i \bmod N)_2 \oplus \tau(j \bmod N)_2,$$

where $N = 2^m, m \in \{0, 1, 2, \dots\}$ and N is a positive integer. The operation \oplus denotes component-wise xor. π and τ are permutations of the components of the binary numbers $(i \bmod N)$ and $(j \bmod N)$ respectively. This is $\pi(i \bmod N)_2 = \pi(i_{m-1} | \dots | i_0)$, where i_k is a binary number. At the moment, we have two commonly used permutations [5] that are following:

$$\text{REVERSE}(i_{m-1} | \dots | i_1 | i_0) = (i_0 | i_1 | \dots | i_{m-1})$$

and for $N = 2^{2m}, m \in \{0, 1, 2, \dots\}$

$$\text{SWAP}(i_{2m-1} | \dots | i_m | i_{m-1} | \dots | i_0) =$$

$$(i_{m-1} | \dots | i_0 | i_{2m-1} | \dots | i_m)$$

Parameters for address function $a(i, j)$. We have one address function implemented: a linear function is formed as $a(i, j) = b + (i + j \cdot L_i)/N$, where b , L_i and N are positive integers. L_i is determined in the L_i column in the page table.

3 Implementation

A block diagram of the configurable address computation unit is depicted in Fig. 5. The inputs to the unit are a bus for the access functions and the scanning point r (Fig. 2).

Inside the unit, a *format control unit* receives the access format function, the scanning point r and the length of coordinates L_i and L_j . This unit specifies the elements (coordinates) that are accessed in the original memory area. The *address decode unit* computes physical addresses for each DRAM memory module, and the *module assignment unit* directs the physical addresses for the proper memory modules.

3.1 Format Control Unit

A detailed block diagram of the format control unit is depicted in Fig. 6a. The scanning point is originally represented by an integer, which has to be converted into the form $r(i, j)$. It is done as follows: $r(i) = r \bmod L_i$ and $r(j) = r / L_i$ (Fig. 6c).

Relative coordinates for the accessed elements in the original memory area are determined in (i, j) format in the *form relative block* (Fig. 6b). One of the access formats, rectangle (R), generate (G) or free (F), is selected according to specified format. A *fix block* ensures that the coordinates fall in the original memory area.

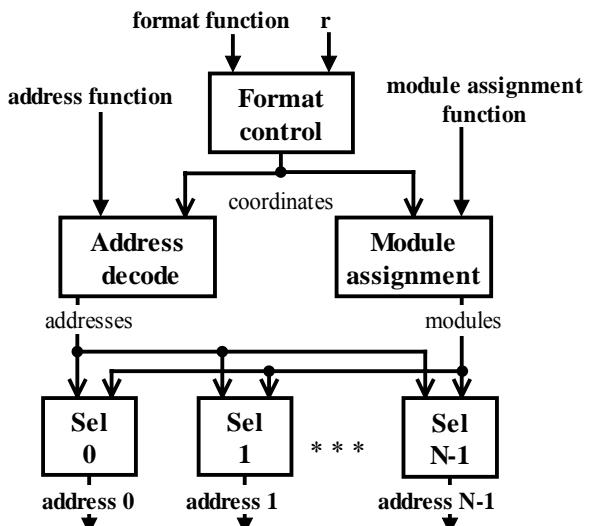


Fig. 5. Configurable address computation unit.

Absolute coordinates are determined in the Form absolute block (not shown). The relative coordinates and $r(i,j)$ are added up in parallel there. Again, with L_i and L_j the coordinates that go over borders are fixed.

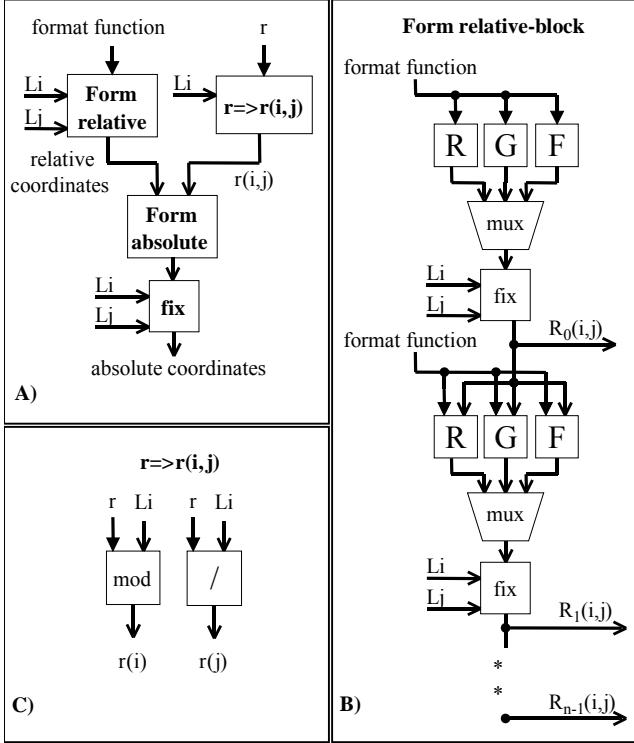


Fig. 6. Format control.

3.2 Address decode and module assignment

Block diagrams of the address decode, module assignment and address select blocks are depicted in Fig. 7. In the address decode block, L_i is first left shifted according to scaling factor of the application. Then, arithmetic operations multiplication, addition, division are carried out in a sequence.

At the same time, the module assignment block determines module numbers that are selected either by linear or dyadic module assignment function. Linear function uses two multipliers, an adder and a modulus operation. Dyadic function uses two modulus operations, which are simple to implement if N is a power of two. After that either REVERSE or SWAP permutation is selected and bitwise XOR operations carried out.

Addresses and module numbers for each absolute coordinate are determined in parallel. Each address select block k ($k \in \{0, \dots, N-1\}$) depicted in Fig. 7 specifies a physical address.

There are N comparators and a multiplexer in the block.

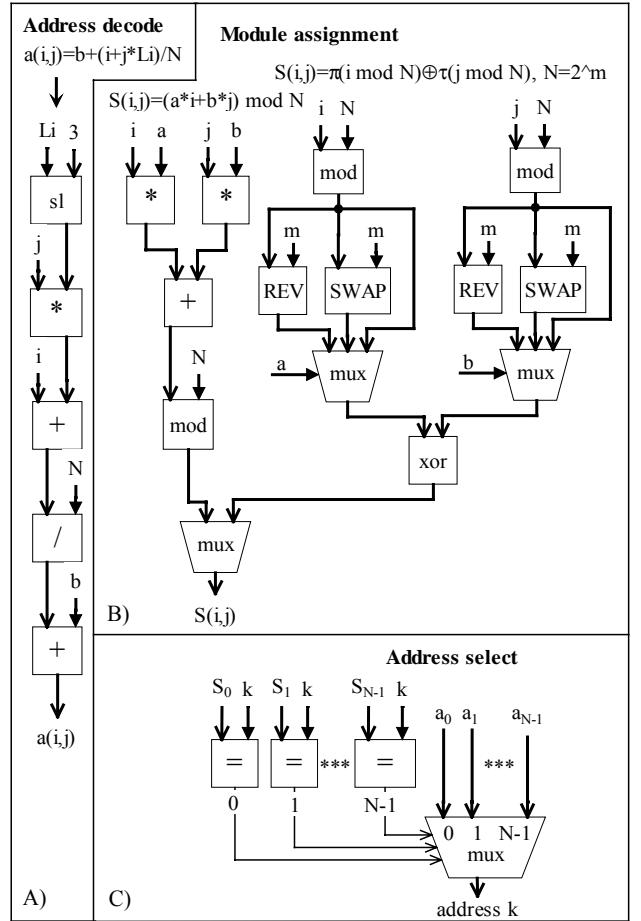


Fig. 7. Address decode, Module assignment, and Address select.

3.3 Memory modules and permutation

Each memory module has its own DRAM controller that gets as inputs control signals from the state machine, row address from the page table and the address from the address computation unit. A bidirectional bus transmits the data to or from the permutation unit.

The data, which comes from or goes to memory modules, might not be in the right order. With help of the Format control and the Module assignment blocks (Fig. 5) the Permutation unit switches the data in the correct order.

4 Simulation Example

In this example, data is at first written to and then read from the parallel memory system.

There are eight memory modules and a generate access format G(1,1,5) when writing and G(2,2,3) when reading is used. The formats are depicted in Fig. 8. The page addresses used are depicted in Fig. 3. The module assignment function and the address function have to be the same when accessing the same page.

At first, the “write” command, data to be written, page address 18, and desired page offset ($r = 35$) are send to CPMA. Values in the row number 18 in the Page table are given to the parallel memory and the scanning point $r(i,j)$ is determined, for example: $r(i,j) = (3,1)$. At the same clock cycle relative coordinates (Fig. 6) are determined in (i,j) format that are (0,0),(1,1),(2,2),(3,3),(4,4). After that the absolute coordinates could be determined as follows: (3,1),(4,2),(5,3),(6,4),(7,5).

The module assignment and address values for the absolute coordinates are determined by $S(i,j)$ and $a(i,j)$ in the page table. The first parameter of $S(i,j)$ tells that the linear module assignment function is used. Others parameters tell that the function is $(1*i+2*j) \bmod 8$.

An address function is determined respectively to be $0+(i+j*L_i)/8$. In our example $S=(5,0,3,6,1)$ and $a=(4,8,12,16,20)$. After that, the right column address is selected for a memory module (e.g. module number 3 have the address 12). With a valid signal the state machine informs that the “write” operation is ready.

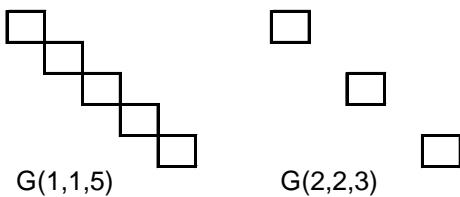


Fig. 8. Writing and reading access formats in the example.

When reading, the command, page address 19 and the same page offset ($r = 35$), as when writing, are send to the system. Relative coordinates ((0,0),(2,2),(4,4)), absolute coordinates ((3,1),(5,3),(7,5)), module assignment values ($S=(5,3,1)$) and address values ($a=(4,12,20)$) are determined and the column addresses are send to right memory modules respectively. Data is read from the

DRAMs and switched to a correct order in the permutation unit. The valid signal goes up to show that the “read” operation is done.

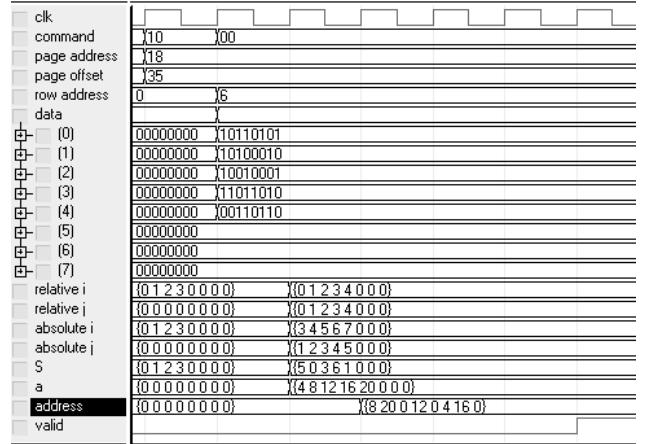


Fig. 9. Simulation wave window.

The example given above is simulated in a register transfer level code. A screen capture of the write operation simulation is depicted in Fig. 9. The “write” command (bit vector “10”), page address (18), page offset (35) are first given as inputs and after a clock cycle data to be written. At the same cycle with the data, the row address (6) is sent from the page table to the DRAM controllers, and parameters for access functions are send to the address computation (not shown). The relative and absolute coordinates, module assignment (S) and address (a) values are computed in one clock cycle. Addresses are routed to the DRAM controllers at the next clock cycle. At the same cycle, incoming data is permuted in the permutation unit. It takes three more clock cycles to write data to the determined address of the DRAMs. At the end of the operation, the valid signal goes up to show that operation is done.

The advantage of using our parallel memory system is apparent if access times are compared when processing a picture with different access formats. If a QCIF (Quarter Common Intermediate Format) picture containing 176*144 pixels, is written to memory a single pixel at a time it takes $176*144 = 25344$ write operations. (Assuming every pixel is written individually.) When writing the same picture with G(1,1,5) access format, as depicted in Fig. 8, it takes 5104 write operations. That is 4.97 times less than with a single access format. On

the other hand, when writing the picture with G(2,2,3) access format it takes 8448 write operations, which is exactly 3 times less than with a single access format. In our implementation, write operations take seven clock cycles.

Area efficient implementation of multiplication, division and modulus operations favors that operands are powers of two. However, this is not always possible. For example, the size of QCIF picture is 176*144 pixels which are not power of two. In addition, restricting the number of memory modules to a power of two would simplify the implementation further.

5 Conclusions

The discussed Configurable Parallel Memory Architecture enables a computer to use a multitude of memory access templates. These access formats can guarantee the usefulness of data and data refresh operations are required less frequently. In addition, the configurability of CPMA facilitates running several different computation threads, each with a unique access scheme, in a single system. This has not been possible in the prior embedded parallel memory systems.

In this paper, we have presented an implementation of the configurable address computation. The access format function, the module assignment function and the address function can flexibly be changed at run-time. Some generally used functions are implemented and the parameters for those functions specified by software.

In the future, our research will focus on the pipelining of the parallel memory system. When a DRAM module is locating its first address, the next address is being computed in the Address computation block simultaneously. Other module assignment functions and access formats are studied for their implementation feasibility.

References

- [1] Batcher K.E., The Multidimensional Access Memory in STARAN. *IEEE Trans. Computers*, Vol. C-26, Feb. 1977, pp. 174-177.
- [2] Budnik, P., Kuck, D.J., The Organization and Use of Parallel Memories. *IEEE Trans. Computers*, Vol. C-20, Dec. 1971, pp. 1566-1569.
- [3] Deb, A., Multiskewing – A Novel Technique for Optimal Parallel Memory Access. *IEEE Trans. Parallel and Distributed Syst.*, Vol. 7, No. 6, June 1996, pp. 595-604.
- [4] Frailong, J.M., Jalby W., Lenfant J., *XOR-Schemes: A Flexible Data Organization in Parallel Memories*. Proc. Int'l Conf. Parallel Processing, Washington DC, 1985, pp. 276-283.
- [5] Gössel, M., Rebel, B., Creuzburg, R., *Memory Architecture & Parallel Access*. Morgan Kaufmann, Amsterdam, The Netherlands, 1994.
- [6] Lawrie, D.H., Access and Alignment of Data in an Array Processor, *IEEE Trans. Computers*, Vol. C-24, Dec. 1975, pp. 1145-1155.
- [7] Patterson, D., Hennessy, J., *Computer organization and design: the hardware/software interface*. Morgan Kaufmann, San Francisco, California, 1994.
- [8] Wijshoff, H.A.G., van Leeuwen, J., On Linear Skewing Schemes and d -Ordered Vectors. *IEEE Trans. Computers*, Vol. C-36, No. 2, Feb. 1987, pp. 233-239.

Publication 2

E. Aho, J. Vanne, K. Kuusilinna, and T. Hämäläinen, “Diamond Scheme Implementations in Configurable Parallel Memory,” in *Proceedings of the IEEE International Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, Brno, Czech Republic, pp. 211–218, April 2002.

Reprinted, with permission, from the proceedings of DDECS 2002.

DIAMOND SCHEME IMPLEMENTATIONS IN CONFIGURABLE PARALLEL MEMORY

Eero Aho, Jarno Vanne, Kimmo Kuusilinna, and Timo Hämäläinen

Institute of Digital and Computer Systems, Tampere University of Technology

Korkeakoulunkatu 1, FIN-33720 Tampere, Finland

eero.aho@tut.fi

Abstract. *Parallel memories increase memory bandwidth with several memory modules working in parallel and can be used by the processor to feed only necessary data. The Configurable Parallel Memory Architecture (CPMA) enables a multitude of access formats and module assignment functions to be used within a single hardware implementation, which has not been possible in prior embedded parallel memory systems. In this paper, a general diamond scheme implementation for a previously implemented CPMA framework is presented. This solution can be utilized to alleviate the processor-memory performance bottleneck.*

1 Introduction

The processor versus memory performance gap has been widening for a number of years. Cache increases memory bandwidth in general purpose systems but it also increases latency. Wide memory can refresh the cache in larger blocks of data. Unfortunately, this can signify the fact that the cache contains a lot of currently unnecessary data.

Some applications, especially in video and graphics, favor special data patterns that can be accessed in parallel. This phenomenon is utilized in *parallel memories*, where the idea is to increase memory bandwidth with several memory blocks working in parallel and feed the processor with only necessary data. Traditionally the parallel memory development has been concentrated on *module assignment functions (skewing schemes)* that try to ensure *conflict free* parallel data accesses to as many *access formats (templates)* as possible. Mostly *linear* module assignment functions and *XOR-schemes* have been used with parallel memories [1, 2, 3]. *Periodic* and *diamond* schemes have also been studied [3, 4, 5].

A generalized block diagram of a parallel memory architecture is shown in Fig. 1. The figure depicts an *Address Computation* unit, N *memory modules* S_0, S_1, \dots, S_{N-1} , and a *Permutation* unit. Depending on the access format F and the address of the first element r , the Address Computation unit computes the addresses and directs them to the right memory modules. The Permutation unit organizes the data into correct order.

A typical problem with previously presented parallel memory implementations is that they are application specific with fixed access formats implemented in hardware. This ensures the fastest speed, but requires redesign if other formats have to be included. In this work, the problem has been addressed by implementing the *Configurable Parallel Memory Architecture (CPMA)* [6], in which access formats and the needed address generation functions can be changed at run-time.

Time-to-market, increasing complexity, and system performance requirements have necessitated the increasing utilization of reusable intellectual property (IP) blocks and programmable platforms in system-on-chip (SoC) designs. CPMA can be viewed as one step to this direction since it is able to cope with a multitude of algorithms needed in different applications without redesign of the memory system.

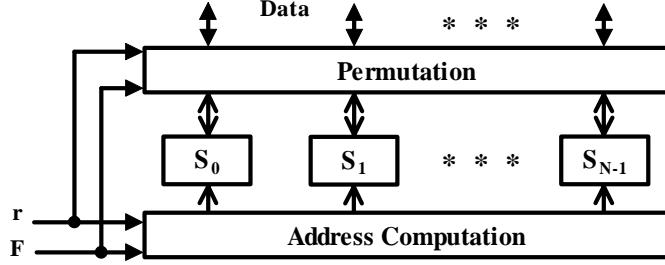


Fig. 1. General parallel memory architecture.

Promising technology development for CPMA includes embedded DRAM designs, where DRAM is embedded to the same die as the processor core. This enables the use of wide data buses, low latency, and vector processor like execution, which are all very attractive properties if data can easily be supplied from parallel memories.

Previous implementations of CPMA included linear module assignment functions and XOR-schemes [7, 8]. In this paper, a generalized implementation of a diamond scheme for the implemented CPMA is presented. An overview of the implemented CPMA is given in Section 2. Diamond scheme and address function implementation details are introduced in Section 3. Results are given in Section 4 and Section 5 concludes the paper.

2 Configurable Parallel Memory Architecture

A block diagram of the implemented CPMA is depicted in Fig. 2. The inputs to CPMA are *virtual address* and *command* (write, read). The virtual address includes a *page address*, which is a pointer to a *Page Table*. *Page offset* directly indicates the *scanning point r* for the parallel memory. The *row address* and the access functions are fed from the *Page Table* to the *Address Computation* unit. In addition, DRAM banks and a Data Permutation unit are required for CPMA operation. [7]

CPMA Address Computation is implemented using multiple dedicated computation units in parallel. There is one unit for each class of access formats, module assignment functions, and address functions that the implementation supports. Moreover, a configurable Data Permutation unit is designed to conform to system specifications.

CPMA is designed to be configurable both at design time and at run-time. In the VHDL implementation, many parameters are generics, which means that the number of DRAMs, size of a DRAM module, width of the data bus, and width of input buses are adjustable. All the required blocks, logic, and buses are implemented with logic synthesis from VHDL. In the following, we describe the run-time configurability and some of the CPMA blocks in more detail.

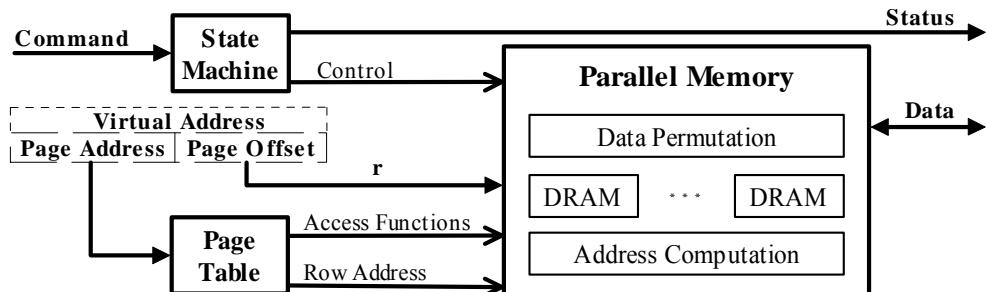


Fig. 2. Configurable Parallel Memory Architecture.

2.1 Page Table

The application software can change the contents of the page table at run-time. The page address is a pointer to a specific row in the page table, which determines the CPMA functionality during that memory access. Page table field contents are explained in the following.

Row address in the page table is like a page number in a physical memory. **Length of coordinates** L_i and L_j indicate a memory area (dimensions i and j) to be accessed with the scheme defined in this page entry. **Access format function** F determines the data elements that are read or stored in parallel. **Module assignment function** $S(i,j)$ indicates the assignment of a data element to the memory module. Just one data element can be accessed by an access format from one memory module at a time for *conflict free* access. The utilized module assignment function determines the access formats that can be used conflict free. Linear module assignment functions and XOR-schemes (also called dyadic functions) are previously implemented in our system [7, 8]. **Address function** $a(i,j)$ determines the address in a memory module for the accessed data element.

When a user specifies an algorithm for CPMA, this process includes defining the contents of the associated page table row. First, the required access formats for the algorithm are evaluated. A module assignment function for conflict free access has to be chosen. Moreover, a suitable address function for the module assignment has to be found. Finally, all this information is stored in the page table.

2.2 Address Computation

A block diagram of the Address Computation unit is depicted in Fig. 3. The inputs to the unit are the access functions and the scanning point r (Fig. 2). The row address is directed to the DRAMs. The specific memory module addresses are the outputs from this unit.

Inside the unit, a *Format Control* unit receives the access format function, the scanning point r , and the length of coordinates L_i and L_j . This unit specifies the coordinates of the data elements that are accessed in the scanning field. The *Module Assignment* unit computes the specific memory modules that contain the accessed data elements. The *Address Decode* unit, in turn, computes the physical (column) addresses for each memory module. After that, the *Address Select* units direct the physical addresses for the proper memory modules.

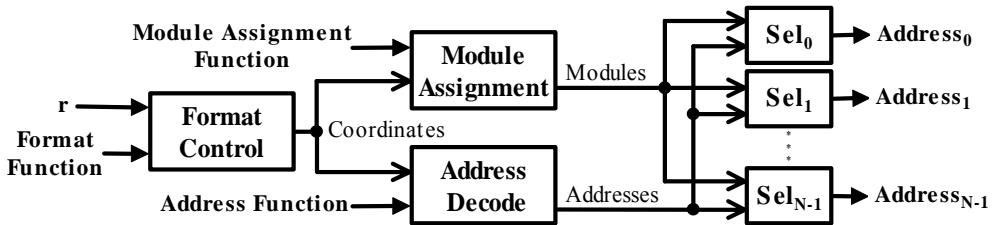


Fig. 3. Address Computation unit.

3 Diamond schemes

A module assignment function is called (N_1, N_2) -periodic ($N_1, N_2 > 0$) if for all (i, j) and all integers k_1 and k_2

$$S(i, j) = S(i + k_1 N_1, j + k_2 N_2). \quad (1)$$

S defines the utilized memory module number. The values (N_1, N_2) form a rectangle, *basic domain*, of side lengths N_1 and N_2 . The basic domain of a periodic function may be any kind of rectangle, but this discussion concentrates on axially oriented rectangles because of their simplicity in hardware implementation. [3, 4]

An example of a (2,2) -periodic scheme is depicted in Fig. 4a where the number of memory modules $N=4$. 8x4 data elements are displayed. The data elements are stored in a memory module with a number shown in the corresponding table cell. The elements are indexed with (i,j) coordinates. For example, a data element located at point $(2, 1)$ is stored in memory module 2. The basic domain (painted gray) is on the left up corner of the scheme containing the memory module numbers 0, 1, 2, and 3. The basic domain is copied to all over the skewing scheme. Square access formats are conflict free all over the scanning field when the number of accessed elements M is equal to N . However, rows and columns are not conflict free when $M=N$.

Multiperiodic skewing schemes are discussed in [5]. Special multiperiodic functions called *diamond schemes* are of interest for specific applications [3]. The basic domain in diamond schemes is formed as with periodic functions, but there are also two permutations π_1 and π_2 . The permutations change the order of the basic domain when i and j increase respectively.

As an example, there is a diamond scheme depicted in Fig. 4b with the same basic domain as in Fig. 4a. The permutations are

$$\pi_1(x) = (x + 2) \bmod 4, \quad \pi_2(x) = (x + 1) \bmod 4. \quad (2)$$

Permutations (2) change the order of the module numbers relative to the periodic rectangle next to it. The following is an example how to determine new module numbers $S(i,j)$:

$$S(2,1) = \pi_1(S(0,1)) = \pi_1(2) = (2 + 2) \bmod 4 = 0,$$

$$S(2,3) = \pi_2(S(2,1)) = \pi_2(0) = (0 + 1) \bmod 4 = 1.$$

This scheme enables conflict free access to rows and columns in all the points, however, square accesses are restrictedly conflict free when $M=N$. For example, a square access format sized 2x2 is conflict free when the scanning point is located at $r(i,j)=(3,0)$ since all the accessed elements are located in different memory modules (Fig. 4b). However, the same access format is not conflict free when $r(i,j)=(3,1)$ because two of the data elements are accessed from the same memory module 1. These data elements are marked with dots “•” in Fig. 4b.

a)	0	1	2	3	4	5	6	7	
0	0	1	0	1	0	1	0	1	i
1	2	3	2	3	2	3	2	3	
2	0	1	0	1	0	1	0	1	
3	2	3	2	3	2	3	2	3	
									j

b)	0	1	2	3	4	5	6	7	
0	0	1	2	3	0	1	2	3	i
1	2	3	0	1	2	3	0	1	
2	1	2	3	0	1	2	3	0	
3	3	0	1	2	3	0	1	2	
									j

Fig. 4. Example of periodic and multiperiodic (diamond) schemes.

3.1 Generalized module assignment for diamond schemes

When implementing diamond schemes, the construction of a basic domain and the two permutations π_1 and π_2 need to be determined.

Let us choose to form an axially oriented basic domain with a linear function (3) or a dyadic function (4).

$$S_b(i_b, j_b) = (a \cdot i_b + b \cdot j_b) \bmod N. \quad (3)$$

$$S_b(i_b, j_b) = \mathbf{A}I_b \oplus \mathbf{B}J_b. \quad (4)$$

\mathbf{A} and \mathbf{B} are $(n \times u)$ and $(n \times v)$ matrices containing binary numbers and $I = (i \bmod 2^u)_2$ and $J = (j \bmod 2^v)_2$ in (4). The dyadic function (4) implementation for CPMA is introduced in [8]. The side lengths of a basic domain, N_1 and N_2 , are restricted to powers of two because of hardware simplicity. Due to the restriction, the number of memory modules N is desirable to be a power of two. The variables (i_b, j_b) are respective basic domain coordinates of the coordinates (i, j) as in (5). The permutations π_1 and π_2 are formed using simple functions (6).

$$i_b = i \bmod N_1, \quad j_b = j \bmod N_2. \quad (5)$$

$$\pi_1(x) = (x + c_1) \bmod N, \quad \pi_2(x) = (x + c_2) \bmod N. \quad (6)$$

The parameters c_1 and c_2 are positive integers in (6) and the number of memory modules N is same as in (3) because all the periodic rectangles use as many memory modules as the basic domain. This kind of permutations are used with diamond schemes in [9] and [10].

Using the functions (3) – (6), a module number can be calculated using the following formula:

$$S(i, j) = \left[S_b(i_b, j_b) + c_1 \cdot \left\lfloor \frac{i}{N_1} \right\rfloor + c_2 \cdot \left\lfloor \frac{j}{N_2} \right\rfloor \right] \bmod N, \quad (7)$$

where $S_b(i_b, j_b)$ is the respective module number in the basic domain. $\lfloor i/N_1 \rfloor$ and $\lfloor j/N_2 \rfloor$ indicate the number of the utilized periodic rectangle in i and j – directions, respectively.

A block diagram of the implementation of the function (7) is depicted in Fig. 5. First, linear or dyadic function is selected. In parallel, the number of the utilized periodic rectangle is counted in i and j -directions and multiplied with the constants c_1 and c_2 respectively. Then, the respective module number in the basic domain $S_b(i_b, j_b)$ and the permutation differences are added and the modulus is taken to get the module number $S(i, j)$. The used modulus and division arithmetic calculations are easy to implement, because N_1 , N_2 , and N are powers of two. Moreover, the modulus in linear function (3), depicted on the right side of Fig. 5, is easy to implement because of N being a power of two. Those calculations can be implemented in hardware by masking and shifting.

For each memory module, four multipliers, two adders, four modulus units (masking), two dividers (shifting), one multiplexer, several AND, and XOR ports are used. Two multipliers and an adder can be shared with the previously implemented linear module assignment function. The dyadic function resources can be used to implement the basic domain of a dyadic function.

Periodic functions are a subclass of diamond schemes. They are also available with the implementation depicted in Fig. 5. Then, the permutation variables c_1 and c_2 are set to zero.

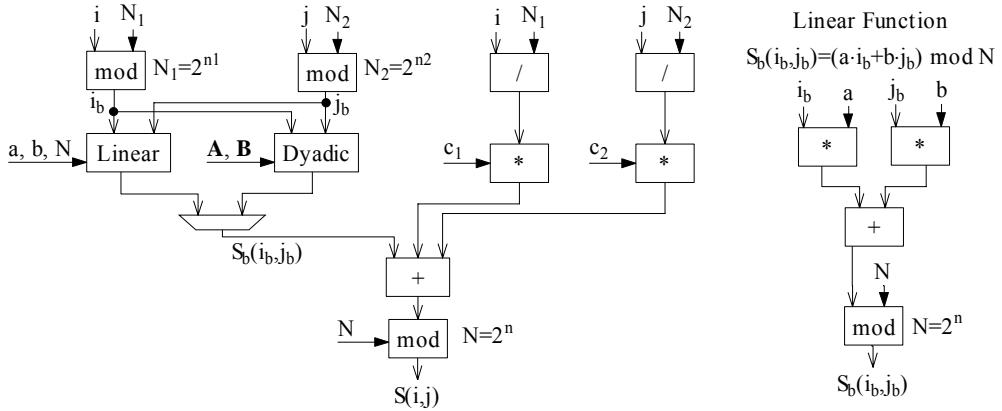


Fig. 5. Generic implementation of the diamond scheme.

The following variables are needed in the page table for the proposed diamond schemes:

- Parameters n_1 and n_2 for lengths of the basic domain N_1 and N_2 ($N_1 = 2^{n_1}, N_2 = 2^{n_2}$).
- Parameters for the used basic domain function, linear (a, b , and N) or dyadic (\mathbf{A} and \mathbf{B}).
- Parameters for the permutations c_1 and c_2 .
- With dyadic function, the parameter n for the final modulus calculation ($N = 2^n$).

3.2 Address function implementations for diamond schemes

Address function is used to determine the physical addresses in the memory modules for the accessed data elements. The address function has to be determined so that two elements are not located in the same address of the same memory module. In other words, an address function has to be suitable for the used module assignment function.

The previously implemented address function [7] is suitable for many module assignment functions. However, the presented diamond scheme implementation can form skewing schemes for which that address function is not suitable. The address function (8) is developed for the diamond schemes using the same notations as in (7). N_1 and N_2 are the side lengths of a basic domain in i and j –directions respectively and L_i is the length of coordinates in i –direction in (8). N_1 and N_2 are restricted to powers of two as were done with the function (7). L_i is should be divisible by N_1 . The address function (8) is used with the diamond schemes in [9].

To ease the implementation and extend the range of usable address functions, the function (8) is modified as function (9). Division L_i/N_1 can be predetermined and, therefore, is changed to d . Moreover, the scanning field can be divided into multiple parts with the added parameter b . Parameters b and d are positive integers but c_1 and c_2 are restricted to powers of two in (9).

$$a(i, j) = \left\lfloor \frac{i}{N_1} \right\rfloor + \frac{L_i}{N_1} \cdot \left\lfloor \frac{j}{N_2} \right\rfloor \quad (8) \quad a(i, j) = b + \left\lfloor \frac{i}{c_1} \right\rfloor + d \cdot \left\lfloor \frac{j}{c_2} \right\rfloor \quad (9)$$

A block diagram of the implementation of the address function (9) is depicted in Fig. 6. The i and j coordinates are divided by c_1 and c_2 respectively and j is multiplied by d . Then, the results and the parameter b are added to get the address $a(i, j)$. The divisions are simple operations because c_1 and c_2 are powers of two.

For each memory module, one multiplier, one adder, and two dividers (shifting) are used. A multiplier and an adder can be shared with the previously implemented address function introduced in [7].

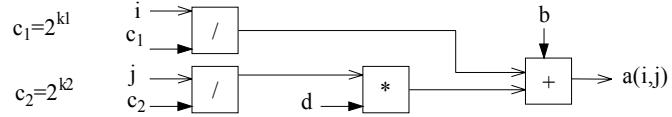


Fig. 6. Implementation of the address function (9).

4 Results

The introduced generic diamond scheme and the address function (9) were implemented with synthesizable register transfer level VHDL. The area of the implementation depends on the implementation parameters, especially, the number of DRAMs, size of a DRAM module, and widths of the buses.

The advantage from using the implemented Configurable Parallel Memory Architecture is illustrated with a video processing example. The half-pixel motion estimation in H.263 video encoding is usually performed in the eight locations around the found full-pixel location. If the search area is interpolated, only every other pixel in the search area is required for the access. Therefore, a crumbled rectangle access format shown in Fig. 7 (colored gray) can be utilized in the half-pixel motion estimation and interpolation when the number of memory modules is sixteen [10]. That access format can be used conflict free within these diamond scheme and address function implementations. For example, the diamond scheme depicted on the left side of Fig. 7 fulfills the requirements. Only part of the needed memory area is shown. A suitable address function mapping is depicted on the right side of Fig. 7.

The search area is interpolated and, as a simple performance example, a sum of absolute difference (SAD) is calculated for a 16x16 pixel block in the half pixel motion estimation. When the search area is accessed a single pixel at a time, it takes 256 read operations. When reading with the access format shown in Fig. 7, it takes 16 read operations. That is 16 times less than with a single pixel access format. Moreover, the configurability of CPMA enables the module assignment function and the address function to be changed at run-time when different access formats are required by other applications.

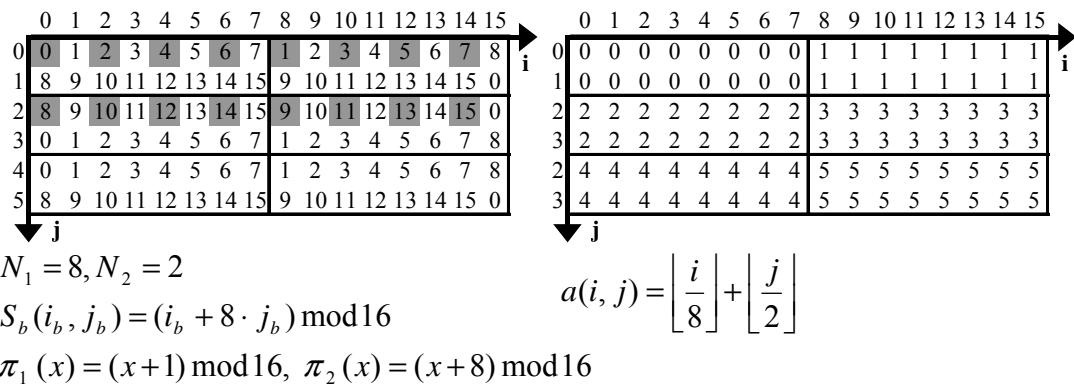


Fig. 7. Access format (colored gray), module assignment function, and address function for half-pixel motion estimation and output of interpolation.

5 Conclusions

The Configurable Parallel Memory Architecture enables a processor to use a multitude of memory access templates. These access formats can guarantee the usefulness of data and data refresh operations are required less frequently. In addition, the configurability of CPMA facilitates running several different computation threads, each with a unique skewing scheme, in a single system.

In this paper, a generic implementation of a diamond scheme for the implemented CPMA was presented. The implemented schemes allow several additional conflict free access formats to be used. Parameters for constructing a diamond scheme can flexibly be changed at run-time without a need to redesign the whole hardware.

In the future, our research will focus on clarifying the access formats that are required in special applications. Especially templates that can be utilized with algorithms used in video and graphics are studied. Moreover, other module assignment functions are under research for their implementation feasibility.

References

- [1] Budnik, P., Kuck, D., J.: The Organization and Use of Parallel Memories. *IEEE Transactions on Computers*, Vol. C-20, No. 12, 1971, pp. 1566-1569.
- [2] Liu, Z., Li, X.: XOR Storage Schemes for Frequently Used Data Patterns. *Journal of Parallel and Distributed Computing*, Vol. 25, No. 2, 1995, pp. 162-173.
- [3] Gössel, M., Rebel, B., Creuzburg, R.: *Memory Architecture & Parallel Access*. Amsterdam (The Netherlands), Elsevier Science B.V. 1994, p. 246.
- [4] Shapiro, H., D.: Theoretical Limitations on the Efficient Use of Parallel Memories. *IEEE Transactions on Computers*, Vol. C-27, No. 5, 1978, pp. 421-428.
- [5] Tel, G., Wijshoff, H., A., G.: Hierarchical Parallel Memory Systems and Multiperiodic Skewing Schemes. *Journal of Parallel and Distributed Computing*, Vol. 7, No. 2, 1989, pp. 355-367.
- [6] Kuusilinna, K.: Studies and Implementations of Bus-based Interconnections. Doctoral Dissertation, Department of Information Technology, Tampere University of Technology, Tampere (Finland), 2001, p. 201.
- [7] Aho, E., Vanne, J., Kuusilinna, K., Hämäläinen, T., Saarinen, J.: Configurable Address Computation in a Parallel Memory Architecture. Antoniou, G., Mastorakis, N., Panfilov, O. (Ed.): *Advances in Signal Processing and Computer Technologies*. Athens (Greece), WSES Press 2001, pp. 390-395.
- [8] Aho, E., Vanne, J., Kuusilinna, K., Hämäläinen, T.: XOR-scheme Implementations in Configurable Parallel Memory. Accepted to International Workshop on System-on-Chip for Real-Time Applications, Banff (Canada), July 2002.
- [9] Creutzburg, R., Niittylahti, J., Sihvo, T., Takala, J., Tanskanen, J.: Parallel Memory Architectures for Video Encoding Systems, Part II: Applications. Proceedings of the International Workshop on Spectral Techniques and Logic Design for Future Digital Systems, Tampere (Finland), June 2000, pp. 567-594.
- [10] Tanskanen, J., Sihvo, T., Niittylahti, J., Takala, J., Creutzburg, R.: Parallel Memory Access Schemes for H.263 Encoder. Proceedings of the IEEE International Symposium on Circuits and Systems, Geneva (Switzerland), May 2000, pp. 691-694.

Publication 3

E. Aho, J. Vanne, K. Kuusilinna, and T. Hämäläinen, “Access Format Implementations in Configurable Parallel Memory,” in *Proceedings of the International Conference on Computer and Information Science (ICIS)*, Seoul, Korea, pp. 59–64, August 2002.

Reprinted from the proceedings of ICIS 2002.

Access Format Implementations in Configurable Parallel Memory

Eero Aho, Jarno Vanne, Kimmo Kuusilinna, and Timo Hämäläinen

Institute of Digital and Computer Systems, Tampere University of Technology

Korkeakoulunkatu 1, FIN-33720 Tampere, Finland

eero.aho@tut.fi

Abstract

Parallel memories increase memory bandwidth with several memory modules working in parallel and can be used to feed a processor with only necessary data. The Configurable Parallel Memory Architecture (CPMA) enables a multitude of access formats and module assignment functions to be used within a single hardware implementation, which has not been possible in prior embedded parallel memory systems. This paper presents a set of access format implementations for the previously implemented CPMA framework. These access formats can be utilized in several applications. In addition, this is a partial solution to the processor-memory performance bottleneck. The utilized resources are shown to be linearly proportional to the number of memory modules.

1. Introduction

The processor versus memory performance gap has been widening for a number of years. A cache memory can increase system memory bandwidth in general purpose systems, but it also tends to increase latency. Wide memories can refresh the cache in large blocks of data. Unfortunately, this might mean that the cache contains a lot of data that the currently running algorithm does not use.

Certain applications, particularly some video and graphics designs, favor special data patterns that can be accessed in parallel. This phenomenon is utilized in *parallel memories*, where the idea is to increase memory bandwidth with several memory blocks working in parallel and to feed processors with only algorithm specific data. Traditionally the parallel memory research has concentrated on *module assignment functions (skewing schemes)* that try to ensure *conflict free* parallel data accesses to the required *access formats (templates)*. *Linear functions, XOR-schemes, periodic schemes and diamond schemes* have been used to implement the module assignment in these parallel memories [1,2]. However, the *Configurable Parallel Memory Architecture (CPMA)* [3]

facilitates the utilization of all these application specific methods in a single embedded system.

A generalized block diagram of a parallel memory architecture is depicted in Figure 1. The functional blocks in Figure 1 are an *Address Computation* unit, N memory modules S_0, S_1, \dots, S_{N-1} , and a *Permutation* unit [2]. Depending on the access format F and the address of the first element r , the Address Computation unit computes the addresses and directs them to the right memory modules. The Permutation unit organizes the data into correct order specified by the access format.

A typical problem with the previously presented parallel memory implementations is that they are application specific with fixed access formats implemented in hardware. This ensures the fastest speed, but requires redesign if other formats have to be included. To address this issue, the Configurable Parallel Memory Architecture can change access formats and the required address generation functions at run-time.

Previous implementations of CPMA included a number of access format implementations [4]. This paper presents an extended set of access formats. An overview of CPMA is given in Section 2. Sections 3 and 4 present the access format implementation details. Results are given in Section 5 and Section 6 concludes the paper.

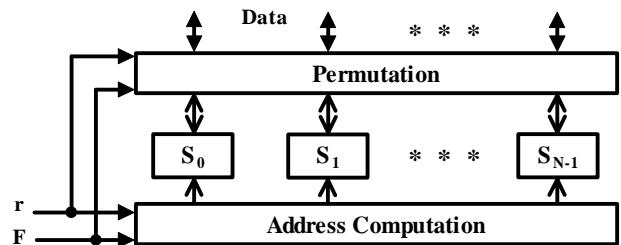


Figure 1. A generalized block diagram of parallel memory architecture.

2. Configurable Parallel Memory Architecture

A block diagram of Configurable Parallel Memory Architecture is depicted in Figure 2. The inputs to CPMA are *Virtual Address* and *Command* (write, read). The Virtual Address includes a *Page Address*, which is a pointer to a *Page Table*. *Page Offset* directly indicates the *scanning point r* for the parallel memory. The *Row Address* and the Access Functions are fed from the *Page Table* to the *Address Computation* unit. In addition, DRAM banks and a Data Permutation unit are required for CPMA operation. [4]

CPMA address computation is implemented using multiple application specific, configurable computation units in parallel. One unit is dedicated for each class of access formats, module assignment functions, and address functions that the implementation supports. Moreover, a configurable Data Permutation unit is designed to conform to system specifications.

CPMA is designed to be configurable both at design time and at run-time. In the VHDL implementation, many parameters are generics, which means that the number of DRAMs, size of a DRAM module, width of the data bus, and width of input buses are adjustable. Except for the memory modules, all the required blocks, logic, and buses are implemented with logic synthesis from VHDL. In the following, we describe the run-time configurability and some of the CPMA blocks in more detail.

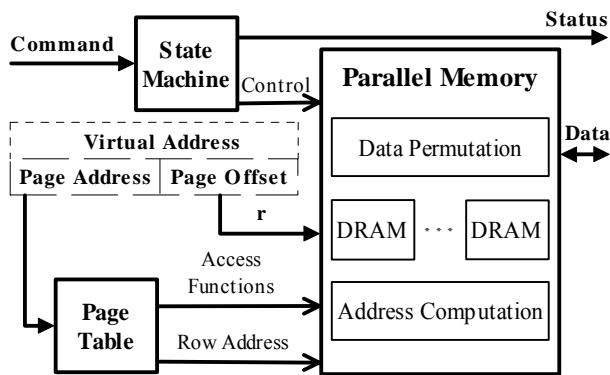


Figure 2. Configurable Parallel Memory Architecture.

2.1. Page Table

The application software can change the contents of the Page Table at run-time. The Page Address is a pointer to a specific row in the Page Table, which determines the CPMA functionality during that memory access. The Page Table field contents are explained in the following.

Row Address has a similar function as a page number in normal memory architecture. **Length of coordinates L_i**

and L_j restrict the addressable two-dimensional memory area, the *scanning field*, in dimensions i and j . The data elements are read or stored in parallel with an **access format F**. Parameters for constructing an access format are set in the Page Table. **Module assignment function $S(i,j)$** indicates the assignment of a data element to a memory module. Only a single data element can be accessed from one memory module at a time for *conflict free* access. The utilized module assignment function determines the access formats that are conflict free. Linear module assignment functions, XOR-schemes (also called dyadic functions), and diamond schemes have been previously implemented for the CPMA framework [4,5]. **Address function $a(i,j)$** determines the physical address in a memory module for a data element.

When a user specifies an algorithm for CPMA, this process includes defining the contents of the associated Page Table row. First, the required access formats for the algorithm are evaluated. A module assignment function for conflict free access is chosen and a suitable address function for the module assignment has to be found. Finally, this information is stored in the Page Table.

2.2. Address Computation

A block diagram of the Address Computation unit is depicted in Figure 3. The inputs to the unit are the Access Functions and the scanning point r (Figure 2). The DRAM row address is passed straight to the memory modules. The outputs from this unit are the DRAM column addresses.

Inside the unit, a *Format Control* unit receives the Access Format Function, the scanning point r , and the length of coordinates L_i and L_j . This unit specifies the coordinates of the data elements that are accessed in the scanning field. The *Module Assignment* unit computes the specific memory modules that contain the accessed data. The *Address Decode* unit, in turn, computes the physical addresses for each memory module. The *Address Select* units direct the physical addresses to the proper memory modules.

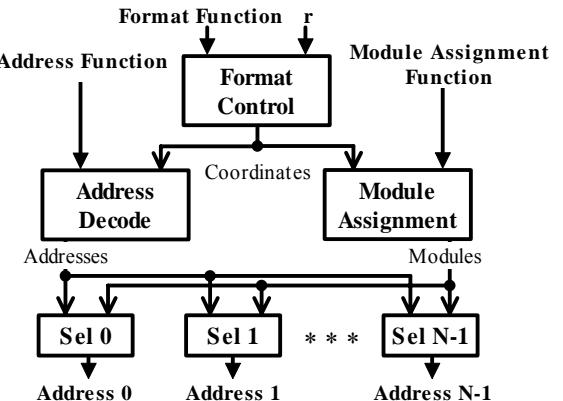


Figure 3. Address Computation unit.

3. Implemented access formats

Various access formats (templates) are utilized in existing applications. Access formats required in video processing include *rows*, *columns*, *rectangles*, and *crumpled rectangles* [6]. Moreover, some exotic access formats can be used. For example, zigzag scanning can utilize quite irregular templates. Besides access formats used in video processing, image processing often utilizes *diagonals* and *backward diagonals* [7]. Particularly, vector computers are specialized in accessing addresses separated by a distance called the *vector stride* [8]. In numerical analysis, useful access formats include *red* and *black chessboards* [9].

The Page Table form of the access formats should be easy to construct. Moreover, the implementation feasibility is one of the main criteria when evaluating access formats. Sharing computational resources can save hardware implementation area. Therefore, the utilized access formats should use similar resources.

Some implementable access formats are depicted in Figure 4. Squares in an access format depict the relative positions of accessed data elements. A scanning point is marked with a dot “•”. The implemented access formats and the parameters required by a Page Table are described in the following.

Generate format $G(a_i, a_j, p)$, where the parameters a_i and a_j are jumps in respective (i, j) directions and p gives the total number of accessed data elements. The parameter a_i may be a positive or a negative integer while a_j has to be positive. All the straight templates, for example rows, columns, and diagonals, can be formed using the *generate* format.

Crumbled rectangle format $CR(a_i, a_j, p_i, p)$, where the parameters a_i and a_j are jumps to i or j directions respectively, p_i shows the number of columns used, and p again corresponds to the number of data elements. All the rectangles and crumbled rectangles can be accessed using this format.

Chessboard format $CB(a, p_i, p)$, where the parameter $a \in \{RC, BC\}$. That is, a is either a red chessboard or a black chessboard. The parameter p_i is the number of accessed data elements in each row and p the total number of data elements.

Vector format $V(a, p)$, where the parameter a is the stride of the vector and p is used as in previous formats. When a coordinate value exceeds the scanning field, a *fit* operation has to be performed to return the coordinate to an acceptable value. The details of this operation are application specific. Vector access is quite similar to the generate format except that the fitting techniques are different. If necessary, the vector format wraps the access to the beginning of the *next row*, as depicted in Figure 4. The *generate* as well as all the other access formats wrap to the beginning of the *same row*. This difference is due to

the fact that vector accesses are intended for one-dimensional memory addressing while the other accesses use two-dimensional memory space.

Free format $F((i_0, j_0), (i_1, j_1), \dots, (i_{h-1}, j_{h-1}), p)$, where the parameter p is utilized as before and the other parameters are the exact coordinates (i, j) from the scanning point r . The parameter i_k may be a positive or a negative integer while j_k has to be positive. The maximum number of free format pixels h directly influences the system bus widths and the Page Table size. The *free* format can access arbitrary memory patterns within the implementation limits. All the other access formats could be formed using this format, but, for the general case, it would be very expensive in terms of implementation area. This is due to Page Table size requirements and resulting system bus widths.

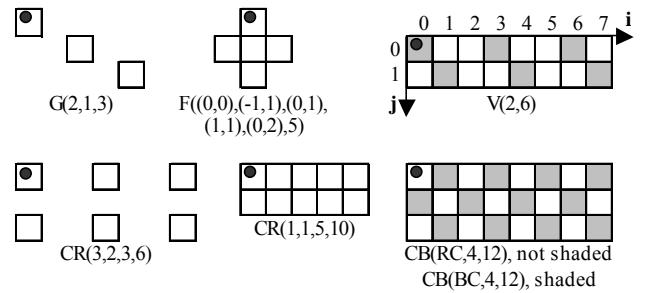


Figure 4. Examples of the implemented access formats.

A scanning point is primarily located to the topmost data element and secondarily to the leftmost element. The data elements are accessed starting from the scanning point and continuing primarily from left to right and secondarily from top downwards. The elements order in the *free* format exceptionally depends on the original order of the coordinate parameters. The scanning point for both of the chessboard formats is in the same, the upper left corner, as shown in Figure 4. Thus, when using the black chessboard, the scanning point is exceptionally located outside the accessed data elements. For all the access formats, the length of coordinates in i and j -directions, L_i and L_j , restricts the distance between the first and the last element. The range is $[-L_i, 2L_i-1]$ in i -direction and $[0, 2L_j-1]$ in j -direction.

4. Access format implementation

A detailed block diagram of the Format Control unit is depicted in Figure 5. The Format Control unit specifies the data elements (coordinates) that are accessed in the scanning field. The inputs to the unit are parameters for the access format function, the scanning point r , and the length of coordinates in i and j -directions, L_i and L_j .

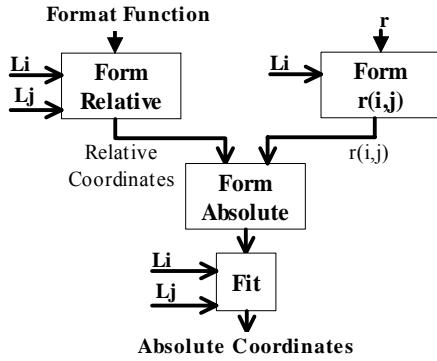


Figure 5. Format Control unit.

The scanning point is originally represented by an integer, which has to be converted into the coordinate form $r(i,j)$. This is accomplished as follows:
 $r(i) = r \bmod L_i$ and $r(j) = \lfloor r / L_i \rfloor$ depicted as the *Form r(i,j)* block in Figure 6c.

Relative coordinates for the accessed data elements in the scanning field are determined in (i,j) format in the *Form Relative* block (Figure 6a). One of the access formats, *generate* (G), *crumpled rectangle* (CR), *chessboard* (CB), *vector* (V), or *free* (F), is selected using a multiplexer depending on the specified format in the Page Table. The Relative Coordinates $R_0(i,j)$, $R_1(i,j)$, ..., $R_{N-1}(i,j)$ are calculated in parallel. The Relative Coordinates could also be formed one by one in a serial fashion [4]. However, that approach has proven to be too slow when complicated access formats are used.

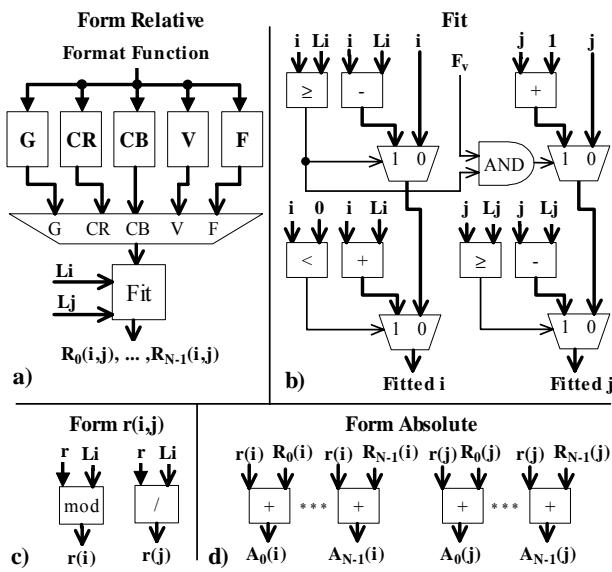


Figure 6. Sub-blocks in the Format Control unit.

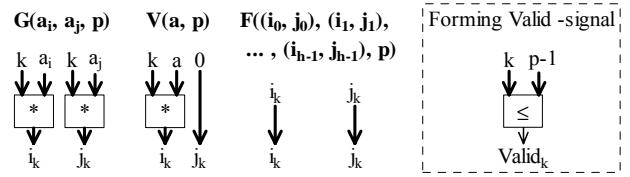


Figure 7. Generate, vector, and free formats and the generation of the Valid -signal.

The *Fit* block (Figure 6b) ensures that the coordinates fall in the scanning field. If the value of an i -coordinate is beyond the scanning field ($i < 0$ or $i \geq L_i$) L_i is either added or subtracted from i to get the *fitted i*. On the other hand, j -coordinate cannot be less than zero and, therefore, it has to be corrected only when the coordinate value is over L_j . When the vector access is used, the F_v signal is asserted and if the coordinate value in i -direction exceeds L_i , the j -coordinate is also incremented by one.

Absolute coordinates are determined in the *Form Absolute* block (Figure 6d). Each absolute coordinate is formed in parallel by adding the relative coordinates and $r(i,j)$. Again, a *Fit* block is used to fix coordinates exceeding L_i and L_j (Figure 5).

The access format implementations for the Form Relative block are depicted in Figure 7, Figure 8, and Figure 9. In Figure 7, the *generate* format is formed by multiplying the jump parameters a_i and a_j with the associated element number in the access format k ($0,1,\dots,N-1$). Thus, the relative coordinates are: $((0,0), (a_i, a_j), (2a_i, 2a_j), \dots, ((N-1)a_i, (N-1)a_j))$. The *vector* format is relatively similar. The i -coordinate is multiplied with the constant k while the j -coordinate is set to zero. The parameters of the *free* format directly indicate the correct coordinates.

It is not necessary to use all the N memory modules in every access. Therefore, *Valid* signals are used to show which relative coordinates are valid. The generation of a *Valid* signal is depicted on the right side of Figure 7. The element number k is compared to the total number of processed data elements p to form the signal.

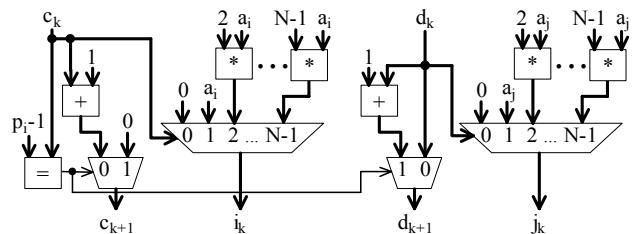


Figure 8. Implementation of the crumbled rectangle access format CR(ai,aj,pi,p).

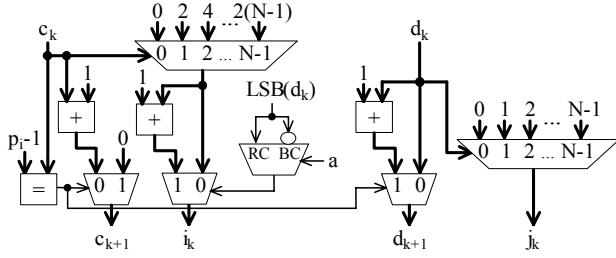


Figure 9. Implementation of the chessboard access format $CB(a,pi,p)$.

The implementations of the *crumpled rectangle* and the *chessboard* formats are depicted in Figure 8 and Figure 9, respectively. Variable c describes the number of solved data elements on the current row and d is the number of solved rows. A new row is started when p_{i-1} is equal to c_k by setting the variable c_k to zero and incrementing the variable d_k by one.

In the *crumpled rectangle* format (Figure 8), the potential i -coordinates are calculated multiplying the parameter a_i with each of the numbers from 0 to $N-1$. The correct i -coordinate is selected using the variable c . Correspondingly, the variable d selects the correct j -coordinate. In the *chessboard* format (Figure 9), the correct i -coordinate is selected from the set of potential coordinates that are all the even numbers from 0 to $2(N-1)$. In addition, the value of the i -coordinate depends on the least significant bit (LSB) of the current row number d_k and whether a red or a black chessboard is selected. The variable d selects the correct j -coordinate.

The control variables c and d are solved in serial but all the other logic is parallel for these two formats. The control signals could also be determined in parallel using a proper modulus and division. However, the utilized area for that solution grows rapidly as a function of the number of memory modules and does not result in significant speed-up.

Since only one of the formats can be selected at a time, some computational resources could be shared. As seen in Figure 8 and Figure 9, the *crumpled rectangle* and the *chessboard* format implementations use very similar units. Moreover, multipliers are used in the implementations of the *generate* and the *vector* formats (Figure 7). All the multipliers could be shared between these four access formats. On the other hand, the *generate* format with parameters $G(a,0,p)$ could implement the *vector* format $V(a,p)$ using the vector access techniques in the Fit block. Each of the access formats has the parameter p specifying the total number of accessed data elements. This parameter could always be located in the same place in the Page Table representation and, therefore, the same comparators could be used to generate Valid signals.

5. Results

The Address Computation unit is the most complex block in CPMA almost irrespectively from the implementation details. The introduced access formats were implemented using synthesizable register transfer level VHDL. The resulting area depends on the implementation parameters, particularly on the number of DRAMs, the size of a DRAM module, and widths of the buses.

The resources required in the Form Relative block are tabulated in Table 1 where the number of memory modules is denoted by N ($N > 1$). The other input of some of the two-input resources is a constant and those units are marked ‘(const)’. The Form Relative block utilizes a total of $3N$ adders, $2N$ comparators, $2N-2$ equality comparators, and $16N-4$ multiplexers. Moreover, $5N-10$ multipliers, $6N-4$ adders, and $6N$ comparators with the other input defined as a constant are required. As similar resources are used to implement the accesses, resource sharing could be utilized. However, this fact is not used to minimize the numbers in Table 1.

The resources required in the Format Control unit are tabulated in Table 2, where $N > 1$. The i -coordinate cannot be less than zero after the Form Absolute block. Therefore, the Fit block in Table 2 requires fewer resources than the Fit block in the Form Relative block (Table 1). The largest units in terms of area are arithmetic units, especially dividers and modulus units, in which the input signals can be arbitrary integers from an implementation specific range. If the length of coordinates in i -direction L_i is restricted to a power of two, the modulus and division operations could be implemented using just shifting and masking. However, some common image sizes like the size of a QCIF (Quarter Common Intermediate Format) picture is 176x144 pixels. These dimensions are not powers of two. The number of resources for each of the resource types is linearly proportional to N . Some of the resources could be shared, but maintaining a parallel architecture requires all the specified operators.

Table 1. Utilized resources in the Form Relative block.

	*	+, -	+, -	<, >	<, >	=	Mux
	(const)		(const)		(const)		
G -format	2N-4					N	
CR -format	2N-4		2N-2		N	$N-1$	$4N-2$
CB -format			3N-2		N	$N-1$	$6N-2$
V -format	$N-2$				N		
F -format					N		
Fit		3N	N	2N	N		$4N$
Other resources							$2N$
Form Relative	5N-10	3N	6N-4	2N	6N	2N-2	16N-4

Table 2. Utilized resources in the Format Control unit.

	/ Mod (const)	*	+, - (const)	+, - (const)	<, > (const)	<, > (const)	=	Mux
Form Rel.			5N-10	3N	6N-4	2N	6N	2N-2 16N-4
Form r(i,j)	N N							
Form Abs.			2N					
Fit			2N	N	2N			3N
Format Cont.	N N	5N-10	7N	7N-4	4N	6N	2N-2	19N-4

The advantages gained by using Configurable Parallel Memory Architecture are illustrated with a video processing example. The half-pixel motion estimation in H.263 video encoding is usually performed in the eight locations around the found full-pixel location. If the search area is interpolated, only every other pixel in the search area is required for the access. Therefore, a crumbled rectangle access format $CR(2,2,8,16)$, depicted in Figure 10 (colored gray), can be utilized in the half-pixel motion estimation and interpolation if the number of accessed data elements is sixteen [6]. A diamond scheme implementation, described in [5], requires only sixteen memory modules for conflict free access. For example, the diamond scheme depicted in Figure 10 allows conflict free access for the access format $CR(2,2,8,16)$. The number of memory modules $N=16$ and 16x6 data elements are illustrated in Figure 10. The data elements are stored in a memory module corresponding to the number in a table cell. The elements are indexed with (i,j) coordinates. For example, a data element located at point (2,1) is stored in memory module 10. In addition, a suitable address function is documented in [5].

The search area is interpolated and, as a performance example, a sum of absolute difference (SAD) is calculated for a 16x16 pixel block in the half pixel motion estimation. With a single conventional memory, the search area is accessed a single pixel at a time and 256 read operations are used. When reading with the access format shown in Figure 10 using CPMA, it takes 16 read operations. That is 16 times less than with a conventional memory architecture. Moreover, the configurability of CPMA enables access formats, module assignment functions, and address functions to be changed at run-time when different kinds of data patterns are required by other applications.

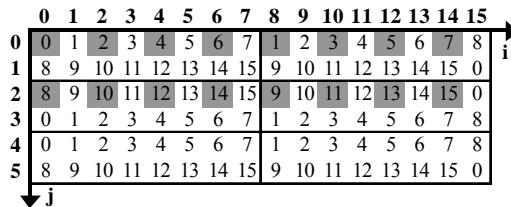


Figure 10. Access format (colored gray) and module assignment function for half-pixel motion estimation and interpolation.

6. Conclusions

The Configurable Parallel Memory Architecture enables a processor to use a multitude of memory access templates. These access formats can guarantee the usefulness of data and data refresh operations are required less frequently. In addition, the configurability of CPMA facilitates running several different computation threads, each with a unique skewing scheme, in a single system.

In this paper, access format implementations for CPMA were presented. Access formats required in many applications can be constructed using the presented implementation. Parameters detailing an access format can flexibly be changed at run-time without a need to redesign the whole hardware. The achieved results are a significant speed-up compared to the conventional memory architecture in terms of required memory accesses. The utilized resources were shown to be linearly proportional to the number of memory modules.

References

- [1] P. Budnik and D. J. Kuck, "The Organization and Use of Parallel Memories," *IEEE Trans. Computers*, Vol. C-20, No. 12, Dec. 1971, pp. 1566-1569.
- [2] M. Gössel, B. Rebel, and R. Creuzburg, *Memory Architecture & Parallel Access*, Elsevier Science B.V, Amsterdam, The Netherlands, 1994, p. 246.
- [3] K. Kuusilinna, *Studies and Implementations of Bus-based Interconnections*, Doctoral Dissertation, Dept. Information Technology, Tampere Univ. of Technology, Tampere, Finland, 2001, p. 201.
- [4] E. Aho, J. Vanne, K. Kuusilinna, T. Hämäläinen, and J. Saarinen, "Configurable Address Computation in a Parallel Memory Architecture," *Advances in Signal Processing and Computer Technologies*, G. Antoniou, N. Mastorakis, and O. Panfilov, ed., WSES Press, Athens, Greece, 2001, pp. 390-395.
- [5] E. Aho, J. Vanne, K. Kuusilinna, and T. Hämäläinen, "Diamond Scheme Implementations in Configurable Parallel Memory," *Proc. IEEE Int'l Workshop Design and Diagnostics of Electronic Circuits and Systems*, Brno, Czech Republic, Apr. 2002, pp. 211-218.
- [6] J. Tanskanen, T. Sihvo, J. Niittylahti, J. Takala, and R. Creutzburg, "Parallel Memory Access Schemes for H.263 Encoder," *Proc. IEEE Int'l Symp. Circuits and Systems*, Geneva, Switzerland, May 2000, pp. 691-694.
- [7] K. Kim and V. K. Prasanna Kumar, "Parallel Memory Systems for Image Processing," *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, San Diego, CA, USA, June 1989, pp. 654-659.
- [8] D. T. Harper III, "Block, Multistride Vector, and FFT Accesses in Parallel Memory Systems," *IEEE Trans. Parallel and Distributed Systems*, Vol. 2, No. 1, Jan. 1991, pp. 43-51.
- [9] J. M. Frailong, W. Jalby, and J. Lenfant, "XOR-Schemes: A Flexible Data Organization in Parallel Memories," *Proc. Int'l Conf. Parallel Processing*, Washington, DC, USA, Aug. 1985, pp. 276-283.

Publication 4

E. Aho, J. Vanne, K. Kuusilinna, and T. Hämäläinen, “XOR-scheme Implementations in Configurable Parallel Memory,” in *System-on-Chip for Real-Time Applications*, W. Badawy and G. A. Jullien, ed., Kluwer Academic Publishers, Boston, MA, USA, pp. 249–261, 2003.

Copyright © Kluwer Academic Publishers.

Reprinted with kind permission of Springer Science and Business Media.

XOR-scheme Implementations In Configurable Parallel Memory

Eero Aho, Jarno Vanne, Kimmo Kuusilinna, and Timo Hämäläinen

Institute of Digital and Computer Systems, Tampere University of Technology

Korkeakoulunkatu 1, FIN-33720 Tampere, FINLAND

eero.aho@tut.fi

Abstract: Being able to continuously feed data to the processing nodes is one of primary problems in parallel data processing. Parallel memories increase memory bandwidth with several memory modules working in parallel and feed the processor with only necessary data. The Configurable Parallel Memory Architecture (CPMA) enables a multitude of access formats and module assignment functions to be used within a single hardware implementation, which has not been possible in prior embedded parallel memory systems. In this paper, a general XOR-scheme implementation for a previously implemented CPMA is presented. Any XOR-scheme can be used for the distribution of data to the memory modules with the introduced implementation. Results show significant reduction in the overall memory accesses, which can be viewed as one of the key approaches to solve the processor-memory performance bottleneck.

Key words: configurable parallel memory architecture, XOR-schemes, module assignment function, conflict free access

1. INTRODUCTION

The development of Dynamic Random Access Memories (DRAMs) has been exponential during the last decades. That is, the size of the memory cell has been shrinking and, thus, the capacity of the DRAMs has been increasing. However, the smaller feature sizes have not made the DRAMs significantly faster. The cell density has been the development driver and speed has partly been a secondary issue. Simultaneously, rapid changes have happened to the transistors in processors. This has allowed both speeds-ups and increased complexity in processors. For these reasons, the processor versus memory performance gap has been widening for a number of years. Several methods and architectures to enhance the memory system performance have been presented [11].

One solution to the problem is to place a cache between the main memory and the processor. Widening the bus between the memory and cache makes it possible to refresh the cache in larger blocks of data. Unfortunately, this can signify the fact that the cache contains a lot of currently unnecessary data.

Some real-time applications, e.g. motion estimation and discrete cosine transformation (DCT) in video processing, favor special data patterns that can be accessed in parallel [6]. This phenomenon is utilized in *parallel memories*, where the idea is to increase memory bandwidth with several memory blocks working in parallel and feed the processor with only necessary data. Traditionally the parallel memory development has been concentrated on *module assignment functions (skewing schemes)* that try to ensure *conflict free* parallel data accesses to as many *access formats (templates)* as possible. *Linear* module assignment functions were first studied with parallel memories [4]. *XOR-schemes* have been shown to allow several additional conflict free access formats [2], [5], [7], [9].

A generalized block diagram of a parallel memory architecture is shown in *Figure 1*. The Figure depicts an *address computation* unit, N *memory modules*, and a *permutation* unit [8]. Depending on the access format F and the address of the first element r , the address computation unit computes the right memory modules and addresses in the memory module. The permutation unit organizes the data into correct order.

A typical problem with previously presented parallel memory implementations is that they are application specific with fixed access formats implemented in hardware. This ensures the fastest speed, but requires redesign if other formats have to be included. In this work, the problem has been addressed by implementing the *Configurable Parallel Memory Architecture (CPMA)* [10], in which access formats and the needed address generation functions can be changed at run-time.

Time-to-market, increasing complexity, and system performance requirements have necessitated the increasing utilization of reusable intellectual property (IP) blocks and programmable platforms in system-on-chip (SoC) designs. CPMA can be viewed as one step to this direction since it is able to cope with a multitude of algorithms needed in different applications without redesign of the memory system.

Promising technology development for CPMA includes embedded DRAM designs, where DRAM is embedded to the same die as the processor core. This enables the use of wide data buses, low latency, and vector processor like execution, which are all very attractive properties if data can easily be supplied from parallel memories. Especially real-time, streaming, applications are deemed to profit from this architecture because the cache memory performance bottleneck can be alleviated.

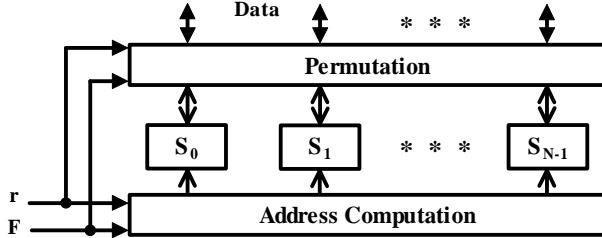


Figure 1. General parallel memory architecture.

Previous implementations of CPMA included linear module assignment functions and some special XOR-schemes [1]. In this paper, a generalized implementation of a XOR-scheme for the implemented CPMA is presented. An overview of the implemented CPMA is given in Section 2. XOR-schemes are introduced in Section 3, which is followed by the XOR-scheme implementation details in Section 4. Results are given in Section 5 and Section 6 concludes the paper.

2. CONFIGURABLE PARALLEL MEMORY ARCHITECTURE

A block diagram of the implemented CPMA is depicted in *Figure 2*. The inputs to CPMA are the *virtual address* and the *command* (write, read). The virtual address includes a *page address*, which is a pointer to a *page table*. *Page offset* directly indicates the *scanning point r* for the parallel memory. The *row address* and the access functions are fed from the page table to the address computation unit. In addition, DRAM banks and a permutation unit are required for CPMA operation. [1]

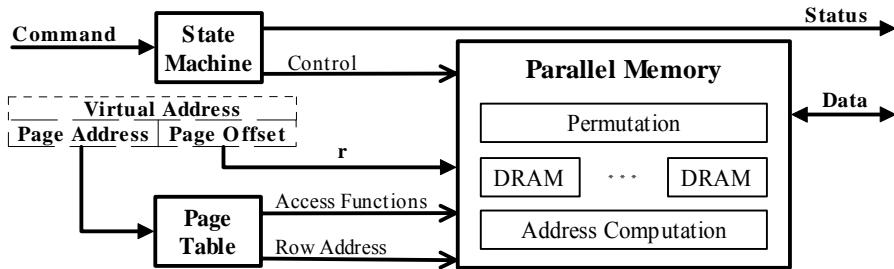


Figure 2. Configurable Parallel Memory Architecture.

CPMA address computation is implemented using multiple dedicated computation units in parallel. There is one unit for each class of access formats, module assignment functions, and address functions that the implementation supports. Moreover, a configurable data permutation unit is designed to conform to system specifications.

CPMA is designed to be configurable both at design time and at run-time. In the VHDL implementation, many parameters are generics, which means that the number of DRAMs, size of a DRAM module, width of the data bus, and width of input buses are adjustable. All the required blocks, logic, and buses are implemented with logic synthesis from VHDL. In the following, we describe some of the CPMA blocks in more detail.

2.1 Page Table

Our page table is a block of memory that is directly addressable from application software. The page address is a pointer to a specific row in the page table, which determines the functionality in the CPMA during that memory access. Sample page table entries are tabulated in *Table 1* and the Table field contents are explained in the following.

Table 1. Page table structure.

Page Address	Row address	L_i	L_j	F	$S(i,j)$	$a(i,j)$
:						
:						

Row address in the page table is like a page number in a physical memory. A single page should usually only carry one module assignment function $S(i,j)$.

Length of coordinates L_i and L_j indicate a memory area (dimensions i and j) to be accessed with the scheme defined in this page entry. The values are scaled according to the application at hand.

Parameters for access format function F . The data elements are read or stored in parallel with an access format F . Five example access formats are shown in *Figure 3*. Squares in an access format depict accessed data elements.

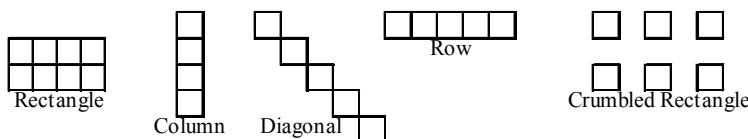


Figure 3. Access format examples.

Parameters for a module assignment function $S(i,j)$. The assignment of a data element to the memory module is determined by a module assignment function. The utilized module assignment function determines the access formats that can be used conflict free. There are two kinds of module assignment functions implemented in our system.

A *Linear* module assignment function is defined as

$$S(i, j) = (a \cdot i + b \cdot j) \bmod N, \quad (1)$$

where a, b and N are positive integers.

Most of the skewing schemes using dyadic functions (also called XOR schemes) are formulated as

$$S(i, j) = \pi(i \bmod N)_2 \oplus \tau(j \bmod N)_2, \quad (2)$$

where $N = 2^m, m \in \{0, 1, 2, \dots\}$. The operation \oplus denotes component-wise XOR. π and τ are permutations of the components of the binary numbers $(i \bmod N)$ and $(j \bmod N)$, respectively. That is $\pi(i \bmod N)_2 = \pi(i_{m-1} | \dots | i_0)$, where i_k is a binary number. [8]

Previously, only two hardwired permutations were implemented within the CPMA framework. These permutations are described by Equations (3) and (4) [8], [9].

$$\text{REVERSE}(i_{m-1} | \dots | i_1 | i_0) = (i_0 | i_1 | \dots | i_{m-1}) \quad (3)$$

and for $N = 2^{2m'}, m' \in \{0, 1, 2, \dots\}$

$$\text{SWAP}(i_{2m'-1} | \dots | i_{m'} | i_{m'-1} | \dots | i_0) = (i_{m'-1} | \dots | i_0 | i_{2m'-1} | \dots | i_{m'}) \quad (4)$$

Parameters for address function $a(i,j)$. Address function determines the address in a memory module for the accessed data element. The current system implements one address function $a(i, j) = b + (i + j \cdot L_i) / N$, where b , L_i and N are positive integers. L_i is specified in the L_i column of the page table.

When a user specifies an algorithm for CPMA, this process includes defining the contents of the associated page table row. First, the required access formats for the algorithm are evaluated. Then, a module assignment function for conflict free access has to be chosen. Moreover, a suitable address function for the module assignment has to be found. Finally, all this information is stored in the page table. Automation of this process requires further research.

2.2 Address Computation

A block diagram of the Address Computation unit is depicted in *Figure 4*. The inputs to the unit are the access functions and the scanning point r (*Figure 2*). The row address is directed straight to the DRAMs. The specific memory module addresses are the outputs from this unit.

Inside the unit, a *Format Control* unit receives the access format function, the scanning point r , and the length of coordinates L_i and L_j . This unit specifies the coordinates of the data elements that are accessed in the scanning field.

The Module Assignment unit computes the specific memory modules that contain the accessed data elements. The Address Decode unit, in turn, computes the physical addresses for each memory module. After that, the Address Select units direct the physical addresses for the proper memory modules.

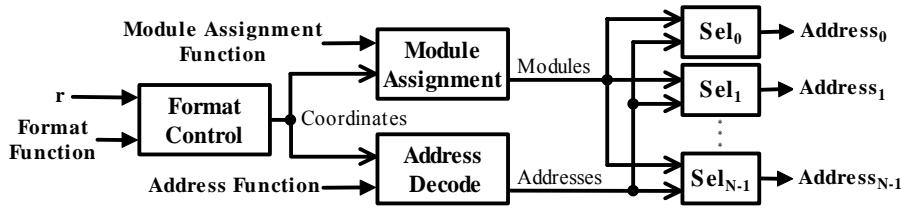


Figure 4. Address Computation unit.

3. XOR-SCHEMES

XOR-schemes always utilize a power of two ($N=2^n$) memory modules. The dyadic function (2) can also be defined as:

$$S(i, j) = \mathbf{A}I \oplus \mathbf{B}J, \quad (5)$$

where $I = (i \bmod 2^u)_2$ and $J = (j \bmod 2^v)_2$. \mathbf{A} and \mathbf{B} are $(n \times u)$ and $(n \times v)$ matrices containing binary numbers. The elements of \mathbf{A} and \mathbf{B} matrices are marked in an unordinary fashion. The top left element of matrix \mathbf{A} is $a_{n-1,u-1}$ and the bottom right is $a_{0,0}$. The scanning field is decomposed into non-overlapping $(2^u, 2^v)$ pages. In every page, the corresponding points are stored within the same memory module. The vector addition and multiplication in the function (5) are achieved by bit-wise logical operations XOR and AND, respectively. [7]

An example XOR-scheme is depicted in *Figure 5* where the number of memory modules $N=8$. A single page including $8*8$ data elements is displayed. The data elements are stored in a memory module with a number shown in the corresponding table cell. The elements are indexed with (i,j) coordinates. E.g., a data element located at point $(001,011)$ is stored in memory module 7. The values along the axels are shown in binary format since that number system is used with the XOR-scheme. The scheme is formed using function (6).

$$\begin{aligned}
 S(i, j) &= I \oplus \text{REVERSE}(J) \\
 &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} i_2 \\ i_1 \\ i_0 \end{bmatrix} \otimes \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} j_2 \\ j_1 \\ j_0 \end{bmatrix} = \begin{bmatrix} i_2 \\ i_1 \\ i_0 \end{bmatrix} \otimes \begin{bmatrix} j_0 \\ j_1 \\ j_2 \end{bmatrix} \\
 &= (i_2 \oplus j_0 \mid i_1 \oplus j_1 \mid i_0 \oplus j_2)
 \end{aligned} \tag{6}$$

	000	001	010	011	100	101	110	111	\rightarrow
000	0	1	2	3	4	5	6	7	\rightarrow
001	4	5	•6	7	•0	1	2	3	
010	2	3	•0	1	•6	7	4	5	
011	6	7	4	5	2	3	0	1	
100	1	0	3	2	5	4	7	6	
101	5	4	7	6	1	0	3	2	
110	3	2	1	0	7	6	5	4	
111	7	6	5	4	3	2	1	0	

$\downarrow j$

Figure 5. Example XOR-scheme.

The function (6) reorganizes each of the J binary vectors as formulated in (3). The scheme is called the EE (Exchange-Expansion) scheme in [9]. The depicted scheme allows row and column accesses to be conflict free in all the points, however, rectangle shaped accesses are restrictedly conflict free when the number of accessed elements M is equal to the number of memory modules N . E.g. a rectangle access format sized $4*2$ is conflict free when the scanning point is located at $r(i,j)=(0,0)$ since all the accessed elements are located in different memory modules (*Figure 5*). However, the same access format is not conflict free when $r(i,j)=(1,1)$ because two of the data elements are accessed from the same memory modules 0 and 6. These data elements are marked with dots “•”.

A more complicated XOR-scheme example is depicted in *Figure 6* where $N=8$. The function is formulated as follows

$$S(i, j) = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} i_2 \\ i_1 \\ i_0 \end{bmatrix} \otimes \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} j_2 \\ j_1 \\ j_0 \end{bmatrix} \quad (7)$$

This formula is called the MG-EE scheme (MG stands for Modified-Gray). The scheme allows several conflict free access formats, e.g. rows, columns, diagonals, rectangles, crumbled rectangles, and chessboards. Some of the access formats are conflict free in arbitrary positions the rest can only be restrictedly placed. [9]

	000	001	010	011	100	101	110	111
000	0	5	7	2	6	3	1	4
001	4	1	3	6	2	7	5	0
010	2	7	5	0	4	1	3	6
011	6	3	1	4	0	5	7	2
100	1	4	6	3	7	2	0	5
101	5	0	2	7	3	6	4	1
110	3	6	4	1	5	0	2	7
111	7	2	0	5	1	4	6	3

Figure 6. 8*8 mapping matrix for the MG-EE scheme.

4. GENERALIZED XOR-SCHEME IMPLEMENTATION

A and **B** in the XOR-scheme Equation (5) are $(n \times u)$ and $(n \times v)$ matrices when the number of memory modules is $N=2^n$. The vector addition and multiplication are achieved by logical operations bitwise XOR and bitwise AND respectively. Therefore, the calculations are done as follows:

$$\begin{aligned} \mathbf{AI} &= \begin{bmatrix} a_{n-1,u-1} & a_{n-1,u-2} & \dots & a_{n-1,0} \\ a_{n-2,u-1} & a_{n-2,u-2} & \dots & a_{n-2,0} \\ \vdots & \vdots & \dots & \vdots \\ a_{0,u-1} & a_{0,u-2} & \dots & a_{0,0} \end{bmatrix} \cdot \begin{bmatrix} i_{u-1} \\ i_{u-2} \\ \vdots \\ i_0 \end{bmatrix} \\ &= (a_{n-1,u-1} \cdot i_{u-1} \oplus a_{n-1,u-2} \cdot i_{u-2} \oplus \dots \oplus a_{n-1,0} \cdot i_0 \mid a_{n-2,u-1} \cdot i_{u-1} \oplus \\ &\quad a_{n-2,u-2} \cdot i_{u-2} \oplus \dots \oplus a_{n-2,0} \cdot i_0 \mid \dots \mid a_{0,u-1} \cdot i_{u-1} \oplus a_{0,u-2} \cdot i_{u-2} \oplus \dots \oplus a_{0,0} \cdot i_0) \end{aligned} \quad (8)$$

The operations \cdot and \oplus depicts logical AND and XOR, respectively. The calculation of \mathbf{BJ} is done using a respective notation where a symbol u is replaced with a symbol v .

Let us denote

$$\mathbf{AI} = (a_{n-1} | a_{n-2} | \dots | a_0),$$

$$\mathbf{BJ} = (b_{n-1} | b_{n-2} | \dots | b_0).$$

Then, the final calculation can be expressed as

$$\mathbf{AI} \oplus \mathbf{BJ} = (a_{n-1} \oplus b_{n-1} | a_{n-2} \oplus b_{n-2} | \dots | a_0 \oplus b_0). \quad (9)$$

A block diagram of the implementation of the XOR-scheme (5) is depicted in *Figure 7*. Matrices **A** and **B** are supplied from a page table. The block **AI** executes the logical operations in function (8) and the respective operations are done in the block **BJ**. Finally, logical bit-wise XOR operations are done as indicated in (9) to get the module number $S(i,j)$.

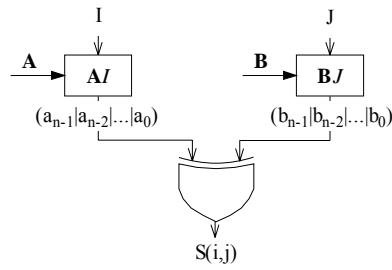


Figure 7. Generic implementation of the XOR-scheme.

The sizes ($n \times u$) and ($n \times v$) of matrices **A** and **B** have to be determined before synthesizing the presented XOR-scheme implementation. $(n*u)$ units of 2 -input AND ports and n units of u -input XOR ports are required in the **AI** block. Respectively, $(n*v)$ units of 2 -input AND ports and n units of v -input XOR ports are required in the **BJ** block. The final bitwise XOR operations need n units of 2 -input XOR ports. Altogether, the required ports in the implementation of the generalized XOR-scheme (5) are tabulated in *Table 2*.

Table 2. Required resources in the implementation.

Port type	Number of ports
2 -input AND ports	$(n*u) + (n*v)$
u -input XOR ports	n
v -input XOR ports	n
2 -input XOR ports	n

The values of the binary matrices \mathbf{A} and \mathbf{B} are required to be stored in the page table. Synthesis only defines the maximum size ($n \times u$) and ($n \times v$) for the matrices. In the case of smaller sized matrices, the elements of the matrices \mathbf{A} and \mathbf{B} have to be placed to the right down corner and the left up corner values are set to zero.

An overview of configuring a XOR-scheme is depicted in *Figure 8*. Before a memory access, the values for the matrices \mathbf{A} and \mathbf{B} in Equation (5) are stored in the page table. When a data pattern is read from CPMA, the Central Processing Unit (CPU) references the page table with a page address to these predetermined matrix values. The memory module number $S(i,j)$ for each of the accessed data elements is determined with the help of the matrix values and the $(i_0, j_0), (i_1, j_1) \dots (i_{N-1}, j_{N-1})$ coordinates from the Format Control unit (*Figure 4*).

All the XOR-schemes using the Equation (5) can be formed using the presented XOR-scheme implementation. These kinds of schemes have been introduced e.g. in [2], [5], [7], and [9]. Those schemes enable the use of a multitude of conflict free access formats. For instance, the XOR-scheme introduced in [7] enables accesses for rows, columns, rectangles, crumbled rectangles and chessboards. The XOR-scheme was also used in the STARAN computer architecture [3]. STARAN was able to access many kinds of templates including rows, columns, *folded lines*, *partial-row-pairs*, and *partial-column-pairs*. The last three access formats are useful in pattern analysis [9].

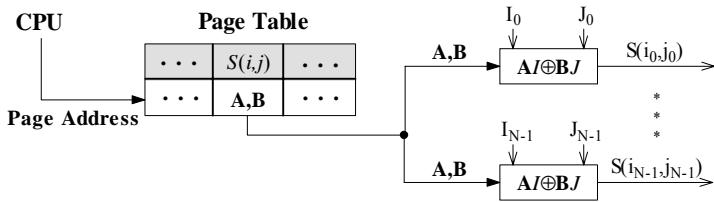


Figure 8. Configuring a XOR-scheme.

5. RESULTS

The introduced XOR-scheme is implemented with synthesizable register transfer level VHDL. The sizes ($n \times u$) and ($n \times v$) of matrices \mathbf{A} and \mathbf{B} have to be determined before synthesizing the design. A test case design was synthesized to confirm the number of required logical ports. The parameters were set as $n=v=u=3$. That makes the number of memory modules $N=2^n=8$. This implementation can form, e.g. the schemes depicted in *Figure 5* and

Figure 6. The synthesis tool estimates an area that equals to 390 2 -input NAND ports.

The required theoretical resources are tabulated in *Table 3* for the test case design when $N=8$. The area estimation for the above ports from the synthesis tool is equivalent to 442 NAND ports. The difference between the synthesized implementation and the theoretical 442 ports indicates that the synthesis tool has managed to optimize the test case design.

Table 3. Required theoretical resources when $N=8$.

Port type	Number of ports
2 -input AND ports	$8*18=144$
3 -input XOR ports	$8*6=48$
2 -input XOR ports	$8*3=24$

The advantage from using the implemented Configurable Parallel Memory Architecture is apparent if access times are compared when processing a picture with different access formats. Crumbled rectangle access formats can be utilized in video processing (e.g. DCT/IDCT, motion estimation). An example of such an access format is shown in *Figure 9*. The access format can be used conflict free within e.g. the XOR-scheme depicted in *Figure 6*.

If a QCIF (Quarter Common Intermediate Format) picture containing $176*144$ pixels, is written to memory a single pixel at a time it takes $176*144 = 25344$ write operations. (Assuming every pixel is written individually.) When writing the same picture with the access format shown in *Figure 9*, it takes 3168 write operations. That is 8 times less than with a single pixel access format.

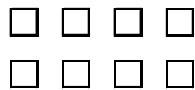


Figure 9. Access format used in the example.

6. CONCLUSION

The discussed Configurable Parallel Memory Architecture enables a system-on-chip processor to use a multitude of memory access templates. These access formats can guarantee the usefulness of data and data refresh operations are required less frequently. In addition, the configurability of the CPMA facilitates running several different computation threads, each with a

unique skewing scheme, in a single system. This has not been possible in the prior embedded parallel memory systems.

In this paper, a general implementation of a XOR-scheme for the implemented CPMA was presented. The implemented schemes allow several additional conflict free access formats to be used. Parameters for constructing a XOR-scheme can flexibly be changed at run-time without a need to redesign the whole hardware.

In the future, our research will focus on clarifying the access formats that are required in special applications. Especially templates that can be utilized with algorithms used in video and graphics are studied. Moreover, other module assignment functions are researched for their implementation feasibility.

7. REFERENCES

- [1] E. Aho, J. Vanne, K. Kuusilinna, T. Hääläinen, and J. Saarinen, "Configurable Address Computation in a Parallel Memory Architecture," *Advances in Signal Processing and Computer Technologies*, G. Antoniou, N. Mastorakis, and O. Panfilov, ed., WSES Press, Athens, Greece, 2001, pp. 390-395.
- [2] M. A. Al-Mouhamed and S. S. Seiden, "Minimization of Memory and Network Contention for Accessing Arbitrary Data Patterns in SIMD Systems," *IEEE Trans. Computers*, Vol. 45, No. 6, June 1996, pp. 757-762.
- [3] K. E. Batcher, "The Multidimensional Access Memory in STARAN," *IEEE Trans. Computers*, Feb. 1977, pp. 174-177.
- [4] P. Budnik and D. J. Kuck, "The Organization and Use of Parallel Memories," *IEEE Trans. Computers*, Vol. C-20, No. 12, Dec. 1971, pp. 1566-1569.
- [5] S. Chen, A. Postula, and L. Jozwiak, "Synthesis of XOR Storage Schemes with Different Cost for Minimization of Memory Contention," *Proc. Euromicro Conf.*, Sep. 1999, pp. 170-177.
- [6] S. Dutta, W. Wolf, and A. Wolfe, "A Methodology to Evaluate Memory Architecture Design Tradeoffs for Video Signal Processors," *IEEE Trans. Circuits and Systems for Video Technology*, Vol. 8, No. 1, Feb. 1998, pp. 36-53.
- [7] J. M. Frailong, W. Jalby, and J. Lenfant, "XOR-Schemes: A Flexible Data Organization in Parallel Memories," *Proc. Int'l Conf. Parallel Processing*, Washington, DC, USA, Aug. 1985, pp. 276-283.
- [8] M. Gössel, B. Rebel, and R. Creuzburg, *Memory Architecture & Parallel Access*, Elsevier Science B.V, Amsterdam, The Netherlands, 1994, p. 246.
- [9] Z. Liu and X. Li, "XOR Storage Schemes for Frequently Used Data Patterns," *Journal of Parallel and Distributed Computing*, Vol. 25, No. 2, Mar. 1995, pp. 162-173.
- [10] K. Kuusilinna, *Studies and Implementations of Bus-based Interconnections*, Doctoral Dissertation, Dept. Information Technology, Tampere Univ. of Technology, Tampere, Finland, 2001, p. 201.
- [11] D. A. Patterson and J. L. Hennessy, *Computer organization and design: the hardware/software interface*, Morgan Kaufmann, San Francisco, California, 1994, p. 648.

Publication 5

E. Aho, J. Vanne, K. Kuusilinna, and T. D. Hämäläinen, “Address Computation in Configurable Parallel Memory Architecture,” *IEICE Transactions on Information and Systems*, vol. E87-D, no. 7, pp. 1674–1681, July 2004.

Copyright © The Institute of Electronics, Information and Communication Engineers (IEICE).
Reprinted with permission.

Authorization number: 06RB0143
<http://www.ieice.org>

Address Computation in Configurable Parallel Memory Architecture

Eero AHO^{†a)}, Jarno VANNE[†], Kimmo KUUSILINNA[†], and Timo D. HÄMÄLÄINEN[†], Nonmembers

SUMMARY Parallel memories increase memory bandwidth with several memory modules working in parallel and can be used to feed a processor with only necessary data. The Configurable Parallel Memory Architecture (CPMA) enables a multitude of access formats and module assignment functions to be used within a single hardware implementation, which has not been possible in prior embedded parallel memory systems. This paper focuses on address computation in CPMA, which is implemented using several configurable computation units in parallel. One unit is dedicated for each type of access formats and module assignment functions that the implementation supports. Timing and area estimates are given for a 0.25-micron CMOS process. The utilized resources are shown to be linearly proportional to the number of memory modules.

key words: *parallel memory, memory access format, skewing schemes, parallel access*

1. Introduction

The processor versus memory performance gap has been widening for a number of years. Cache memories can increase system memory bandwidth in general purpose systems, but they also tend to increase latency. Wide memories can refresh the cache in large blocks of data. Unfortunately, this might mean that the cache contains a lot of data that the currently running algorithm does not use. Multiport memories enable several simultaneous memory accesses but they are an expensive solution especially when the number of ports is large.

Certain applications, particularly some video and graphics designs, favor special data patterns that can be accessed in parallel. This phenomenon is utilized in *parallel memories*, where the idea is to increase memory bandwidth with several memory blocks working in parallel and to feed processors with only algorithm specific data. Traditionally, the parallel memory research has concentrated on *module assignment functions (skewing schemes)* that try to ensure *conflict free* parallel data accesses to the required *access formats (templates)*. *Linear functions* [1], *XOR-schemes* [2], *periodic schemes* [3], and *diamond schemes* [4] have been used to implement the module assignment in these parallel memories. However, the *Configurable Parallel Memory Architecture* (CPMA) [5] facilitates the utilization of all these application specific methods in a single embedded system.

Innovative memory implementations are gaining popularity in contemporary high-performance systems. Impulse

memory controller [6] uses an additional level of address indirection to remap data structures in memory. Therefore, applications may access just the required data from memory keeping the cache coherent at the same time. This data is also in the right order. However, Impulse does not really increase memory bandwidth like parallel memories do. HiPAR-DSP [7] is a parallel processor utilizing parallel memories. Common data access patterns for image processing algorithms are supported by use of a shared on-chip memory with parallel matrix type access patterns. A motion-stereo processor implementation with parallel memories is shown in [8].

A generalized block diagram of a parallel memory architecture is depicted in Fig. 1. The functional blocks in Fig. 1 are an *Address Computation* unit, *N memory modules* S_0, S_1, \dots, S_{N-1} , and a *Permutation* unit [4]. Depending on the *access format F* and the *address of the first element r*, the Address Computation unit computes the addresses and directs them to the appropriate memory modules. The Permutation unit organizes the *data* into correct order specified by the access format.

A typical problem with the previously presented parallel memory implementations is that they are application specific with fixed access formats implemented in hardware. This ensures the fastest speed, but requires redesign if other formats need to be included. To address this issue, the Configurable Parallel Memory Architecture can change access formats and the required address generation functions at run-time.

Time-to-market, increasing complexity, and system performance requirements have necessitated the increasing utilization of reusable intellectual property (IP) blocks and programmable platforms in system-on-chip (SoC) designs. CPMA can be viewed as one step to this direction since it is able to cope with a multitude of algorithms needed in different applications without redesign of the memory system.

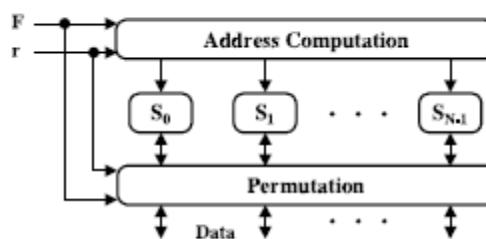


Fig. 1 A generalized block diagram of a parallel memory architecture.

Manuscript received October 15, 2003.

Manuscript revised January 15, 2004.

[†]The authors are with the Institute of Digital and Computer Systems, Tampere University of Technology, Tampere, Finland.

a) E-mail: eero.aho@tut.fi

The Address Computation unit is the most complex block in CPMA almost irrespectively from the implementation details. Hence, this paper concentrates on the address computation implementation. The work is based on the previous research results found in [9]–[12]. As new results, resource counts in addition to logic synthesis-based area and timing estimates are given for an implementation utilizing resource sharing.

Useful access formats for different applications are explained in Sect. 2 and Sect. 3 discusses module assignment function differences. An overview of CPMA is given in Sects. 4 and 5. Sections 6 and 7 examine the hardware implementation of the address computation. Section 8 concludes the paper.

2. Access Formats for a Selection of Applications

The address space in parallel memories cannot be assigned arbitrarily since the data has to be stored in memory using predetermined patterns, called access formats or templates. The data elements of an access format can be read from the memory modules in parallel when a conflict free access is performed.

Different applications utilize different access formats. Examples of templates are depicted in Fig. 2. Further discussion about the applications can be found in the references as detailed in the following paragraphs. Squares in an access format depict the relative positions of accessed data elements.

Image processing: An image can be represented by a two-dimensional array. Images can be stored, viewed from a display, rotated, scaled, and compressed, to name a few image specific operations. Image processing can utilize access formats like *rows*, *columns*, *rectangles*, *crumpled rectangles* (also called *distributed blocks*), *chessboards*, *diagonals*, and *backward diagonals* [2], [13]–[15]. Moreover, three dimensional (3D) stereo vision benefits from *square* templates. One such FPGA implementation utilizes a 3*3 window in parallel memory [8].

Video processing: Some of the basic video coding operations, for example in H.263 and MPEG-4 standards,

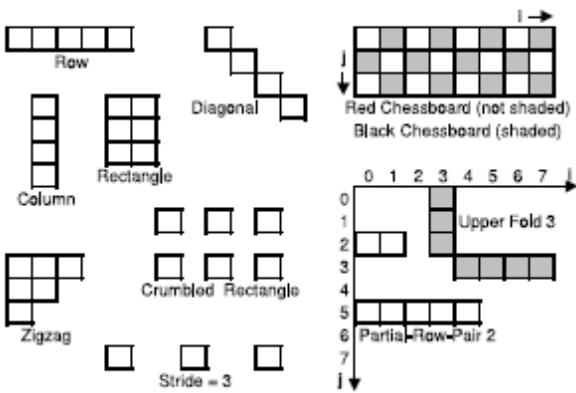


Fig. 2 Access formats.

are motion estimation, interpolation, motion compensation, discrete cosine transformation (DCT), quantization, inverse quantization, and inverse discrete cosine transformation (IDCT). Suitable access formats are *rows*, *columns*, *crumpled rows*, *rectangles*, and *crumpled rectangles* [5], [16]. Moreover, some exotic access formats can be used. For example, the basic zigzag scan as well as the additional alternate-horizontal scan and alternate-vertical scan in MPEG-4 can utilize quite irregular templates. These formats are similar to the “Zigzag” format in Fig. 2.

Vector processing: Vector computers are specialized in accessing addresses separated by a distance called the vector *stride*. In addition, *rectangle* formats are utilized. Many applications in the fields of digital signal processing and telecommunications benefit from the use of strides. Vector/matrix computation, fast fourier transform (FFT) and Viterbi algorithms are some examples [15], [17].

Numerical analysis: Useful access formats include *red* and *black chessboards* for red-black relaxation methods [2]. The red chessboard consists of data elements in the rectangle in such a way that the sum of their *i* and *j* coordinates is even. The black chessboard consists of the remaining elements in the rectangle. Bitonic sorting benefits from the use of *crumpled row* type accesses [18]. *Rectangle* templates help data accesses of convolution algorithms and a *plus* (+) shaped access format is useful for successive over relaxation (SOR) [19].

Pattern analysis: Two dimensional template matching and pattern recognition can utilize a *rectangle* access format [14], [20]. *Chessboard* and *crumpled rectangle* access formats are useful in image sampling [21]. Hierarchical clustering, cluster validity, feature extraction, and classification can benefit from *upper folds*, *lower folds*, *partial-row-pairs*, and *partial-column-pairs* [22]. These formats are also desirable patterns for image processing. Those access formats are defined for an $N \times N$ matrix where N is the number of memory modules. Only $N - 1$ memory accesses are executed and, therefore, the number of accessed data elements is $M = N - 1$. The STARAN computer architecture was able to access a number of templates including rows, columns, folded lines, partial-row-pairs, and partial-column-pairs [23]. Examples of an upper fold 3 and a partial-row-pair 2 are depicted in Fig. 2. If the size of the matrix is 8×8 , the number of memory modules is $N = 8$, and the number of accessed data elements $M = N - 1 = 7$.

3. Module Assignment Functions

The distribution of data to the memory modules is called a module assignment function S and also known as a skewing scheme. The used module assignment function determines access formats that can be used conflict free. Different types of module assignment functions inherently support specific types of access formats. These differences are shortly introduced in the following.

3.1 Linear Function

Linear module assignment functions were first introduced in the 1970's by Budnik and Kuck [1]. A prime number of memory modules N enables access to many straight access formats (for example rows, columns, diagonals, constant stride accesses) [1]. Moreover, the number of accessed data elements M may be equal to the number of memory modules N . On the other hand, special properties of *Fibonacci numbers* (1, 1, 2, 3, 5, 8, 13, ...) can be utilized when a conflict free access to all axially oriented rectangles with a minimum number of memory modules is desired [4].

A case with a prime number of memory modules $N = M = 5$ is illustrated in Fig. 3. The used module assignment function is $S(i, j) = i + 3j \bmod 5$. Some conflict free straight access formats are shown: row, column, forward diagonal, and forward jumps of three. The access format with jumps of $(i, j) = (2, 1)$ is not conflict free because the data elements are accessed from the same memory module 4. It is illustrated with dots ‘•’.

However, a fast hardware implementation of the arithmetic modulus of a prime number is expensive. If the number of memory modules is a power of two, the arithmetic operations are implemented using just shifting and masking. Unfortunately, restricting the number of memory modules restricts also the allowed access formats. Budnik and Kuck [1] described linear skewing schemes that use more memory modules than there are accessed data elements to ease the implementation of the modulus operation. Unfortunately, this approach generates unused memory locations, *holes*.

If two elements are stored in the same memory module and their respective neighbors are also stored in the corresponding memory modules, the module assignment function is called *isotropic* [4]. If an access format F is conflict free in *any* point with respect to an isotropic module assignment function S , then the access format F is conflict free in *every* point [4]. In other words, if an isotropic module assignment function is used, an access format can be placed conflict free everywhere in memory or not at all. In addition, the same is true for linear module assignment functions because they are a subset of isotropic functions.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	i
0	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4	0	
1	3	4	0	1	2	3	4	0	1	2	3	4	0	1	2	3	
2	1	2	3	4	0	1	2	3	4	0	1	2	3	4	0	1	
3	4	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4	
4	2	3	4	0	1	2	3	4	0	1	2	3	4	0	1	2	
5	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4	0	
6	3	4	X	1	2	X	4	0	X	2	3	X	0	1	X	3	
j																	

Fig. 3 Straight access formats with a prime number of memory modules ($N = 5$).

3.2 XOR-Schemes

XOR-schemes always utilize a power of two ($N = 2^n$) memory modules. All the required calculations are bitwise logical operations [2].

As previously noted, if a linear module assignment function is used and an access format is conflict free at any point in memory then the access format is always conflict free. With XOR-schemes, this is not true. To show whether or not an access format is conflict free with respect to an XOR-scheme, methods of linear algebra can be applied [2].

Furthermore, no linear module assignment function exists which is conflict free with respect to rows, columns, and rectangles (p, q) , when the number of accessed data elements $M = N = pq$. On the other hand, it is always possible to find an XOR-scheme that meets the requirements, but access formats are then restrictedly conflict free [4].

Finding an XOR-scheme for a set of conflict free access formats is shown to be NP-complete when the number of memory modules is arbitrary. To solve the problem, a heuristic algorithm and automated heuristic methods have been presented [24].

3.3 Periodic and Multiperiodic Functions

Periodic module assignment functions have some kind of periodicity in the access scheme [3]. A periodic rectangle is called a *basic domain*. The basic domain of a periodic function may be any kind of rectangle, but this discussion concentrates on axially oriented rectangles because of their simplicity in hardware implementation.

Diamond schemes are a special class of multiperiodic functions [4]. The basic domain in diamond schemes is formed as with periodic functions, but there are also two permutations π_1 and π_2 . The permutations change the order of the basic domain when i and j increase respectively.

3.4 Relationships between Functions

The relationships between different classes of module assignment functions are illustrated in Fig. 4 [4]. For example, linear functions are isotropic, periodic, belong to diamond schemes, and general module assignment functions. On the other hand, a periodic function does not have to be a dyadic function but dyadic functions are periodic functions. XOR-schemes are a subset of dyadic functions and multiperiodic

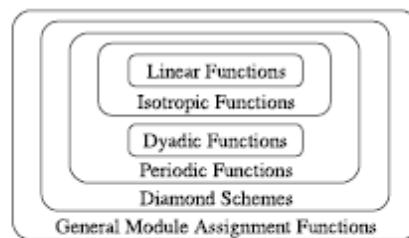


Fig. 4 Relationships between module assignment functions [4].

schemes are an extension of diamond schemes [4].

One important parameter with a conflict free access is the diversity of available scanning points. Having parallel access to an access format at any memory position induces strong limits on the variety of shapes of accessible data templates [3]. However, restricting the positions of access format with a linear module assignment function does not obtain additional conflict free access formats. That is because linear functions are a subset of isotropic functions. On the other hand, using dyadic functions and other periodic and multiperiodic functions may increase the amount of available access formats that can restrictedly be placed on the scanning field.

4. Configurable Parallel Memory Architecture

A block diagram of the Configurable Parallel Memory Architecture is depicted in Fig. 5. The inputs to CPMA are *Virtual Address* and *Command* (write, read). The Virtual Address includes a *Page Address*, which is a pointer to a *Page Table*. *Page Offset* directly indicates the *scanning point r* for the parallel memory. The *Row Address* and the Access Functions are fed from the Page Table to the Address Computation unit. In addition, memory banks and a Data Permutation unit are required for CPMA operation. The CPMA concept allows the memory banks to be any kind of RAM. In the following, single port DRAM is considered.

CPMA address computation is implemented using several configurable computation units in parallel. One unit is dedicated for each type of access formats, module assignment functions, and address functions that CPMA supports. Presented implementation aims at the fastest operation still maintaining sufficient configurability; with pipelining used to enhance the system throughput [25]. Moreover, a configurable Data Permutation unit was designed to conform to system specifications.

CPMA is designed to be configurable both at design time and at run-time. In the hardware implementation, many parameters are VHDL generics, which means that the number of memory blocks, size of a memory module, width of

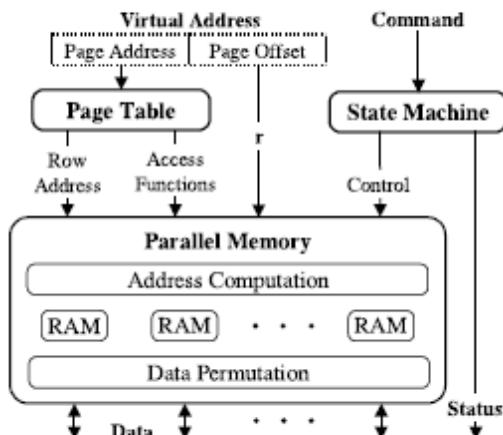


Fig. 5 Configurable parallel memory architecture.

the data bus, and width of input buses are adjustable. Except for the memory modules, all the required blocks, logic, and buses are implemented with logic synthesis from VHDL. In the following, the run-time configurability and some of the CPMA blocks are described in more detail.

4.1 Page Table

The application software can change the contents of the Page Table at run-time. The Page Address is a pointer to a specific row in the Page Table, which determines the CPMA functionality during that memory access. The contents of a single Page Table row are explained in the following.

Row Address has a similar function as a page number in normal memory architecture. **Length of coordinates** L_i and L_j restrict the addressable two-dimensional memory area, the *scanning field*, in dimensions i and j . As mentioned earlier, the data elements are read or written in parallel with an **access format F**. A Page Table row contains a format type indicator and the parameters for constructing the current access format (Access Format Function). All the templates introduced in Chapter 2 can be formed. Moreover, a **module assignment function** $S(i, j)$ indicates the assignment of a data element to a memory module. Linear module assignment functions, XOR-schemes, and diamond schemes have been implemented for the CPMA framework [9]–[11]. **Address function** $a(i, j)$ determines the physical address in a memory module for a data element.

When a user specifies an algorithm for CPMA, this process includes defining the contents of the associated Page Table row. First, the required access formats for the algorithm are evaluated. A module assignment function for conflict free access is chosen and a suitable address function for the module assignment has to be found. Finally, this information is stored in the Page Table.

5. Address Computation

A block diagram of the Address Computation unit is depicted in Fig. 6. The outputs from this unit are the DRAM column addresses. The DRAM row address is passed straight to the memory modules.

Inside the unit, the *Format Control* block specifies the coordinates of the data elements that are accessed in the scanning field. The *Address Decode* block computes the physical addresses for each memory module. The *Module*

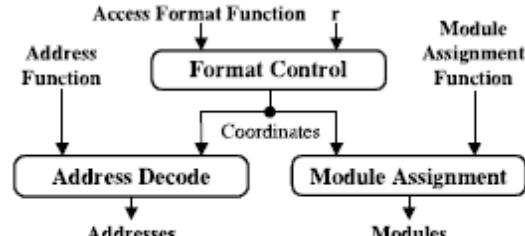


Fig. 6 Address computation unit.

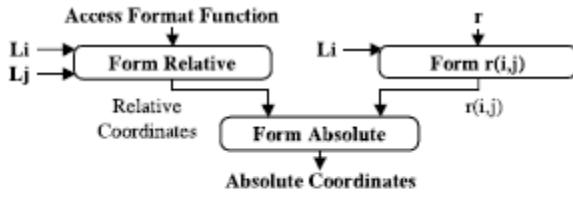


Fig. 7 Format control block.

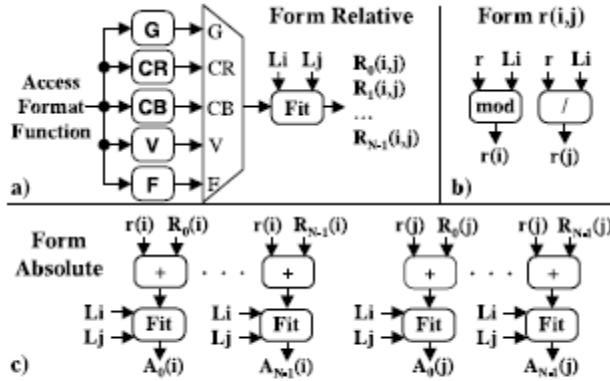


Fig. 8 Sub-blocks in the format control block.

Assignment block, in turn, directs the physical addresses to the proper memory modules.

5.1 Format Control Block

A more detailed block diagram of the Format Control block is depicted in Fig. 7. The inputs to the unit are the Access Format Function, the scanning point r , and the length of coordinates in i and j -directions, L_i and L_j [12].

Relative coordinates for the accessed data elements in the scanning field are determined with (i, j) format in the *Form Relative* block in Fig. 8a. One of the access formats, *generate* (G), *crumbled rectangle* (CR), *chessboard* (CB), *vector* (V), or *free* (F), is selected using a multiplexer depending on the specified format type in the Page Table. The Relative Coordinates $R_0(i, j), R_1(i, j), \dots, R_{N-1}(i, j)$ are calculated in parallel. The *Fit* block ensures that the coordinates fall in the scanning field.

The scanning point is originally represented by an integer, which has to be converted into the coordinate form $r(i, j)$. This is accomplished as follows: $r(i) = r \bmod L_i$ and $r(j) = \lfloor r/L_j \rfloor$ shown as the *Form r(i, j)* block in Fig. 8b.

Absolute coordinates are determined in the *Form Absolute* block depicted in Fig. 8c. Each absolute coordinate is formed in parallel by adding the relative coordinates and $r(i, j)$. Again, a *Fit* block is used to fix coordinates exceeding the maximum values L_i and L_j .

5.2 Module Assignment Block

Figure 9 depicts a block diagram of the Module Assignment block. The choice of module assignment is between a linear module assignment function [9], an XOR-scheme [10], and

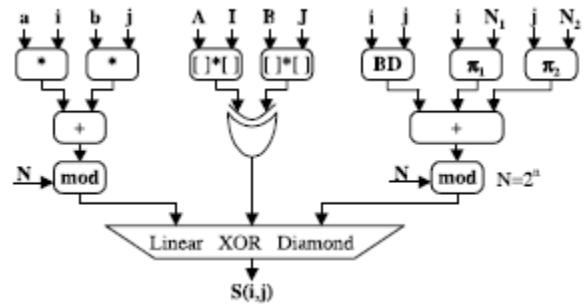


Fig. 9 Module assignment block.

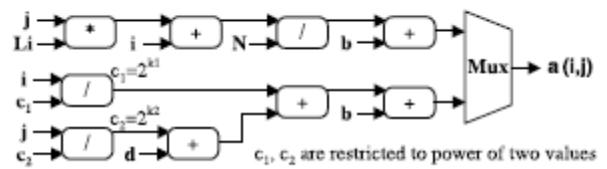


Fig. 10 Address decode block.

a diamond scheme [11]. In Fig. 9, i and j are the absolute coordinates from the Format Control block. I and J correspond to the same values but in binary format. All the other values come from the Page Table.

Linear function uses two multipliers, an adder, and a modulus operation as shown in Fig. 9. In XOR-schemes, A and B are matrices containing binary numbers. Matrix multiplication is achieved by bit-wise logical operations XOR and AND. Moreover, additional bit-wise XOR operations are required for the result. Diamond schemes require the construction of a basic domain (BD) and the two permutations π_1 and π_2 . The three values are added to get a module number. Diamond scheme implementation requires just a power of two modulus operation.

5.3 Address Decode Block

Figure 10 depicts the Address Decode block. The Format Control block provides the coordinates (i, j) as inputs. The other inputs are parameters from the Page Table. Two types of address functions are implemented [9], [11]. The upper one in Fig. 10 is mainly suitable for linear module assignment functions and XOR-schemes. The bottommost one is designed for diamond schemes. Two of the divisions are restricted to power of two operations.

6. Complexity

The Address Computation unit is the most complex block in CPMA almost irrespectively from the implementation details. All of the separated resources required in the Address Computation unit are tabulated in Table 1 where the number of memory modules is denoted by N ($N > 1$). The resources are separated into Format Control (FC), Module Assignment (MA) and Address Decode (AD) blocks. One of the inputs in some of the two-input resources can be a constant and

Table 1 Utilized resources in the address computation unit (resources separated).

	/	Mod	*	+, -	<, >	=	Mux	2^k	/	Mod	*	+, -	<, >	Const
FC	1	1		$7N$	$4N$	$2N - 2$	$19N - 4$				$5N - 10$	$7N - 4$	$6N$	
MA		N	$6N$	$4N$			$3N$		$2N$	$4N$				
AD		N	$2N$	$4N$			$2N$		$2N$					
All	$N + 1$	$N + 1$	$8N$	$15N$	$4N$	$2N - 2$	$24N - 4$	$4N$	$4N$	$5N - 10$	$7N - 4$	$6N$		

Table 2 Utilized resources in the address computation unit (resources shared).

	/	Mod	*	+, -	<, >	=	Mux	2^k	/	Mod	*	+, -	<, >	Const
FC	1	1		$5N$	$2N$	$N - 1$	$15N - 4$				$2N - 4$	$4N - 3$	$2N$	
MA		N	$4N$	$3N$			$7N$		$2N$	$4N$				
AD		N	$2N$	$3N$			$2N$		$2N$					
All	$N + 1$	$N + 1$	$6N$	$11N$	$2N$	$N - 1$	$24N - 4$	$4N$	$4N$	$2N - 4$	$4N - 3$	$2N$		

those units are separately marked ‘Const’. Moreover, specific divider and modulus units are restricted to power of two operation and those units are marked ‘ 2^k ’. They can be implemented using just shift and mask operations. The Format Control block utilizes a total of 1 divider, 1 modulus unit, $7N$ adders, $4N$ comparators, $2N - 2$ equality comparators, and $19N - 4$ multiplexers. In addition, $5N - 10$ multipliers, $7N - 4$ adders, and $6N$ comparators with the other input defined as a constant are required.

As similar resources are used in the implementation, resource sharing can be utilized. Table 2 shows the Address Computation unit with resources shared. Same modulus and divider units are required but the number of multipliers and adders is reduced by $2N$ and $4N$, respectively. Resource sharing may increase multiplexer count as can be seen from the Module Assignment block. The number of resources for each of the resource types is linearly proportional to N .

The largest units in terms of area are arithmetic units, especially dividers and modulus units, in which the input signals can be arbitrary integers from an implementation specific range. If the width of the scanning field is restricted to a power of two, the modulus and division operations could be implemented using just shifting and masking. However, some common image sizes like the size of a CIF (Common Intermediate Format) picture is 352×288 pixels which is not a power of two.

7. Area and Timing

The CPMA blocks were implemented using synthesizable register transfer level VHDL. The area and timing of CPMA is dependent on the selected implementation parameters. This paper concentrates on analyzing the area and the timing of the Address Computation unit. As noted earlier, the Address Computation unit dominates the CPMA implementation area and delay almost irrespective of the selected parameters. Area and timing of the Data Permutation unit and the DRAM Controllers are evaluated elsewhere [25].

The area and timing results are based on logic synthesis. The synthesis technology is a 0.25-micron CMOS process. The evaluations are performed for systems consisting

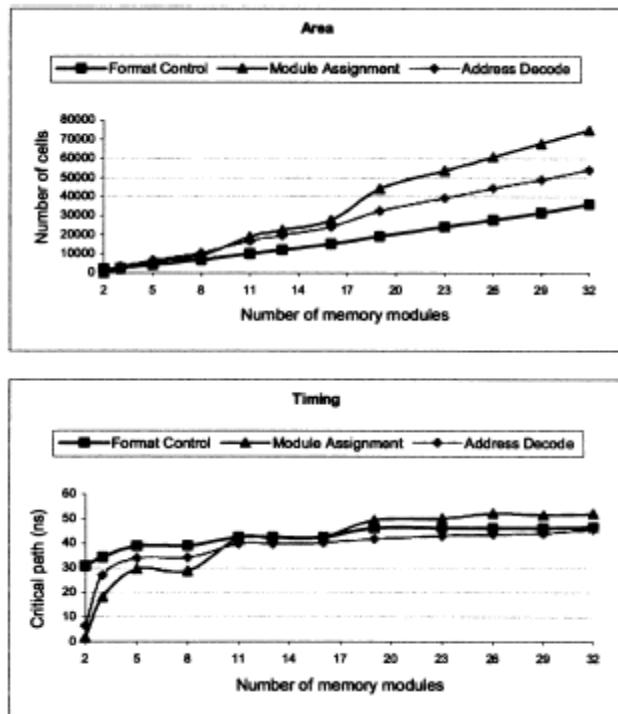


Fig. 11 Area and timing evaluation of the address computation sub-blocks.

of 2–32 memory modules. The two charts in Fig. 11 illustrate the gathered results. The numbers indicating memory modules (N) used to obtain the results in the charts are 2, 3, 5, 8, 11, 13, 16, 19, 23, 26, 29, and 32. Memory capacity of a single memory module is kept constant. Therefore, the total memory capacity increases linearly as a function of the number of memory modules.

Figure 11 shows the contribution of the three sub-blocks of the Address Computation unit to the results. The first diagram indicates the influence of the number of memory modules on area and the second diagram the influence on timing. In the area charts, the cell count is based on 2-input NAND-gates. In the timing charts, the critical path is the slowest combinatorial path between two successive se-

quential elements.

As expected, areas of all the three blocks increase approximately linearly as a function of the number of memory modules. However, the widths of most implementation resources increase substantially when the number of memory modules increases over a power of two boundary. This accounts for most of the area nonlinearity in Fig. 11.

The block critical path delay increases mainly after each power of two boundary. The delay stays relatively constant when the number of memory modules is greater than four. When $N = 2$, Module Assignment and Address Decode blocks have very short delay values because just one bit precision is required for modulus and division operations.

Synthesis tools are affected by the given options and parameters. Therefore, the results in this study are most useful in illustrating relative changes in designs.

8. Conclusions

The Configurable Parallel Memory Architecture enables a processor to use a multitude of memory access templates. These access formats can guarantee the usefulness of data and, therefore, data refresh operations are required less frequently.

In this paper, we have presented an address computation implementation for CPMA. Access formats required in many applications and several skewing schemes can be constructed using the presented implementation. Parameters detailing a template and module assignment function can flexibly be changed at run-time without a need to redesign the whole hardware implementation. The utilized resources were shown to be linearly proportional to the number of memory modules.

Acknowledgments

This research was financially supported by the Academy of Finland (grants 104487 and 105328), Nokia Foundation, Ulla Tuominen Foundation, Foundation of Technology (TES), and Graduate School in Electronics, Telecommunications and Automation (GETA).

References

- [1] P. Budnik and D.J. Kuck, "The organization and use of parallel memories," *IEEE Trans. Comput.*, vol.C-20, no.12, pp.1566–1569, Dec. 1971.
- [2] J.M. Fraile, W. Jalby, and J. Lenfant, "XOR-schemes: A flexible data organization in parallel memories," Proc. Int'l Conf. Parallel Processing, pp.276–283, Washington, DC, USA, Aug. 1985.
- [3] H.D. Shapiro, "Theoretical limitations on the efficient use of parallel memories," *IEEE Trans. Comput.*, vol.C-27, no.5, pp.421–428, May 1978.
- [4] M. Gössel, B. Rebel, and R. Creuzburg, *Memory Architecture & Parallel Access*, Elsevier Science, Amsterdam, The Netherlands, 1994.
- [5] K. Kuusilinna, J. Tanskanen, T. Hääläinen, J. Niitylahti, and J. Saarinen, "Configurable parallel memory architecture for multimedia computers," *J. Systems Architecture*, vol.47, no.14–15, pp.1089–1115, Aug. 2002.
- [6] L. Zhang, Z. Fang, M. Parker, B.K. Mathew, L. Schaelicke, J.B. Carter, W.C. Hsieh, and S.A. McKee, "The impulse memory controller," *IEEE Trans. Comput.*, vol.50, no.11, pp.1117–1132, Nov. 2001.
- [7] W. Hinrichs, J.P. Wittenburg, H. Lieske, H. Kloos, M. Ohmacht, and P. Pirsch, "A 1.3-GOPS parallel DSP for high-performance image-processing applications," *IEEE J. Solid-State Circuits*, vol.35, no.7, pp.946–952, July 2000.
- [8] S. Lee, M. Haniyama, and M. Kameyama, "An FPGA-oriented motion-stereo processor with a simple interconnection network for parallel memory access," *IEICE Trans. Inf. & Syst.*, vol.E83-D, no.12, pp.2122–2130, Dec. 2000.
- [9] E. Aho, J. Vanne, K. Kuusilinna, T. Hääläinen, and J. Saarinen, "Configurable address computation in a parallel memory architecture," Proc. WSES Int'l Conf. Circuits, Systems, Communications and Computers, pp.4941–4946, Rethymnon, Greece, July 2001.
- [10] E. Aho, J. Vanne, K. Kuusilinna, and T. Hääläinen, "XOR-scheme implementations in configurable parallel memory," Proc. Int'l Workshop System-on-Chip for Real-Time Applications, pp.287–298, Banff, Canada, July 2002.
- [11] E. Aho, J. Vanne, K. Kuusilinna, and T. Hääläinen, "Diamond scheme implementations in configurable parallel memory," Proc. IEEE Int'l Workshop Design and Diagnostics of Electronic Circuits and Systems, pp.211–218, Brno, Czech Republic, April 2002.
- [12] E. Aho, J. Vanne, K. Kuusilinna, and T. Hääläinen, "Access format implementations in configurable parallel memory," Proc. Int'l Conf. Comput. and Information Science, pp.59–64, Seoul, Korea, Aug. 2002.
- [13] D.C. van Voorhis and T.H. Morrin, "Memory systems for image processing," *IEEE Trans. Comput.*, vol.C-27, no.2, pp.113–125, Feb. 1978.
- [14] K. Kim and V.K. Prasanna Kumar, "Parallel memory systems for image processing," Proc. IEEE Conf. Computer Vision and Pattern Recognition, pp.654–659, San Diego, CA, USA, June 1989.
- [15] J. Takala and T. Järvinen, "Stride permutation access in interleaved memory systems," in *Domain-Specific Processors: Systems, Architectures, Modeling and Simulation*, ed. S. Bhattacharyya, E. Deprettere, and J. Teich, pp.63–84, Marcel Dekker, New York, NY, USA, 2003.
- [16] J. Tanskanen, T. Sihvo, J. Niitylahti, J. Takala, and R. Creutzburg, "Parallel memory access schemes for H.263 encoder," Proc. IEEE Int'l Symp. Circuits and Systems, pp.691–694, Geneva, Switzerland, May 2000.
- [17] D.T. Harper III, "Block, multistride vector and FFT accesses in parallel memory systems," *IEEE Trans. Parallel Distrib. Syst.*, vol.2, no.1, pp.43–51, Jan. 1991.
- [18] M. Al-Mouhamed, L. Bic, and H. Abu-Hamed, "A compiler transformation to improve memory access time in SIMD systems," Proc. Int'l Conf. Parallel Architectures and Compilation Techniques, pp.174–178, Boston, MA, USA, Oct. 1996.
- [19] P.R. Panda, N. Dutt, and A. Nicolau, *Memory Issues in Embedded Systems-On-Chip: Optimizations and Exploration*, Kluwer Academic Publishers, London, Great Britain, 2002.
- [20] K.H. Al-Saqabi and E.W. Davis, "Extending parallelism to memory hierarchies in massively parallel systems," IEE Proc. — Computers and Digital Techniques, vol.138, no.4, pp.193–202, July 1991.
- [21] Z. Liu and X. Li, "XOR storage schemes for frequently used data patterns," *J. Parallel and Distributed Computing*, vol.25, no.2, pp.162–173, March 1995.
- [22] X. Li, "Parallel algorithms for hierarchical clustering and cluster validity," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol.12, no.11, pp.1088–1092, Nov. 1990.
- [23] K.E. Batcher, "The multidimensional access memory in STARAN," *IEEE Trans. Comput.*, vol.C-26, no.2, pp.174–177, Feb. 1977.
- [24] M.A. Al-Mouhamed and S.S. Seiden, "Minimization of memory and network contention for accessing arbitrary data patterns in SIMD systems," *IEEE Trans. Comput.*, vol.45, no.6, pp.757–762, June 2006.

1996.

- [25] J. Vanne, E. Aho, K. Kuusilinna, and T. Hämäläinen, "Enhanced configurable parallel memory architecture," Proc. Euromicro Symp. Digital System Design, pp.28–35, Dortmund, Germany, Sept. 2002.



Eero Aho received the M.S. degree in Electrical Engineering from Tampere University of Technology (TUT), Finland in 2001. His research interests include memory systems and parallel processing. Currently, he is working toward his PhD degree as a research scientist at the Institute of Digital and Computer Systems (DCS) at TUT.



Jarno Vanne received the M.S. degree in Information Technology from Tampere University of Technology, Finland in 2002. His research interests include memory systems and hardware accelerators. Currently, he is working toward his PhD degree as a research scientist at the Institute of Digital and Computer Systems at TUT.



Kimmo Kuusilinna PhD '01, Tampere University of Technology. His main research interests include hardware emulation, interconnection networks, system-on-chip design, and parallel memories. Currently he is working as a senior research scientist at the Institute of Digital and Computer Systems at TUT.



video.

Timo D. Hämäläinen MSc '93, PhD '97, TUT. He acted as a senior research scientist in the DCS at TUT in 1997–2001 and during the time founded a research group named "Parallel DSP processing and wireless multimedia systems." He has acted as a project manager and research supervisor for several academic and industrial projects. In 2001 he has taken the position of full professor at TUT and continues the research on wireless local and personal area networking as well as SoC solutions for wireless

Publication 6

E. Aho, J. Vanne, K. Kuusilinna, and T. D. Hämäläinen, “Comments on “Winscale: An Image-Scaling Algorithm Using an Area Pixel Model”,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 15, no. 3, pp. 454–455, March 2005.

Copyright © The Institute of Electrical and Electronics Engineers (IEEE).
Reprinted with permission.

Copyright© 2005 IEEE. Reprinted from IEEE Transactions on Circuits and Systems for Video Technology 2005.□□□□

This material is posted here with permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any of Tampere University of Technology's products or services. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org.□□□□

By choosing to view this document, you agree to all provisions of the copyright laws protecting it.□□

Comments and Corrections

Comments on “Winscale: An Image-Scaling Algorithm Using an Area Pixel Model”

Eero Aho, Jarno Vanne, Kimmo Kuusilinna, and Timo D. Hämäläinen

Abstract—In the paper by Kim *et al.* (2003), the authors propose a new image scaling method called *winscale*. The presented method can be used for scaling up and down. However, scaling down utilizing the *winscale* concept gives exactly the same results as the well-known bilinear interpolation. Furthermore, compared to bilinear, scaling up with the proposed *winscale* “overlap stamping” method has very similar calculations. The basic *winscale* upscaling differs from the bilinear method.

Index Terms—Area pixel model, image scale, interpolation.

The paper by Kim *et al.* [1] presents a new scaling algorithm, *winscale*. The method scales up or down using an area pixel model instead of the common point pixel model. In the following, *bilinear* interpolation [2] is evaluated using a similar area pixel model interpretation and the similarities with the *winscale* algorithm are shown.

Bilinear interpolation can be calculated by three one-dimensional (1-D) *linear* interpolations [2]. First, two 1-D interpolations in the *x* direction are done, and then one interpolation in the *y* direction. Fig. 1 illustrates bilinear interpolation. The scaled pixel value *P* can be filtered by the original image pixels *C*0–*C*3 as

$$P = B_0 \cdot C_0 + B_1 \cdot C_1 + B_2 \cdot C_2 + B_3 \cdot C_3. \quad (1)$$

Distances between the scaled pixel *P* and the original pixel *C*0 are depicted by *bW* and *bH* in *x* and *y* directions, respectively. Distance from *C*0 to *C*2 and from *C*0 to *C*1 equals 1.0. The coefficients *B*0–*B*3 are defined as follows:

$$\begin{cases} B_0 = (1 - bW) \cdot (1 - bH) \\ B_1 = (1 - bW) \cdot bH \\ B_2 = bW \cdot (1 - bH) \\ B_3 = bW \cdot bH. \end{cases} \quad (2)$$

Coefficient values *B*0–*B*3 correspond to the areas shown in Fig. 1. Let us define coordinates for *C*0 and *C*3 as (*C*0_x, *C*0_y) and (*C*3_x, *C*3_y), respectively. The scaled pixel coordinates are (*P*_x, *P*_y). Therefore, for the coefficients *B*0–*B*3

$$\begin{cases} bW = P_x - C_0_x \\ bH = P_y - C_0_y \\ (1 - bW) = C_3_x - P_x \\ (1 - bH) = C_3_y - P_y. \end{cases} \quad (3)$$

Manuscript received May 24, 2004; revised October 5, 2004. This paper was recommended by Associate Editor L.-G. Chen.

The authors are with the Institute of Digital and Computer Systems, Tampere University of Technology, FIN-33101 Tampere, Finland (e-mail: eero.aho@tut.fi).

Digital Object Identifier 10.1109/TCSVT.2004.842599

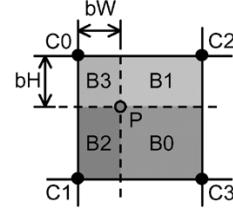


Fig. 1. Scaling with bilinear.

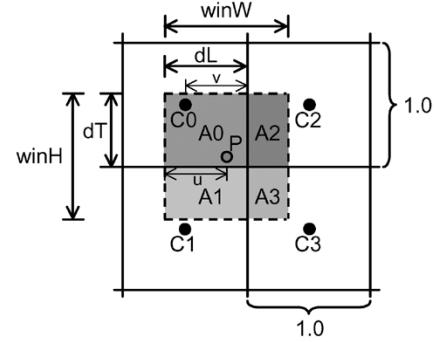


Fig. 2. Scaling with winscale.

When (3) is applied to (2), the coefficients are defined as

$$\begin{cases} B_0 = (C_{3x} - P_x) \cdot (C_{3y} - P_y) \\ B_1 = (C_{3x} - P_x) \cdot (P_y - C_{0y}) \\ B_2 = (P_x - C_{0x}) \cdot (C_{3y} - P_y) \\ B_3 = (P_x - C_{0x}) \cdot (P_y - C_{0y}). \end{cases} \quad (4)$$

Scaling with *winscale* is depicted in Fig. 2. Pixel intensity is assumed to be evenly distributed on the enclosing rectangular area. Dots in Fig. 2 are located in the middle of the respective pixel intensity rectangles. The area of an original image pixel is 1.0. Each scaled pixel is obtained by weighted-averaging the original image pixel values with area coverage ratio. *Winscale* defines the scaled pixel value *P* as

$$P = SF \cdot (A_0 \cdot C_0 + A_1 \cdot C_1 + A_2 \cdot C_2 + A_3 \cdot C_3). \quad (5)$$

The filter window size is restricted with *winW* and *winH*, the width and the height of the scaled pixel region, respectively. The scaled pixel *P* is located in the middle of the filter window. The scale factor (SF) depends on the filter window size. Fractional parts of the filter window boundary coordinates are denoted by *dL* and *dT*. The area coverage (*A*0–*A*3) of the filter window can be defined as

$$\begin{cases} A_0 = dL \cdot dT \\ A_1 = dL \cdot (winH - dT) \\ A_2 = (winW - dL) \cdot dT \\ A_3 = (winW - dL) \cdot (winH - dT). \end{cases} \quad (6)$$

When scaling down, SF is defined as 1.0 and *winW* = *winH* = 1.0. Therefore, *dL* and *dT* can be calculated as in (7). Half of the original

pixel $C0$ intensity rectangle width and half of the filter window width are denoted by v and u , respectively.

$$\left\{ \begin{array}{l} dL = (C0_x + v) - (P_x - u) = (C0_x + 0.5) - (P_x - \text{winW}/2) \\ \quad = (C0_x + 1.0) - P_x = C3_x - P_x \\ dT = (C0_y + 0.5) - (P_y - \text{winH}/2) = (C0_y + 1.0) - P_y \\ \quad = C3_y - P_y \\ \text{winW} - dL = 1 - dL = 1 - (C3_x - P_x) = P_x + (1 - C3_x) \\ \quad = P_x - (C3_x - 1) = P_x - C0_x \\ \text{winH} - dT = 1 - dT = 1 - (C3_y - P_y) = P_y + (1 - C3_y) \\ \quad = P_y - (C3_y - 1) = P_y - C0_y. \end{array} \right. \quad (7)$$

When (7) is applied to (6), the winscale downscaling coefficients are defined as

$$\left\{ \begin{array}{l} A0 = (C3_x - P_x) \cdot (C3_y - P_y) \\ A1 = (C3_x - P_x) \cdot (P_y - C0_y) \\ A2 = (P_x - C0_x) \cdot (C3_y - P_y) \\ A3 = (P_x - C0_x) \cdot (P_y - C0_y). \end{array} \right. \quad (8)$$

As can be seen from (4) and (8), $B0 = A0$, $B1 = A1$, $B2 = A2$, and $B3 = A3$. This means that with the assumptions above, winscale downscaling gives exactly the same results as bilinear downscaling.

The two methods use different pixel models. Winscale uses the area pixel model, and bilinear uses the point pixel model. As stated above, winscale treats pixels as rectangles and pixel intensity is evenly distributed in rectangle area. Therefore, pixel center position is always in the middle of the pixel intensity rectangle area. With these two algorithms, the distance between an original pixel center position and a downscaled pixel center position is exactly the same, irrespective of the used pixel model. Moreover, with winscale downscaling, the filter window size is defined as 1.0. These issues make the filtering coefficients identical in downscaling and result in the same downscaled pixel values with winscale and bilinear.

The same was also verified with C language implementations. Input video images were scaled down with bilinear and winscale and the results were compared. Scaled values differed at maximum 0.4%. The difference comes from the limited accuracy of the implementations. The 0.4% difference equals to 1/255 that is the minimum possible difference when 8 bits are used for one pixel color component. This difference comes when two values are rounded to different directions, one rounded up and the other down.

When scale up ratio is below 1/2, winscale utilizes *prescaler*, which shrinks the image by two's power to adjust the scale up ratio to be between 1 and 1/2. This can make the downscaled image different from bilinear. However, the prescaler does not relate to the actual winscale concept but it is just an additional part to ensure that all original pixels are used at least once. Similar prescaler can be used with bilinear as well.

The winscale algorithm uses similar hardware operations to bilinear. However, winscale is a more complex algorithm in terms of operations per pixel, as can be seen from [1, Table I].

The paper by Kim *et al.* [1] also introduces a modified winscale *scaling up* method called *overlap stamping*. When compared to bilinear, scaling up with the proposed method has very similar calculations. In the overlap stamping method, the area of the filter window is defined as 1.0. This leads to the same restrictions as with downscaling: SF = 1.0 and winW = winH = 1.0. However, the paper defines an incremental value for the next scaled pixel exceptionally to $(N-1)/(M-1)$ when an image is scaled up from $N \times N$ to $M \times M$ pixels. With the normal incremental value N/M , the results would be same as scaling up with bilinear.

Increasing the ratio between original image and scaled image (scaling factor) increases the differences between winscale overlap stamping method and bilinear results. When incremental values for the next scaled pixel are determined differently, the scaled pixel locations differ more when a scaling factor increases.

The difference between the winscale overlap stamping method and bilinear was evaluated with C language implementations. The input video sequence was CarPhone with frame size 176×144 (QCIF). Scaling up to sizes 240×192 and 1024×768 (XGA) produces scaled value differences at maximum 6.3% and 20.0%, respectively.

Scaled image qualities were evaluated in the paper [1]. As shown in [1, Fig. 8], the winscale overlap stamping method achieves worse image quality than bilinear interpolation with all of the shown test images.

The normal scaling up with winscale (without the overlap stamping method) uses SF and filtering window size different from 1.0. These factors give different results from bilinear.

The winscale algorithm has good edge characteristics, which are better than bilinear [1]. Therefore, winscale is useful when scaling up text or other fine edge images, and the small complexity overhead, compared to bilinear interpolation, is acceptable.

To conclude, the new image scaling method called winscale can be used for scaling up and down [1]. In this paper, we have shown that scaling down utilizing the winscale concept gives exactly the same results as the well-known bilinear interpolation. Furthermore, compared to bilinear, scaling up with the proposed winscale overlap stamping method has very similar calculations. The basic winscale upscaling differs from the bilinear method and may be useful when text or other fine edge images are scaled up.

REFERENCES

- [1] C.-H. Kim, S.-M. Seong, J.-A. Lee, and L.-S. Kim, "Winscale: An image-scaling algorithm using an area pixel model," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 13, no. 6, pp. 549–553, Jun. 2003.
- [2] T. M. Lehmann, C. Gönner, and K. Spitzer, "Survey: Interpolation methods in medical image processing," *IEEE Trans. Med. Imag.*, vol. 18, no. 11, pp. 1049–1075, Nov. 1999.

Publication 7

E. Aho, J. Vanne, T. D. Hämäläinen, and K. Kuusilinna, “Block-Level Parallel Processing for Scaling Evenly Divisible Images,” *IEEE Transactions on Circuits and Systems I*, vol. 52, no. 12, pp. 2717–2725, December 2005.

Copyright © The Institute of Electrical and Electronics Engineers (IEEE).
Reprinted with permission.

Copyright© 2005 IEEE. Reprinted from IEEE Transactions on Circuits and Systems I 2005.

This material is posted here with permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any of Tampere University of Technology's products or services. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org.

By choosing to view this document, you agree to all provisions of the copyright laws protecting it.

Block-Level Parallel Processing for Scaling Evenly Divisible Images

Eero Aho, Jarno Vanne, Timo D. Hämäläinen, and Kimmo Kuusilinna

Abstract—Image scaling is a frequent operation in medical image processing. This paper presents how two-dimensional (2-D) image scaling can be accelerated with a new coarse-grained parallel processing method. The method is based on evenly divisible image sizes which is, in practice, the case with most medical images. In the proposed method, the image is divided into slices and all the slices are scaled in parallel. The complexity of the method is examined with two parallel architectures while considering memory consumption and data throughput. Several scaling functions can be handled with these generic architectures including linear, cubic B-spline, cubic, Lagrange, Gaussian, and sinc interpolations. Parallelism can be adjusted independent of the complexity of the computational units. The most promising architecture is implemented as a simulation model and the hardware resources as well as the performance are evaluated. All the significant resources are shown to be linearly proportional to the parallelization factor. With contemporary programmable logic, real-time scaling is achievable with large resolution 2-D images and a good quality interpolation. The proposed block-level scaling is also shown to increase software scaling performance over four times.

Index Terms—Image zoom, interpolation, parallelization, two-dimensional (2-D) image scale.

I. INTRODUCTION

RESAMPLING is utilized when a discrete image is transformed to a new set of coordinate points. Some resampling examples are image scaling, spatial distortion correction, and image rotation. Resampling can be divided conceptually into two sequential processes. At first, a discrete image is interpolated to a continuous image. Then, the interpolated image is sampled. In practical implementations, interpolation and sampling are often combined so that the image is interpolated at only those points that are sampled.

Many medical imaging systems use scaling as a part of their normal operation. Aspect ratios are corrected by scaling a two-dimensional (2-D) image in one dimension [1]. Temporal registration of retinal images is used, for example, to follow disease evolution. Scaling is required when the images are of different resolutions or the eye and the camera are at different distances [2]. Ultrasound is particularly suitable for imaging abdominal

Manuscript received February 1, 2005; revised July 22, 2005. This work was supported in part by the Academy of Finland by Grant 104487, by the Nokia Foundation, the Heikki and Hilma Honkanen Foundation, and the Graduate School in Electronics, Telecommunications and Automation (GETA). This paper was recommended by Guest Editor Y. Lian.

E. Aho, J. Vanne, and T. D. Hämäläinen are with the Institute of Digital and Computer Systems, Tampere University of Technology, FI-33101 Tampere, Finland (e-mail: eero.aho@tut.fi).

K. Kuusilinna is with Nokia Research Center, FI-33721 Tampere, Finland. Digital Object Identifier 10.1109/TCSI.2005.856894

and thoracic organs, such as the heart. Registration of ultrasound images benefits from real-time scaling [3]. Real-time, high quality scaling is also applicable in magnetic resonance imaging (MRI) [4], [5] and computer aided surgery (CAS) [6].

Complexity and quality of scaling varies depending on the used interpolation method. Several medical imaging studies have compared interpolation methods [1], [7], [8]. The most recent [1] evaluates *nearest neighbor*, *linear*, *quadratic*, *cubic B-spline*, *cubic*, *truncated sinc*, *windowed sinc*, *Lagrange*, and *Gaussian* interpolation methods. The simplest and fastest one, *nearest neighbor*, also incurs the largest interpolation error. The runtimes of software interpolation implementations were measured and the most complex one, *Gaussian* interpolation, used on average 22 times more time than *linear* interpolation [1]. Scaling large resolution images in real-time requires lots of computation performance, particularly with complex interpolation algorithms needed when faithful preservation of image details is important. Parallel processing is one possible solution to these challenges.

Most 2-D image sizes introduced in medical imaging publications are divisible by some integer other than one (768×584 [9], 256×256 [10], 3300×2600 [11], 760×570 [12]). Standard image sizes like VGA, XGA, PAL, NTSC, QSIF, and CIF are divisible by the number eight. One explanation for this property is that image and video compression standards divide an image to 8×8 or 16×16 blocks (JPEG, MPEG-4, H.264). However, image sizes are not always compatible with those block sizes. Therefore, before compression, the image size is changed to fulfill the requirements of the standard (for example, with padding). After that, the divisibility of the image width and height are known and this knowledge can be utilized during the scaling.

For efficient computation, many practical optimizations have been presented. Constant filtering coefficients for a specific scaling ratio are used in a video scaler presented in [13]. Image sizes are restricted to the power of two in a compressed domain scaling method shown in [14]. A new linear type scaling algorithm *winscale* is implemented on an FPGA in [15].

The aforementioned scaling implementations use simple interpolation algorithms without hard real-time constraints. An example of parallel scaling implementation is given in [16]. Pixels are processed in a very fine-grained fashion which is the most common approach. Real-time (20 frames/s) cardiac MRI system is implemented with four floating point digital signal processors (DSPs) (TMS320C40) [4]. Two of the DSPs are dedicated to image calculations, such as rotation and scaling. The parallelization is very coarse-grained.

This paper presents a novel method for parallelizing scaling. A 2-D image is divided into slices and all the slices are scaled in parallel. Here, this is called *block-level parallelism*. The parallelization granularity is between the above mentioned implementations. The method is applicable for both software and hardware implementation, however, the circuit implementation is the focus of this paper. The idea of the parallelization method is given in [18]. This paper presents implementation details in addition to resource counts and performance evaluation.

Two basic parallel architectures are presented to demonstrate the implementation of block-level parallelism. The proposed parallel architectures can be used with several interpolation algorithms. The parallelization and, therefore, performance can be easily adjusted because of the computation regularity. In addition, the parallelism can adapt independently from the computational unit structure. In this work, based on preliminary evaluation, the most promising architecture was implemented as a simulation model. The original and scaled image sizes can be arbitrary integers from an implementation specific range. The required hardware resources were estimated and the performance evaluated.

Background for scaling evenly divisible images is given in Section II. Block boundaries are analyzed in Section III. Section IV introduces the proposed parallel architectures and Section V evaluates memory requirements. Section VI presents the implemented architecture in detail and Section VII gives the results. Section VIII concludes the paper.

II. SCALING EVENLY DIVISIBLE IMAGES

When scaling an image, pixels are required to be interpolated only at those points that are sampled. The interpolated 2-D image sampling point distances are Δx and Δy , horizontal distance and vertical distance, respectively. Normally, these distances are defined as follows:

$$\begin{aligned}\Delta x &= \frac{\text{original_image_width}}{\text{scaled_image_width}} \\ \Delta y &= \frac{\text{original_image_height}}{\text{scaled_image_height}}.\end{aligned}\quad (1)$$

In the following, the *original block* is always a part of the original image. The width and/or height of the original image are multiples of the original block. In addition, the original block can be scaled with the same scaling ratio as the original image. A *scaled block* is scaled from the original block with the scaling ratio. Naturally, the width and height of the original and the scaled block must be natural integers.

Fig. 1 depicts an original block size $c_1 \times d_1$ that is scaled to size $c_2 \times d_2$. The original image is scaled with the same ratios and, in this example, the width a_1 is three times and the height b_1 twice the size of the original block. Therefore, the sampling point distances Δx and Δy are the same in the both scaling examples. Hence, block-level parallelism can be utilized when scaling the original image. In Fig. 1, if six units designed for block scaling are processing in parallel, each can scale a separate part of the original image.

In general, the same scaling can be done in $\gcd(a_1, a_2)$ times in x -direction and $\gcd(b_1, b_2)$ times in y -direction where \gcd is the greatest common divisor. This method is applicable for

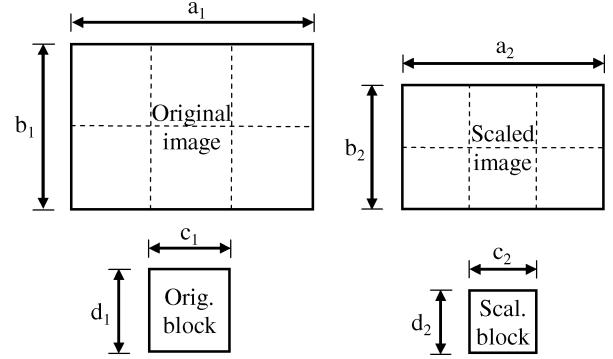


Fig. 1. Evenly divisible images scaled down.

both upscaling and downscaling providing that the interpolated image sampling point distance is defined as shown in (1). All the interpolation methods shown in [1] fulfill the requirement and the algorithms can be parallelized in block-level. This is not the case with all the interpolation methods. For example, winscale overlap stamping method [15] exceptionally defines the distances as follows:

$$\begin{aligned}\Delta x &= \frac{(\text{original_image_width} - 1)}{(\text{scaled_image_width} - 1)} \\ \Delta y &= \frac{(\text{original_image_height} - 1)}{(\text{scaled_image_height} - 1)}.\end{aligned}\quad (2)$$

This overlap stamping technique does not fulfill the requirement and cannot be parallelized with the proposed method.

III. BLOCK BOUNDARIES

An image can be divided row-wise, column-wise, or, when extensive parallelism is required, utilizing both methods. In the row-wise approach, the image is divided into slices in horizontal direction and each slice is scaled separately. Respectively, in column-wise method, the image is divided into slices in vertical direction.

When image blocks are scaled separately, the pixels close to the block boundary may need the neighboring block pixels for interpolation. This overlapping amount depends on the utilized interpolation algorithm, image division direction (row- or column-wise), image sizes (original and scaled), and whether scaling up or down is used. Moreover, if overlapping occurs, blocks located in the middle of an image utilize more overlapped pixels than the blocks next to an image boundary. Within the interpolation algorithm, essential information for overlapping amount is how many original image pixels with the proposed algorithm are used for scaling a single pixel.

As mentioned, it is also possible to divide an image simultaneously both row- and column-wise. In that case, overlapped pixels can exist in all of the surrounding eight blocks, complicating image loading and temporary pixel storage. Hence, it is advisable to do the image division only row- or column-wise if sufficient parallelism is achieved that way. For these cases, the overlapped pixels exist at maximum in the two neighboring blocks. In the following, merely a row- or column-wise division is considered.

Table I tabulates the maximum number of overlapped rows or columns per a block depending on whether a row- or column-

TABLE I
MAXIMUM NUMBER OF OVERLAPPED ROWS OR COLUMNS PER A BLOCK
FOR A SELECTION OF WELL KNOWN SCALING ALGORITHMS
($\lfloor k/2 \rfloor \leq c_1, \lfloor k/2 \rfloor \leq c_2$)

Algorithm	$k \times k$	Downscale		Upscale	
		p_e	p_m	p_e	p_m
Nearest neighbor	1×1	0	0	0	0
Linear	2×2	0	0	1	2
Quadratic	3×3	1	2	1	2
Cubic B-spline	4×4	1	2	2	4
Lagrange	5×5	2	4	2	4
Cubic	6×6	2	4	3	6
Truncated sinc	7×7	3	6	3	6
Gaussian	8×8	3	6	4	8

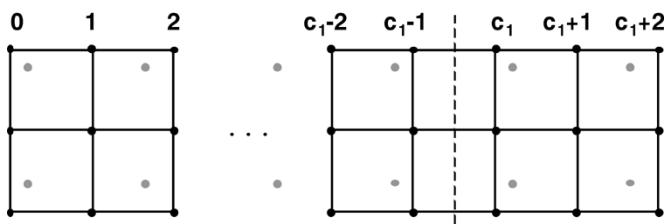


Fig. 2. Block boundary in downscaling.

wise approach is used, respectively. Each of the eight interpolation methods selected utilizes a different number of original image pixels $k \times k$ for scaling a single pixel. The variants of the same interpolation methods can also utilize a different number of samples than shown in Table I [1]. However, the sample count of the algorithm does not alter with a single implementation. Moreover, it is possible to use different interpolation algorithm in x and y -directions as done in [16]. However, some additional control and coefficient computation logic is demanded. In this paper, both x and y -directions are considered to use the same interpolation method. Middle blocks p_m have neighbor blocks on two sides whereas edge blocks p_e have a block just on one side. Variables k , c_1 , and c_2 are restricted as $\lfloor k/2 \rfloor \leq c_1, \lfloor k/2 \rfloor \leq c_2$ to guarantee that overlapped pixels exist only in the neighboring blocks.

A. Downscaling

Let us assume an image size $a_1 \times b_1$ that is scaled to size $a_2 \times b_2$ (Fig. 1). The original block size is $c_1 \times d_1$. Details of a block boundary in x -direction are shown in Fig. 2. Downscaling is calculated using the interpolated image sampling point distances as shown in (1). The black dots in Fig. 2 depict original image pixels and the gray dots downscaled image pixels. The x -coordinates of the original image are shown above the grid. A block boundary is shown with a vertical broken line.

All the downscaled pixels are located inside the outermost pixels of the corresponding original image block (Fig. 2). Therefore, overlapping does not occur when a maximum of four original image pixels per scaled pixel are used and the filtered pixels are those surrounding the scaled pixel location (Table I).

When more than four pixels are utilized, scaled boundary pixels may require additional pixels from the surrounding original blocks. For example, cubic B-spline interpolation utilizes the surrounding 4×4 original pixels for filtering a scaled pixel. Depending if row- or column-wise division is used, at maximum

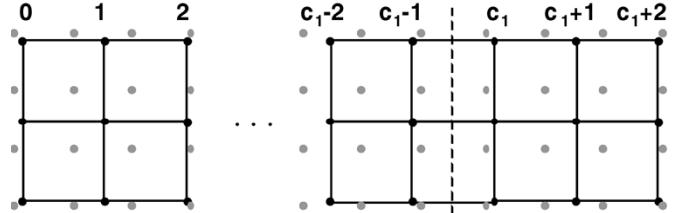


Fig. 3. Block boundary in upscaling.

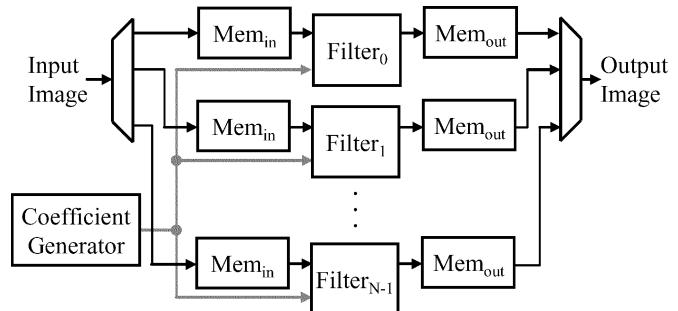


Fig. 4. Distributed memory architecture for image scaling.

one row or column is overlapped with each surrounding block (Table I). In the worst case situation, two original image rows or columns are overlapped with the surrounding blocks.

B. Upscaling

Fig. 3 depicts upscaling with corresponding original image and block sizes as in Fig. 2. As with downscaling, the shown upscaling utilizes the normal interpolated image sampling point distances (1). As can be seen, the upscaled boundary pixels are located outside of the original block. However, when the scaled image is centered relative to the original image, the closest original pixel is always in the currently processed original block. Therefore, the nearest neighbor algorithm, which utilizes just one original pixel for filtering a scaled pixel, does not require overlapped pixels (Table I).

When more than one pixel is utilized, original image blocks may overlap some rows or columns. The maximum number of overlapped rows and columns is equal or one more in each direction than with the respective downscaling algorithm (Table I).

IV. BLOCK-LEVEL PARALLEL ARCHITECTURES FOR IMAGE SCALING

Two general block-level parallel architectures for image scaling are depicted in Figs. 4 and 5. Both of the architectures have N Filter units, N being the parallelization factor. In addition, a Coefficient Generator unit and memory units are needed. Same coefficients are fed to all the Filter units. However, each Filter unit scales a separate image block. The Filter and Coefficient Generator units are implemented to comply with the interpolation algorithm. Filters interpolate first horizontally and then vertically. Therefore, a small buffer is required inside the Filter units. The simplest possible Filter unit can be implemented with a multiply accumulate (MAC) unit. However, more complex structures can also be used to further increase performance. Coefficients can be calculated in advance as done in [17] or on the fly using the Coefficient Generator unit.

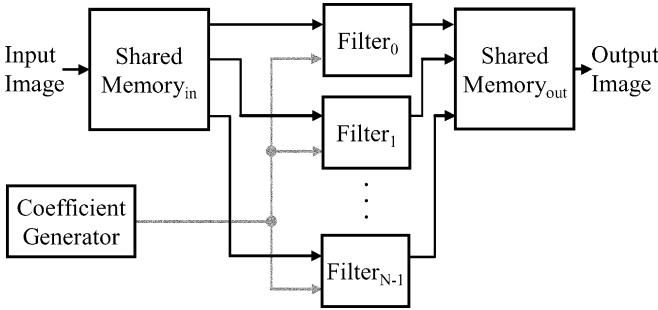


Fig. 5. Shared memory architecture for image scaling.

The architecture in Fig. 4 utilizes distributed memory whereas the one in Fig. 5 has a shared memory. The parallelization can be increased by adding more Filter units. However, the block count in an image restricts the maximum parallelism. These architectures are suitable for all the interpolation algorithms that fulfill the block-level parallelism restrictions mentioned previously.

A. Distributed Memory Architecture

In the distributed memory architecture (Fig. 4), each filter has its own memory module. The input image slices are distributed to each memory module, and, therefore, the overlapped pixels in block boundaries have to be taken into account. Perhaps the simplest way is to handle the block boundaries similarly to the image boundary, for example, by padding the required pixels. However, the scaled image quality is reduced. This solution is later called mode *di1*.

Another possibility is to exchange the overlapped pixels with neighboring filters (mode *di2*). This solution requires additional communication between the filters.

The third option is to duplicate the overlapped pixels (mode *di3*). No communication is necessary between filters, but additional memory is required. Furthermore, a more complex control is needed when overlapped pixels are simultaneously directed to more than one memory module.

B. Shared Memory Architecture

All the filters utilize the same memory address space in the shared memory architecture (Fig. 5). The entire original image is stored to the shared memory, optionally in smaller pieces multiplexed in time to save memory. With this architecture, the problem with overlapped pixels does not exist because the whole image is in the same memory. A straightforward way to implement the shared memory is to utilize multiport memories (mode *sh1*). However, they are an expensive solution especially when the number of ports is large.

Parallel memories [19], [20] are another way to implement the shared memory (mode *sh2*). However, two additional permutation networks are required, one for writing and the other for reading. The data in parallel memories cannot be assigned arbitrarily, but it is accessed with the predetermined patterns, called *access formats*. Row access format is required for writing and a constant *stride* access format for reading the original image. The former format writes N adjacent pixels. In the latter format, the memory addresses are separated by

TABLE II
MEMORY CONSUMPTION IN TERMS OF PIXEL COUNT

Arch. Mode	Distributed				Shared	
	<i>di1 and di2</i>		<i>di3</i>		<i>sh1 and sh2</i>	<i>sh1 and sh2</i>
	<i>Mem_{in}</i>	<i>Mem_{out}</i>	<i>Mem_{in}</i>	<i>Mem_{out}</i>	<i>Mem_{in}</i>	<i>Mem_{out}</i>
Col. Wise	$N[c_1k]$	$N[c_2]$	$2[(c_1+p_e)k] + (N-2)[(c_1+p_m)k]$	$N[c_2]$	a_1k	a_2
Row Wise	$N[a_1d_1]$	$N[a_2d_2]$	$2[a_1d_1+a_1p_e] + (N-2)[a_1d_1+a_1p_m]$	$N[a_2d_2]$	a_1b_1	a_2b_2

TABLE III
MEMORY BANDWIDTH (PIXELS/IMAGE)

Arch. Mode	Distributed				Shared	
	<i>di1 and di2</i>		<i>di3</i>		<i>sh1 and sh2</i>	<i>sh1 and sh2</i>
	<i>Mem_{in}</i>	<i>Mem_{out}</i>	<i>Mem_{in}</i>	<i>Mem_{out}</i>	<i>Mem_{in}</i>	<i>Mem_{out}</i>
Write	c_1d_1	c_2d_2	$c_1d_1+p_{xZ_1}$	c_2d_2	a_1b_1	a_2b_2
Read	$c_2d_2k^2$	c_2d_2	$c_2d_2k^2$	c_2d_2	$a_2b_2k^2$	a_2b_2

a distance called vector stride [21]. The formats for read and write are reversed for Shared Memory_{out}.

V. MEMORY REQUIREMENTS

The complexity of the two architectures is evaluated considering the memory consumption and bandwidth. The memory consumption in terms of pixel count is tabulated in Table II. Loading pattern of the image pixels is assumed to be the normal row after row, from up to down. The distributed memory architecture has three modes, *di1*, *di2*, and *di3*. The modes of shared memory architecture are *sh1* and *sh2*. Each of the modes has input and output memory modules abbreviated as Mem_{in} and Mem_{out}. Memory consumption depends on the image division direction, row or column. The number of filters is $N \geq 2$. The parameters in formulas are as defined earlier in this paper.

With the distributed memory architecture, the parameters in square brackets denote single memory module size in pixels and the parameters outside the square brackets the memory module count. For example, when utilizing the distributed memory architecture with mode *di1* or *di2* and dividing the image column-wise, an input memory module (Mem_{in}) of size c_1k pixels is required N times. The distributed memory architecture with mode *di3* requires additional memory for the overlapped pixels. Two of the memory modules scale edge blocks of the image and the rest process middle blocks (Table II). The total memory consumption between the modes *di1*, *di2*, *sh1*, and *sh2* is equal. However, the data is distributed to several memory modules in distributed memory architecture.

The memory consumption with column-wise division is considerably less than with the row-wise method (Table II). This happens with all the five modes due to the data loading order mentioned. Modes *sh1* and *sh2* require buffering the complete original and the scaled image with the row-wise method, whereas just parts of the images are required at a time when column-wise division is used.

The required memory bandwidth is evaluated in Table III. The bandwidth is tabulated in terms of pixels per image. The write and read bandwidths are separated. Distributed memory architecture with mode *di3* requires a reading bandwidth of c_2d_2

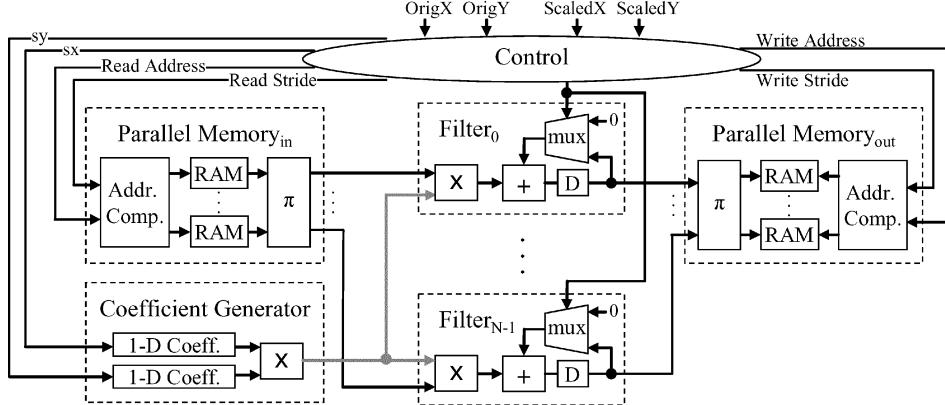


Fig. 6. Implemented image scaling architecture.

pixels/image for Mem_{out} . Overlapped pixels affect the Mem_{in} write bandwidth of mode $di3$. Variable p_x denotes p_e or p_m depending on whether an edge or middle block is stored to the current memory module. Variable z_1 denotes c_1 or d_1 depending on whether row- or column-wise division is used.

The required memory bandwidth is divided between several memory modules when the distributed memory architecture is used. Respectively, all the data is accessed from the two shared memories in the shared memory architecture. This can be seen from the bandwidth results in Table III.

The results can be improved by optimizing the memory usage. Therefore, the shown memory consumption and bandwidth values are most useful in illustrating relative differences between the separate methods.

According to the estimation shown above, column-wise division demands considerably less memory than row-wise method. However, the scaled image quality is reduced with mode $di1$ of distributed memory architecture. Additional communication between the filters is needed with mode $di2$ and mode $di3$ requires additional memory and a more complex control compared to the other solutions. Shared memory architecture with mode $sh1$ uses expensive multiport memories. As a result, shared memory architecture with parallel memories (mode $sh2$) and column-wise image division is evaluated in more detail in the following.

VI. IMPLEMENTATION CASE STUDY

The implemented architecture is depicted in Fig. 6. In the architecture, original image pixels located in Parallel Memory_{in} are scaled to Parallel Memory_{out}. Original and scaled image width and height (OrigX, OrigY, ScaledX, ScaledY) are fed as inputs. The two Parallel Memory units include an Address Computation unit, N memory modules, and a Data Permutation (π) unit. The data is read from Parallel Memory_{in} according to the given address and stride length. The Coefficient Generator includes two similar units that compute one-dimensional (1-D) coefficients for x and y -directions. The 1-D coefficients are multiplied to get the required 2-D coefficient. N Filter units operate in parallel by multiplying and accumulating original image pixels and coefficients. A delay register is denoted with ‘D’.

When the scaled pixels are ready, accumulation is reset and data is written to Parallel Memory_{out} according to Write Address and Write Stride. To save hardware resources, the number of filters and the number of memory modules is restricted to a power of two $N = 2^n$. Padding is used to add extra pixels if the original and scaled image widths are not multiples of N .

In image boundaries, undefined pixels beyond image boundaries may be required for interpolation. Those pixels are replaced by corresponding edge pixel values in the implemented architecture. In y -direction, the Control unit limits the referenced memory accesses between image height boundaries. In x -direction, edge pixel values are stored to several memory locations to extend the image width as demanded. While the image is divided column-wise and shared memory architecture with mode $sh1$ is used, that increases the input memory consumption (Mem_{in}) shown in Table II from $a_1 k$ to $(a_1 + 2\lfloor k/2 \rfloor)k$ pixels. The respective write memory bandwidth for Mem_{in} is increased from $a_1 b_1$ to $(a_1 + 2\lfloor k/2 \rfloor)b_1$ pixels/image (Table III). Normally, $k \ll a_1$, and, therefore, the increases are not significant.

According to [1], linear and nearest neighbor interpolations are the most often analyzed interpolation algorithms in recent publications. However, they also are simple ones resulting poor quality. More complex and also frequently analyzed interpolations are cubic B-splines [22] and cubic interpolation [23]. Those were the two interpolation algorithms implemented for this study. Cubic B-splines use 4×4 original image pixels to filter one scaled pixel with moderate quality. Cubic interpolation functions introduced in [1] use 2×2 , 4×4 , 6×6 , or 8×8 pixels to scale a single one. Cubic 6×6 interpolation is a compromise with good quality and moderate complexity. Lehmann *et al.* [1] recommend it for most common interpolation tasks in medical image processing. However, it is more complex than cubic B-splines. One-dimensional cubic B-splines determines scaling coefficients as follows [1], [22]

$$\text{Coeff}_1(s) = \begin{cases} \left(\frac{1}{2}\right)|s|^3 - |s|^2 + \frac{2}{3}, & 0 \leq |s| < 1 \\ -\left(\frac{1}{6}\right)|s|^3 + |s|^2 - 2|s| + \frac{4}{3}, & 1 \leq |s| \leq 2 \\ 0, & |s| > 2. \end{cases} \quad (3)$$

The distance between a scaled pixel location and an original pixel location is denoted by s . Coefficient is multiplied with

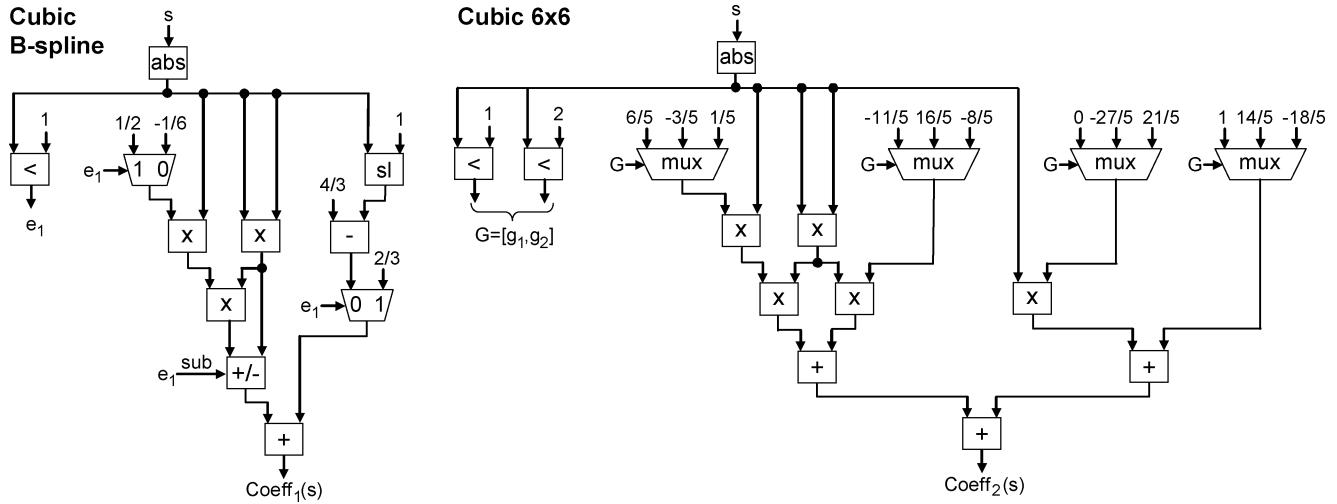


Fig. 7. One-dimensional interpolation functions.

the respective original image pixel. Cubic 6×6 interpolation function is determined as follows [1], [23]:

$$\text{Coeff}_2(s) = \begin{cases} \left(\frac{6}{5}\right)|s|^3 - \left(\frac{11}{5}\right)|s|^2 + 1, & 0 \leq |s| < 1 \\ -\left(\frac{3}{5}\right)|s|^3 + \left(\frac{16}{5}\right)|s|^2 - \left(\frac{27}{5}\right)|s| + \frac{14}{5}, & 1 \leq |s| < 2 \\ \left(\frac{1}{5}\right)|s|^3 - \left(\frac{8}{5}\right)|s|^2 + \left(\frac{21}{5}\right)|s| - \frac{18}{5}, & 2 \leq |s| \leq 3 \\ 0, & |s| > 3. \end{cases} \quad (4)$$

The detailed implementation of the 1-D cubic B-spline and cubic 6×6 algorithms is shown in Fig. 7. For cubic B-spline, the absolute value of the input is determined first. Following the comparator output, constant values are selected using multiplexers. Multiplying with two is implemented with hardwired left shifting (sl). One addition/subtract (+/-) operation uses the comparator output to determine whether addition or subtraction is utilized. Only values between $-2 \dots 2$ are given to the cubic B-spline implementation shown in Fig. 7. Respectively, only values between $-3 \dots 3$ are allowed as input for the cubic 6×6 implementation. The 1-D Coefficient unit in Fig. 6 includes either one of the unit diagrams in Fig. 7.

The distribution of data to the parallel memory modules is called a *module assignment function*, also known as a *skewing scheme*. The data elements of an access format can be read from the memory modules in parallel when a *conflict free* access is performed. The utilized module assignment function determines the access formats that can be used conflict free. As stated above, here the use of a parallel memory necessitates implementing stride access formats. The stride length used for read accesses equals the used original block size and the write stride equals the scaled block size. The block sizes depend on the original and scaled image sizes and the filter count N . While original and scaled image sizes can be arbitrary integers from an implementation specific range, the parallel memory needs control that allows conflict free accesses for all the strides.

Unfortunately, there is no single skewing scheme that fulfills the requirements when the number of memory modules equals the number of accessed data elements [19]. One solution is to use the *Configurable Parallel Memory Architecture* (CPMA)

TABLE IV
UTILIZED RESOURCES IN CUBIC B-SPLINE COEFFICIENT GENERATOR

Unit \ Res.	const						
	x	+/-	abs	mux	+/-	shift	<, >
1-D	3	2	1	2	1	1	1
2-D	7	4	2	4	2	2	2

[24] that allows several access formats and skewing schemes to be used with a single hardware when the number of memory modules is arbitrary [20]. However, a more dedicated system for changeable stride access is to use *dynamic storage schemes* that allow multiple stride specific schemes to be used within a single system [21], [25]. Nevertheless, the number of memory modules is restricted to a power of two ($N = 2^n$). The restriction also applies to the example implementation and, therefore, the dynamic storage schemes introduced in [25] are used. *XOR-schemes* based module assignment function hardware requires just n two-input XOR gates and a shift operation [25]. A skewing scheme is changed at run-time according to the required access format. However, the scheme can only be changed before writing new data to parallel memory. The data permutation network is implemented with a full crossbar.

VII. RESULTS

Using the C programming language, a simulation model corresponding to the functionality shown in Fig. 6 was constructed. The block-level parallelism was modeled with loops. That is, the image scaling was done in such order that the respective pixels in each of the image blocks were scaled consecutively. The original images, used as test input, were from [26]. They are retinal images using the RGB color model with resolution 700×605 . Each color component is encoded with eight bits. The images were scaled both down and up. The scaled images were verified against another reference C-code implementation. In the reference implementation, the pixel rows were scaled from left to right in a normal fashion. Moreover, the scaled images were visually inspected.

TABLE V
UTILIZED RESOURCES FOR UNITS CORRESPONDING TO THE ARCHITECTURE IN FIG. 6

Res.								const				gates	
Unit	x	+/-	abs	shift	<,>	=	mux	x	+/-	<,>	=	XOR	NOR
CG	11	6	2				8				4		
PM_{in}		<i>N</i>		<i>N</i>			2 <i>N</i>	<i>N</i> -2			<i>N</i> ²	<i>N</i> × <i>n</i>	<i>N</i>
PM_{out}		<i>N</i>		<i>N</i>			2 <i>N</i>	<i>N</i> -2			<i>N</i> ²	<i>N</i> × <i>n</i>	<i>N</i>
F (# <i>N</i>)	<i>N</i>	<i>N</i>						<i>N</i>					
Control	4	8			1	2	5		5	1	2		
Total	<i>N</i> +15	3 <i>N</i> +14	2	2 <i>N</i>	1	2	5 <i>N</i> +13	2 <i>N</i> -4	5	5	2 <i>N</i> ² +2	2 <i>N</i> × <i>n</i>	2 <i>N</i>

A. Hardware Complexity

All the resources required in the architecture were evaluated from the C-code implementation. As an example of the evaluation method, the resources needed in cubic B-spline Coefficient Generator unit, depicted in Fig. 7, are tabulated in Table IV. The resources (abbreviated as “Res.”) are separated into 1- and 2-D implementations. One of the inputs in some of the two-input resources can be a constant and those units are separately marked “const.” The 1-D system utilizes a total of three multipliers, two addition/subtraction units, one absolute unit, and two multiplexers. In addition, one addition/subtraction unit, one shift unit, and one comparator with the other input defined as a constant are required. The 2-D implementation requires twice the resources of the 1-D system and one extra multiplier.

Resources of the other units, corresponding to the architecture depicted in Fig. 6, are tabulated in Table V. The number of Filters is denoted by $N = 2^n$ ($N \geq 2$). The resources are separated into Coefficient Generator (CG), Parallel Memory_{in}(PM_{in}), Parallel Memory_{out}(PM_{out}), *N* Filter, and Control units. Cubic 6×6 is used as the interpolation function in Table V. Compared to Table IV, also equality comparators (=) and some basic ports (XOR, NOR) are necessary. Data direction and, therefore, control signals are different for Data Permutation units of Parallel Memory_{in} and Parallel Memory_{out}. However, equal amount of resources are required.

The number of most of the resources is directly proportional to N . The number of equality comparators with the other input being a constant is exceptionally proportional to N^2 in the Parallel Memory units. However, those comparators can be implemented in hardware using N^2 *n*-bit AND gates when the other input is a constant. The bit widths of the other resources depend on the demanded accuracy. Bit width evaluation is a complex issue by itself and, therefore, out of the scope of this study.

The resources shown in Table V do not include the logic required for storing the original image in Parallel Memory_{in} and reading the scaled image from Parallel Memory_{out}. Implementing this logic depends on the utilized handshaking method and whether single port or dual port memory is used making it very implementation specific. An additional row access format (N adjacent data elements) with word addressability is necessary for both of the parallel memories. Word addressability means that the memory can be addressed only at word boundaries when the word is defined as N data elements. The implemented module assignment function allows these requirements. Address computation and data permutation implementations do

TABLE VI
PERFORMANCE EVALUATION (FRAME/S)

<i>N</i>	fps (grayscale)	fps (RGB)	
1	5.0	1.7	ScaledX = 800
2	10.1	3.4	ScaledY = 690
4	20.1	6.7	<i>f</i> = 100 MHz
8	40.3	13.4	<i>k</i> = 6
16	80.5	26.8	

not demand such many resources as Parallel Memories shown in Table V because of the constant access format requirement.

B. Hardware Performance Estimation

The maximum frame rate (frames/second) can be computed as follows:

$$\text{Frame rate}_{\text{RGB}}^{\max} = \frac{f}{3 \cdot \text{ScaledX} \cdot \text{ScaledY}} \cdot \frac{N}{k \cdot k}. \quad (5)$$

The RGB color model is assumed and the utilized clock frequency is denoted by *f*. It is supposed that storing original image to Parallel Memory_{in} do not restrict the performance. For greyscale images, the maximal frame rate is three times bigger because the image color components need not to be processed.

This kind of system can be implemented with contemporary field-programmable logic chips, FPGAs. For example, Altera Stratix [27] and Xilinx Virtex-II [28] components contain several dual port SRAM modules and hardware multipliers. According to the preliminary logic synthesis evaluation, some results can be estimated. Utilizing moderate pipelining with the implementation of the architecture in Fig. 6, at least 100-MHz clock frequency can be achieved with Altera Stratix family. Using this clock frequency, the maximum frame rates for different parallelization options can be evaluated and they are tabulated in Table VI. Scaled image size is 800 × 690 and cubic 6×6 interpolation algorithm is used. As stated above, original image size does not affect the frame rate (5). Therefore, just scaled image size is defined in Table VI. In the case of 16 filters, the RGB image can be scaled in real-time (26.8 frames/s).

When bigger images are scaled or more complex interpolation algorithms used, more throughput can be achieved by increasing the filter count or adding pipeline stages (increased clock frequency). Extending the discussion beyond block-level parallelism, a more complex filter could be used to access several pixels of a single image block at a time. Single pixel scaling

TABLE VII
SOFTWARE PERFORMANCE IMPROVEMENT (PROPOSED/REFERENCE)

<i>N</i>	Scaled image size		Original image size = 700 × 605
	420 × 360	800 × 690	
1	0.94	0.92	
2	1.66	1.69	
5	2.25	2.22	RGB color system
10	2.89	2.92	Cubic 6 × 6 interpolation
20	4.09	4.15	

operations could then be parallelized in addition to scaling several image blocks concurrently.

C. Software Evaluation

The proposed block-level scaling can also be utilized with software implementations and using just single processor. As noted above, all the respective pixels in each of the block use the same coefficients. Therefore, N times fewer coefficients have to be calculated when N is the number of blocks in an image. However, memory accesses with the proposed implementation are not as linear as with the reference implementation. The performance of this kind of an implementation was evaluated with a 1.7-GHz Pentium 4, 1-GB RAM, and Windows XP operating system. The original image size was 700 × 605 with RGB color system [26]. The interpolation algorithm was Cubic 6 × 6 and the image was scaled down to size 420 × 360 and scaled up to size 800 × 690. Both of the software implementations were implemented with floating point arithmetic. Several iterations and averaging were used to get reliable results. The performance improvement (proposed implementation versus reference implementation) results are tabulated in Table VII. The scaled pixel values of the two implementations differed at maximum 0.0004%. The diminutive difference comes from the limited accuracy of the implementations. When the number of utilized blocks N is one, the proposed implementation is worse because of the unlinear memory accesses. However, the performance improvement is clear with larger block counts, exceeding the factor of four when the block count is 20. The slight code size overhead is a drawback of the method.

VIII. CONCLUSION

Image and video scaling complexity and quality varies a lot depending on the used interpolation methods. Complex algorithms with stringent real-time requirements require parallel processing. Most image sizes are divisible by some natural integer other than one, which facilitates the parallelization of the problem. This paper shows how this property can be exploited with parallel block-level implementations of scaling algorithms to achieve target performance. The proposed method is applicable to several common interpolation algorithms and different kinds of computation units performing the filtering. According to the detailed evaluation of one promising architecture, the resource counts were shown to be acceptable and good quality real-time 2-D image scaling feasible in modern FPGA implementations. In addition, the proposed method was shown to be able to increase the performance of a software implementation.

REFERENCES

- [1] T. M. Lehmann, C. Gönner, and K. Spitzer, "Survey: interpolation methods in medical image processing," *IEEE Trans. Med. Imag.*, vol. 18, no. 11, pp. 1049–1075, Nov. 1999.
- [2] F. Laliberté, L. Gagnon, and Y. Sheng, "Registration and fusion of retinal images—an evaluation study," *IEEE Trans. Med. Imag.*, vol. 22, no. 5, pp. 661–673, May 2003.
- [3] R. Shekhar and V. Zagrodsky, "Mutual information-based rigid and non-rigid registration of ultrasound volumes," *IEEE Trans. Med. Imag.*, vol. 21, no. 1, pp. 9–22, Jan. 2002.
- [4] P. N. Morgan, R. J. Iannuzzelli, F. H. Epstein, and R. S. Balaban, "Real-time cardiac MRI using DSP's," *IEEE Trans. Med. Imag.*, vol. 18, no. 7, pp. 649–653, Jul. 1999.
- [5] V. A. Kovalev, F. Kruggel, H.-J. Gertz, and D. Y. von Cramon, "Three-dimensional texture analysis of MRI brain datasets," *IEEE Trans. Med. Imag.*, vol. 20, no. 5, pp. 424–433, May 2001.
- [6] W. Birkfellner *et al.*, "A head-mounted operating binocular for augmented reality visualization in medicine—design and initial evaluation," *IEEE Trans. Med. Imag.*, vol. 21, no. 8, pp. 991–997, Aug. 2002.
- [7] J. A. Parker, R. V. Kenyon, and D. E. Troxel, "Comparison of interpolating methods for image resampling," *IEEE Trans. Med. Imag.*, vol. MI-2, no. 1, pp. 31–39, Mar. 1983.
- [8] E. Maeland, "On the comparison of interpolation methods," *IEEE Trans. Med. Imag.*, vol. 7, no. 3, pp. 213–217, Sep. 1988.
- [9] J. Staal, M. D. Abràmoff, M. Niemeijer, M. A. Viergever, and B. van Ginneken, "Ridge-based vessel segmentation in color images of the retina," *IEEE Trans. Med. Imag.*, vol. 23, no. 4, pp. 501–509, Apr. 2004.
- [10] J. Zhong, R. Ning, and D. Conover, "Image denoising based on multiscale singularity detection for cone beam CT breast imaging," *IEEE Trans. Med. Imag.*, vol. 23, no. 6, pp. 696–703, Jun. 2004.
- [11] J. Lowell *et al.*, "Measurement of retinal vessel widths from fundus images based on 2-D modeling," *IEEE Trans. Med. Imag.*, vol. 23, no. 10, pp. 1196–1204, Oct. 2004.
- [12] Y. V. Haeghen, J. M. A. D. Naeyaert, I. Lemahieu, and W. Philips, "An imaging system with calibrated color image acquisition for use in dermatology," *IEEE Trans. Med. Imag.*, vol. 19, no. 7, pp. 722–730, Jul. 2000.
- [13] A. Ramaswamy, Y. Nijim, and W. B. Mikhael, "Polyphase implementation of a video scalar," in *Conf. Rec. Asilomar Conf. Signals, Systems Computers*, Pacific Grove, CA, Nov. 1997, pp. 1691–1694.
- [14] R. Dugad and N. Ahuja, "A fast scheme for image size change in the compressed domain," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 11, no. 4, pp. 461–474, Apr. 2001.
- [15] C.-H. Kim, S.-M. Seong, J.-A. Lee, and L.-S. Kim, "Winscale: an image-scaling algorithm using an area pixel model," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 13, no. 6, pp. 549–553, Jun. 2003.
- [16] F. Tao, X. Wen-Lu, and Y. Lian-Xing, "An architecture and implementation of image scaling conversion," in *Proc. Int. Conf. ASIC*, Shanghai, China, Oct. 2001, pp. 409–410.
- [17] R. Li and S. Levi, "An arbitrary ratio resizer for MPEG applications," *IEEE Trans. Consum. Electron.*, vol. 46, no. 3, pp. 467–473, Aug. 2000.
- [18] E. Aho, J. Vanne, T. D. Hääläinen, and K. Kuusilinna, "Block-level parallel processing for scaling evenly divisible frames," in *Proc. IEEE Int. Symp. Circuits Syst.*, Kobe, Japan, May 2005, pp. 1134–1137.
- [19] P. Budnik and D. J. Kuck, "The organization and use of parallel memories," *IEEE Trans. Comput.*, vol. C-20, no. 12, pp. 1566–1569, Dec. 1971.
- [20] E. Aho, J. Vanne, K. Kuusilinna, and T. D. Hääläinen, "Address computation in configurable parallel memory architecture," *IEICE Trans. Inf. Syst.*, vol. E87-D, no. 7, pp. 1674–1681, Jul. 2004.
- [21] D. T. Harper III and D. A. Linebarger, "Conflict-free vector access using a dynamic storage scheme," *IEEE Trans. Comput.*, vol. 40, no. 3, pp. 276–283, Mar. 1991.
- [22] H. S. Hou and H. C. Andrews, "Cubic splines for image interpolation and digital filtering," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. ASSP-26, no. 6, pp. 508–517, Dec. 1978.
- [23] R. G. Keys, "Cubic convolution interpolation for digital image processing," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. ASSP-29, no. 6, pp. 1153–1160, Dec. 1981.
- [24] K. Kuusilinna, J. Tanskanen, T. Hääläinen, J. Niittylahti, and J. Saarinen, "Configurable parallel memory architecture for multimedia computers," *J. Syst. Architecture*, vol. 47, no. 14–15, pp. 1089–1115, Aug. 2002.
- [25] D. T. Harper III, "Increased memory performance during vector accesses through the use of linear address transformations," *IEEE Trans. Comput.*, vol. 41, no. 2, pp. 227–230, Feb. 1992.

- [26] A. Hoover and M. Goldbaum, "Locating the optic nerve in a retinal image using the fuzzy convergence of the blood vessels," *IEEE Trans. Med. Imag.*, vol. 22, no. 8, pp. 951–958, Aug. 2003.
- [27] Stratix Datasheet, Altera. [Online]. Available: www.altera.com
- [28] Virtex-II Datasheet, Xilinx. [Online]. Available: www.xilinx.com



Eero Aho received the M.S. degree in electrical engineering from the Tampere University of Technology (TUT), Tampere, Finland, in 2001. Currently, he is working toward the Ph.D. degree as a Research Scientist with the Institute of Digital and Computer Systems, TUT.

His research interests include memory systems and parallel processing.



Jarno Vanne received the M.S. degree in information technology from the Tampere University of Technology (TUT), Tampere, Finland, in 2002. Currently, he is working toward the Ph.D. degree as a Research Scientist with the Institute of Digital and Computer Systems, TUT.

His research interests include memory systems and hardware accelerators.



Timo D. Hääläinen received the M.Sc. and Ph.D. degrees in 1993 and 1997, respectively, from the Tampere University of Technology (TUT), Tampere, Finland.

He was a Senior Research Scientist with the Institute of Digital and Computer Systems (DCS), TUT, during 1997–2001, and during that time, he founded a research group named "Parallel DSP Processing and Wireless Multimedia Systems." He has acted as a project manager and research supervisor for several academic and industrial projects. In 2001, he became Full Professor at TUT and continues research on wireless local and personal area networking, as well as SoC solutions for wireless video.



Kimmo Kuusilinna received the Ph.D. degree in 2001 from the Tampere University of Technology, Tampere, Finland.

Currently, he is working as a Senior Research Engineer with the Nokia Research Center, Tampere. His main research interests include system-level design and verification, interconnection networks, and parallel memories.

Publication 8

E. Aho, J. Vanne, and T. D. Hämäläinen, “Parallel Memory Architecture for Arbitrary Stride Accesses,” in *Proceedings of the IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, Prague, Czech Republic, pp. 65–70, April 2006.

Copyright © The Institute of Electrical and Electronics Engineers (IEEE).
Reprinted with permission.

Copyright© 2006 IEEE. Reprinted from Proceedings of the IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS) 2006.□□□□

This material is posted here with permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any of Tampere University of Technology's products or services. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org.□□□□

By choosing to view this document, you agree to all provisions of the copyright laws protecting it.□□

Parallel Memory Architecture for Arbitrary Stride Accesses

Eero Aho, Jarno Vanne, and Timo D. Hämäläinen

Institute of Digital and Computer Systems

Tampere University of Technology

Tampere, Finland

{eero.aho, jarno.vanne, timo.d.hamalainen}@tut.fi

Abstract—Parallel memory modules can be used to increase memory bandwidth and feed a processor with only necessary data. Arbitrary stride access capability with interleaved memories is described in previous research where the skewing scheme is changed at run time according to the currently used stride. This paper presents the improved schemes which are adapted to parallel memories. The proposed novel parallel memory architecture allows conflict free accesses with all the constant strides which has not been possible in prior application specific parallel memories. Moreover, the possible access locations are unrestricted and the data patterns have equal amount of accessed data elements as the number of memory modules. The complexity is evaluated with resource counts.

I. INTRODUCTION

The processor versus memory performance gap has been widening for a number of years. Cache memories can increase system memory bandwidth in general purpose systems, but they are not typically used in digital signal processing applications due to latency and real-time restrictions. Wide memories allow accessing large blocks of data. Unfortunately, they may also produce data that the currently running algorithm does not use. Multiport memories enable several simultaneous memory accesses but they are expensive solutions especially when the number of ports is large.

Parallel memory modules can be used to access special data patterns and feed the processors with only algorithm specific data [1], [2]. In specific data patterns, accessed data elements are separated by a distance called a *stride*. Many applications in the fields of digital signal processing and telecommunications benefit from the use of strides. Vector/matrix computation, fast Fourier transform (FFT), and Viterbi algorithm are some examples [3], [4].

In this study, *interleaved memories* use time-multiplexed parallel memory modules that receive access requests serially one by one. Respectively, *parallel memories* are defined as space-multiplexed memories that are used e.g. in SIMD processing. Parallel memories have wide address and data buses and the memory modules receive access requests in parallel.

Traditionally, the research on stride accesses has concentrated on *module assignment functions (skewing schemes)* that try to ensure conflict free parallel data accesses to a set or maximal amount of access strides [3], [5]. Normally, the scheme can be changed only at design time. This paper considers merely constant stride accesses and data patterns having equal amount of accessed data elements as the number of memory modules. Moreover, access locations of the data patterns are supposed to be unrestricted. Unfortunately, no single skewing scheme has been found that allows unrestrictedly placed conflict free accesses for all the constant strides when the number of memory modules is power of two [6]. The generalized proof is given later in this paper.

One solution is to use the *Configurable Parallel Memory Architecture (CPMA)* that allows several data *access formats (templates)* and skewing schemes to be used with a single hardware when the number of memory modules is arbitrary [4]. However, a more dedicated system for changeable constant stride access utilizes *dynamic storage schemes* that allow multiple stride specific schemes to be used within a single system [6], [7]. Nevertheless, the memory module count is restricted to a power of two and only interleaved memory system is considered in [6] and [7].

Arbitrary stride accessibility with parallel memories is demanded in real-time image scaling architecture shown in [8]. However, the main aspect in [8] is a new image scaling parallelization method and the implementation utilizing it. The parallel memory theory and implementation details are not discussed.

This paper presents a novel parallel memory architecture allowing conflict free arbitrary constant stride accesses. For hardware simplicity, the number of memory modules is restricted to a power of two ($N = 2^n$). From the hardware point of view, the skewing schemes are simplified from the ones in [7]. A skewing scheme is changed at run time according to the used stride. However, when a skewing scheme is changed, the data locating already in the memory modules is naturally invalidated. This is not a problem, for example, in the image scaling system in [8]. A new stride with a possible new scheme is demanded only when new input image data is written to parallel memories.

Background for the used conflict free skewing schemes is given in Section II. The proposed parallel memory architecture is described in detail in Section III. Complexity is evaluated in Section IV and Section V concludes the paper.

II. BACKGROUND AND THEORY

The address space in parallel memories cannot be assigned arbitrarily but the data has to be referenced using predetermined patterns, called access formats or templates. The distribution of data to the memory modules is called a module assignment function S that is also known as a skewing scheme. The used module assignment function determines access formats that can be used conflict free. A data element at location i is assigned to a memory module according to $S(i)$. An address function $a(i)$ (also called row number) determines the physical address in a memory module for a data element.

Theorem 1: No single skewing scheme can be found that allows conflict free accesses for all the constant strides when the access format can be unrestrictedly placed. This theorem is valid with arbitrary number of memory modules and when at least two data elements are accessed in parallel.

Proof: Let $S(i)$ be the used module assignment function and i_1 the first accessed data location. Therefore, the second accessed data location $i_2 = i_1 + \text{stride}$. The two elements i_1 and i_2 cannot be accessed conflict free with a stride that is selected such that $S(i_1) = S(i_2)$.

Theorem 2: Let σ be relatively prime to 2 (i.e. σ is an odd positive integer), s a nonnegative integer, and the number of memory modules $N = 2^n$. Any skewing scheme that allows a conflict free access for a stride 2^s also provides a conflict free access for any stride defined as

$$\text{stride} = \sigma \cdot 2^s. \quad (1)$$

Theorem 2 is proved in [6]. All the strides can be formed by changing σ and s . According to Theorem 2, a stride with a particular s and any σ values can be used conflict free in a single skewing scheme. Therefore, by finding conflict free storage schemes for each of the values of s (i.e. single scheme for a single s) enables conflict free access for arbitrary strides.

The conventional low-order interleaved scheme utilizes the module assignment function

$$S(i) = i \bmod N, \quad (2)$$

and the address function

$$a(i) = \lfloor i / N \rfloor. \quad (3)$$

The n least significant bits (LSBs) of the data location i are used as module number and the rest of the bits as an address

in a module. All the odd strides can be accessed with the low-order interleaved scheme [5]. The same can be proved with Theorem 2. It is trivial to see that stride $2^0 = 1$ is always conflict free in the low-order interleaved scheme. Therefore, according to Theorem 2, also all the other odd strides are conflict free.

Dynamic schemes for arbitrary stride accesses were first introduced by Harper *et al.* in [6]. Row rotation schemes [9] with $N = 2^n$ was used. XOR-schemes allow simpler computation since merely logic gates are demanded and no carry bits are used for computation [1], [10]. XOR-schemes always require power of two memory module count. Another Harper's approach concerning arbitrary stride accesses utilizes XOR-schemes [7].

In this study, XOR-schemes and $N = 2^n$ memory modules are used. The definitions here differ a slightly from that in [7]. In a parallel memory architecture with 2^l data locations, i and z are l -bit vectors and T is an $l \times l$ transformation matrix containing binary numbers. Vector z is partitioned into two fields: the module number $S(i)$ is defined with n least significant bits and the address in the memory module $a(i)$ with the remaining bits. The vector z is defined as follows:

$$z = i \cdot T. \quad (4)$$

The elements of the matrix T are marked in a custom fashion. The top left element of the matrix is $t_{l-1,l-1}$ and the bottom right is $t_{0,0}$. The vector addition and multiplication in (4) are achieved by bit-wise logical operations XOR and AND, respectively. The matrix T is identity matrix having also some off-diagonal 1-elements. Let $T_{y,x}$ represent an elementary transformation with an additional 1-element in a matrix position (y, x) . Several off-diagonal 1-elements can be represented by a product of elementary transformations. For example,

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = T_{1,0} \cdot T_{2,1}. \quad (5)$$

In [7], T is defined as

$$T = \prod_{k=0}^{\min(n,s)-1} T_{\max(n,s)+k,k}, \quad (6)$$

which ensures conflict free access for any stride. The proposed transformation simplifies (6) by defining

$$T = \prod_{k=0}^{n-1} T_{s+k,k}. \quad (7)$$

The proposed transformation (7) equals to (6) when $s \geq n$.

TABLE I
EXAMPLE OF THE PROPOSED SKEWING SCHEME ($n = 2$, $s = 1$)

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$S(i)$	0	1	3	2	2	3	1	0	0	1	3	2	2	3	1	0
$a(i)$	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3	3

An example of the proposed skewing scheme is tabulated in Table I where $n = 2$ and $s = 1$. The memory module count $N = 2^n = 4$ and the address depth in a memory module is restricted to four in Table I. Sixteen data elements are displayed. For example, a data element at location $i = 5$ is stored in memory module $S(i) = 3$ under address $a(i) = 1$. The proposed transformation matrix T (7) in Table I is defined as (5). With $s = 1$ in (1), strides of 2, 6, 10, 14, ... are conflict free in arbitrary location. For example, an access format with stride = 2 at location $i = 1$ (painted gray) is conflict free since all the four accessed data elements are located in different memory modules.

A. Hardware Implementation

All the off-diagonal 1-elements of the matrix T (6) locate in LSB side. Therefore, the address in a memory module $a(i)$ is received straight from the $l\text{-}n$ MSBs of the data element location i and no additional hardware is demanded [7]. This address function is defined in (3). The same situation holds for the proposed transformation (7).

The module number $S(i)$ calculation is pretty simple because just at most two elements of any column of T (6) are nonzero [7]. Therefore, the module number can be calculated as follows:

$$S(i) = i[n + s - 1 : \max(n, s)] \oplus i[n - 1 : 0], \quad (8)$$

where a data element location i is stated in binary form [MSB : LSB] and \oplus denotes bit-wise logical XOR operation.

The proposed transformation matrix T (7) also has at most two nonzero elements in a single column. This leads the following simple implementation:

$$S(i) = i[n + s - 1 : s] \oplus i[n - 1 : 0]. \quad (9)$$

The implementation of (8) and (9) demands defining n and s . In a particular implementation, the number of memory modules $N = 2^n$ is a constant. Respectively, s is determined by a stride (1). Therefore, n is defined at design time and s at run time.

With $n = 2$ and $s = 1$ (Table I), the proposed module assignment function $S(i)$ (9) and address function $a(i)$ are defined as

$$\begin{aligned} S(i) &= i[2 : 1] \oplus i[1 : 0], \\ a(i) &= i[3 : 2]. \end{aligned}$$

The hardware implementations of [6], [7], and [11] as well as (9) do not function properly when odd strides are used ($s = 0$). As mentioned above, all the odd strides can be accessed with a conventional low-order interleaved scheme (2). In the proposed implementation, the odd strides are separately taken into account. When $s = 0$, low-order interleaved scheme is formed as follows:

$$S(i) = 0 \oplus i[n - 1 : 0] = i[n - 1 : 0]. \quad (10)$$

B. Discussion

As stated above, the proposed transformation T (7) equals to (6) when $s \geq n$. In the other even stride lengths, the proposed skewing schemes (9) have been verified with Matlab software simulations. The memory organizations under simulation had power of two number of memory modules $N = 2, 4, 8, \dots, 1024 | n \in \{1, \dots, 10\}$. All the possible lengths of strides were verified to be conflict free when $s \in [1, n-1]$. Respectively, all the odd strides ($s = 0$) are managed with (10). Therefore, it is stated that the proposed skewing schemes (9) and (10) provide conflict free access for all the constant strides with practical number of memory modules.

A row access format (N adjacent data elements with stride 1) is commonly used. Word addressability means that the memory can be addressed only at word boundaries when the word is defined as N data elements. The proposed module assignment function $S(i)$ (9) allows conflict free word addressable row accesses with all the values of n and s . This property is also available with (8) unless not considered in [7]. For example with $N = 4$ (Table I), data locations $i = 8, 9, 10, 11$ or $i = 12, 13, 14, 15$ can be accessed conflict free. However, locations $i = 9, 10, 11, 12$ are not conflict free because data elements at $i = 11$ and $i = 12$ are located in the same memory module 2. Therefore, when a row access format is demanded to reference arbitrarily in the memory, not only according to word boundaries, the skewing scheme needs to be modified accordingly.

The differences between the proposed implementation and the Harper's implementations [6], [7], and [11] are discussed in the following. The proposed implementation use parallel memories instead of interleaved memory. In parallel memories, each of the N addresses $a(i)$ of the data locations i is assigned to the correct memory module $S(i)$ simultaneously. Therefore, a lot of parallel logic is demanded. Moreover, address and data permutation networks are required around the memory modules.

The proposed schemes are optimized from Harper's ones [7], [11] in terms of hardware complexity. The hardware in (8) includes n two-input XOR gates, a comparator (max determination), two multiplexers, a mask generator and a

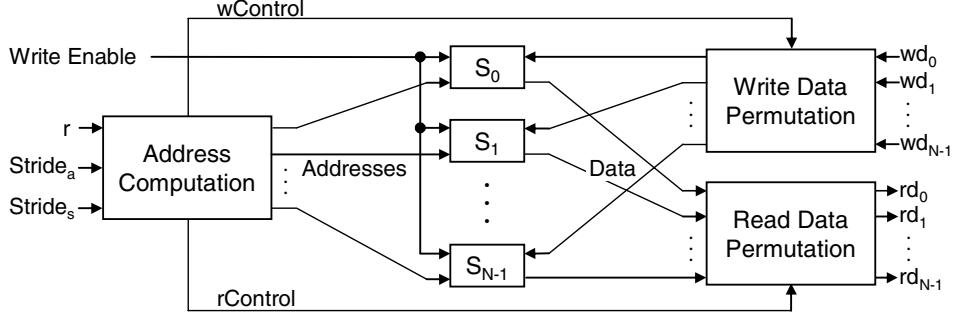


Fig. 1. Proposed parallel memory architecture.

shifter with a capability of bit masking [7], [11]. Respectively, the proposed implementation (9) demands only n two-input XOR gates and a shifter. Moreover, the proposed hardware implementation determining s according to a stride differs from Harper's one.

III. PROPOSED PARALLEL MEMORY ARCHITECTURE

A block diagram of the proposed parallel memory architecture is shown in Fig. 1. The referenced data locations are defined with the *first element location (scanning point)* r and the stride length $Stride_a$. The skewing scheme is formed according to the $Stride_s$ input. In write operation, *Write Enable* input is asserted simultaneously with the data to be written $wd_0, wd_1, \dots, wd_{N-1}$. The two *Data Permutation* units permute the written or read data according to the control signals from the *Address Computation* unit.

As mentioned above, a word addressable row access format (stride = 1) is conflict free in all the used skewing schemes but the scheme can only be changed before writing a new data to memory. Therefore, the access stride ($Stride_a$) and the stride defining the used skewing scheme ($Stride_s$) are

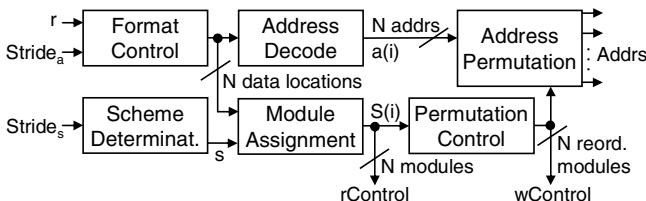


Fig. 2. Address Computation unit.

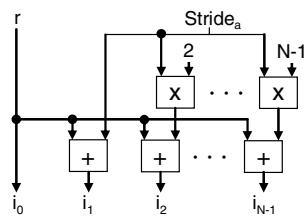


Fig. 3. Format Control unit.

separated to allow controlling the access stride and skewing scheme separately. For example, the data storage in Table I can be referenced with a row access format without changing the skewing scheme by assigning $Stride_a = 1$ and $Stride_s = 4$. However, the row access format can be addressed only at word boundaries.

The proposed parallel memory system is implemented in VHDL. The system is designed to be configurable both at design time and at run time. In the hardware implementation, most parameters are VHDL generics, which means that the memory module count ($N = 2^n$), size of a memory module, and width of buses are adjustable. The run-time configurability refers to the ability to change the skewing scheme at run time.

Fig. 2 depicts a block diagram of the Address Computation unit. The outputs of the unit are the memory module addresses for each of the memory module (*Addrs*). Moreover, control signals (*rControl*, *wControl*) are formed for the Data Permutation units.

Inside the unit, the *Format Control* unit specifies the accessed data locations. The *Scheme Determination* unit defines the used skewing scheme. The *Module Assignment* unit computes the specific memory modules that contain the accessed data and the *Address Decode* unit, in turn, computes the physical addresses for each memory module. The *Permutation Control* unit reorders the memory module numbers. According to these values, the *Address Permutation* unit directs the physical addresses to the proper memory modules.

A detailed block diagram of the Format Control unit is shown in Fig. 3. The stride is multiplied with constant numbers $2, \dots, N-1$ and the referenced data locations i_0, i_1, \dots, i_{N-1} are formed by additions.

Fig. 4 depicts the Scheme Determination unit. The input to the unit is the stride defining the used skewing scheme $Stride_s = Str[d-1 : 0]$. The relation between a stride and s is given in (1). The output s is constructed by determining the count of adjacent zeros before first '1' by starting from LSB. For example, with $Stride_s = 12_{10} = 01100_2$, the output $s = 2_{10}$. Another input of the equality comparators is a constant zero but the width of the other input is different in each of

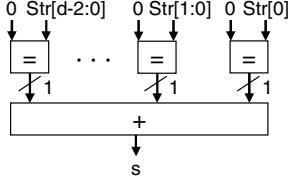


Fig. 4. Scheme Determination unit.

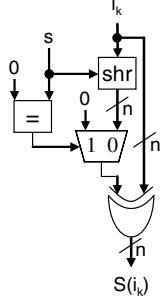


Fig. 5. Module Assignment unit (1/N part of the unit).

the comparators. The output of the equality comparators is one bit wide. Those one bit wide results are summed up to get the output s . Since the $\text{Stride}_s \geq 1$ (i.e. stride cannot be 0), the most significant bit $\text{Str}[d-1]$ is not actually demanded.

Fig. 5 shows a block diagram of the Module Assignment unit. In fact, there are N parallel similar logic blocks in the Module Assignment unit. The input s from the Scheme Determination unit defines the used scheme. When $s = 0$, low-order interleaved scheme is used as shown in (10). Zero value is selected with a multiplexer and the n LSBs of the data location i_k is used to form $S(i_k)$ when $k \in [0, N-1]$. Otherwise, the data location i_k is shifted right (shr) according to s and, after that, n parallel bit-wise XOR operations are performed as shown in (9).

The Address Decode unit is presented in Fig. 6. The unit implements the address function (3) and contains no logic, only hardwired shifting. The MSBs of a data location i_k are selected as address $a(i_k)$ output.

The Permutation Control unit is shown in Fig. 7. The unit changes the order of the module numbers received from the Module Assignment unit. The comparators are used to find the memory module number $S(i_b)$ ($b \in [0, N-1]$) that equals the constant k , $k \in [0, N-1]$. The respective index b of the data element i_b is forwarded to output $S'(i_k)$. As an example with $N = 4$ and with the memory module numbers $\{S(i_0), S(i_1), S(i_2), S(i_3)\} = \{1, 2, 3, 0\}$, the reordered module number are $\{S'(i_0), S'(i_1), S'(i_2), S'(i_3)\} = \{3, 0, 1, 2\}$.

The parallel memory system includes three Permutation units. One of the units is depicted in Fig. 8. Full crossbar permutation networks are used. Memory module numbers in original or reordered way are used as control signals ($\text{Ctrl}_0, \text{Ctrl}_1, \dots, \text{Ctrl}_{N-1}$). In the Write and Read Data Permutation

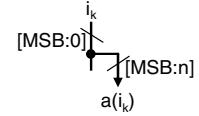


Fig. 6. Address Decode unit (1/N part of the unit).

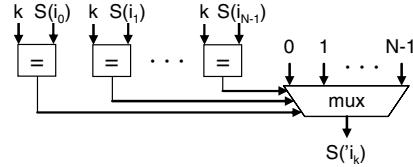


Fig. 7. Permutation Control unit (1/N part of the unit).

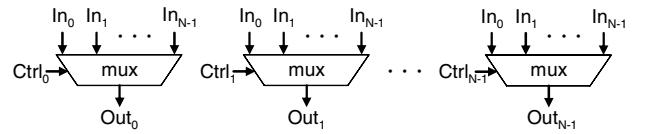


Fig. 8. Permutation unit.

units, the inputs ($\text{In}_0, \text{In}_1, \dots, \text{In}_{N-1}$) and outputs ($\text{Out}_0, \text{Out}_1, \dots, \text{Out}_{N-1}$) are data buses. Respectively, addresses are permuted in the Address Permutation unit.

IV. COMPLEXITY

All of the resources required in the sub-blocks of the proposed parallel memory system are tabulated in Table II. The resources are separated into Format Control (FC), Scheme Determination (SD), Address Decode (AD), Module Assignment (MA), Permutation Control (PC), and three Permutation units (3xP). The number of memory modules is denoted by $N = 2^n$ ($N \geq 2$) and the width of the Stride_s input by d (Fig. 4).

Except for multiplexers, just two inputs are supposed in all the resources. Multiplexers use an additional control signal to select one of the two inputs. The bit widths of the inputs depend on the design-time configurations, for example, the used memory module width and depth. In the Scheme Determination unit (Fig. 4), the utilized adder is multi-input but the multiple inputs are just one bit wide. Therefore, it is marked as a single adder.

One of the inputs in some of the two-input resources can be a constant and those units are separately marked ‘const’. With multiplexers, one or two of the three inputs can be constant.

The Format Control unit utilizes $N-1$ adders and $N-2$ multipliers with the other input defined as a constant. Respectively, the Module Assignment unit demands N shifters and $N \times n$ 2-bit XOR gates. In addition, N multiplexers and N equality comparators with some constant inputs are required.

TABLE II
UTILIZED RESOURCES IN THE PARALLEL MEMORY SYSTEM

	$+$ / $-$	shift	mux	x	const mux	=	gates XOR
FC				$N-1$	$N-2$		
SD				1		$d-1$	
AD							
MA				N	N	N	Nxn
PC					$N(N-1)$	NxN	
3xP				$3N(N-1)$			
All		N	N	$3N(N-1)$	$N-2$	NxN	$N(N+1)+d-1$
							Nxn

Adder, shifter, and multiplier amounts are linearly proportional to N . The number of multiplexers and equality comparators are proportional to N^2 in the Permutation Control unit. However, while the other comparator input is a constant, the comparators can be implemented in hardware using N^2 AND ports with some inverted input bits. Moreover, two of the three inputs are constants in the multiplexers of the Permutation Control unit (Fig. 7). When increasing the number of memory modules, the most significant area increase is with the Permutation units, where the multiplexer amount is proportional to N^2 . As a case study, full crossbar permutation networks dominate the logic area in a video coding specific parallel memory system with memory module count four or more [12].

The two Data Permutation units could be combined in the proposed system. The inputs of the combined Data Permutation unit would be selected with “Write Enable” signal between the current inputs of the Read or Write Data Permutation units (Fig. 1). The output of the combined unit would be directed to the inputs of the memory modules and the output of the whole parallel memory system. The number of additional multiplexers would be $2N$ instead of another Data Permutation amount $Nx(N-1)$. This would decrease the resource count when $N > 2$. As a drawback, the delay of the combined Data Permutation unit would increase due to additional multiplexers. Moreover, write after read turnaround time of the proposed parallel memory system would also increase. Another or additional possibility to decrease the demanded logic is to utilize multistage interconnection networks like Benes [13].

V. CONCLUSION

Stride accesses are frequently required in the fields of digital signal processing and telecommunications. In this paper, we have presented a novel parallel memory architecture allowing all the constant stride accesses which has not been possible in prior application specific parallel memory systems. The proposed architecture is mainly useful in SIMD type of processing.

REFERENCES

- [1] S. Chen, A. Postula, and L. Jozwiak, “Synthesis of XOR storage schemes with different cost for minimization of memory contention,” in *Proc. Euromicro Conf.*, Milan, Italy, pp. 170–177, Sep. 1999.
- [2] R. Hartenstein, J. Becker, M. Herz, and U. Nageldinger, “An embedded accelerator for real world computing,” in *Proc. IFIP Int'l Conf. Very Large Scale Integration*, Gramado, Brazil, Aug. 1997.
- [3] J. Takala and T. Järvinen, “Stride permutation access in interleaved memory systems,” in *Domain-Specific Multiprocessors – Systems, Architectures, Modeling, and Simulation*, ed. S. S. Bhattacharyya, E. F. Deprettere, and J. Teich, pp. 63–84, Marcel Dekker, New York, NY, USA, 2004.
- [4] E. Aho, J. Vanne, K. Kuusilinna, and T. D. Hämäläinen, “Address computation in configurable parallel memory architecture,” *IEICE Trans. Inf. & Syst.*, vol. E87-D, no. 7, pp. 1674–1681, July 2004.
- [5] M. Valero, T. Lang, M. Peiron, and E. Ayguadé, “Conflict-free access for streams in multimodule memories,” *IEEE Trans. Computers*, vol. 44, no. 5, pp. 634–646, May 1995.
- [6] D. T. Harper III and D. A. Linebarger, “Conflict-free vector access using a dynamic storage scheme,” *IEEE Trans. Comput.*, vol. 40, no. 3, pp. 276–283, Mar. 1991.
- [7] D. T. Harper III, “Increased memory performance during vector accesses through the use of linear address transformations,” *IEEE Trans. Comput.*, vol. 41, no. 2, pp. 227–230, Feb. 1992.
- [8] E. Aho, J. Vanne, T. D. Hämäläinen, and K. Kuusilinna, “Block-level parallel processing for scaling evenly divisible images,” *IEEE Trans. Circuits Syst. I*, vol. 52, no 12, pp. 2717–2725, Dec. 2005.
- [9] P. Budnik and D. J. Kuck, “The organization and use of parallel memories,” *IEEE Trans. Comp.*, vol. C-20, no. 12, pp. 1566–1569, Dec. 1971.
- [10] J. M. Frailong, W. Jalby, and J. Lenfant, “XOR-schemes: A flexible data organization in parallel memories,” in *Proc. Int'l Conf. Parallel Processing*, Washington, DC, USA, pp. 276–283, Aug. 1985.
- [11] D. T. Harper III and D. A. Linebarger, “Dynamic address mapping for conflict-free vector access,” U.S. Patent 4 918 600, Apr. 17, 1990.
- [12] J. K. Tanskanen and J. T. Niittylahti, “Scalable parallel memory architectures for video coding,” *Journal of VLSI Signal Processing*, vol. 38, no. 2, pp. 173–199, Sep. 2004.
- [13] S. Dutta, W. Wolf, and A. Wolfe, “A methodology to evaluate memory architecture design tradeoffs for video signal processors,” *IEEE Trans. Circuits Syst. Video Technol.*, vol. 8, no. 1, pp. 36–53, Feb. 1998.

Publication 9

E. Aho, J. Vanne, and T. D. Hämäläinen, “Parallel Memory Implementation for Arbitrary Stride Accesses,” in *Proceedings of the Embedded Computer Systems: Architectures, Modeling, and Simulation Conference (IC-SAMOS)*, Samos, Greece, pp. 1–6, July 2006.

Copyright © The Institute of Electrical and Electronics Engineers (IEEE).
Reprinted with permission.

Copyright© 2006 IEEE. Reprinted from Proceedings of the Embedded Computer Systems: Architectures, Modeling, and Simulation Conference (IC-SAMOS) 2006.□□□□

This material is posted here with permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any of Tampere University of Technology's products or services. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org.□□□□

By choosing to view this document, you agree to all provisions of the copyright laws protecting it.□□

Parallel Memory Implementation for Arbitrary Stride Accesses

Eero Aho, Jarno Vanne, and Timo D. Hämäläinen

Institute of Digital and Computer Systems, Tampere University of Technology

Tampere, Finland

{eero.aho, jarno.vanne, timo.d.hamalainen}@tut.fi

Abstract—Parallel memory modules can be used to increase memory bandwidth and feed a processor with only necessary data. Arbitrary stride access capability with interleaved memories is described in previous research where the skewing scheme is changed at run time according to the currently used stride. This paper presents the improved schemes which are adapted to parallel memories. The proposed novel parallel memory implementation allows conflict free accesses with all the constant strides which has not been possible in prior application specific parallel memories. Moreover, the possible access locations are unrestricted and the data patterns have equal amount of accessed data elements as the number of memory modules. Timing and area estimates are given for Altera Stratix FPGA and 0.18 micrometer CMOS process with memory module count from 2 to 32. The FPGA results show 129 MHz clock frequency for a system with 16 memory modules when read and write latencies are 3 and 2 clock cycles, respectively.

I. INTRODUCTION

The processor versus memory performance gap has been widening for a number of years. Cache memories can increase system memory bandwidth in general purpose systems, but they are not typically used in digital signal processing applications due to latency and real-time restrictions. Wide memories allow accessing large blocks of data. Unfortunately, they may also produce data that the currently running algorithm does not use. Multiport memories enable several simultaneous memory accesses but they are expensive solutions especially when the number of ports is large.

Parallel memory modules can be used to access special data patterns and feed the processors with only algorithm specific data [1], [2]. In particular data patterns, accessed data elements are separated by a distance called *stride*. Many applications in the fields of digital signal processing and telecommunications benefit from the use of strides. Vector/matrix computation, fast Fourier transform (FFT), and Viterbi algorithm are some examples [3], [4].

In this study, *interleaved memories* use time-multiplexed parallel memory modules that receive access requests serially one by one. Respectively, *parallel memories* are defined as space-multiplexed memories that are used e.g. in SIMD processing. Parallel memories have wide address and data buses and the memory modules receive access requests in parallel.

Traditionally, the research on stride accesses has concentrated on *module assignment functions (skewing schemes)* that try to ensure conflict free parallel data accesses to a set or maximal amount of access strides [3], [5]. Normally, the scheme can be changed only at design time. This paper considers merely constant stride accesses and data patterns having equal amount of accessed data elements as the number of memory modules. Moreover, access locations of the data patterns are supposed to be unrestricted. Unfortunately, no single skewing scheme can be found that allows unrestrictedly placed conflict free accesses for all the constant strides [6].

One solution is to use the *Configurable Parallel Memory Architecture (CPMA)* that allows several data access formats (*templates*) and skewing schemes to be used with a single hardware when the number of memory modules is arbitrary [4]. However, CPMA is relatively complex architecture to use only for stride accesses. A more dedicated memory system for changeable constant stride access utilizes *dynamic storage schemes* that allow multiple stride specific schemes to be used within a single system [7], [8]. Nevertheless, the memory module count is restricted to a power of two and only interleaved memory system is considered in [7] and [8].

Arbitrary stride accessibility with parallel memories is demanded in real-time image scaling architecture shown in [9]. However, the main aspect in [9] is a new image scaling parallelization method and the implementation utilizing it. The parallel memory theory and implementation details are not discussed.

This paper presents a novel parallel memory implementation allowing conflict free arbitrary constant stride accesses. For hardware simplicity, the number of memory modules is restricted to a power of two ($N = 2^n$). From the hardware point of view, the skewing schemes are simplified from the ones in [8]. A skewing scheme is changed at run time according to the used stride. However, when a skewing scheme is changed, the data locating already in the memory modules is naturally invalidated. This is not a problem, for example, in the image scaling system in [9]. A new stride with a possible new scheme is demanded only when new input image data is written to parallel memories. Theoretical details of the proposed parallel memory system are given in [6]. This paper analyzes the impact of pipelining and parallelism degree on performance and complexity.

Synthesis results in FPGA and ASIC are given for two pipeline architectures with memory module counts 2, 4, 8, 16, and 32.

Background for the used conflict free skewing schemes is given in Section II. The proposed parallel memory architecture is described in detail in Section III. Pipelining is described in Section IV and Section V gives the results. Finally, Section VI concludes the paper.

II. BACKGROUND

The address space in parallel memories cannot be assigned arbitrarily but the data has to be referenced using predetermined patterns, called access formats or templates. The distribution of data to the memory modules is called a module assignment function S that is also known as a skewing scheme. The used module assignment function determines access formats that can be used conflict free. A data element at location i is assigned to a memory module according to $S(i)$. An address function $a(i)$ (also called row number) determines the physical address in a memory module for a data element.

Dynamic schemes for arbitrary stride accesses were first introduced by Harper *et al.* in [7]. Row rotation schemes [10] with $N = 2^n$ was used. Another Harper's approach concerning arbitrary stride accesses utilizes XOR-schemes [8]. XOR-schemes allow simpler computation since merely logic gates are demanded and no carry bits are used for computation [2], [11]. XOR-schemes always require power of two memory module count. Also in this study, XOR-schemes and $N = 2^n$ memory modules are used.

Let a stride be defined as

$$\text{stride} = \sigma \cdot 2^s, \quad (1)$$

where σ is an odd positive integer and s a nonnegative integer. All the strides can be formed by changing σ and s . Any skewing scheme that allows a conflict free access for a specific stride 2^s also provides a conflict free access for strides (1) with all σ [7]. Therefore, by finding conflict free storage schemes for each of the values of s (i.e. single scheme for a single s) enables conflict free access for arbitrary strides.

The proposed module assignment is defined as

$$S(i) = i[n+s-1:s] \oplus i[n-1:0], \quad (2)$$

where a data element location i is stated in binary form [MSB : LSB] and \oplus denotes bit-wise logical XOR operation. The implementation of (2) demands defining n and s . In a particular implementation, the number of memory modules $N = 2^n$ is a constant. Respectively, s is determined by a stride (1). Therefore, n is defined at design time and s at run time.

The proposed schemes are optimized from Harper's ones [8], [12] in terms of hardware complexity. The hardware in [8] and [12] includes n two-input XOR gates, a comparator

(max determination), two multiplexers, a mask generator and a shifter with a capability of bit masking. Respectively, the proposed implementation (2) demands only n two-input XOR gates and a shifter.

It is proved in [6] that the proposed skewing schemes (2) provide conflict free access for all the constant *even* strides with practical number of memory modules $N = 2, 4, 8, \dots, 1024 | n \in \{1, \dots, 10\}$. However, the hardware implementations of [7], [8], and [12] as well as (2) do not function properly when *odd* strides are used ($s = 0$). In the proposed implementation, the odd strides are separately taken into account. When $s = 0$, the conventional *low-order interleaved scheme* is formed as follows:

$$S(i) = 0 \oplus i[n-1:0] = i[n-1:0]. \quad (3)$$

All the odd strides can be accessed with the low-order interleaved scheme [5]. Therefore, all the constant strides are conflict-free by using (2) and (3) with practical memory module count.

A row access format (N adjacent data elements with stride 1) is commonly used. Word addressability means that the memory can be addressed only at word boundaries when the word is defined as N data elements. The module assignment functions $S(i)$ (2) and (3) allow conflict free word addressable row accesses with all the values of n and s .

The proposed and a suitable address function for (2) and (3) is defined as

$$a(i) = \lfloor i/N \rfloor. \quad (4)$$

In (4), the address in a memory module $a(i)$ is received directly from the MSBs of the data element location i and no additional hardware is required.

III. PROPOSED PARALLEL MEMORY ARCHITECTURE

A block diagram of the proposed parallel memory architecture is shown in Fig. 1. The referenced data locations are defined with the *first element location (scanning point)* r and the stride length $Stride_a$. The skewing scheme is formed according to the $Stride_s$ input. In write operation, *Write Enable* input is asserted simultaneously with the data to be written $wd_0, wd_1, \dots, wd_{N-1}$. The two *Data Permutation* units permute the written or read data according to the control signals from the *Address Computation* unit.

As mentioned above, a word addressable row access format (stride = 1) is conflict free in all the used skewing schemes but the scheme can only be changed before writing new data to memory. Therefore, the access stride ($Stride_a$) and the stride defining the used skewing scheme ($Stride_s$) are separated to allow controlling the access stride and skewing scheme separately. As an example to illustrate the achieved benefit, data is first written to the parallel memory system with stride four by assigning $Stride_a = Stride_s = 4$. The same data can be referenced with a row access format with $Stride_a$

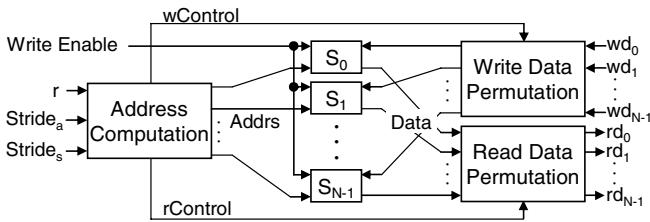


Fig. 1. Proposed parallel memory architecture.

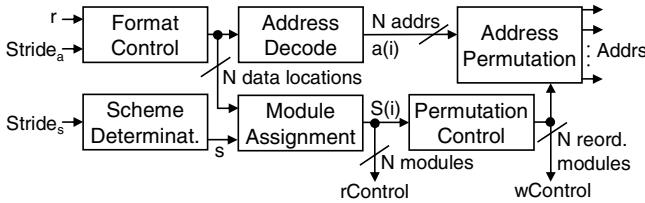


Fig. 2. Address Computation unit.

$= 1$ and $\text{Stride}_s = 4$. However, the row access format can be addressed only at word boundaries.

The proposed parallel memory system is implemented in VHDL. The system is designed to be configurable both at design time and at run time. In the hardware implementation, most parameters are VHDL generics, which means that the memory module count ($N = 2^n$), size of a memory module, and width of buses are adjustable. The run-time configurability refers to the ability to change the skewing scheme at run time.

Fig. 2 depicts a block diagram of the Address Computation unit. The outputs of the unit are the memory module addresses for each of the memory module (*Addrs*). Moreover, control signals (*rControl*, *wControl*) are formed for the Data Permutation units.

Inside the unit, the *Format Control* unit specifies the accessed data locations. The *Scheme Determination* unit defines the used skewing scheme. The *Module Assignment* unit computes the specific memory modules that contain the accessed data and the *Address Decode* unit, in turn, computes the physical addresses for each memory module. The *Permutation Control* unit reorders the memory module numbers. According to these values, the *Address Permutation* unit directs the physical addresses to the proper memory modules.

A detailed block diagram of the Format Control unit is shown in Fig. 3. The stride is multiplied with constant numbers $2, \dots, N-1$ and the referenced data locations i_0, i_1, \dots, i_{N-1} are formed by additions.

Fig. 4 depicts the Scheme Determination unit. The input to the unit is the stride defining the used skewing scheme $\text{Stride}_s = \text{Str}[d-1 : 0]$. The relation between a stride and s is given in (1). The output s is constructed by determining the count of adjacent zeros before first ‘1’ by starting from LSB. For example, with $\text{Stride}_s = 12_{10} = 01100_2$, the output $s = 2_{10}$. Another input of the equality comparators is a constant zero but the width of the other input is different in each of the

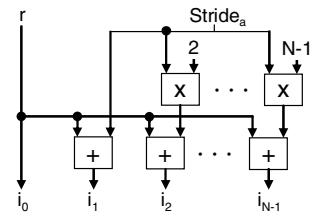


Fig. 3. Format Control unit.

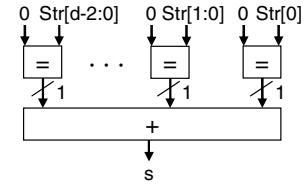


Fig. 4. Scheme Determination unit.

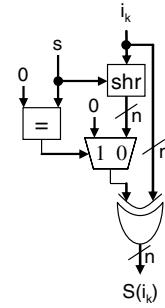


Fig. 5. Module Assignment unit (1/ N part of the unit).

comparators. The output of the equality comparators is one bit wide. Those one bit wide results are summed up to get the output s . Since the $\text{Stride}_s \geq 1$ (i.e. stride cannot be 0), the most significant bit $\text{Str}[d-1]$ is not actually demanded.

Fig. 5 shows a block diagram of the Module Assignment unit. In fact, there are N parallel similar logic blocks in the Module Assignment unit. The input s from the Scheme Determination unit defines the used scheme. When $s = 0$, low-order interleaved scheme is used as shown in (3). Zero value is selected with a multiplexer and the n LSBs of the data location i_k is used to form $S(i_k)$ when $k \in [0, N-1]$. Otherwise, the data location i_k is shifted right (shr) according to s and, after that, n parallel bit-wise XOR operations are performed as shown in (2).

The Address Decode unit is presented in Fig. 6. The unit implements the address function (4) and contains no logic, only hardwired shifting. The MSBs of a data location i_k are selected as address $a(i_k)$ output.

The Permutation Control unit is shown in Fig. 7. The unit changes the order of the module numbers received from the Module Assignment unit. The comparators are used to find the memory module number $S(i_b)$ ($b \in [0, N-1]$) that equals the constant k , $k \in [0, N-1]$. The respective index b of the data element i_b is forwarded to output $S(i_b)$. As an example with $N = 4$ and with the memory module numbers { $S(i_0)$, $S(i_1)$, $S(i_2)$,

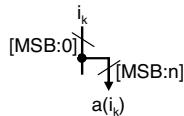


Fig. 6. Address Decode unit (1/N part of the unit).

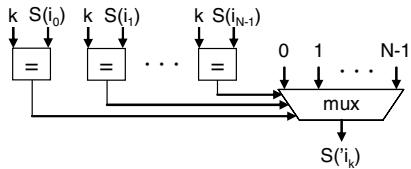


Fig. 7. Permutation Control unit (1/N part of the unit).

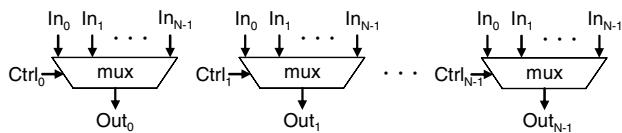


Fig. 8. Permutation unit.

$S(i_3) = \{ 1, 2, 3, 0 \}$, the reordered module number are $\{ S(i'_0), S(i'_1), S(i'_2), S(i'_3) \} = \{ 3, 0, 1, 2 \}$.

The parallel memory system includes three Permutation units. One of the units is depicted in Fig. 8. Full crossbar permutation networks are used. Memory module numbers in original or reordered way are used as control signals ($Ctrl_0, Ctrl_1, \dots, Ctrl_{N-1}$). In the Write and Read Data Permutation units, the inputs ($In_0, In_1, \dots, In_{N-1}$) and outputs ($Out_0, Out_1, \dots, Out_{N-1}$) are data buses. Respectively, addresses are permuted in the Address Permutation unit.

IV. PIPELINE

The proposed parallel memory implementation is divided into several pipeline stages. As in a normal pipeline, several operations can overlap in execution. Fig. 9 and Fig. 10 depict the implemented two pipelines. In both of the Figs. 9 and 10, read and write pipelines are separated. The *high throughput (HT)* architecture (Fig. 9) utilizes two clock cycles for the Address Computation unit whereas the *low latency (LL)* architecture (Fig. 10) consumes only one cycle. The arrows in Figs. 9 and 10 depict one-way links between the units.

At the first cycle in read operation of the HT architecture (Fig. 9), the Format Control and Scheme Determination units operate in parallel and give stimulus to the Address Decode and Module Assignment units. All of these four units are executed at the same cycle. In the next clock cycle, the Permutation Control and Address Permutation units operate in sequence. The third clock cycle is consumed by the memory modules. The output of the Module Assignment unit is delayed two clock cycles (second and third cycle) and used as control signal to permute the data into correct order in the Read Data Permutation unit at fourth cycle. Totally, the read latency of the HT architecture is four clock

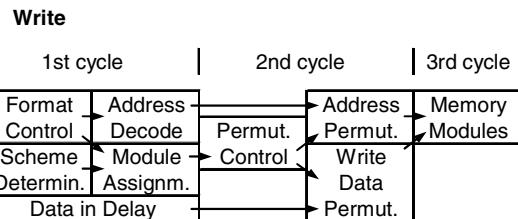
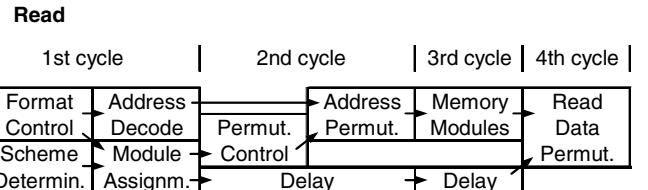


Fig. 9. High throughput (HT) architecture pipelining.

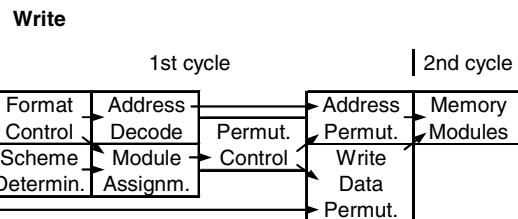
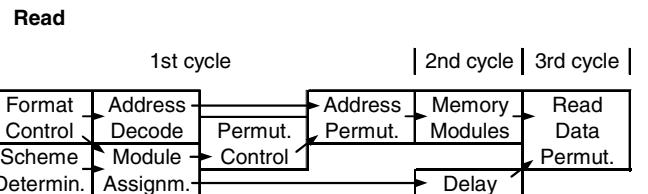


Fig. 10. Low latency (LL) architecture pipelining.

cycles. The LL architecture consumes three cycles with the same operation (Fig. 10).

In write operation, the Write Data Permutation unit operates simultaneously with the Address Computation unit. Therefore, the write latency in both of the HT and LL architectures is one clock cycle less than with read operation.

V. RESULTS

Both of the architectures (HT and LL) have been implemented in synthesizable register transfer level VHDL. The area and timing results are received by logic synthesis both in ASIC and FPGA. The evaluations are performed for systems consisting of 2, 4, 8, 16, and 32 memory modules. Memory capacity of a single memory module is kept constant 512 bytes. Therefore, the total memory capacity increases linearly as a function of the number of memory modules. The data width of a single memory module is eight bits and the maximum allowed stride is restricted to 255 in this study. The implementation is parametrizable and, therefore, all these values can be changed at design time.

TABLE I
ASIC AREA EVALUATION IN GATE COUNT

	N = 2	N = 4	N = 8	N = 16	N = 32
FC	49	235	779	2 161	5 603
SD	48	48	48	48	48
AD	0	0	0	0	0
MA	36	99	348	874	2 043
PC	1	29	191	1 015	5 056
AP	48	225	965	3 840	15 829
RDP	43	203	869	3 472	14 346
WDP	43	203	869	3 472	14 346
Pip, HT	479	1 001	2 106	4 424	9 249
Pip, LL	318	667	1 410	2 933	6 138
All, HT	746	2 042	6 175	19 306	66 520
All, LL	586	1 709	5 479	17 815	63 408

A. ASIC Results

The used synthesis technology is 0.18 μm CMOS process. All the other units but memory modules were synthesized.

Table I tabulates the area results of the proposed parallel memory system. The gate count is based on 2-input NAND gates. The area results are separated into Format Control (FC), Scheme Determination (SD), Address Decode (AD), Module Assignment (MA), Permutation Control (PC), Address Permutation (AP), Read Data Permutation (RDP), and Write Data Permutation (WDP) units. Moreover, the high throughput (HT) and low latency (LL) pipeline gate counts are shown. The pipelines include all the pipeline registers in the parallel memory system. The total gate count includes either one of the pipelines.

The Format Control unit, Module Assignment unit and both of the pipelines increase close to linearly as a function of the number of memory modules. The Scheme Determination unit does not depend on the memory module count whereas the Address Decode unit contains only hardwired shifting. The most dramatic increase as a function of N is with the Permutation Control unit. However, with $N = 32$, the Permutation Control unit consumes just 8 % of the total area. In this study, the three full crossbar permutation networks AP, RDP, and WDP have the most impact of the area as a function of N . With memory module count $N = 2$, pipelines in both of the HT and LL architectures consume over half of the total area. However, with $N = 32$, the three permutation units utilize 68 and 71 % of the HT and LL architecture area, respectively.

In addition to the number of memory modules, the Address Permutation unit depends on address width whereas the Data Permutation (RDP and WDP) units depend on data width. In the evaluations, as mentioned above, a single memory module depth is restricted to 512 locations implying 9-bit address buses whereas the data buses are restricted to eight bits. This explains the area difference between the Address and Data Permutation units.

TABLE II
ASIC TIMING EVALUATION

	N = 2	N = 4	N = 8	N = 16	N = 32
HT / MHz	633	552	461	368	272
LL / MHz	508	373	256	199	162

Another case study of a parallel memory system is implemented in [13]. Full crossbar permutation networks dominate the logic area in a video coding specific parallel memory system with memory module count four or more [13]. However, pipelining is not considered in [13].

The two Data Permutation units could be combined in the proposed system. The inputs of the combined Data Permutation unit would be selected with “Write Enable” signal between the current inputs of the Read or Write Data Permutation units (Fig. 1). The output of the combined unit would be directed to the inputs of the memory modules and the output of the whole parallel memory system. As a drawback, the delay of the combined Data Permutation unit would increase due to additional multiplexers. Moreover, write after read turn-around time of the proposed parallel memory system would also increase. Another or additional possibility to decrease the demanded logic is to utilize multistage interconnection networks like Benes [14].

Table II tabulates the achieved clock frequency of the proposed parallel memory system with 0.18 μm CMOS process. The high throughput architecture can utilize a clock frequency of 633 MHz to 272 MHz with memory module count $N = 2$ to 32, respectively. The low latency architecture attains from 54 to 80 % of the respective HT architecture clock frequency.

B. FPGA Results

The proposed parallel memory system is implemented in Altera Stratix EP1S40F780C5 logic device [15]. Compared to the above mentioned ASIC results, also the data memory is implemented by utilizing the embedded memory modules on the FPGA. According to synthesis results, a single 4096-bit memory module achieves 291 MHz clock frequency.

The area and timing results are tabulated in Table III. Both of the architectures (HT and LL) utilize the same amount of memory, from 8192 to 131072 bits. The high throughput architecture demands from 139 to 21369 logic cells (LCs) with memory module count $N = 2$ to 32. This is 0.3 to 51.8 % of the total logic cell count and 0.2 to 3.8 % of the memory on the used FPGA. The low latency architecture utilizes 2 to 17 % less resources than the respective HT architecture.

The used synthesis tool demands registered inputs to provide proper timing estimates. Therefore, the results shown in Table III contain some extra registers. Rough estimate is about 5 % overhead in the logic cell count with HT architecture and $N = 8$. The used FPGA device also contains embedded multipliers. However, the evaluated parallel memory system does not utilize them.

TABLE III
FPGA AREA AND TIMING EVALUATION

	N = 2	N = 4	N = 8	N = 16	N = 32
Mem / bits	8 192	16 384	32 768	65 536	131 072
HT / LCs	139	378	1 466	5 349	21 369
LL / LCs	115	338	1 381	5 139	20 969
HT / MHz	209	183	143	120	80
LL / MHz	210	115	81	64	46

The achieved clock frequency varies between 209 and 80 MHz with HT architecture (Table III). The LL architecture attains 53 to 101 % of the respective HT architecture frequency. For reference with the same Altera Stratix FPGA device, a 32-bit Nios processor [16] with a timer and a serial connection (UART) peripherals utilizes 2316 logic cells and achieves 117 MHz clock frequency [17].

C. FPGA Results with Balanced Pipeline

The ASIC and FPGA pipelines are not well balanced, especially in LL architecture. For example, with $N = 16$ in Table III, a read operation in LL (low latency) architecture has higher latency in terms of seconds than a respective operation with HT architecture, 47 ns and 33 ns, respectively. The pipeline should be balanced for each of the memory module count separately. This can be done by hand at the expense of design-time configurability. In this study, the automatic methods given by a synthesis tool are used and the FPGA results are tabulated in Table IV.

The logic cell count with balanced pipeline increases between 6 to 49 % from the respective amount shown in Table III. With $N = 16$ and 32, the LL architecture exceeds the logic cell count of HT architecture (Table IV).

The maximum clock frequency in the high throughput architecture is from 232 to 96 MHz. The high improvement in clock frequency is achieved with the low latency architecture (10 to 102 % increase). The improvements are mainly due to the fast on-chip SRAM memory modules which can manage clock frequency over 250 MHz. With much lower clock frequencies, plenty of logic can be executed at the same clock cycle with the memory modules.

VI. CONCLUSION

Stride accesses are frequently required in the fields of digital signal processing and telecommunications. In this paper, we have presented a novel parallel memory architecture allowing all the constant stride accesses which has not been possible in prior application specific parallel memory systems. Powers of two memory module counts from 2 to 32 have been evaluated. The results show that the proposed system can be implemented in contemporary programmable logic device with acceptable latency and clock frequency.

TABLE IV
FPGA AREA AND TIMING EVALUATION WITH BALANCED PIPELINE

	N = 2	N = 4	N = 8	N = 16	N = 32
Mem / bits	8 192	16 384	32 768	65 536	131 072
HT / LCs	173	565	1 991	6 015	22 608
LL / LCs	146	505	1 761	6 199	22 796
HT / MHz	232	229	184	141	96
LL / MHz	232	206	158	129	86

REFERENCES

- [1] G. Kuzmanov, G. Gaydadjiev, and S. Vassiliadis, "Multimedia rectangularly addressable memory," *IEEE Trans. Multimedia*, vol. 8, no. 2, pp. 315–322, Apr. 2006.
- [2] S. Chen, A. Postula, and L. Jozwiak, "Synthesis of XOR storage schemes with different cost for minimization of memory contention," in *Proc. Euromicro Conf.*, Milan, Italy, pp. 170–177, Sep. 1999.
- [3] J. Takala and T. Järvinen, "Stride permutation access in interleaved memory systems," in *Domain-Specific Multiprocessors – Systems, Architectures, Modeling, and Simulation*, ed. S. S. Bhattacharyya, E. F. Deprettere, and J. Teich, Marcel Dekker, New York, NY, USA, pp. 63–84, 2004.
- [4] E. Aho, J. Vanne, K. Kuusilinna, and T. D. Hämäläinen, "Address computation in configurable parallel memory architecture," *IEICE Trans. Inf. & Syst.*, vol. E87-D, no. 7, pp. 1674–1681, July 2004.
- [5] M. Valero, T. Lang, M. Peiron, and E. Ayguadé, "Conflict-free access for streams in multimodule memories," *IEEE Trans. Computers*, vol. 44, no. 5, pp. 634–646, May 1995.
- [6] E. Aho, J. Vanne, and T. D. Hämäläinen, "Parallel Memory Architecture for Arbitrary Stride Accesses," in *Proc. IEEE Workshop Design and Diagnostics of Electronic Circuits and Systems*, Prague, Czech Republic, pp. 65–70, Apr. 2006.
- [7] D. T. Harper III and D. A. Linebarger, "Conflict-free vector access using a dynamic storage scheme," *IEEE Trans. Comput.*, vol. 40, no. 3, pp. 276–283, Mar. 1991.
- [8] D. T. Harper III, "Increased memory performance during vector accesses through the use of linear address transformations," *IEEE Trans. Comput.*, vol. 41, no. 2, pp. 227–230, Feb. 1992.
- [9] E. Aho, J. Vanne, T. D. Hämäläinen, and K. Kuusilinna, "Block-level parallel processing for scaling evenly divisible images," *IEEE Trans. Circuits Syst. I*, vol. 52, no 12, pp. 2717–2725, Dec. 2005.
- [10] P. Budnik and D. J. Kuck, "The organization and use of parallel memories," *IEEE Trans. Comp.*, vol. C-20, no. 12, pp. 1566–1569, Dec. 1971.
- [11] J. M. Frailong, W. Jalby, and J. Lenfant, "XOR-schemes: A flexible data organization in parallel memories," in *Proc. Int'l Conf. Parallel Processing*, Washington, DC, USA, pp. 276–283, Aug. 1985.
- [12] D. T. Harper III and D. A. Linebarger, "Dynamic address mapping for conflict-free vector access," U.S. Patent 4 918 600, Apr. 17, 1990.
- [13] J. K. Tanskanen and J. T. Niittylahti, "Scalable parallel memory architectures for video coding," *Journal of VLSI Signal Processing*, vol. 38, no. 2, pp. 173–199, Sep. 2004.
- [14] S. Dutta, W. Wolf, and A. Wolfe, "A methodology to evaluate memory architecture design tradeoffs for video signal processors," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 8, no. 1, pp. 36–53, Feb. 1998.
- [15] Altera, Stratix Device Handbook, Volume 1, Version 3.2, Jan. 2005.
- [16] Altera, Nios 3.0 CPU Data sheet, Version 2.2, Oct. 2004.
- [17] E. Salminen, A. Kulmala, and T. D. Hämäläinen, "HIBI-based Multiprocessor SoC on FPGA," in *Proc. IEEE Int'l Symp. Circuits Syst.*, Kobe, Japan, pp. 3351–3354, May 2005.

Publication 10

E. Aho, J. Vanne, T. D. Hämäläinen, and K. Kuusilinna, “Configurable Implementation of Parallel Memory Based Real-Time Video Downscaler,” *Microprocessors and Microsystems*, accepted for publication.

Copyright © Elsevier B.V.
Printed with permission.

Configurable implementation of parallel memory based real-time video downscaler

Eero Aho¹, Jarno Vanne¹, Timo D. Hämäläinen¹, and Kimmo Kuusilinna²

¹Institute of Digital and Computer Systems, Tampere University of Technology,
FI-33101 Tampere, Finland

²Nokia Research Center, FI-33721 Tampere, Finland

Abstract

Image downscaling is necessary in multiresolution video streaming and when a camera captures larger resolution frames than required. This paper presents an implementation of a downscaler capable of real-time scaling of color video. The scaler can be configured to support nearly arbitrary scaling ratios. The scaling method is based on evenly divisible image sizes, which is, in practice, the case in most video and image standards. Bilinear interpolation is utilized as the scaling algorithm. Fine-grained parallel processing is utilized to increase performance and parallel memories are used to attain the required bandwidth. The results show that an FPGA implementation can downscale 16VGA and HDTV video in real-time with a complexity of less than half of the reference implementations.

Key words: Image scale, Bilinear interpolation, Parallel processing, Real-time, Parallel memory

1. Introduction

Resampling is utilized when a discrete image is transformed to a new set of coordinate points. Some resampling examples are image scaling, spatial distortion correction, and image rotation. Resampling can be divided conceptually into two sequential processes. At first, a discrete image is interpolated to a continuous image. Then, the interpolated image is sampled. In practical implementations, interpolation and sampling are often combined so that the image is interpolated at only those points that are sampled.

Spatial scalability refers to the feature of image representation in different sizes or spatial resolutions. Real-time spatial scalable video coding with several frame resolutions demands lots of computation performance for scaling. In addition, video resolutions used in typical applications tend to increase. Nowadays, consumer video cameras capturing frames with the size of 1440 x 1080 pixels at real-time are available [1]. High performance downscaling is required when the post processed video resolution is smaller.

Complexity and quality of scaling varies depending on the used interpolation method. Several studies have compared existing interpolation methods [2], [3], [4]. The most recent [2] evaluates *nearest neighbor*, *bilinear*, *quadratic*, *cubic B-spline*, *cubic*, *truncated sinc*, *windowed sinc*, *Lagrange*, and *Gaussian* interpolation methods. The simplest and fastest one, nearest neighbor, also incurs the largest interpolation error. Bilinear interpolation is a frequently used operation in graphics hardware and the quality is sufficient for many applications [5]. The most complex one, Gaussian interpolation, used on average 22 times more time than bilinear interpolation [2]. Hence, a trade-off between scaling quality and complexity has to be made.

Standard image sizes such as VGA, HDTV, PAL, NTSC, and CIF are divisible by the number sixteen. One explanation for this property is that video compression standards divide the image to 16 x 16 macro blocks (MPEG-2, MPEG-4, H.263, H.264). However, image sizes are not always

compatible with those block sizes. In those cases and before compression, the image size is changed to fulfill the requirements of the standard, for example with padding. After that, the divisibility of the image width and height are known and this knowledge can be utilized during the scaling.

For efficient computation, many practical optimizations have been presented. Image sizes are restricted to multiple of 8 x 8 size blocks in a compressed domain scaling method shown in [6]. Lagrange interpolation algorithm is parallelized in [7]. A new bilinear type scaling algorithm, *winscale*, is implemented on an FPGA in [8]. A hybrid bilinear scaling, *Qscale*, produces lower but comparable quality to bilinear interpolation with less complex hardware [9]. The above mentioned scaling implementations do not utilize fixed image sizes.

The fixed image size is used in [10] utilizing an FIR filter with constant coefficients. In [11], all the coefficients are stored in registers. Similarly, an adaptive scaling algorithm with pre-calculated coefficients is implemented in [12]. In these two designs, the coefficients change according to the utilized interpolation algorithm but only few scaling ratios can be used. In [13], certain pre-calculated parameters are stored in memory in advance to simplify the complexity of the cubic spline interpolation based scaling hardware.

This paper presents a hardware implementation capable of downscaling large resolution YUV 4:2:0 color format video in real-time. Similarly as in the above mentioned implementations, the original and scaled image sizes are fixed at design time. A new scaling ratio demands hardware reconfiguration and the ratio can be from a nearly arbitrary range. In addition, bilinear interpolation is used in our implementation and the above mentioned image divisibility is utilized to further reduce the required hardware resources. In the implementation, all the four original image pixels necessary in bilinear interpolation are processed in parallel. The implementation is verified on an FPGA and the results are compared to four reference solutions found from literature.

Background for image scaling is introduced in Section 2 and scaling down with the design-time fixed image sizes is analyzed in Section 3. The implemented downscaler is described in detail in Sections 4 and 5. Section 6 examines pipelining of the execution and the results are given in Section 7. Section 8 concludes the paper.

2. Image scaling

2.1. Scaling with bilinear interpolation

In image scaling, pixels are required to be interpolated only at the sampling points. The interpolated image sampling point distances are horizontal distance Δx and vertical distance Δy . Normally, these distances are defined as follows.

$$\begin{aligned}\Delta x &= \text{original_image_width} / \text{scaled_image_width}, \\ \Delta y &= \text{original_image_height} / \text{scaled_image_height}.\end{aligned}\tag{1}$$

Fig. 1 illustrates the bilinear interpolation. At the sampling point, the scaled pixel value P can be filtered by the original image pixels $C0-C3$ as

$$P = W0 \cdot C0 + W1 \cdot C1 + W2 \cdot C2 + W3 \cdot C3.\tag{2}$$

Distances between the scaled pixel P and the original pixel $C0$ are depicted by Δw and Δh in x and y directions, respectively. Distance from $C0$ to $C2$ and from $C0$ to $C1$ equals 1.0. The coefficients $W0-W3$ are defined as follows.

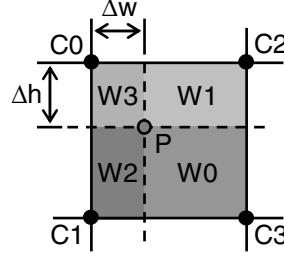


Fig. 1. Scaling with bilinear interpolation.

$$\begin{cases} W0 = (1 - \Delta w) \cdot (1 - \Delta h) \\ W1 = (1 - \Delta w) \cdot \Delta h \\ W2 = \Delta w \cdot (1 - \Delta h) \\ W3 = \Delta w \cdot \Delta h. \end{cases} \quad (3)$$

The coefficient values $W0-W3$ correspond to the areas shown in Fig. 1.

Bilinear interpolation consists of four operation phases [8]. In *pixel moving*, the location of the scaled pixel P is calculated from the previous scaled pixel location utilizing the distances Δx and Δy (1). *Memory accesses* are needed to load the four original image pixels $C0-C3$ around the scaled pixel location. *Weighting factor calculation* defines the coefficients $W0-W3$ (3). Finally, *pure filter* operations are performed in order to calculate the scaled pixel value P (2).

2.2. Scaling evenly divisible images

In the following discussion, the *original block* is defined to be a part of the original image. The width and/or height of the original image are multiples of the original block. In addition, the original block can be scaled with the same scaling ratio as the original image. With the scaling ratio, a *scaled block* is scaled from the original block. The width and height of the original and the scaled block must be natural integers.

Fig. 2 depicts an original block size $c_1 \times d_1$ that is scaled to a size $c_2 \times d_2$. The original image is scaled with the same ratios and, in this example, the width a_1 is three times and the height b_1 twice the size of the original block. Therefore, the sampling point distances Δx and Δy are the same in both scaling examples. In general, the minimum sized original block is $\text{gcd}(a_1, a_2)$ times in x -direction and $\text{gcd}(b_1, b_2)$ times in y -direction smaller than the original image where gcd is the greatest common divisor [14]. For this statement to be true, the interpolated image sampling point distance needs to be defined as shown in (1).

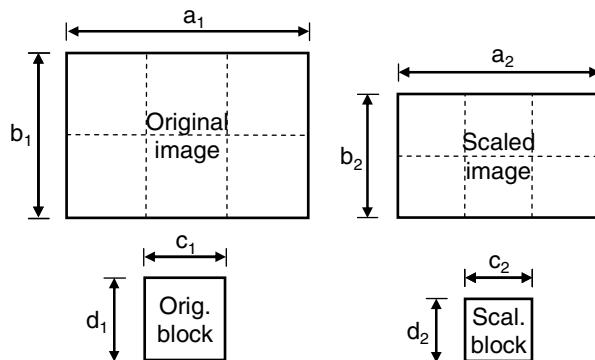


Fig. 2. Evenly divisible images scaled down.

Table 1. Required ROM depth with the bilinear interpolation.

	ROM	ROM, block property	ROM, block property + logic
Pixel moving, x	a_2	-	$a_2/c_2 + c_2$
Pixel moving, y	b_2	-	$b_2/d_2 + d_2$
Weighting factor calculation	$4a_2b_2$	$4c_2d_2$	$c_2 + d_2$

The evenly divisible image property allows optimized image scaling. A straightforward way is to utilize block-level parallelism [14]. In Fig. 2, if six units designed for block scaling are processing in parallel, each can scale a separate part of the original image. As a drawback, this kind of block-level parallel implementation demands an additional buffer to gather the parallel scaled pixels into correct order [14]. This paper shows a more application specific implementation and fine-grained parallelization method without the additional buffer.

With a 4:2:0 YUV color image, both of the chrominance components have a quarter of the luminance component sample count. The used block scaling size is the same for both luminance and chrominance components. However, the number of block scaling iterations is twice for luminance data, both horizontally and vertically.

3. Downscaling with fixed image sizes

The original and scaled image sizes are fixed at design time to make the pixel moving and weighting factor calculation operations simpler. For example, some of the parameters can be calculated with accurate floating point arithmetic and stored in ROM in advance.

When scaling evenly divisible images, each of the image blocks has the respective sampling point locations and same weighting factors. Therefore, less memory is required in ROM based implementations. The same aspect was partly noticed with windowed sinc interpolation function in [15]. In our case, we optimize the trade-off between memory and logic in bilinear interpolation based scaling. The implementation details with fixed image sizes are analyzed in detail in the following.

3.1. Pixel moving

A straightforward way to implement the pixel moving operation is to use two fractional adders, one for x -direction and the other for y -direction. With fixed scaling ratios, the sampling point distances Δx and Δy (1) are constant. Therefore, in both of the fractional adders, one of the two inputs is constant making the implementation easier. However, the error accumulates due to restricted accuracy of the adders.

Another way is to compute the pixel moving operations in advance and store the results in memory. To diminish the memory width, only the integer location of $C0$ is stored in memory instead of the fractional location of P . The required ROM depths (number of words) are a_2 and b_2 in x and y -directions, respectively (Table 1).

With the proposed implementation, evenly divisible image property and additional logic are utilized, which reduces the memory consumption further. The $C0$ locations in a block as well as each block location in an image are stored in memory (Table 1). Moreover, an adder is used to sum those values together. For example, if an HDTV (1920 x 1088) size downsampled image is divided to 16 blocks in x and y -directions, this method diminishes the memory consumption from 4140 bytes to 210 bytes.

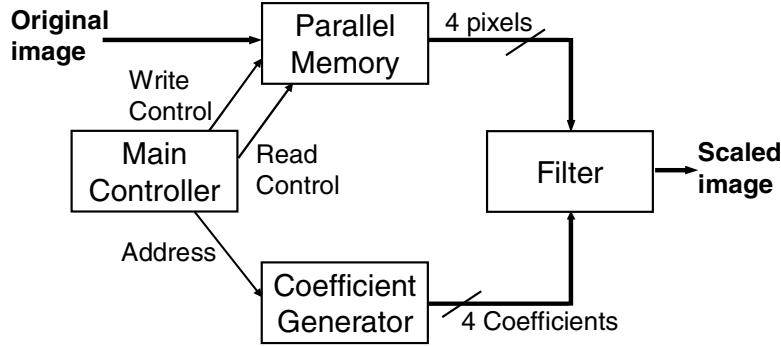


Fig. 3. Downscaler architecture.

3.2. Weighting factor calculation

The fractional location of the sampling point P has to be delivered from the pixel moving stage, if the weighting factors are calculated with logic. In any case, the total error is squared since errors in x and y -coordinates of the sampling point are multiplied in (3). By calculating all the weighting factors to memory in advance, a ROM depth of $4a_2b_2$ is required with bilinear interpolation (Table 1). An HDTV size downscaled image with 16-bit factors consumes 16 MB of ROM.

The memory consumption for an HDTV image is reduced to 64 KB when the image is divided to 16 blocks in x and y -directions. In the proposed implementation, interpolation relies on a trade-off between storage and calculation of the weighting factors. Only Δw and Δh values of each image block are stored in memory and the rest are computed with logic. The utilized memory depth is $c_2 + d_2$. As a result, only 376 bytes of ROM is consumed with the same image size and block count as above.

4. Downscaler architecture

A block diagram of our downscaler architecture is shown in Fig. 3. The original image pixels are received as input, four pixels at a time. The scaled image is given to the output pixel by pixel. Scaling operations are performed as follows. Pixel moving operations are done in the *Main Controller* unit and memory accesses are handled in the *Parallel Memory* unit. Weighting factor calculation is carried out in the *Coefficient Generator* unit. Finally, pure filter operations are performed in the *Filter* unit. Four original image pixels read from the *Parallel Memory* unit and four coefficients computed in the *Coefficient Generator* unit are used to filter one scaled pixel at a time.

The downscaler is designed to be design-time configurable. In the hardware implementation, many parameters are VHDL generics, which means that the original and scaled image sizes as well as the used evenly divisible image block count and weighting factor calculation bit widths are adjustable. According to the generic values, the ROM contents are automatically calculated at design time. All the required logic, ROM, and buses are implemented with logic synthesis from VHDL. The ready-made memory modules on FPGA are used as data memory.

4.1. Main Controller

The Main Controller unit contains two sub-units. A *Write Controller* unit controls the writing of an original image to the Parallel Memory unit. A *Scale Controller* unit controls reading of the original image pixels from the Parallel Memory unit and determining the addresses to the ROM located in the Coefficient Generator unit. The two controllers maintain the correct order of data accesses to the Parallel Memory unit.

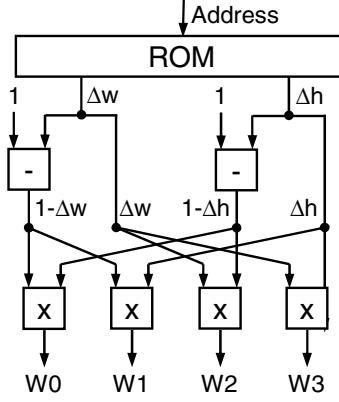


Fig. 4. Coefficient Generator unit.

4.2. Coefficient Generator

The Coefficient Generator and the Filter units operate on fractional fixed point numbers. The accuracy of the computation is affected by bit widths. The pixel values are eight bits, but the bit widths of the weighting factors have to be defined according to the desired accuracy. As noted above, the bit widths of the weighting factor calculation is configurable at design time.

A block diagram of the Coefficient Generator unit is depicted in Fig. 4. The input to the unit is a ROM address. The ROM defines the Δw and Δh values for the weighting factor calculation (3). The Δw and Δh values are between [0..1]. The subtractions from the constant value 1 are done and the results are multiplied to get the coefficients W_0-W_3 . Two subtractors with the other input as a constant and four fractional multipliers are used. The required ROM depth is $c_2 + d_2$ (Table 1).

4.3. Filter

A block diagram of the Filter unit is shown in Fig. 5. At first, the coefficients W_0-W_3 are multiplied with the original image pixels C_0-C_3 . Then, the products are summed up. This implements the functionality shown in (2). The intermediate results are truncated but the final result is rounded with the round to nearest method by adding the value 0.5. The 8-bit integer values C_0-C_3 are multiplied with coefficients W_0-W_3 having a magnitude between [0..1) and, therefore, the integer part of the products can be expressed with eight bits. Three carry bits have to take into account when the four products and the value 0.5 are summed up. The three carry bits are fed to an OR gate to control the saturation of the result. The resulted scaled pixel value is saturated to the maximum 8-bit pixel value. The implementation takes four multipliers and one 5-input adder with one input as a constant. Moreover, an OR gate in addition to a 2-input multiplexer is used.

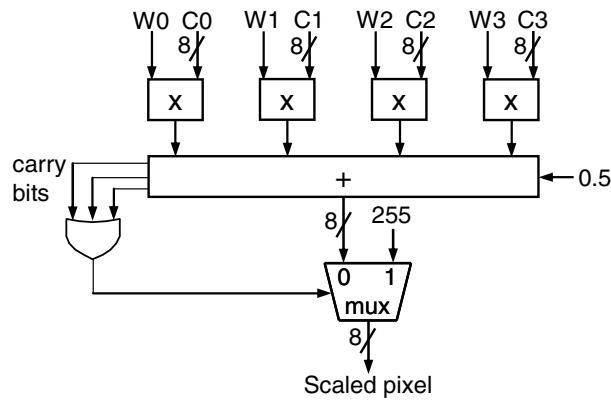


Fig. 5. Filter unit.

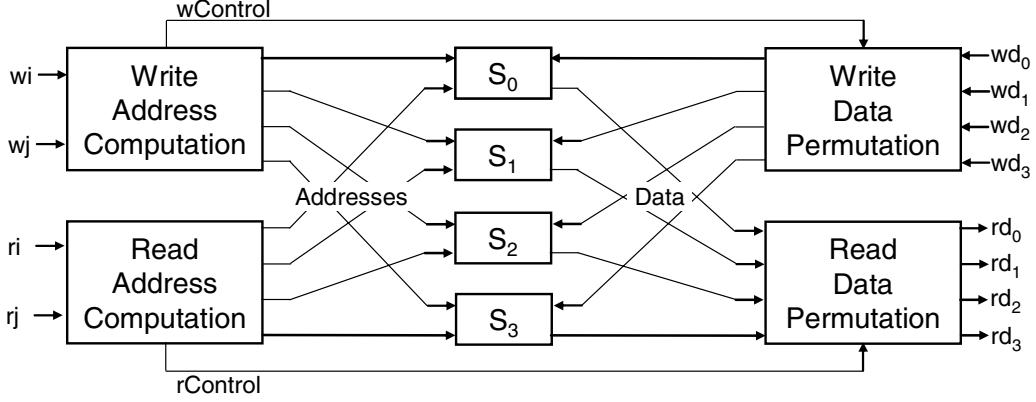


Fig. 6. Parallel Memory unit.

5. Parallel Memory

Parallel memories increase memory bandwidth with several memory modules working in parallel [16], [17]. A block diagram of the implemented Parallel Memory unit is shown in Fig. 6. The unit includes two *Address Computation* and two *Data Permutation* units as well as four dual port memory modules S_0, S_1, S_2, S_3 . Each memory module is capable of simultaneous read and write operations. Depending on the *write scanning point* (wi, wj) or the *read scanning point* (ri, rj), the respective Address Computation unit computes the addresses and directs them to the appropriate memory modules. The Data Permutation units organize the data into correct order according to the control signals received from the Address Computation units. In the write operation, four data elements wd_0, wd_1, wd_2, wd_3 are permuted and stored in the memory modules. With reading, the data elements rd_0, rd_1, rd_2, rd_3 are outputs of the unit.

Each memory module is one byte (pixel) wide. Two original image rows are demanded for filtering and two rows for buffering. Therefore, memory capacity of the four original image rows is required. Circular/module memory addressing [18] is used in y-direction.

5.1. Functionality

The data in parallel memories cannot be assigned arbitrarily, but it is accessed with the predetermined patterns, called *access formats*. The data elements of an access format can be read from the memory modules in parallel when a *conflict free* access is performed. Data is distributed to the memory modules according to a *module assignment function* S . The used module assignment function determines access formats that can be used conflict free. A *scanning point* defines the access format location in the addressable two-dimensional memory area, the *scanning field*. A *placement set* defines the memory locations where an access format can be placed conflict free.

In the proposed implementation, four bytes wide row access format is needed for writing and a square (2×2) access format for reading. The unrestricted placement set is required for reading, meaning that the square access format can be placed anywhere in the scanning field. For writing, an unrestricted placement set is not required, but the row access format needs to cover the scanning field.

For the proposed implementation with four memory modules, a suitable module assignment function for the two access formats is

$$S(i, j) = (i + 2j) \bmod 4. \quad (4)$$

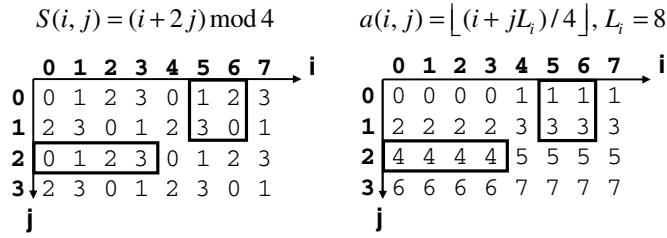


Fig. 7. Used module assignment and address functions.

This is a *linear* module assignment function [19]. An *address function* $a(i, j)$ determines the physical address in a memory module for a data element. A feasible address function for the module assignment function is

$$a(i, j) = \lfloor (i + jL_i)/4 \rfloor, \quad (5)$$

where L_i is the width of the scanning field. Four memory modules with capacity of four original image rows are occupied in this case. In addition, although the width of the original image can be arbitrary, the size of the memory module is normally power of two. Therefore, the width of the scanning field L_i can be restricted to power of two without extra memory consumption compared to situation when L_i equals to the original image width. Naturally, L_i has to be a power of two number that is equal or greater than the original image width.

The used module assignment and address functions are illustrated in Fig. 7. The width of the scanning field L_i is defined as eight and 8×4 data elements are displayed. A single data element is stored in a memory module with a number shown in the corresponding table cell on the left-hand side of Fig. 7. The respective address in the memory module is shown on the right-hand side. The elements are indexed with (i, j) coordinates. For example, a data element located at point $(3, 2)$ is stored in a memory module number 3 under address 4. The row and square (2×2) access formats are enclosed by a fat line in Fig. 7.

5.2. Write Address Computation and Data Permutation

As mentioned above, the row access format used for writing needs only to cover the scanning field. Therefore, the accesses in the scanning field can be restricted to non-overlapped locations starting from the top left corner of the scanning field. Due to the address function (5), the same address is fed to each memory module. Only hardwired masking and shifting is used in the Write Address Computation unit since L_i is restricted to the power of two with (5).

There are two data permutation possibilities with the utilized module assignment function (4) and a non-overlapped covering placement set. For this reason, the logic in the Write Data Permutation unit includes four 2-input multiplexers with each of the input and output widths being eight bits.

5.3. Read Address Computation and Data Permutation

As stated before, the read operation demands square access format with an unrestricted placement set. Therefore, the addresses have to be calculated separately for each memory module. Fig. 8 depicts the Read Address Computation unit in more detail. The outputs of the unit are addresses for each memory module (a_0, a_1, a_2, a_3) and rControl signal for controlling the Read Data Permutation unit. At first, the module number of the scanning point is calculated with the module assignment function (4). The function can be refined as follows:

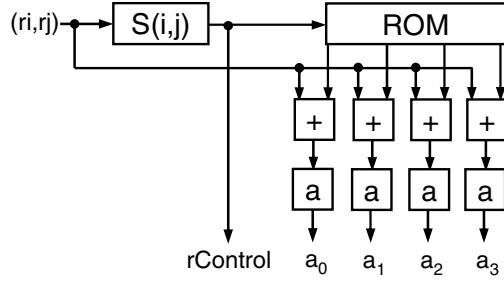


Fig. 8. Read Address Computation unit.

$$S(i, j) = (i + 2j) \bmod 4 = ((i \bmod 4) + (2j \bmod 4)) \bmod 4 = ([i_1, i_0] + [j_0, 0]) \bmod 4, \quad (6)$$

where $[i_1, i_0]$ and $[j_0, 0]$ mean the two least significant bits. It should be noted that only a two bit adder is required. After calculation of the module number, four relative (i, j) coordinates are determined with a ROM (Fig. 8) having a size of four 8-bit locations. These four i and four j values are added in parallel with the scanning point using eight (2×4) adders. Finally, addresses are calculated utilizing the same address function $a(i, j)$ (5) as with the Write Address Computation unit. Merely hardwired masking and shifting is utilized for address calculation.

The *isotropic* property [19] of the used linear module assignment function (4) can be utilized for implementing the Read Data Permutation unit. At most N different permutations have to be carried out with an isotropic module assignment function when the number of accessed elements is N [19]. Therefore, the proposed system demands four different permutations that can be implemented with a rotator. The Read Data Permutation unit is depicted in more detail in Fig. 9. With $rControl$ signal the correct data is selected for each memory module. Four 4-input multiplexers are required for the implementation.

6. Pipeline

The Downscaler unit has two separate pipelines, one for writing the original image to memory and the other to perform the actual scaling. The two pipelines depend on each other so that the scaling cannot be started before the required data is in the memory. Respectively, writing has to wait until the accessed memory location is free. Fig. 10 and Fig. 11 depict the pipeline stages without any pipeline stalls. As in a normal pipeline, several writing and scaling operations can be overlapped in execution.

The pipeline depths are formed according to initial synthesis delay results on FPGA. The latency of the write pipeline (Fig. 10) without wait stages is three clock cycles. The first stage is for the Write Controller unit in the Main Controller unit. The next two stages are for the Parallel Memory unit. In the second stage, the Write Address Computation unit determines the addresses for the memory modules. At the same stage, the Write Permutation unit permutes the incoming data into correct order. The reordered data is stored in the memory modules at stage three.

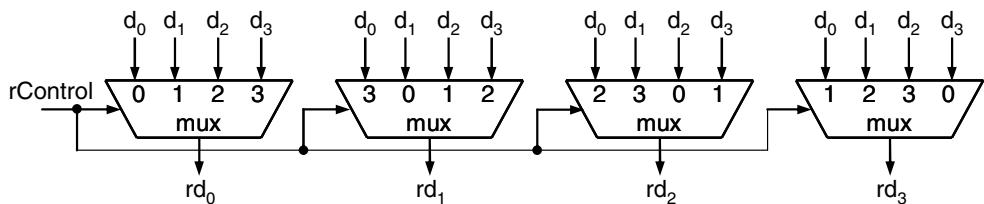


Fig. 9. Read Data Permutation unit.

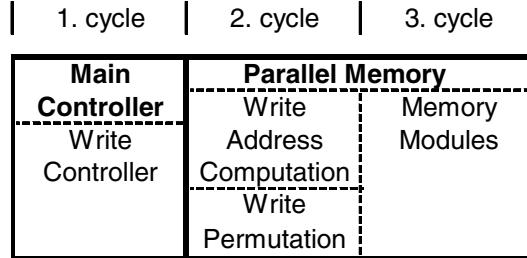


Fig. 10. Write pipeline.

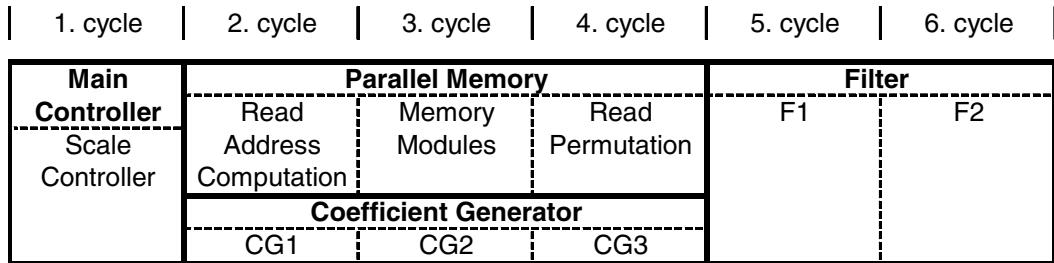


Fig. 11. Scale pipeline.

The scale pipeline (Fig. 11) has six cycle latency when no stalling is required. The Parallel Memory unit uses three stages, because the data cannot be permuted simultaneously with the Read Address Computation unit. In parallel with the Parallel Memory unit, the Coefficient Generator unit determines the weighting factors within three pipeline stages CG1, CG2, and CG3. The stages five and six are for the Filter unit including the pipeline stages F1 and F2.

7. Results

The downscaler has been coded in synthesizable register transfer level VHDL. The system has been implemented on Altera Stratix FPGA [20] and verified against a floating point software implementation. Fig. 12 depicts the verification environment. Two Nios soft processors [21] are used to control the Ethernet and downscaler interfaces. An external host PC computer transfers raw video data via Ethernet to the implemented downscaler. The downscaled video is moved back to the host PC and compared to the reference scaled video.

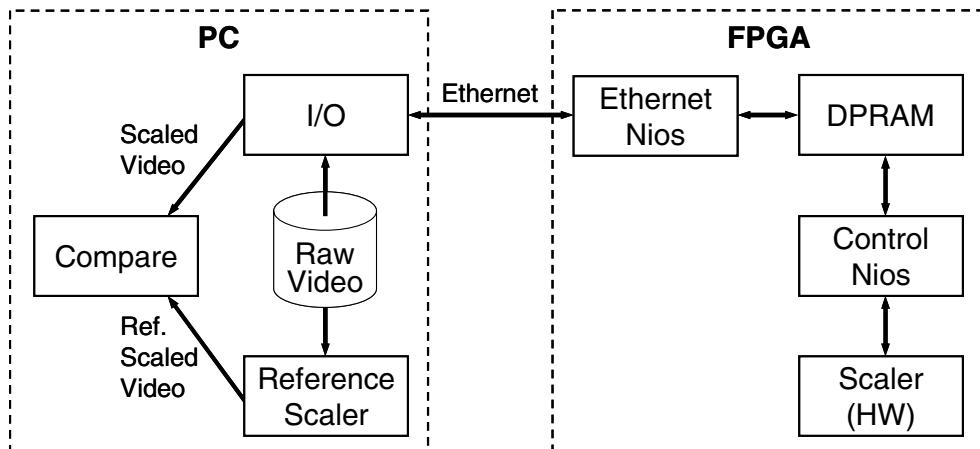


Fig. 12. Verification environment.

Table 2. Area in CMOS technology (16VGA -> SXGA).

Unit	Gate Count
Main Control	650
Parallel Memory	550
Coefficient Generator	3900
Filter	3120
Pipeline	2280
Total	10500

The purpose of the verification is to test downscaler functionality in relatively real environment and with actual delays. However, the verification environment is not specially optimized for the downscaler since the environment is intended to suit for testing several kinds of hardware accelerators. Due to limited Nios performance and Ethernet throughput, maximum performance cannot be tested with this environment.

7.1. Downscaler complexity

As noted, the original and scaled image sizes as well as the coefficient precision are determined at design time. In the analyzed implementation, the original image size is 16VGA (2560 x 1920) and the scaled image size is SXGA (1280 x 1024). The precision in weighting factor calculation is restricted so that minimum complexity with sufficient accuracy is achieved. Due to integer pixel values, the minimum possible error is +/- 1 that is also aimed to be the maximum error per pixel with the proposed implementation. With several bit widths, the hardware scaling results are compared to the accurate reference floating point software implementation results. This leads 12 bits accuracy for the values in the ROM of the Coefficient Generator unit as well as 13 bits precision for weighting factor calculation. With these bit widths, the maximum error is +/- 1 per pixel and the mean absolute error is 0.15 per pixel.

The used resources in Altera Stratix EP1S40F780C5 logic device are 371 logic cells, 8 embedded multipliers (18 bits wide) and 131,072 memory bits. That makes 1% of the total logic cell amount, 14% of the embedded multipliers, and 4% of the memory in the used logic device. The maximum clock frequency for the implementation is 105 MHz. For reference, a 32-bit Nios processor with a timer and a serial connection (UART) peripherals utilizes 2316 logic cells [22].

In addition, the area results based on logic synthesis are given for 0.18-micron CMOS process technology. The gate count is based on 2-input NAND gates and tabulated in Table 2. The gate count of the implementation includes all the logic, registers, and ROMs but the dual port memory is excluded. Most of the complexity is caused by the arithmetic multiplier and adder units which are mainly located in the Coefficient Generator and Filter units. The Main Control and Parallel Memory units are primarily built from simple logic gates. The pipeline includes all the pipeline registers. The total gate count is 10500.

7.2. Implementation comparison

Five scaling implementations are compared in Table 3. Unfortunately, they have different features making a fair comparison difficult. However, these are the most relevant designs found in the literature. The complexity and quality of the used scaling algorithms differ between the scalers. Moreover, some implementations can only scale either up or down whereas others can scale to both directions. The utilized color format is not available in all the references. Two ASIC and three FPGA implementation results are shown. The used resources in the designs are represented with gate count and memory consumption, where the input and output image widths are denoted by W_i and W_o ,

respectively. The used clock frequency is also tabulated when available. The achieved performance is shown as the rate (frames/second) in which the original image size is scaled to the target size.

The two ASIC implementations do not report a gate count ([12], [13]). To ease the comparison, the area results have been converted to a gate count estimate. However, the equivalent gate count depends, for instance, on the semiconductor vendor. Routing and the efficiency of placement also introduce some uncertainty. According to ITRS Roadmap 2003 [23], the area of a 2-input NAND gate equals to $320F^2$ where F is defined as the minimum feature size. Therefore, a 2-input NAND gate with 0.5- and 0.8-micron processes corresponds to $80.0 \mu\text{m}^2$ and $204.8 \mu\text{m}^2$, respectively. These values are used to get the maximum values for [12] and [13] in Table 3.

However, the area in [12] includes routing in addition to logic whereas the situation is not known for [13]. To give a reference point, two widely used cryptography algorithm (DES and AES) are implemented with 0.18-micron CMOS technology in [24]. On average, the area after place and route is shown to be 40% larger than the area given by logic synthesis. The wire versus logic ratio of the cryptography implementations is much higher than in normal designs, which increases the difference [24]. Despite the roughness of this method, the minimum gate amounts of the two ASIC implementations [12] and [13] are estimated with this proportionality factor.

The implementation in [12] uses a novel and high quality adaptive algorithm for scaling luminance component (Y). The chrominance components (U, V) are scaled with bilinear interpolation. The scaler in [13] utilizing cubic spline interpolation gives better quality than bilinear interpolation [2].

The design in [11] utilizes Altera EPF10K50 FPGA device [26]. The consumed resources include 2160 logic cells for logic operations and memory consumption of 24 embedded array blocks. The gate count is not given in the reference. The used FPGA device contains typically 50000 gates [26]. 33000 gates can be estimated by diminishing the embedded array block proportion from the typical gate count and taking the proportional logic cell usage level into account. The other way to estimate the logic amount is to use the given typical gate count 12 in a logic cell [26]. With 2160 logic cells, that equals to 25920 gates. The scaling ratios are restricted to 4/5, 9/10, and 24/25 in [11]. The used method in [11] is shown to give slightly better quality than bilinear interpolation based scaling [11].

A new scaling algorithm, winscale, is used in [8]. With downscaling, the winscale algorithm has been shown to give exactly same results as bilinear [25]. However, the good edge characteristics of the winscale algorithm make it useful when scaling up text or other fine edge images [8]. The hardware structure of winscale algorithm is a bit more complex, but similar to bilinear.

Our implementation is dedicated to YUV 4:2:0 video downscaling with design-time configurable image sizes. With the original image size 16VGA and scaled image size SXGA, the achieved performance is 50 frames/second. With less than half of the gate count of the second smallest implementation, the achieved performance is considerably better and with larger images than the other implementations.

Table 3. Scaling implementation comparison.

	Scaling algorithm	Scaling direction	Color format	Tech.	Gate count	Memory [bytes]	Clk freq. [MHz]	Original image	Scaled image	Frames/second
[12]	Adaptive (Y), Bilinear (U, V)	Up	YUV 4:2:0	ASIC, 0.5u	123000-172000	$5W_i+6W_o$	130	144x176 (QCIF)	576x704 (4CIF)	30
[13]	Cubic spline	Up/down	-	ASIC, 0.8u	87000-122000	-	-	720x480 (NTSC)	720x576 (PAL)	30
[11]	Cubic and Bisigmoidal	Up	RGB	FPGA	26000-33000	$6W_i$	46	720x576 (PAL)	800x600	25
[8]	Winscale	Up/down	-	FPGA	29000	$4W_i$	65	640x480 (VGA)	1024x768 (XGA)	-
Prop.	Bilinear	Down	YUV 4:2:0	FPGA	10500	$4W_i$	105	2560x1920 (16VGA)	1280x1024 (SXGA)	50

As mentioned, winscale and bilinear are similar in terms of hardware complexity and resulting image quality. Therefore, the design in [8] and the proposed implementation are compared in more detail in the following. In [8], image sizes can be changed at run time and pixel moving operations are implemented with six adders. However, the restricted adder accuracy produces accumulated error for weighting factor calculation and pure filter operations. Therefore, up to 30-bit filter coefficients are used to obtain sufficient precision [8]. Respectively, we define image sizes and compute a lot of information to ROM with accurate floating point arithmetic at design time. Therefore, 13-bit coefficient accuracy is sufficient in the proposed implementation. Moreover, ROM consumption is diminished by utilizing evenly divisible image properties.

7.3. Discussion

The gate count of the proposed implementation depends on the original and scaled image sizes. With larger image sizes, larger ROMs and some wider operations are needed. Moreover, the ROM contents also influence the gate count since ROM is implemented with combinatorial logic. In any case, the most area consuming arithmetic units remain the same. The gate count varies at maximum 15% depending on the image sizes.

The achieved performance (frames/second) also varies based on the original and scaled image sizes. Smaller images have fewer pixels to handle resulting in increased performance. Moreover, the maximum clock frequency increases with smaller image sizes up to 10%. As another example implementation, with the original image size QUXGA (3200 x 2400) and the downscaled image size HDTV (1920 x 1088), real-time scaling (30 frames/second) is achieved with the complexity of 11300 gates.

8. Conclusion

Video scaling quality varies depending on the used interpolation method. Therefore, a trade-off between quality and complexity has to be made. In several applications, design-time fixed scaling ratios are sufficient. This has been utilized in the proposed fine-grained parallel downscaling implementation. With the evenly divisible image concept, an even more efficient system has been designed. With contemporary FPGAs, the implemented design is able to scale down large resolution color videos in real-time.

In the future, the implemented downscaler will be used as a part of a high performance MPEG-4 video encoder [27]. The video captured by a camera is downscaled to the encoding resolution with the proposed implementation. Several Nios processors as well as hardware accelerators will be implemented on an FPGA.

Acknowledgements

This research was financially supported by the Academy of Finland (grant 104487), Nokia Foundation, Heikki and Hilma Honkanen Foundation, and Graduate School in Electronics, Telecommunications and Automation (GETA).

References

- [1] Sony, HDR-FX1 HDV Features, [Online], Available: www.sony.com.
- [2] T. M. Lehmann, C. Gönner, K. Spitzer, Survey: Interpolation methods in medical image processing, IEEE Transactions on Medical Imaging 18 (11) (1999) 1049-1075.
- [3] J. A. Parker, R. V. Kenyon, D. E. Troxel, Comparison of interpolating methods for image resampling, IEEE Transaction on Medical Imaging MI-2 (1) (1983) 31-39.

- [4] E. Maeland, On the comparison of interpolation methods, *IEEE Transactions on Medical Imaging*, 7 (3) (1988) 213–217.
- [5] E. H. W. Meijering, K. J. Zuiderveld, M. A. Viergever, Image reconstruction by convolution with symmetrical piecewise n th-order polynomial kernels, *IEEE Transactions on Image Processing* 8 (2) (1999) 192-201.
- [6] R. Dugad, N. Ahuja, A fast scheme for image size change in the compressed domain, *IEEE Transactions on Circuits and Systems for Video Technology* 11 (4) (2001) 461-474.
- [7] H. Sarbazi-Azad, M. Ould-Khaoua, L. M. Mackenzie, A parallel algorithm for Lagrange interpolation on the cube-connected cycles, *Microprocessors and Microsystems* 24 (2000) 135–140.
- [8] C.-H. Kim, S.-M. Seong, J.-A. Lee, L.-S. Kim, *Winscale*: An image-scaling algorithm using an area pixel model, *IEEE Transactions on Circuits and Systems for Video Technology* 13 (6) (2003) 549–553.
- [9] J. R. Nottingham, Hardware-efficient system for hybrid-bilinear image scaling, U.S. Patent 6,252,576, June 2001.
- [10] A. Ramaswamy, Y. Nijim, W. B. Mikhael, Polyphase implementation of a video scalar, in: Conference Record of the Asilomar Conference on Signals, Systems & Computers, Pacific Grove, CA, USA, November 1997, pp. 1691-1694.
- [11] F. Tao, X. Wen-Lu, and Y. Lian-Xing, An architecture and implementation of image scaling conversion, in: Proceedings of the International Conference on ASIC, Shanghai, China, October 2001, pp. 409–410.
- [12] A. Raghupathy, N. Chandrachoodan, K. J. R. Liu, Algorithm and VLSI architecture for high performance adaptive video scaling, *IEEE Transactions on Multimedia* 5 (4) (2003) 489–502.
- [13] L. A. de Barros Naviner, VLSI architecture for real-time moving image resampling, in: Proceedings of the IEEE Midwest Symposium on Circuits and Systems, Rio de Janeiro, Brazil, August 1995, pp. 1260-1263.
- [14] E. Aho, J. Vanne, T. D. Hääläinen, K. Kuusilinna, Block-level parallel processing for scaling evenly divisible images, *IEEE Transactions on Circuits and Systems I* 52 (12) (2005) 2717-2725.
- [15] R. Li, S. Levi, An arbitrary ratio resizer for MPEG applications, *IEEE Transactions on Consumer Electronics* 46 (3) (2000) 467-473.
- [16] P. Budnik, D. J. Kuck, The organization and use of parallel memories, *IEEE Transactions on Computers* C-20 (12) (1971) 1566–1569.
- [17] E. Aho, J. Vanne, K. Kuusilinna, T. D. Hääläinen, Address computation in configurable parallel memory architecture, *IEICE Transactions on Information and Systems* E87-D (7) (2004) 1674–1681.
- [18] J. K. Tanskanen, T. Sihvo, J. Niittylahti, Byte and modulo addressable parallel memory architecture for video coding, *IEEE Transactions on Circuits and Systems for Video Technology* 14 (11) (2004) 1270-1276.
- [19] M. Gössel, B. Rebel, R. Creuzburg, *Memory Architecture & Parallel Access*, Elsevier Science B.V, Amsterdam, 1994.
- [20] Altera, Stratix Device Handbook, Volume 1, Version 3.2, January 2005.
- [21] Altera, Nios 3.0 CPU Data sheet, Version 2.2, October 2004.
- [22] E. Salminen, A. Kulmala, T. D. Hääläinen, HIBI-based Multiprocessor SoC on FPGA, in: Proceedings of the IEEE International Symposium on Circuits and Systems, Kobe, Japan, May 2005, pp. 3351-3354.

- [23] Semantech, International Technology Roadmap for Semiconductors 2003 Edition System Drivers, Report, 2003.
- [24] I. Papaefstathiou, V. Papaefstathiou, C. Sotiriou, Design-space exploration of the most widely used cryptography algorithms, *Microprocessors and Microsystems* 28 (2004) 561-571.
- [25] E. Aho, J. Vanne, K. Kuusilinna, T. D. Hämäläinen, Comments on “*Winscale*: An image-scaling algorithm using an area pixel model”, *IEEE Transactions on Circuits and Systems for Video Technology* 15 (3) (2005) 454-455.
- [26] Altera, FLEX 10K Data sheet, Version 4.1, January 2003.
- [27] O. Lehtoranta, E. Salminen, A. Kulmala, M. Hännikäinen, T. D. Hämäläinen, A parallel MPEG-4 encoder for FPGA based multiprocessor SoC, in: Proceedings of the International Conference on Field Programmable Logic and Applications, Tampere, Finland, August 2005, pp. 380-385.



Eero Aho received the M.S. degree in Electrical Engineering from Tampere University of Technology (TUT), Finland in 2001. His research interests include memory systems and parallel processing. Currently he is working toward his PhD degree as a research scientist at the Institute of Digital and Computer Systems (DCS) at TUT.



Jarno Vanne received the M.S. degree in Information Technology from Tampere University of Technology, Finland in 2002. His research interests include memory systems and hardware accelerators. Currently he is working toward his PhD degree as a research scientist at the Institute of Digital and Computer Systems at TUT.



Timo D. Hämäläinen, MSc '93, PhD '97, TUT. He acted as a senior research scientist in the DCS at TUT in 1997-2001 and during the time founded a research group named "Parallel DSP processing and wireless multimedia systems." He has acted as a project manager and research supervisor for several academic and industrial projects. In 2001 he has taken the position of full professor at TUT and continues the research on wireless local and personal area networking as well as SoC solutions for wireless video.



Kimmo Kuusilinna, PhD '01, Tampere University of Technology. His main research interests include system-level design and verification, interconnection networks, and parallel memories. Currently he is working as a senior research engineer at the Nokia Research Center.

Tampereen teknillinen yliopisto
PL 527
33101 Tampere

Tampere University of Technology
P.O. Box 527
FIN-33101 Tampere, Finland