Avoiding Conversion and Rearrangement Overhead in SIMD Architectures

Asadollah Shahbahrami

Avoiding Conversion and Rearrangement Overhead in SIMD Architectures

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus Prof. dr. ir. J.T. Fokkema,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen

op maandag 15 september 2008 om 15:00 uur

door

Asadollah SHAHBAHRAMI

Master of Science in Computer Engineering-Machine Intelligence,
Shiraz University, Shiraz, Iran
geboren te Chaloos, Mazandaran, Iran

Dit proefschrift is goedgekeurd door de promotor:

Prof. dr. K.G.W. Goossens

Copromotor: Dr. B.H.H. Juurlink

Samenstelling promotiecommissie:

Rector Magnificus	voorzitter
Prof. dr. K.G.W. Goossens	Technische Universiteit Delft, promotor
Dr. B.H.H. Juurlink	Technische Universiteit Delft, copromotor
Prof. dr. ir. H.J. Sips	Technische Universiteit Delft
Prof. dr. ir. A.J. van der Veen	Technische Universiteit Delft
Dr. K. Flautner	ARM Ltd., Cambridge
Dr. A. Ramirez	Universitat Politècnica de Catalunya, Barcelona
Prof. dr. ir. G.J.M. Smit	Universiteit Twente
Prof. dr. ir. R.L. Lagendijk, reservelid	Technische Universiteit Delft

My first promotor Professor Stamatis Vassiliadis† has provided substantial guidance and support for this thesis.

Shahbahrami, Asadollah
Avoiding Conversion and Rearrangement Overhead in SIMD Architectures
Computer Engineering Laboratory
Delft University of Technology
Keywords: SIMD Architectures, Vectorization, SIMD Programming, Multimedia Application, Cache Optimization.

ISBN 978-90-807957-9-2
Cover page: Sketch design of an SIMD unit by Author.

Typeset by the author with the LaTeX Documentation system.

Author email: A.Shahbahrami@TUDelft.nl, shahbahrami@guilan.ac.ir
Printed in The Netherlands

This dissertation is dedicated to all my teachers and family with gratitude and love

Avoiding Conversion and Rearrangement Overhead in SIMD Architectures

Asadollah Shahbahrami

Abstract

In this dissertation, a novel SIMD extension called Modified MMX (MMMX) for multimedia computing is presented. Specifically, the MMX architecture is enhanced with the extended subwords and the matrix register file techniques. The extended subwords technique uses SIMD registers that are wider than the packed format used to store the data. It uses 32 bits extra for each 64-bit register. The extended subwords technique avoids data type conversion overhead and increases parallelism in SIMD architectures. This is because promoting the subwords of the source SIMD registers to larger subwords before they can be processed and demoting the results again before they can be written back to memory incurs conversion overhead. The matrix register file technique allows to load data that is stored consecutively in memory into a column of the register file, where a column corresponds to the corresponding subwords of different registers. In other words, this technique provides both row-wise as well as column-wise accesses to the media register file. It is a useful approach for matrix operations that are common in multimedia processing. In addition, in this work, new and general SIMD instructions addressing the multimedia application domain are investigated. It does not consider an ISA that is application specific. For example, special-purpose instructions are synthesized using a few general-purpose SIMD instructions. The performance of the MMMX architecture is compared to the performance of the MMX/SSE architecture for different multimedia applications and kernels using the `sim-outorder` simulator of the SimpleScalar toolset. Additionally, three issues related to the efficient implementation of the 2D Discrete Wavelet Transform (DWT) on general-purpose processors, in particular the Pentium 4, are discussed. These are 64K aliasing, cache conflict misses, and SIMD vectorization. 64K aliasing is a phenomenon that happens on the Pentium 4, which can degrade performance by an order of magnitude. It occurs if two or more data items whose addresses differ by a multiple of 64K need to be cached simultaneously. There are also many cache conflict misses in the implementation of vertical filtering of the DWT, if the filter length exceeds the number of cache ways. In this dissertation, techniques are proposed to avoid 64K aliasing and to mitigate cache conflict misses. Furthermore, the performance of the 2D DWT is improved by exploiting the data-level parallelism using the SIMD instructions supported by most general-purpose processors.

Asadollah Shahbahrami Delft, The Netherlands, 2008

Abbreviations

	Full Name	Description
ASIC	Application-Specific Integrated Circuit	An integrated circuit that implements a specific function.
CISC	Complex Instruction Set Computers	CISC is an instruction set architecture in which each instruction consists of many microcode and take many clock cycles to execute.
CPU	Central Processing Unit	A unit that executes the programs.
DCT	Discrete Cosine Transform	The DCT is a transform to convert image or video pixels from the time domain to the frequency domain.
DLP	Data-Level Parallelism	DLP is a technique to execute a large number of operations by a single instruction.
DMPs	Dedicated Multimedia Processors	DMPs are typically custom designed architectures intended to perform specific multimedia functions.
DSPs	Digital Signal Processors	DSPs are microprocessors, which have specifically been designed for digital signal processing.
DWT	Discrete Wavelet Transform	The DWT provides a time-frequency representation of image or video signals.
FIR	Finite Impulse Response	FIR filters are digital filters that have an impulse response which reaches zero in a finite number of steps.
FP	Floating-Point	FP presents a numerical representation system for real numbers.
FPGA	Field Programmable Gate Array	An FPGA is a reprogrammable hardware device that can be used to implement arbitrary circuits.
GPPs	General-Purpose Processors	Processors that are designed to execute a variety of applications. GPPs have a higher degree of flexibility than other processors such as DSPs.
HDTV	High Definition TeleVision	HDTV is the new standard in television technology which enhances the quality of the picture on the screen.
IDCT	Inverse Discrete Cosine Transform	The IDCT is the inverse of the DCT, which converts the transformed image to the time domain.
ILP	Instruction-Level Parallelism	ILP is a technique to execute several instructions in each cycle by exploiting the independent instructions.
ISA	Instruction Set Architecture	ISA includes the set of instructions of either a particular processor or a family of processors.
JPEG	Joint Photographic Experts Group	The committee that has developed the JPEG and JPEG2000 standards.

LBWT	Line-Based Wavelet Transform	The LBWT is a traversal technique that is used to implement the 2D discrete wavelet transform. In this technique the vertical filtering starts as soon as a sufficient number of lines, as determined by the filter length, has been horizontally filtered.
LUT	Look-Up Table	A LUT is a group of memory cells, which consists of all the possible results of a function for a given set of its input values.
MDMX	MIPS Digital Media eXtension	MDMX is a SIMD extension unit developed for the MIPS family of processors.
MMA	MultiMedia Application	Multimedia applications use and process different media elements including text, graphics, images, audio, 2D and 3D animation, and video.
MMX	Multi-Media Extensions	MMX is a multimedia extension, provided on the Intel microprocessors, which consists of 64-bit integer SIMD instructions on packed elements.
MMMX	Modified Multi-Media Extensions	The MMMX architecture is MMX enhanced with extended subwords, the matrix register file, and a few general-purpose instructions that are not present in MMX.
MPEG	Motion Picture Experts Group	The committee that has developed the MPEG compression standards.
MRF	Matrix Register File	The MRF is a media register file that provides both row-wise as well as column-wise access to the register file.
NSPs	Native Signal Processing	NSP is an enhancement to a GPP to process multimedia data.
RCWT	Row-Column Wavelet Transform	The RCWT is a traversal technique that is used to implement the 2D discrete wavelet transform. In the RCWT approach, the 2D DWT is divided into two 1D DWTs, namely horizontal and vertical filtering.
RISC	Reduced Instruction Set Computer	RISC is opposite of CISC. RISC represents a microprocessor design strategy that reduces chip complexity by using simpler instructions, removing microcode layer, than the CISC design.
RUU	Register Update Unit	The RUU determines which instruction should be issued to the functional units for execution.
SAD	Sum-of-Absolute Differences	The SAD function is a similarity measurement algorithm that is usually used in motion estimation algorithms to remove temporal redundancies between video frames.
SIMD	Single-Instruction Multiple-Data	Computation concept of executing the same instruction on multiple data elements.

SLP	Subword Level Parallelism	SLP is a form of DLP that packs several small data elements into a media register in order to process them simultaneously.
SPE	Synergistic Processing Element	SPEs are SIMD processors with local stores. The Cell processor contains 8 SPEs.
SPI	Special-Purpose Instruction	Special-purpose instructions are provided in order to accelerate some specific functions.
SPU	Synergistic Processing Unit	Each SPE of Cell processor has an SPU. SPU includes a 256KB local memory and two SIMD datapaths, and a 128x128b register file.
SSD	Sum-of-Squared Differences	The SSD function is a similarity measurement algorithm that is usually used in motion estimation algorithms to remove temporal redundancies between video frames.
SSE	Streaming SIMD Extensions	SSE is another multimedia extension that provides floating-point SIMD instructions on packed elements.
TLP	Thread-Level Parallelism	TLP is a technique to execute multiple threads of an application or multiple applications at once.
VIS	Visual Instruction Set	VIS is a multimedia instruction set extension designed by Sun and implemented on the UltraSPARC processor.
VLIW	Very Long Instruction Word	VLIW is a technique to execute many operations in a single instruction.
VMX	Vector Multimedia eXtension	VMX consists of 162 PowerPC instructions that target multimedia applications, and was co-developed by IBM, Motorola, and Apple.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

MultiMedia Applications (MMAs) have been becoming one of the most prominent workloads in computer systems [41, 93]. They are used in many environments ranging from desktop systems to mobile systems. There are a variety of multimedia algorithms for capturing, manipulating, storing, and transmitting multimedia objects such as text, handwritten data, image, video, graphics, and audio objects [53, 116, 18, 93, 88, 36]. Multimedia standards such as MPEG-1/2/4/7, JPEG, JPEG2000, and H.263/4 put challenges on hardware architectures for executing different multimedia algorithms efficiently. This is because MMAs are associated with multiple standards and multiple formats. However, the efficient processing of MMAs is currently one of the main challenges in the media processing field.

Different architectures have been proposed to process MMAs ranging from fully custom to domain-specific architectures, and to General-Purpose Processors (GPPs) with multimedia extensions. None of them, however, can provide high-performance with programmability. The main reason is that the dynamic nature of MMAs has been not matched well with the ability of the existing architectures. In this thesis, architectural enhancements for GPPs equipped with Single-Instruction Multiple-Data (SIMD) extensions are proposed to provide much more performance than and the same programmability compared to existing multimedia extensions such as MMX and SSE.

The purpose of this chapter is to provide a brief overview of recent architectural approaches for multimedia processing and state a number of challenges for GPPs enhanced with multimedia extensions, which will be addressed in this dissertation. Section 1.1 presents an overview of multimedia characteristics. Section 1.2 describes different classifications of processors that have been proposed for processing MMAs and they are compared to each other in Section 1.3. Section 1.4 evaluates some

Operand size	Usage Frequency
8-bit	40%
16-bit	51%
32-bit	9%

Table 1.1: *Different data types that are used by multimedia data [52].*

Operation type	Percentage
ALU	40%
Load/Store	26-27%
Branch	20%
Shift	10%
Integer Mult.	2%
Floating point	3-4%

Table 1.2: *Operations distribution that are needed to implement the multimedia algorithms [52].*

SIMD architectures using multimedia kernels on the Pentium 4 processor, in order to determine their bottlenecks. Section 1.5 presents dissertation challenges, and finally, Section 1.6 gives an overview of the different chapters of this thesis.

1.1 Characteristics of Multimedia Applications

In this section, the characteristics of MMAs are briefly discussed.

MMAs have certain characteristics that make them different from other applications, for example, scientific benchmarks [41, 107, 7, 8, 50, 51]. The most important ones are as follows. First, MMAs typically contain a significant amount of Data-Level Parallelism (DLP). This means that multimedia algorithms perform the same operations on different data items. Second, most of the execution time of MMAs is spent in a few small loops or kernels. Third, multimedia data is usually narrow. For example, image and video pixels can be represented in 8-bit. Table 1.1 depicts the distribution of operand sizes used in MMAs. It shows that most operands are smaller than or equal to 16-bit. Finally, many multimedia algorithms process two-dimensional (2D) data, process data along the rows as well as along the columns.

Additionally, MMAs perform significantly more fixed-point operations than floating-point operations. Table 1.2 depicts the distribution of the operations needed to implement MMAs. As this table shows, the overall usage of an integer ALU that can perform arithmetic operations, compares, logic operations, and moves is about 40%. Furthermore, MMAs have high spatial locality but little temporal locality. Typically, the processor loads a small amount of data, processes it, and it never or rarely reuses the data again. Based on these characteristics, MMAs require different architectures than other applications.

In order to understand the limitations of existing architectures for processing MMAs, they are investigated in the next section.

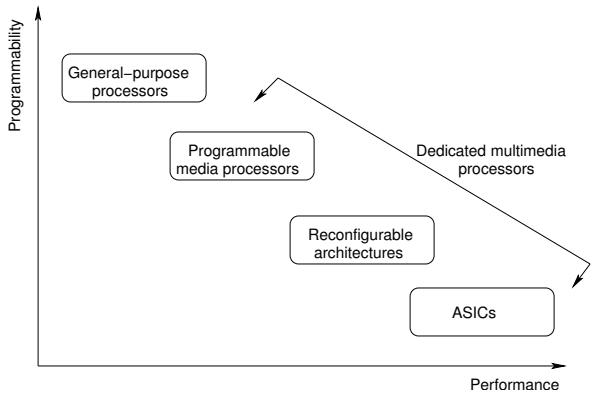


Figure 1.1: *Different proposed architectures for processing of MMAs.*

1.2 Processor Architectures to Support MMAs

Many architectures ranging from application-specific processors to domain-specific processors have been proposed to process MMAs [107, 8, 83, 80]. A number of programmable Digital Signal Processors (DSP) have been used since 1980. They support specific instructions such as the multiply-accumulate (MAC) instruction, in order to improve both performance and programmability. Hen [60] has given a summary of characteristics of early DSPs as well as recent DSPs. Hen classified them into four groups based on different implementation methods: DSP chip, DSP core, multimedia DSPs, and Native Signal Processing (NSP) instruction set processors. Multimedia DSPs are specifically designed for audio/video applications. One example of this group is the Trimedia TM 1300 [43, 119]. The NSP processors extend the instruction set of a GPP to process multimedia data.

Other researchers [36, 104, 37, 53, 123] have provided different classifications of the media processors. All proposed architectures can be divided into two categories, Dedicated Multimedia Processors (DMPs) and GPPs enhanced with multimedia extensions. DMPs can also be divided into three groups, Application Specific Integrated Circuits (ASICs), reconfigurable architectures, and programmable media processors. This classification is depicted in Figure 1.1 in the performance-programmability space. Each architecture is briefly discussed in the following sections.

1.2.1 Dedicated Multimedia Processors (DMPs)

DMPs are typically custom designed architectures intended to perform specific multimedia functions such as video and audio compression and decompression, and 2D and 3D graphics applications. DMPs can be divided into ASICs, reconfigurable ar-

chitectures, and programmable media processors.

The ASIC implementation is a direct mapping of a multimedia algorithm to hardware. The implemented hardware is optimized to execute that specific algorithm. Matching the individual hardware modules to the processing requirements results in area-efficient implementations. The ASIC design is usually used to accelerate specific multimedia algorithms such as the Discrete Cosine Transform (DCT), quantization, entropy encoding, and motion estimation, while a host processor takes care of the main control.

Reconfigurable architectures [16] offer a compromise between the performance advantages of ASICs and the flexibility of programmable architectures. Reconfigurable architectures are able to directly implement specialized functions in hardware and also contain functional resources that can be modified. However, reconfiguration involves an additional cost of time and power.

Programmable media architectures can be divided into flexible programmable architectures, which provide high flexibility, and adapted programmable architectures, which provide higher efficiency but less flexibility. These architectures can support a complete MMA. There are different mechanisms in the design of programmable architectures for achieving high-performance such as DLP, Instruction-Level Parallelism (ILP), and Thread-Level Parallelism (TLP) or adaptation to special algorithm characteristics by implementing specialized instructions and dedicated hardware modules that result in higher efficiency for a limited application field [44].

Advanced dedicated multimedia processors use Very Long Instruction Word (VLIW) architectural schemes to exploit a high degree of ILP [77]. This is because VLIW architectures have many advantages compared to superscalar processors. For example, VLIW processors employ static instruction scheduling performed at compile-time rather than dynamic scheduling performed at run-time as in superscalar processors, which requires much more hardware [45]. Furthermore, hardware does not need to determine which instructions can be issued in parallel. One example of this group is Philips' TM1000 [43, 119]. This architecture contains a VLIW processor, as well as a video and audio I/O subsystem. The processor has an instruction set that is optimized for processing audio, video, and graphics.

Other researchers [87, 28, 79, 32, 75] have proposed some dedicated programmable architectures for the multimedia domain. Lee et al. [87] have shown that a vector architecture is a cost-effective solution for MMAs because these applications exhibit a large amount of DLP. An ISA extension called Complex Streamed Instructions (CSI) for increasing parallelism by processing 2D data streams has been proposed in [28]. This ISA extension has several advantages. First, CSI does not put an architectural limitation on the number of subwords that are processed in parallel, because CSI processes data streams of arbitrary length. Thus, the number of bits or data elements that

are processed in parallel is not visible to the programmer. Second, CSI minimizes the overhead caused by data misalignment by performing alignment in hardware. CSI also eliminates loop control instructions, because CSI processes 2D streams of arbitrary length. The instructions represent the overhead necessary to put data in a format suitable to SIMD operations, these are called overhead instructions, such as *packing/unpacking* and data *re-shuffling* instructions.

Matrix registers with accumulators are introduced in the Matrix-Oriented Multimedia (MOM) ISA [32, 33]. The MOM architecture combines traditional pipelined vector processing with subword processing. It relies on having a vector register file where every element contains subwords that are processed in parallel. It supports stride-n access, where every element is loaded separated by an n-byte gap. Two key features distinguish MOM from CSI. First, MOM is a register-to-register architecture that uses sectioning when the data do not fit into the MOM registers. Second, MOM requires overhead instructions for data conversion.

Another related dedicated architecture for processing MMAs is the Imagine processor [75, 103], which has a load/store architecture for 1D streams of data records. Imagine is a stand-alone multimedia coprocessor. The focus of the Imagine project is to develop a programmable architecture for graphics and image/signal processing.

1.2.2 GPPs Enhanced with Multimedia Extension

In order to increase the performance of MMAs, GPPs vendors have extended their ISAs. These ISA extensions use the Subword Level Parallelism (SLP) concept [89]. A subword is a smaller precision unit of data contained within a word. In SLP, multiple subwords are packed into a word and then whole word is processed. SLP is used in order to exploit DLP with existing hardware without sacrificing the general-purpose nature of the processor. SLP provides a very low-cost form of small-scale SIMD parallelism, which is called microSIMD in [91], in a word-oriented processor. This is because there is no need to replicate the functional units, and the memory port can supply multiple elements at no additional cost. In addition, SLP is a form of vector processing. A register is viewed as a small vector with elements that are smaller than the register size. This requires small data types and wide registers.

As mentioned previously, multimedia kernels process small data types, and the registers of GPPs satisfy these requirements. In particular, the double-precision Floating-Point (FP) registers can hold several of such elements. The same operation is applied to the different subwords simultaneously.

In SLP, a word-wide functional unit is partitioned into parallel subword functional units, with small hardware overhead. As illustrated in Figure 1.2, a 64-bit ALU can be partitioned into four 16-bit ALUs. Such a partitionable ALU allows either four

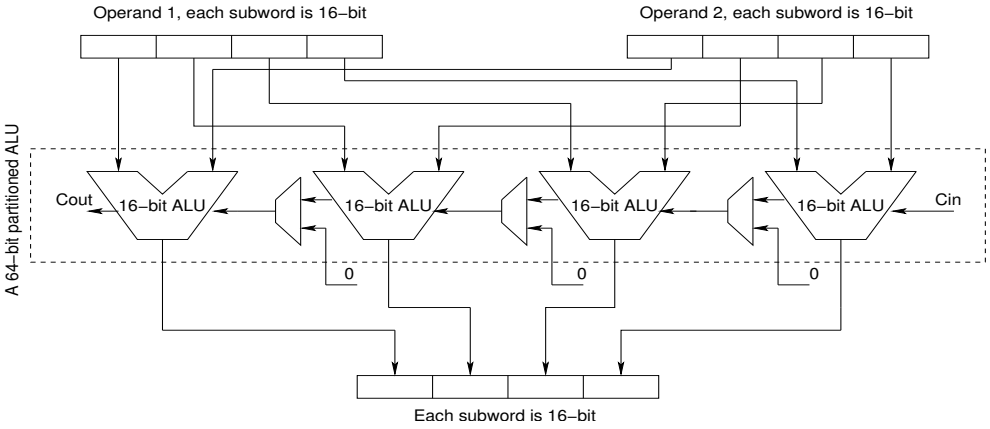


Figure 1.2: *A 64-bit partitioned ALU that is divided into four parallel functional units using the subword level parallelism concept.*

16-bit, two 32-bit ALU operations, or a single 64-bit ALU operation to be performed in a single clock cycle. The overhead is very small since the same datapaths are used in all cases. Furthermore, unlike VLIW and superscalar processors, SLP does not require additional ports to the register file. A processor with two 64-bit partitionable ALUs could support eight parallel 16-bit operations with just a 6-ported (4 read and 2 write ports) register file, while a processor with eight independent 16-bit functional units requires a 24-ported register file.

The first multimedia extensions are Intel's MMX [106, 105], Sun's Visual Instruction Set (VIS) [140], Compaq's Motion Video Instructions (MVI) [10], MIPS Digital Media eXtension (MDMX) [57, 72], and HP's Multimedia Acceleration eXtension (MAX) [89, 90]. These extensions supported only integer data types and were introduced in the mid-1990's. 3DNow [2] was the first to support floating-point media instructions. It was followed by Streaming SIMD Extension (SSE) and SSE2 from Intel [111, 139]. Motorola's AltiVec [118, 42] supports integer as well as floating-point media instructions. In addition, high-performance processors also use SIMD processing. An excellent example of this is the Cell processor [49, 62, 67] developed by a partnership of IBM, Sony, and Toshiba. Cell is a heterogeneous chip multiprocessor consisting of a PowerPC core that controls eight high-performance Synergistic Processing Elements (SPEs). Each SPE has one SIMD computation unit that is referred to as Synergistic Processor Unit (SPU). Each SPU has 128 128-bit registers. SPUs support both integer and floating-point SIMD instructions.

The main differences between these multimedia extensions are the following. First, they reconfigure the internal register file structure different from each other to accommodate microSIMD operations. Second, they choose and add different multimedia instructions in their ISA. Multimedia instruction set can be broadly categorized ac-

GPP with Multimedia Extension ISA Name	**AltiVec/VMX**	**MAX-1/2**	**MDMX**	**MMX/ 3DNow**	**VIS**	**MMX/ SIMD**	**SSE**	**SSE2**	**SPU ISA**
Company	Motorola/IBM	HP	MIPS	AMD	Sun	Intel	Intel	Intel	IBM/Sony/Toshiba
Instruction set	Power PC	PARISC2	MIPS-V	IA32	P. V.9	IA32	IA64	IA64	-
Processor	MPC7400	PA RISC	R1000 PA8000	K6-2	Ultra Sparc	P2	P3	P4	Cell
Year	1999/2002	1995	1997	1999	1995	1997	1999	2000	2005
Datapath width	128-bit	64-bit	64-bit	64-bit	64-bit	64-bit	128-bit	128-bit	128-bit
Size of register file	32x128b	(31) /32x64b	32x64b	8x64b	32x64b	8x64b	8x128b	8x128b	128x128b
Dedicated or shared with	Dedicated	Int. Reg.	FP Reg.	Dedicated	FP Reg.	FP Reg.	Dedicated	Dedicated	Dedicated
Integer data types:									
8-bit	16	-	8	8	8	8	8	16	16
16-bit	8	4	4	4	4	4	4	8	8
32-bit	4	-	-	2	2	2	2	4	4
64-bit	-	-	-	-	-	-	-	2	2
Shift right/left	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Multiply-add	Yes	No	No	Yes	Yes	Yes	Yes	Yes	Yes
Shift-add	No	Yes	No	No	No	No	No	No	No
Floating-point	Yes	No	Yes	Yes	No	No	Yes	Yes	Yes
Single-precision	4x32	-	2x32	4x16 2x32	-	-	4x32	4x32	4x32
Double-precision	-	-	-	1x64	-	-	-	2x64	2x64
Accumulator	No	No	1x192b	No	No	No	No	No	
# of instructions	162	(9) 8	74	24	121	57	70	144[1]	213
# of operands	3	3	3-4	2	3	2	2	2	2/3/4
Sum of absolute-differences	No	No	No	Yes	Yes	No	Yes	Yes	Yes
Modulo addition/ subtraction	8, 16, 32	16	8, 16	8, 16 32	16, 32	8, 16 32, 64	8, 16 32, 64	8, 16 32,64	8, 16 32,64
Saturation addition/ subtraction	U8, U16, U32 S8, S16, S32	U16, S16	S16	U8, U16 S8, S16	No	U8, U16 S8, S16	U8, U16 S8, S16	U8, U16 S8, S16	- -

Table 1.3: *Summary of available multimedia extensions. Sn and Un indicate n-bit signed and unsigned integer packed elements, respectively. Values n without a prefix U or S in the last row, indicate operations work for both signed and unsigned values.* [1] *Note that 68 instructions of the 144 SSE2 instructions operate on 128-bit packed integer in XMM registers, wide versions of 64-bit MMX/SSE integer instructions.*

cording to the location and geometry of the register file upon which microSIMD instructions operate. The alternatives are reusing the existing integer or floating point register files, or implementing an entirely separate one. The type of the register file affects the width and therefore the number of packed elements that can be operated on simultaneously (vector length). Despite the similarities, each approach to subword extensions is unique [72]. Key differences include the amount of additional hardware required, ranging from MAX-2, which reuses the integer registers and execution units and requires virtually no additional execution hardware, to AltiVec, which requires an entirely new execution unit. Table 1.3 summarizes the common and distinguishing features of existing multimedia instruction set extensions [8, 60, 129, 47].

1.3 A Comparison Between Processor Architectures for MMAs

In this section, different processor architectures for MMAs are compared based on the metrics programmability, performance, and cost.

Architectures	Performance	Flexibility	Power	Cost	Density	Design effort
ASIC	High	Low	Low	High	Medium	High
Reconfigurable hardware	Medium	High	High	Medium	Medium	Medium
Dedicated media processors	Medium	High	Medium	Medium	Medium	Medium
GPPs with multimedia extensions	Low	High	Medium	Low	High	Low

Table 1.4: *Comparison of different architectures for multimedia processing [60, 137].*

Various metrics have been developed to compare the quality of different media processor implementations. For example, flexibility has been considered as one of the key advantage in media processors since it allows changes to system functionality at various points in the design life cycle. Table 1.4 compares different solutions for multimedia processing. The ASIC approaches offer the advantages of high-performance and low power, but their design and debugging phases involve a significant amount of time. Because the development cost cannot be spread across multiple applications, the cost of ASICs are generally higher than, for example, conventional microprocessor-based solutions. In addition, they are suitable only for specific functions, and future extensions are not possible without redesigning the hardware.

Reconfigurable architectures are more flexible than ASIC designs, while their power consumption is high. Dedicated media architectures provide dedicated modules for several multimedia tasks, but they are not suitable for multiple standards and multiple formats of media applications. They have high-performance compared to GPPs enhanced with multimedia extensions but they have narrow applicability. GPPs equipped with media ISA extensions are more flexible than other architectures, but their performance is lower. One main reason why their performance is lower than other architectures is because they incur many overhead instructions. For example, Ranghanathan et al. [112] have shown that the implementations of the MPEG/JPEG codecs using the VIS ISA require on average 41% overhead instructions

In this dissertation, some (micro-)architectural enhancements are proposed to avoid overhead instructions and to exploit more DLP than existing multimedia extensions such as MMX and SSE can. Subword level parallelism is used, a concept that has already been used by microSIMD extensions. Exploiting SLP requires mapping the algorithm to the partitioned ALUs in such a way that the maximum number of subwords are executed in parallel, while the time for overhead instructions must not waste the speedup achieved by the partitioned ALUs.

SLP is more cost efficient than other parallel architectures such as Multiple Instruction, Multiple Data (MIMD), macroSIMD (compared to microSIMD), superscalar processor, and VLIW architectures. This is because of the following reasons.

First, an MIMD architecture consists of multiple processors, and each processor can execute a different instruction in each clock cycle. Each processor has its own register file. For example, for four processors, four instructions must be issued for 4-way

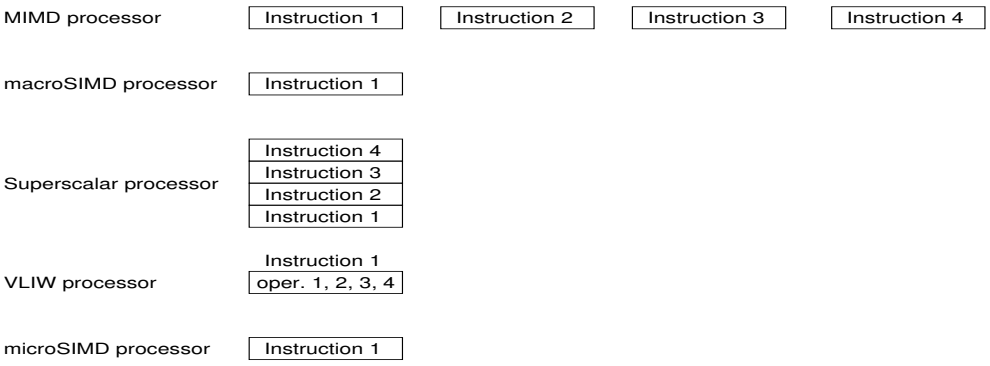


Figure 1.3: *Instructions needed per cycle to provide 4-way parallelism [91].*

parallelism. An interconnection network is needed to transfer data between the processors. Second, a macroSIMD architecture has the same datapaths as an MIMD architecture, except that a single instruction is issued to all the processors in a single clock cycle. Third, in the superscalar architecture, the register file is shared between m parallel ALUs. In each clock cycle, at most n different instructions are issued for n-way parallelism, where $n \leq m$. Finally, the VLIW architectures is almost the same as a superscalar architecture except that only a single instruction is issued in each clock cycle, while this single instruction consists of up to n different operations for n parallel ALUs to provide n-way parallelism.

Figure 1.3 shows the number of instructions that need to be issued in order to achieve 4-way parallelism in the different parallel architectures. In addition, Table 1.5 shows the approximate area of the register files of the different architectures to support 4-way parallelism on 16-bit elements. Each register file has 32 16-bit registers. Both MIMD and macroSIMD architectures have four register files, with 128 registers in total. Their area requirements are proportional to the total number of bits in all four register files, with an overhead of d per register file, and an addressing overhead of e per register. The microSIMD architecture can hold the same number of 16-bit operands in one quarter of the number of registers, since these are packed as four 16-bit subwords in one 64-bit register. Hence, it has slightly smaller area requirements due to area overhead for the registers and register files than the MIMD or macroSIMD architectures.

In the rest of this thesis, the word "SIMD" is used instead of "microSIMD". In the next section, the performance of some SIMD architectures such as the MMX and SSE extensions is evaluated using multimedia kernels on the Pentium 4 processor.

Parallel Architecture	# of Register Files	Total # of Registers	Width of Register	Max. Number of 16-bit Operands	Approximate Area for all Registers
MIMD	4	128	16-bit	128	F(4*32*16)+4(d+32e)
MacroSIMD	4	128	16-bit	128	F(4*32*16)+4(d+32e)
Superscalar	1	32	16-bit	32	F(32*16)+d+32e
VLIW	1	32	16-bit	32	F(32*16)+d+32e
MicroSIMD	1	32	64-bit	128	F(4*32*16)+d+32e

Table 1.5: *Storage capacity and area requirements with fixed number of bits per register address. Number of registers per register file is 32 registers. "d": overhead per register file, "e": addressing overhead per register [91].*

1.4 An Evaluation of SIMD Architectures Using Multimedia Kernels

In order to identify the bottlenecks of existing SIMD architectures, some important multimedia kernels have been implemented using MMX and SSE and their performance was measured on the Pentium 4. The selected kernels are the sum-of-absolute differences (SAD) [111], SAD with interpolation, SAD for histogram similarity measurement [39], sum-of-squared differences (SSD) [144], color space conversions (RGB-to-YCbCr and YCbCr-to-RGB) [125], matrix transpose for integers (Transp. (int)) and FP numbers (Transp. (real)) [122], Paeth prediction [114], 2D DCT, $(5, 3)$ lifting scheme [135], and Daubechies' transform with four coefficients [141] (Daub-4) were selected. It is important to note that the last three kernels process data in both horizontal and vertical directions. This means that these algorithms consist of both horizontal filtering, process data along the rows and vertical filtering, process data along the columns. This implies that in order to employ SIMD instructions, the matrix needs to be transposed frequently. All kernels were implemented using the MMX architecture except Transp. (real) and both horizontal filtering and vertical filtering of the Daub-4 transform, which were implemented using the SSE architecture.

In the following sections, the methodology and metrics and the results obtained on the Pentium 4 processor are discussed. Finally, the performance bottlenecks are determined.

1.4.1 Methodology and Metrics

Two versions of each kernel were implemented: one in C and one in assembly using MMX and SSE. The different versions of each kernel employ the same algorithm and data types. Each program consists of three parts, for reading the input data, for performing the computation, and for storing the calculated data. Only the computation part was implemented in MMX and SSE and only the time taken by this part is

Processor	Intel Pentium 4
CPU Clock Speed	3.0GHz
L1 Data Cache	8 KBytes, 4-way set associative, 64 Bytes line size
L2 Cache	512 KBytes, 8-way set associative, 64 Bytes line size, On Chip

Table 1.6: *Parameters of the experimental platform.*

reported.

The reasons why the assembly language has been used for the SIMD programming of multimedia kernels are the following. First, the assembly language is the most effective technique because it may produce the required performance gain. Second, as indicated by Kuroda et al. [83], the efficient programming of processors with multimedia extensions can only be attained if experts tune their software using assembly language, just as in DSP approaches. Third, the goal is to determine the impact of SIMD instructions on the performance and not the performance of the intermediate tools. Additionally, multimedia kernels are usually small functions and their implementations by assembly language is not so difficult.

All C programs were compiled using gcc with optimization level *-O2*. As experimental platform a 3.0GHz Pentium 4 processor was employed. The main architectural parameters of this system are summarized in Table 1.6.

All programs were executed on a lightly loaded system. The number of cycles was obtained using the IA-32 cycle counter [70]. Cycle counters provide a very precise tool for measuring the time that elapses between two different points in the execution of a program [17, 131]. In order to eliminate the effects of context switching and compulsory cache misses, the *K-best* measurement scheme and a *warmed up* cache were used [17]. That means that the function was repeatedly (K times) executed, and the fastest time is reported. Executing the function at least once before starting the measurement minimizes the effects of both instruction and data cache misses. The speedup was measured by the ratio of execution cycle count for the computational part of each kernel and this metric formed the basis of the comparative study in this thesis.

1.4.2 Analysis of Results

Figure 1.4 depicts the speedup of the MMX and SSE implementations of the multimedia kernels over the C implementation on the Pentium 4 processor. As can be seen the speedup of the MMX/SSE implementation of the SAD kernel is the largest due to the Special-Purpose `psadbw` Instruction (SPI) [111]. The speedup for other similarity measurements (SAD with interpolation, SAD for histogram similarity mea-

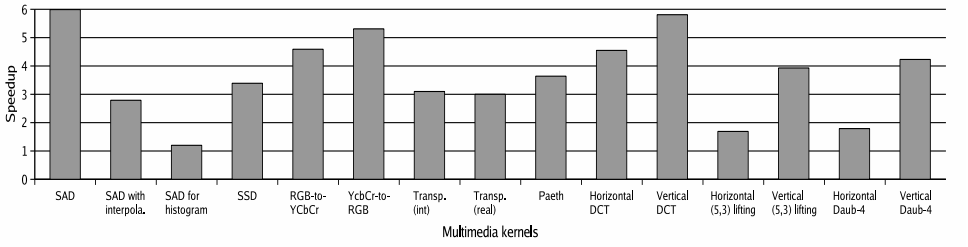


Figure 1.4: *Speedup of the MMX and SSE implementations of the multimedia kernels over the scalar versions on the Pentium 4 processor.*

surements, and SSD) is less than for the SAD kernel. This is because there are no SPIs for these functions. In addition, the speedup of the RGB-to-YCbCr color space conversion is less than the speedup of the YCbCr-to-RGB color space conversion. The reason for this is that in the former kernel more overhead instructions are required than in the latter kernel. Furthermore, the performance improvements of the SIMD implementations of the vertical filtering of the DCT, $(5, 3)$ lifting, and Daub-4 are larger than the implementations of the corresponding horizontal filtering phases of these kernels. This is because under the row-major image layout, it is easier and more efficient to vectorize vertical filtering than horizontal filtering.

In general, multimedia extensions provide significant performance benefits for multimedia kernels as is shown in Figure 1.4 and also by other researchers [112, 130]. Existing extensions, however, have a number of bottlenecks that limit the performance improvement. In the next section, some of these bottlenecks are determined.

1.4.3 Performance Bottlenecks

SIMD extensions generally provide two kinds of SIMD instructions. The first are the SIMD computational instructions such as arithmetic instructions. The second are the SIMD overhead instructions that are necessary for data movement, data type conversions, and data reorganization. The latter instructions are needed to bring data in a form amenable to SIMD processing. These instructions constitute a large part of the SIMD codes. For example, Ranghanathan et al. [112] indicated that the SIMD implementations of the MPEG/JPEG codecs using the VIS ISA require on average 41% overhead instructions such as packing/unpacking and data re-shuffling. In addition, the dynamic instructions count of the EEMBC consumer benchmarks running on the Philips TriMedia TM32 shows that over 23% of instructions are data alignment instructions such as pack/merge bytes (16.8%) and pack/merge half words (6.5%) [61]. The execution of this large number of the SIMD overhead instructions decreases the performance and increases pressure on the fetch and decode steps.

To illustrate where overhead instructions are needed in the SIMD implementations of multimedia kernels, two motivational examples are shown in Figure 1.5. To the left, the MMX implementation of the RGB-to-YCbCr color space conversion is shown and to the right the SSE implementation of horizontal filtering of the Daub-4 transform is shown. In addition, the Figure 1.5 in the middle shows the different steps in the processing of multimedia data. The main reasons to select these kernels are as follows. The image data in the color space conversion is usually interleaved. SIMD vectorization of kernels that use interleaved data is difficult due to the fact that multimedia extensions provide access only to continuous elements. As already mentioned, the Daub-4 transform consists of two 1D transforms, horizontal and vertical filtering, as do the other 2D transforms. To vectorize the horizontal filtering, the matrix needs to be transposed frequently. Transposition takes a significant amount of time, however. For example, Figure 1.6 shows how to transpose a 4×4 block of single-precision floating-point values using SSE instructions. As this figure shows, the first two low-order and two high-order values of rows 0 and 2, and 1 and 3 are unpacked. The obtained results are also unpacked. For this operation eight load/store, eight unpcklps/unpckhps, and four data movement instructions are required. In other words, 20 SIMD instructions are needed to transpose a matrix of size 4×4 as depicted in Figure 1.7. As a result, vectorizing such applications efficiently is a challenge in SIMD architectures.

As Figure 1.5 shows, data reordering and data type conversion instructions are used in Steps 3, 4, and 6 in the MMX implementation after loading the input data and before storing the outputs. The overhead instructions are used in Step 3 in the SSE implementation to transpose a block. In the MMX code, the number of overhead instructions is 41 in each loop iteration, while the number of the SIMD instructions in the processing stage (Step 5) is 78. This means that the number of overhead instructions is significant compared to the number of SIMD instructions in the processing stage. Consequently, it is important either to eliminate, to alleviate, or to overlap these instructions with other SIMD instructions. In addition, 30 instructions in the processing stage of the color space conversion are data movement instructions between registers and memory. This is because there are not enough registers to keep the temporary results and coefficients. Therefore, data has to be frequently loaded or stored from or to memory. Since the MMX and SSE architectures have only eight architectural media registers that is not sufficient to implement the multimedia kernels efficiently.

1.5 Dissertation Challenges

As indicated earlier, many data type conversion and data rearrangement instructions are needed to implement MMAs using existing SIMD architectures. The main reason

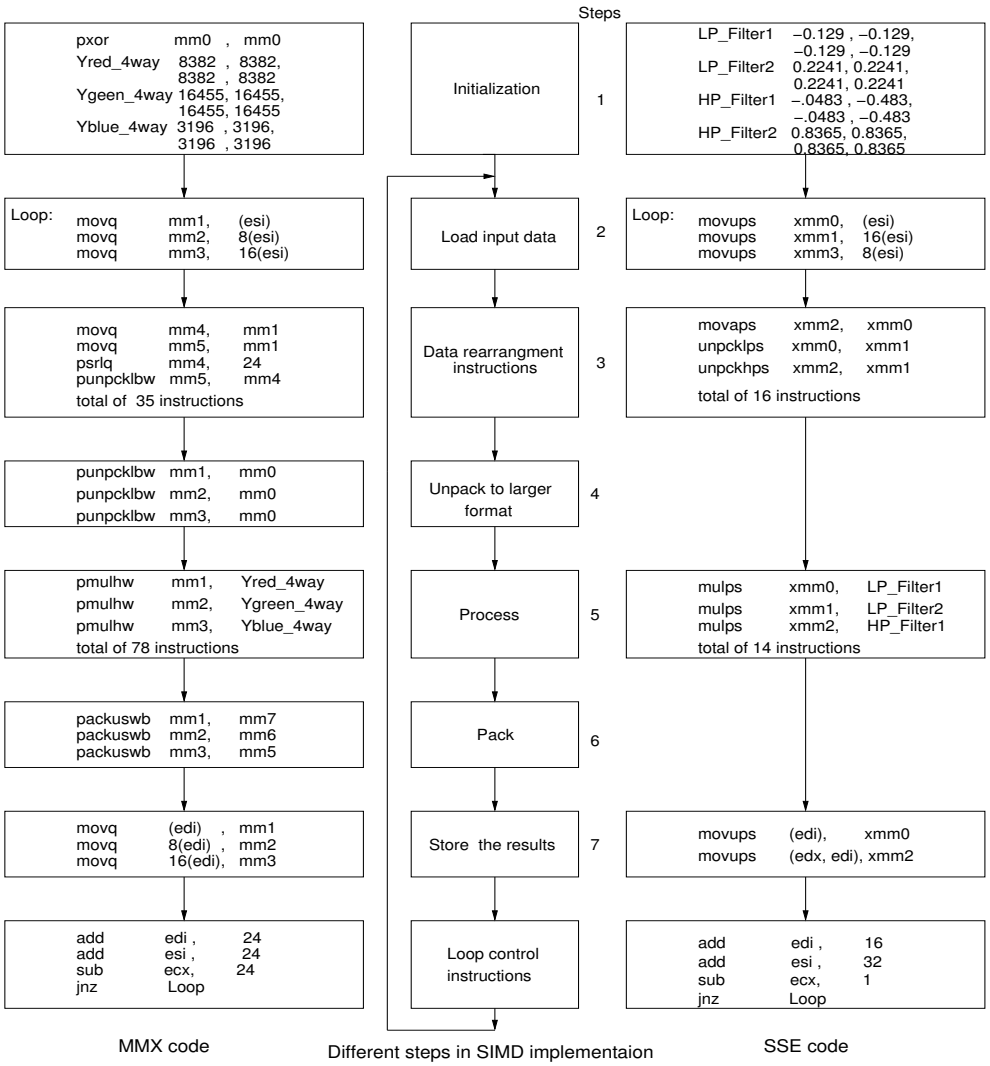


Figure 1.5: *Illustration of where overhead instructions are used in the MMX implementation of the RGB-to-YCbCr kernel and the SSE implementation of the horizontal filtering of the Daub-4 transform.*

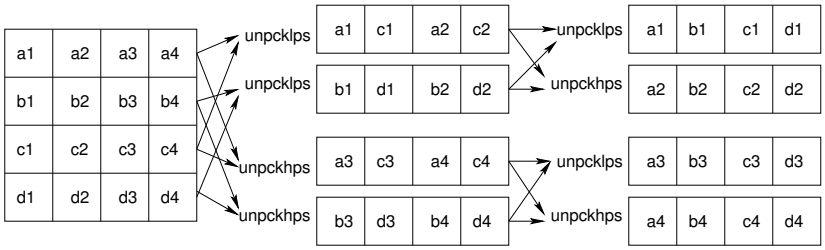


Figure 1.6: *Matrix transpose of a* 4×4 *block using SSE instructions.*

`movaps`	`xmm0,`	`(blk1)`	; xmm0 =	a4	a3	a2	a1
`movaps`	`xmm1,`	`16(blk1)`	; xmm1 =	b4	b3	b2	b1
`movaps`	`xmm2,`	`32(blk1)`	; xmm2 =	c4	c3	c2	c1
`movaps`	`xmm3,`	`48(blk1)`	; xmm3 =	d4	d3	d2	d1
`movaps`	`xmm4,`	`xmm0`	; xmm4 =	a4	a3	a2	a1
`movaps`	`xmm6,`	`xmm1`	; xmm6 =	b4	b3	b2	b1
`unpcklps`	`xmm0,`	`xmm2`	; xmm0 =	c2	a2	c1	a1
`unpcklps`	`xmm1,`	`xmm3`	; xmm1 =	d2	b2	d1	b1
`movaps`	`xmm5,`	`xmm0`	; xmm5 =	c2	a2	c1	a1
`unpcklps`	`xmm0,`	`xmm1`	; xmm0 =	d1	c1	b1	a1
`unpckhps`	`xmm5,`	`xmm1`	; xmm5 =	d2	c2	b2	a2
`unpckhps`	`xmm4,`	`xmm2`	; xmm4 =	c4	a4	c3	a3
`unpckhps`	`xmm6,`	`xmm3`	; xmm6 =	d4	b4	d3	b3
`movaps`	`xmm7,`	`xmm4`	; xmm7 =	c4	a4	c3	a3
`unpcklps`	`xmm4,`	`xmm6`	; xmm4 =	d3	c3	b3	a3
`unpckhps`	`xmm7,`	`xmm6`	; xmm7 =	d4	c4	b4	a4
`movaps`	`(blk2),`	`xmm0`	; (blk2) =	d1	c1	b1	a1
`movaps`	`16(blk2),`	`xmm5`	; 16(blk2) =	d2	c2	b2	a2
`movaps`	`32(blk2),`	`xmm4`	; 32(blk2) =	d3	c3	b3	a3
`movaps`	`48(blk2),`	`xmm7`	; 48(blk2) =	d4	c4	b4	a4

Figure 1.7: *Illustration of the SSE instructions to transpose a* 4×4 *block.*

Multimedia Extension	Number of Instructions
VIS	106
MAX/MAX2	64
AltiVec	24
MMX/SSE	88

Table 1.7: *The number of instructions needed to transpose an* 8×8 *block on the different multimedia extensions, each element of the block is two bytes.*

for this is that the requirements of MMAs do not match the abilities of GPPs enhanced with SIMD extensions. This is for the following reasons:

- There is a mismatch between the computational format and the storage format of multimedia data. The precision of the intermediate results are usually larger than the storage format. Consequently, data type conversion instructions such as unpacking are required before operations are performed and the results also have to be packed before they can be stored back to memory. As a result, performance is lost due to the execution of data type conversion instructions, and fewer subwords can be processed in parallel. These operations are shown in Steps 4 and 6 in Figure 1.5.
- Existing SIMD computational instructions cannot efficiently exploit DLP of the 2D multimedia data. As already mentioned, 2D multimedia algorithms such as the 2D Discrete Wavelet Tranform (DWT) and 2D (I)DCT consist of two 1D transforms called horizontal and vertical filtering. The horizontal filtering processes the rows while vertical filtering processes the columns. SIMD vectorization of the vertical filtering is straightforward, since the corresponding data of each column are adjacent in memory. Therefore, several columns can be processed without any rearranging of the subwords. For horizontal filtering on the other hand, corresponding elements of adjacent rows are not continuous in memory. In order to employ SIMD instructions, data rearrangement instructions are needed to transpose the matrix. This step takes a significant amount of time. For example, transposing an 8×8 block of bytes, requires 56 MMX/SSE instructions, if the elements are two bytes wide, then 88 instructions are required as shown in Table 1.7. This table depicts the number of instructions needed to transpose an 8×8 block for different multimedia extensions. This was shown in Step 3 in Figure 1.5 for horizontal filtering of the Daub-4 transform.
- Vector instructions of conventional SIMD extensions execute the same operations on multiple data that is adequately packed in vector registers. Computations, on the other hand, may execute the same operations on multiple interleaved data. SIMD memory architectures typically provide access to contiguous memory items. This means that multimedia extensions cannot load or

store strided data. Therefore, vectorizing interleaved data requires many rearrangement instructions to allow for parallel computation, which can lead to actual performance degradation. This was shown in Step 3 in Figure 1.5 for color space conversion.

- Special-purpose instructions such as the SAD instruction have limited usefulness except for the specific kernels they were designed to accelerate. This has several drawbacks. First, if the SAD becomes obsolete because a different similarity metric is employed, then the SAD SPI is no longer useful. For example, MIPS' MDMX [72] provides no SAD SPI but advocates using the SSD instead. Second, as indicated in [85], the complex CISC-like semantics of SPIs makes automatic code generation difficult. Third, the SAD SPI only supports the packed byte data type. While useful for the SAD kernel used in motion estimation, this precision is not sufficient for multimedia kernels such as motion estimation in the transform domain or for cost functions used in image and video retrieval [94]. Finally, since these instructions process eight 8-bit subwords, they are most useful if the vector length is a multiple of 8. In the H.264 standard, however, variable block sizes, for instance 8×4 and 4×4 are used [136].

All of the above limitations are critical to efficient SIMD implementation of multimedia applications. Solving these limitations is the scope of this dissertation. In order to reach this goal, a novel SIMD ISA extension called the Modified MMX (MMMX) is proposed. The MMMX architecture uses the *extended subwords* and the *Matrix Register File* (MRF) techniques. Extended subwords use registers that are wider than the packed format used to store the data. Extended subwords avoid data type conversion instructions. The MRF allows to load data stored consecutively in memory to a column of the register file, where a column corresponds to corresponding subwords of different registers. This technique avoids the need of data rearrangement instructions. The proposed architecture is validated by studying the performance of a wide range of multimedia kernels and applications. The performance obtained by the MMMX architecture is compared with that achieved by the MMX/SSE architectures.

In addition, it was found that the implementation of the 2D DWT on the Pentium 4 suffers from the following problems.

- 64K aliasing, which occurs when two data blocks need to be cached simultaneously whose addresses differ by a multiple of 64K. This can degrade performance by an order of magnitude. Two code transformation techniques, loop fission and offsetting, are proposed to avoid 64K aliasing.

- Cache conflict misses. Cache performance can be improved by applying loop interchange, but there will still be many conflict misses if the filter length exceeds the cache associativity. Therefore, two code transformation techniques,

associativity-conscious loop fission and lookahead, are proposed to reduce the number of conflict misses.

The proposed techniques are implemented on actual machines and their performance is compared to an optimized reference implementation.

An overview of how the research challenges have been addressed and how they are presented in this dissertation follows.

1.6 Structure of the Thesis

This dissertation contains six chapters.

Chapter 2 gives a brief discussion about the background information related to data type conversion, data permutation instructions, SIMD vectorization, and cache optimization. In some SIMD architectures data type conversion instructions are used to convert the smaller data type to a larger data type. In addition, in this chapter different techniques to deal with data rearrangement are explained. For example, providing pre-defined data permutation instructions, designing a separate permutation unit, and changing the architecture of the media register file are such techniques. Furthermore, some techniques for SIMD vectorization and cache optimization that have been proposed by other researchers are described.

Chapter 3 describes the proposed MMMX architecture. First, extended subwords and the matrix register file techniques are discussed in detail. Second, the new load/store SIMD instructions, ALU, and multiplication instructions are presented. Third, the main differences between the MMMX and MMX architectures are explained. Finally, the hardware cost of the MMMX architecture is discussed.

Chapter 4 describes the performance evaluation of the proposed architecture using multimedia kernels and applications. Some important MMAs such as MPEG-2 and JPEG are selected. These applications are profiled in order to find the most time consuming kernels, and those media kernels are implemented using the MMMX and MMX/SSE architectures. The performance of the MMMX architecture is compared to the MMX/SSE architectures at kernel-, image-, and application-level.

Chapter 5 discusses the implementation of the 2D DWT on SIMD-enhanced general-purpose processors. First, the different algorithms to implement the 2D DWT are explained. Second, the memory behavior of the vertical filtering is analyzed in order to identify the problems of 64K aliasing and cache conflict misses. Third, the proposed techniques to avoid 64K aliasing and to alleviate the cache conflict misses are discussed. Finally, the SIMD vectorization of the 2D DWT and its performance improvement are explained.

The thesis ends with Chapter 6, in which the conclusions, the contributions, and future work are described.

Chapter 2

Background

A register file consists of an array of registers that participates as a part of the Central Processing Unit (CPU). The register file provides the source operands and stores the calculated results of most instructions. In traditional GPPs, a register can hold a single data, while in the GPPs enhanced with multimedia extensions, a register can hold many packed data items and is called a media register. Consequently, a media register file can be viewed as an $N \times M$ matrix, where N is the number of registers and M is the maximum number of subwords in each register. Since a media register file has capacity of $N \times M \times n$ bits, where n is the number of bits for the smallest subword. For example, in the MMX extension $N = 8$, $M = 8$, and $n = 8$.

The previous chapter has shown that in the SIMD implementations of many multimedia kernels, an n-bit subword is sometimes not sufficient to keep the intermediate results. Then the n-bit subword should be unpacked to $2 \times n$-bit subword. This means that data type conversion instructions are needed to unpack a subword to a subword that is twice as wide. In addition, the previous chapter has discussed that data permutation instructions are used because of the data reordering within and between media registers. Therefore, SIMD architectures have provided different SIMD instructions to deal with data type conversion and data permutation instructions.

In this chapter, the background information on data type conversion instructions, data rearrangement techniques, SIMD vectorization, and cache optimization techniques are discussed. Section 2.1 describes the data type conversion instructions. Section 2.2 explains different techniques for data reordering. Section 2.3 explains different techniques for SIMD vectorization of multimedia applications. Section 2.4 describes the techniques that are used for cache optimization. Finally, concluding remarks are presented in Section 3.4.

Instructions	Description
paddb	packed add byte, any carry out of the sum is lost.
psubb	packed difference byte, any borrow out of a difference is lost.
pavgb	average packed unsigned byte integers.
pmaxub	packed unsigned byte maximum.
pminub	packed unsigned byte minimum.
paddsb	packed add signed saturation byte ($-128 \leq x \leq 127$).
psubsb	packed difference signed saturation byte ($-128 \leq x \leq 127$).
paddusb	packed add unsigned saturation byte ($0 \leq x \leq 255$).
psubusb	packed difference unsigned saturation byte ($0 \leq x \leq 255$).

Table 2.1: *The MMX instruction set to process 8-bit data type.*

2.1 Data Type Conversion

This section first explains the different data type conversion instructions that have been provided by some SIMD architectures and then it discusses how those instructions can be avoided by using a larger precision.

2.1.1 Data Type Conversion Instructions

Some multimedia extensions have provided 8×8-bit SIMD ALU and saturation instructions for 8-way parallelism. Saturation clips the results of an arithmetic operation to the range that can be represented by the data type. In other words, the result of an operation that exceeds the range of a data type, saturates to the maximum value of the range. On the other hand, if a result that is less than the range of a data type then it saturates to the minimum value of the range. Table 2.1 depicts the MMX instructions that support the byte data type. These instructions support three types of arithmetic operations. First, wraparound operations that truncate the calculated result to the result register size. Second, signed saturation that saturates the obtained result to either the largest positive byte or the negative byte depending on the result. Finally, unsigned saturation that saturates the result to either the largest unsigned byte or zero.

Image and video data is typically stored as packed 8-bit elements, but intermediate results usually require more than 8-bit precision. As a consequence, most 8-bit SIMD ALU instructions are wasted. In the SIMD extensions, if the packed subwords are filled by a maximum value representable using the subword data type, the choice is either to be imprecise using saturation operation at every stage, or to loss parallelism by unpacking to a larger format. Using saturation instructions produce unexpected results. This is because saturation is usually used at the end of computation. It is more precise to saturate once at the end of the computation rather than at every step of the algorithm. For instance, adding three signed 8-bit values $120 + 48 - 10$, using

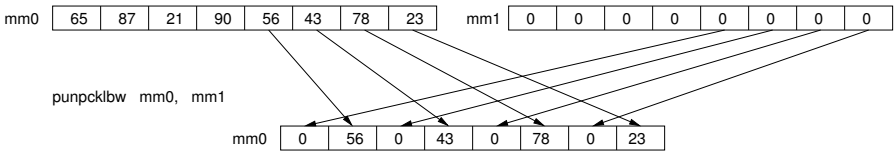


Figure 2.1: *Illustration of the* `punpcklbw mm0, mm1` *instruction.*

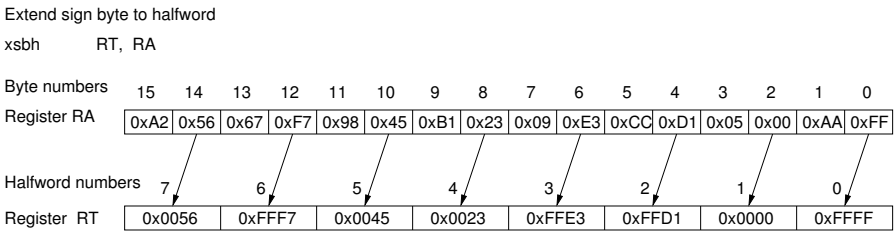


Figure 2.2: *An example of the* extend sign byte halfword *instruction that has been provided in the synergistic processor unit of the Cell processor [67].*

signed saturation at every step produces 117 and using signed saturation at the last step produces 127.

SIMD architectures support different packing, unpacking, and extending instructions to convert the different data types to each other. For example, the MMX/SSE architectures provide `packss{wb,dw,wb}` and `punpck{hbw,hwd,hdq,lbw,lwd,ldq}` instructions for data type conversions. The `punpckl{bw,wd,dq}` (unpack low packed data) instructions interleave the low-order data subwords of the destination and source operands into the destination operand. If the source operand has a value of all zeros, the result is a zero extension of the low order subwords of the destination operand. Figure 2.1 illustrates an example for the `punpcklbw` instruction. However, these instructions do not support sign extension and the programmer should carefully consider this problem.

The AltiVec extension and the ISA of the Cell SPE, on the other hand, provide extend instructions to convert from a smaller data type to a larger data type, which have one source operand and have the capability of sign extension. For instance, the Cell SPE provides three such instructions. The first is *extend sign byte halfword* that propagates the sign of the right byte of the source register to the left byte. The resulting 16-bit signed integer is stored in the destination operand. Figure 2.2 illustrates this instruction.

The second is *extend sign halfword to word* instruction, where the sign of the halfword in the right half of the source operand is extended to the left halfword. The 32-bit signed integer result is stored in the destination operand. The third data type conversion instruction is *extend sign word to doubleword*. In this instruction the sign

of the word in the right slot is propagated to the left word and the resulting 64-bit integer is placed in the destination operand.

In addition, the AltiVec extension supports pack instructions for 8-bit and 16-bit signed and unsigned saturation. These instructions concatenate two source operand registers, producing a single result of either sixteen bytes or eight halfwords for 8- and 16-bit data types, respectively. Additionally, there are some merge instructions in the AltiVec ISA for byte, halfword, and word data type. For example, the halfword SIMD merge instructions interleave the four low halfwords or four high halfwords of two source operands and produce a result of 8 halfwords.

2.1.2 Avoiding Data Type Conversion

Some architectures provide a larger precision to avoid the use of data type conversion instructions. For example, some DSP processors such as TMS320C64x/C64x+ DSP [138] and MIPS' MDMX extension have wide accumulators. In the C64x and C64x + DSPs two 32-bit registers are used to hold a 40-bit value. The MIPS' MDMX extension uses a 192-bit accumulator. This 192-bit register can be partitioned into eight 24-bit values or four 48-bit values. It is mainly used for MAC operations, which are common in many signal processing algorithms.

The ISA of the Cell SPE has 128-bit registers. This allows to use a computational format of, e.g., 16-bit when the storage format is 8-bit. In fact, the Cell SPE does not provide arithmetic instructions for the packed byte data type except for SPIs such as average bytes and absolute differences. This is because as was mentioned in Chapter 1, this data type is not sufficient for computational format. The SPE processor of the Cell has provided explicit addition/subtraction and multiply instructions for 16-, 32-bit and 16-bit data types, respectively. In this thesis, however, it is shown that 12 and 24 bits are sufficient for many media kernels and, therefore, the additional four and eight bits are not needed.

Slingerland and Smith have also proposed an idea of inserting four extra bits to each byte of a register [130], where it is called fat subwords technique. However, they have not evaluated this technique. Furthermore, as shown in Chapter 1 many 2D media algorithms process data along the rows as well as along the columns, and the fat subwords technique is not sufficient for those kernels.

2.2 Data Rearrangement

This section describes different approaches to permute data within a register and across registers.

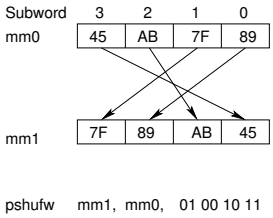


Figure 2.3: *Illustration of the packed shuffle word instruction of the SSE architecture.*

Generally, there are three approaches to deal with data rearrangement. First, the data is reordered within the registers by instructions provided in the ISA. Second, the data is reorganized during memory operations. Finally, the data is rearranged using a special register file organization. These approaches are discussed in detail in the following sections.

2.2.1 Explicit Instructions

Some SIMD architectures such as MIPS' MDMX, MMX, and SSE have a set of permutation instructions with limited capabilities. For example, the MMX/SSE `pshufw` (packed shuffle word) instruction uses an immediate operand to select which of the four words in the source operand will be placed in each of the words in the destination operand. Figure 2.3 illustrates an example of this instruction. MIPS' MDMX uses a predefined set of eight 8-bit and eight 16-bit wide shuffles to implement partial shuffle operations.

The AltiVec extension, the Cell SPE, and the Texas Instruments C64x VLIW DSP [120] provide a separate permutation unit that allows an arbitrary permutation of any subword in one instruction. In the AltiVec extension, this permutation unit is one of four integer pipelines that are fed by the dynamic dispatch unit. In the SPE, this unit is one of four units that are located in the odd pipeline. The Texas Instruments C64x VLIW DSP has eight execution units. Two of them are used to handle permutations.

The permute unit can generate almost any permutation of data from two input registers. For instance, the AltiVec extension and Cell SPE use `vperm(vd, va, vb, vc)` and `shufb(vd, va, vb, vc)` (shuffle bytes) instructions, respectively. These instructions can take an arbitrary collection of bytes from the two source operands va and vb, and shuffle them into the destination operand vd based on the permute vector vc. Figure 2.4 depicts an example of this instruction.

Although designing a separate permutation unit provides more flexibility than predefined permutation instructions, it comes at a significant hardware cost, consuming

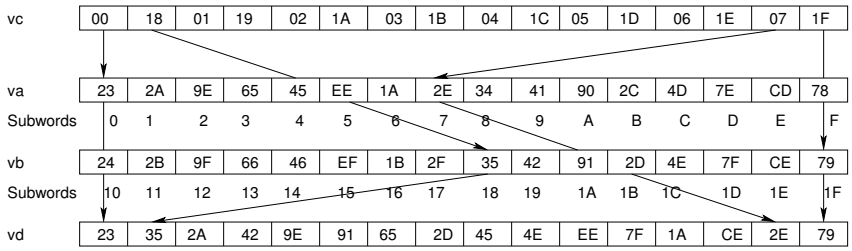


Figure 2.4: *Illustration of the vector permute instruction of the AltiVec extension and Cell SPE to permute sixteen subwords from the concatenation of registers va and vb by the byte index values in the vc register.*

chip area, and complexity. The most important problem with this unit is that it can only access two media registers at a time. This results in serious limitations on the data reordering across more than two media registers that is used in some applications such as matrix transposition and 2D media kernels as was discussed in previous chapter. In addition, it is costly in terms of the number of registers because every required data reordering pattern must be kept in a register. Additionally, it increases the memory bandwidth requirements to load the rearrangement patterns from memory to registers.

Lee presented a new subword permutation instruction across multiple registers, which can perform all permutations of a 2×2 matrix [92]. The `Mix` instruction in HP's MAX [89] can perform any permutation of the four 16-bit elements within a 64-bit register.

In general, in this approach, the data permutation instructions are explicitly used to rearrange the subwords within an SIMD register, two SIMD registers, or across more than two SIMD registers. The execution overhead of these instructions is the main drawback of this approach.

2.2.2 Memory Operations

Slingerland and Smith [130] proposed that SIMD architectures implement strided loads and stores to gather non-adjacent data elements. This technique is useful for some multimedia kernels such as color space conversion. This is because the strided memory accesses can eliminate the overhead instructions. The MOM ISA provides gather and scatter instructions to reorder the data. However, this approach has the following drawbacks. First, since gather, scatter, and strided instructions are usually slower than normal load and store instructions, the memory latency is increased and this increases the memory gap between the processor and the memory system hierarchy. Second, loading several strided elements may cause a number of cache

misses and reduce the spatial locality. Furthermore, higher bandwidth is required to load many strided data simultaneously. In [22] it has been indicated that one reason for poor VIRAM [78] memory performance for color space conversion is the strided memory accesses.

ARM's Neon Technology [11, 56] treats memory as an Array of Structures (AoS). This means that a load instruction loads subwords stored consecutively in memory into different SIMD registers. For example, the `vld3.16 {D0, D1, D2}, [R0]` instruction transfers four 3×16-bit structures stored in memory as $x_0, y_0, z_0, x_1, \ldots, z_3$ to the registers `D0`, `D1`, and `D2` so that `D0` contains the values $x_0, \ldots, x_3$, `D1` the values $y_0, \ldots, y_3$, and `D2` the values $z_0, \ldots, z_3$. This instruction is very useful for the color space conversion kernel where the stride is 3 but cannot be used for other data rearrangement operations that use strides different from 3 such as matrix transposition. For this operation other load instructions such as `vld4` can be used.

2.2.3 Register File Organization

Many researchers [73, 63, 115, 58, 71, 101] have eliminated the data permutation instructions by changing the organization of the media register file. For example, Jung et al. [73] have proposed a register file organization that provides both row- and column-wise accesses. Hsieh et al. [63] have discussed a Transpose Switch Matrix Memory (TSMM). The TSMM can be interpreted as 4 columns of 4 16-bit registers where each column corresponds to a particular processing element. This architecture supports normal access as well as transposed read. Ronner et al. [115] have proposed a highly parallel DSP (HiPAR-DSP) for image and video processing. Each data path of the HiPAR-DSP architecture has access to a shared memory with regular matrix access patterns and a matrix-memory. The matrix-memory is accessed using 2D addresses. The 2D addresses consist of the position of the upper left requested data element and the horizontal and vertical distance between adjacent matrix elements. That is the address format is (h, v, h_d, v_d), where $(h, v, h_d,$ and $v_d)$ are the address of the upper left requested data and the horizontal and vertical distance between adjacent matrix elements, respectively.

Hanounik et al. [58] have proposed to use diagonal registers to execute matrix transpose efficiently. In their technique, in addition to accessing the data in each row as is done in a traditional vector register file, the data along the diagonal direction can also be accessed. They have introduced three kind of registers, row registers, diagonal-down registers, and diagonal-up registers. The diagonal-down registers are accessed from the top left corner of the vector register file to the bottom right corner of register file. The diagonal-up registers extend from the bottom left corner of vector register file to the top right corner of vector register file.

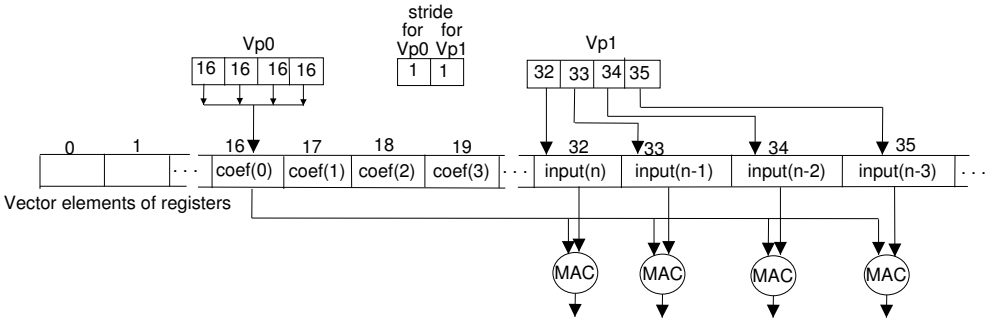


Figure 2.5: *Vector pointers are used to index the coefficients and input entries in the single-instruction multiple disjoint data implementation of the finite impulse response filter.*

Jayasena et al. [71] have proposed a stream register file with indexed access. They have shown that arbitrary access to a stream register file provides more temporal locality and reduces data replication in the stream register file compared to sequential stream register file.

A different approach to eliminate data permutation instructions named Single-Instruction Multiple disjoint Data (SIMdD) has been proposed in the eLite DSP architecture [40, 100, 66, 101]. Instead of a vector register file, the eLite DSP employs a large scalar register file, the vector element file (VEF). The elements in the VEF are addressed by four indices contained in a vector pointer register. In other words, vectors are dynamically composed. While very flexible, this approach requires four read ports to the VEF and can process at most four values in parallel. To process more, more read ports are required. The eLite DSP also has vector accumulator registers and a vector accumulator unit. For example, in case of the Finite Impulse Response (FIR) filter, two vector pointers are used. All elements of a vector pointer point to the same coefficient in the VEF and are incremented by one for the next coefficient. All elements of the other vector pointer point to consecutive input entries in the VEF and are incremented by one for addressing the next input data. Figure 2.5 illustrates this algorithm. As this figure shows, V_{p0} indexes the coefficient $coef[0]$ and V_{p1} indexes the input data. The indexed coefficients and input values are used as the inputs to the vector MAC operations.

In this method, there is overhead both in hardware and in software because in hardware, vector pointer registers and a vector pointer unit are needed. In software, the programmer or the compiler has to use initializations instructions for loading pointers to two source vector pointers and a destination vector pointer.

2.3 SIMD Vectorization

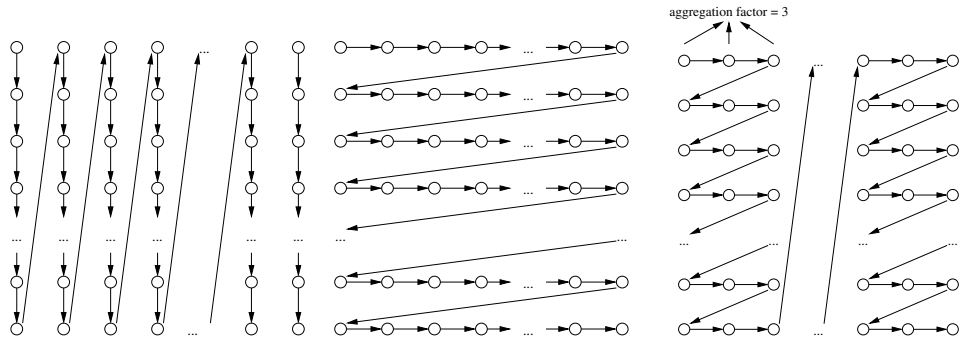
This section describes different techniques to vectorize MMAs efficiently.

Several studies have recently been performed to minimize the number of permutation instructions by compilers [113, 102, 81]. Kudriavtsev et al. [81] have proposed an algorithm to generate permutation instructions without using intrinsics or built-in functions. In their technique, the memory operations are grouped into SIMD instructions based on their effective address. Other operations are grouped starting from the memory operation groups. The number of permutation instructions is optimized with integer linear programming. Nuzman et al. [102] have extended a classic loop-based vectorizer to generate efficient permutation instructions for interleaved data, whose strides are powers of 2. Ren et al. [113] have used an algorithm to optimize the generation of permutation instructions. Their optimization technique reduces the number of data rearrangement instructions by propagating permutations across statements and merging consecutive permutations whenever possible.

Chaver et al. [25, 24] have used automatic vectorization to implement the Cohen, Daubechies and Feauveau 9/7 filter [31] (CDF-9/7) using SSE instructions. They used the single-precision floating-point format for both image pixels and wavelet coefficients. The Intel compiler, however, can only vectorize simple loops, and therefore some manual code modifications had to be performed. Furthermore, only horizontal filtering could be automatically vectorized (they assumed column-major order). In addition, they focused on the memory hierarchy and considered several techniques such as tiling to improve spatial and temporal locality. For example, they combined aggregation with a line-based approach for their SIMD implementation. The aggregation technique filters a number of adjacent columns consecutively before moving to the next row. The line-based algorithm uses a single loop to process both rows and columns together.

Bernabé et al. [15] have implemented the Daub-4 transform using the SSE instructions in order to reduce the execution time of the 3D wavelet transform. They as well as other researchers [46, 23] have shown that vertical filtering of the DWT requires more time than horizontal filtering because vertical filtering lacks spatial locality. However, as shown in the first chapter, in order to vectorize horizontal filtering the subwords in a media register have to be rearranged, which incurs significant overhead.

Kutil [84] has implemented the $(9, 7)$ lifting scheme using built-in SSE functions. He proposed a single loop approach to SIMD vectorization. In this approach horizontal and vertical filtering are combined into a single loop. This is called line-based computation in [30] and pipeline computation in [24], where it has been used to vectorize the CDF-9/7 transform. The single-loop approach requires a buffer whose size is



(a) Normal **(b)** Loop interchange **(c)** Aggregation

Figure 2.6: *Different implementations of the vertical filtering of discrete wavelet transform.*

equal to 16 rows of data. If this buffer does not fit in the cache, the temporal locality will be reduced.

2.4 Cache Optimization

This section describes different techniques to optimize the cache performance of the 2D DWT.

Meerwald et al. [99] have shown that a straightforward implementation of vertical filtering of the DWT that processes the elements along the columns can generate many cache misses and proposed two techniques, row extension and aggregation, to avoid this problem. Row extension adds some dummy elements to each row so that the image width is no longer a power of two but co-prime with the number of cache sets. According to [99], a disadvantage of this method is that the final coded bitstream is changed. As mentioned previously, aggregation approach processes a number of adjacent columns consecutively before moving to the next row. The number of columns filtered consecutively is called the "aggregation factor". If the aggregation factor is equal to the image width, aggregation is identical to loop interchange, which is a well-known compiler technique. Figure 2.6 illustrates the normal straightforward, the loop interchanged, and the aggregated implementation of vertical filtering.

A potential advantage of aggregation over loop interchange is that it can exploit the reuse between the rows of input values needed to compute consecutive rows of output values, while loop interchange cannot for large images. To investigate this, the aggregation technique was implemented. Figure 2.7 depicts the speedup of loop interchange over aggregation for different aggregation factors on the Pentium 4. Contrary

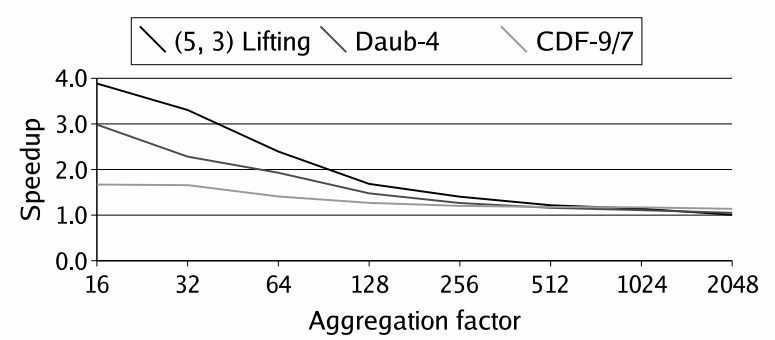


Figure 2.7: *Speedup of the loop interchanged implementation of vertical filtering over the aggregated implementation for different aggregation factors on the Pentium 4. The image size is* 2048×2048.

to the initial expectations, aggregation performs worse than loop interchange. This is due to three reasons. First, aggregation incurs more loop overhead than loop interchange, since it consists of three loop nests instead of two. Second, on a cache miss the P4 prefetches the next block. Loop interchange takes advantage of this but aggregation does not when it reaches the end of a group of columns. Third, cache performance is not critical in these scalar implementations.

Chatterjee and Brooks [21] proposed two optimizations: strip-mining (which is identical to aggregation) and recursive data layout. The second optimization modifies the layout of the image data so that each sub-band is stored contiguously. This increases the locality for subsequent decomposition levels, but only the execution time of the first level is reported. Furthermore, the first decomposition level takes more time than all subsequent decomposition levels together.

Chaver et al. [24] combined aggregation with a line-based approach [30], which starts vertical filtering as soon as a sufficient number of rows (determined by the filter length) has been filtered horizontally. This approach reduces the amount of memory required. In addition, they considered different layouts.

Komi et al. [76] as well as Lee et al. [94] proposed block-based approaches to improve the cache efficiency for the 2D DWT. In [76] equations are presented that allow to find the optimal block size, assuming a fully associative data cache. In [94] the block size is equal to one way of the two-way set-associative L1 data cache.

Andreopoulos et al. [3] identified three categories of DWT implementations based on the order the transform coefficients are produced: strictly breadth-first (SBF), roughly depth-first (RDF), and strictly depth-first (SDF). SBF implementations filter all rows horizontally before filtering all columns vertically. RDF implementations interleave periods of horizontal filtering with periods of vertical filtering. The line-based and

block-based approaches belong to this category. SDF corresponds to RDF with minimal interleaving period. Andreopoulos et al. have shown that RDF implementations incur fewer misses than SBF implementations.

2.5 Conclusions

This chapter presented the background information for data type conversion and data rearrangement instructions. Some multimedia extensions have provided different instructions to convert from a smaller data type to a larger data type and vice versa. SIMD architectures use different techniques for data permutation. Generally, there are three approaches for data reordering. First, explicit instructions are employed in order to reorder the data within and between SIMD registers. Some SIMD architectures have provided a set of pre-defined instructions with limited capabilities, while others have used a separate permutation unit to allow an arbitrary permutation of any subword. Second, data rearrangement is performed during memory operations using gather, and scatter instructions. In these methods the media register file is only accessible in the horizontal direction. Finally, data permutation is performed by the organization of the media register file. For example, the media register file is accessible in both horizontal and vertical directions.

In addition, this chapter described some SIMD vectorization and cache optimization techniques that have been used in implementations of multimedia kernels, for example, in the 2D discrete wavelet transform.

In the next chapter, the MMMX architecture is presented. The MMMX architecture is an SIMD extension for multimedia domains. It focuses on maintaining programmability and accelerates MMAs by exploiting DLP. The MMMX architecture eliminates the data type conversion and data rearrangement instructions. Its instructions are applicable in multiple domains. It is not an application-specific ISA.

Chapter 3

MMMX Architecture

In previous chapters, it was discussed that the performance of multimedia applications can be accelerated by exploiting DLP in a programmable SIMD architecture. This was because the media applications have been changing and this promotes the use of programmable processors instead of custom ASICs or highly specialized application-specific processors. Some multimedia kernels were usually supported by dedicated and ASIC hardware. To avoid the added cost and complexity of these dedicated hardware units, this chapter focuses on maintaining programmability while increasing performance using a novel SIMD extension. In addition, new and general SIMD instructions addressing the multimedia application domain are investigated. It does not consider an ISA that is application specific. For example, special-purpose instructions are synthesized using a few general-purpose SIMD instructions.

This chapter describes a novel SIMD ISA extension called Modified MMX (MMMX, pronounced as triple-MX). The MMMX extension is multimedia oriented. It is based on two micro-architectural techniques: extended subwords and the Matrix Register File (MRF). Extended subwords use registers that are wider than the packed format used to store the data. The MRF allows to load data that is stored consecutively in memory into a column of the register file, where a column corresponds to the corresponding subwords of different registers. The MRF is useful for matrix operations that are common in multimedia processing.

The MMMX architecture is MMX enhanced with extended subwords, the MRF, and a few general-purpose instructions that are not present in MMX. It is important to note that although the MMX/SSE integer extension has been enhanced with the proposed techniques, the extended subwords and the MRF techniques are general and can be applied to basically any SIMD ISA extension. The main reason why MMX was

```
unsigned char blk1[16][16], blk2[16][16];
int ssd = 0;
for (i=0; i<16; i++)
  for (j=0; j<16; j++)
    ssd += (blk1[i][j] - blk2[i][j])
        *  (blk1[i][j] - blk2[i][j]);
```

Figure 3.1: *C code of the sum-of-squared differences kernel.*

selected is that it is representative of SIMD extensions and the author's familiarity with the x86 instruction set.

The remainder of the chapter is organized as follows. Extended subwords and the MRF are discussed in Section 3.1 and Section 3.2, respectively. The MMMX instructions are described in Section 3.3, and concluding remarks presented in Section 3.4.

3.1 Extended Subwords

Image, video, and audio data are usually small 8- or 16-bit integers, while computations on these small data types often require larger data types. Consider, for example, the code that is depicted in Figure 3.1. This code computes the sum-of-squared differences between two 16×16 blocks.

The difference between `blk1[i][j]` and `blk2[i][j]` is a 9-bit value between -255 and 255, and $\sum_{i=0}^{15}\sum_{j=0}^{15}(blk1[i][j] - blk2[i][j])^2$ does not fit in neither an 8- nor a 16-bit subword. This is because as shown in Equation (3.1), a 24-bit value is needed for the final result.

$$\sum_{i=0}^{15}\sum_{j=0}^{15}(255)^2 < (2^8)^3 = 2^{24} \tag{3.1}$$

Some architectures provide an absolute difference operation for such computations and therefore, do not need a larger format for their intermediate results. For instance, in the discussed example, instead of calculating `blk1[i][j] - blk2[i][j]` that needs a precision of 9-bit, `|blk1[i][j] - blk2[i][j]|` can be calculated. This means that for two 8-bit values x and y, $x - y$ needs 9 bits but $|x - y|$ can be represented in 8 bits. This concept is also used in the special-purpose `psadbw` instruction that has been provided in the SSE extension. The C code of the sum-of-absolute differences function is depicted in Figure 3.2. This code computes the sum-of-absolute differences between two 16×16 blocks. Since `blk1[i][j] - blk2[i][j]` is a 9-bit value, eight of these intermediate results do not fit in a single 64-bit register, while the SSE extension uses the absolute differ-

```
unsigned char blk1[16][16], blk2[16][16];
int sad = 0; short diff;
for (i=0; i<16; i++)
  for (j=0; j<16; j++) {
    diff = blk1[i][j]  - blk2[i][j];
    if (diff<0) diff = - diff;
    sad += diff;
  }
```

Figure 3.2: *C code of the sum-of-absolute differences kernel.*

ence operation to keep the carry bit internally. As shown in Equation (3.2), a 16-bit value is needed for the final result.

$$\sum_{i=0}^{15}\sum_{j=0}^{15}(255) < (2^8)^2 = 2^{16} \tag{3.2}$$

The absolute difference operation cannot be used by other multimedia kernels. Therefore, those media kernels need an extra bit to keep the output carry. For example, consider the following loop which computes the arithmetic average of two images:

```
unsigned char src1[], src2[], dst[];
for (i=0; i<n; i++)
  dst[i] = (src1[i] + src2[i]) >> 1;
```

Even though the final result `dst[i]` is an 8-bit value, the intermediate result `src1[i] + src2[i]` is 9-bit. As previously discussed in Section 2.1 in Chapter 2, if the packed subwords are filled by a maximum value representable using the subword data type, the choice is either to be imprecise using saturation operation at every stage, or to loose parallelism by unpacking to a larger format. Using saturation instructions produce unexpected results.

Converting or promoting the source operands to a larger format before they are processed, and packing or demoting the results again before they are written to memory, causes conversion overhead. In addition, the number of subwords that are processed in parallel by a single SIMD instruction is reduced. The main reason for the data type conversion instructions is the mismatch between the storage format and the computational format.

Some multimedia kernels have been examined to determine their storage and computational formats. The result is depicted in Table 3.1. It shows that four extra bits for every byte in a register is sufficient for those kernels. This is also supported by [37], where it was shown that a 12-bit data format is sufficient for the most functions involved in the MPEG-4 encoding (core profile). For example, results presented there indicate that the MPEG-4 encoding consists of 21 functions and 17 of them need a

Multimedia kernels	Storage format	Computational format
RGB-to-YCbCr	unsigned byte	12-bit
YCbCr-to-RGB	unsigned byte	12-bit
SAD function	unsigned byte	9-bit
SAD function with interpolation	unsigned byte	10-bit
SSD function	unsigned byte	16-bit
SSD function with interpolation	unsigned byte	16-bit
Add block	unsigned byte	9-bit
2D DCT	(un)signed byte	12-bit
2×2 Haar transform	unsigned byte	10-bit
Inverse Haar transform	half word	10-bit
Paeth prediction	unsigned byte	10-bit
Repetitive padding	unsigned byte	9-bit
Arithmetic average	unsigned byte	9-bit

Table 3.1: *The storage and computational formats of some multimedia kernels.*

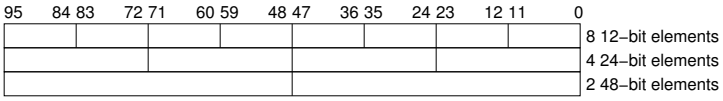


Figure 3.3: *Different subwords in the media register file of the MMMX architecture.*

precision equal or smaller than 12-bit. The remaining functions need a larger precision than 12-bit, for instance, 32- and 64-bit.

To avoid the data type conversion overhead and to increase parallelism, extended subwords are employed. This means that the registers are wider than the data loaded into them. Specifically, for every byte of data, there are four extra bits. This implies that MMMX registers are 96 bits wide, while MMX has 64-bit registers. The number of elements that can be placed into a register is determined by the size of elements. Based on that, the MMMX registers can hold 2×48-bit, 4×24-bit, or 8×12-bit elements as is depicted in Figure 3.3. This means that a 96-bit register can be broken into independent subwords of 12, 24, and 48 bits that are operated in parallel. Registers are used to hold fixed-point data. In other words, the MMMX architecture supports 8-, 16-, and 32-bit data types in memory and 12-, 24-, and 48-bit in the datapath during computations.

The extended subwords technique increases the number of subwords that can be packed into a media register. This feature allows to perform more operations in parallel by packing more data elements into a single media register. The extended subwords technique also eliminates most of the pack/unpack instructions required to convert between different data types.

This technique, however, is not sufficient for 2D multimedia algorithms that process data along the rows as well as along the columns. In other words, 2D operations

access data in both horizontal and vertical directions. The register files of existing SIMD architectures only allow simultaneous access to a row register. A register is referred to a row register if it is horizontally (row-wise) accessed. In other words, row registers correspond to conventional media registers. A register is called a column register if it is vertically (column-wise) accessed. In order to employ SIMD instructions in 2D algorithms, the matrix needs to be transposed frequently. Transposition takes a significant amount of time, however. This is because the matrix transpose operations like any 2D operation accesses data elements along the rows as well as along the columns. For example, to implement this operation in MMX/SSE requires many rearrangement instructions such as `punpckh`, `punpckl`, and `pshufw`. Specifically, to implement an 8×8 matrix transposition using MMX/SSE requires 56 instructions if the elements are 8 bits wide. If the elements are two bytes wide, then 88 instructions are required.

The next section describes the matrix register file to overcome this limitation.

3.2 The Matrix Register File

This section describes the matrix register file (MRF), the number and configuration of its registers, and its usefulness in the implementation of multimedia kernels.

As mentioned in previous chapters, one of the bottlenecks of any SIMD architecture is the restricted access to the data stored in the media register file. SIMD instructions can only access the data in row registers. The restriction is due to the conventional design of the media register file, which allows concurrent accesses only to data residing in a row. However, the ability to efficiently rearrange subwords within and between registers is crucial to the performance of many media kernels. Matrix transposition, in particular, which is needed in several block-based algorithms, is a very expensive operation. Therefore, a way to accelerate this operation is that the architecture allows access to data in both dimensions. To overcome this problem, a matrix register file is employed, which allows data loaded from memory to be written to a column of the register file as well as to a row register.

Figure 3.4 illustrates an MRF with 12-bit subwords. It has eight row registers $3mxi, 0 \leq i \leq 7$ (corresponding to conventional media registers) and eight column registers $3mxci, 0 \leq i \leq 7$ (corresponding subwords in different row registers). Both row and column registers are 96 bits wide. Data loaded from memory can be written to a row register as well as to a column register. Seven 2:1 12-bit multiplexers are needed per register/row to select between row-wise and column-wise access. For example, for register `3mx0` it needs to be able to select between the most significant subword of the data for column-wise access and another subword in case of row-wise access. Multiplexers are not needed for the subwords on the main diagonal.

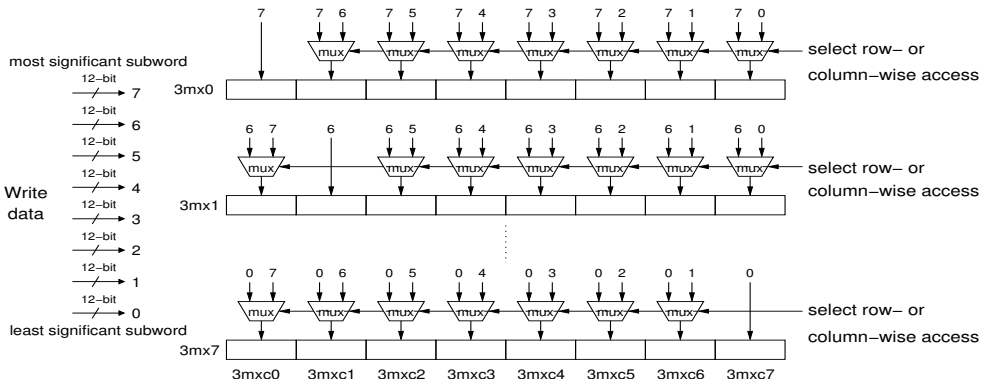


Figure 3.4: *A matrix register file with 12-bit subwords. For simplicity, write and clock signals have been omitted.*

Each 12-bit subword $MRF[i, j], 0 \leq i, j \leq 7$ belongs to row register i and column register j. Eight row and eight column registers are accessed in row-wise and column-wise, respectively. Therefore, the MRF architecture provides parallel access to 12-, 24-, and 48-bit subwords of the row registers that are horizontally located. This is similar to conventional SIMD architectures, which provide parallel access to 8-, 16-, and 32-bit data elements of media registers. In addition, the MRF provides parallel access to 12-bit subwords of the column registers that are vertically arranged. It does not support access to 24- or 48-bit data elements of the column registers. Although it can be extended to support those subwords, supporting 12-bit subwords is sufficient to implement many media kernels, which has been done in the next chapter.

There are two ways to transpose an 8×8 block whose elements are smaller than 12-bit while they use 16-bit storage format. First, normal write to row registers followed by column-wise read from column registers. Second, column-wise write to column registers followed by normal read from row registers. The MRF requires additional hardware for each port that support column-wise access. Since the media register file has fewer write ports than read ports, the latter approach is selected to implement the MRF in this thesis.

To illustrate the usefulness of the MRF, two examples are discussed in the remainder of this section. The first example is computing an 8-point 1D DCT using the LLM algorithm [96] which is depicted in Figure 3.5. An 8×8 2D DCT can be accomplished by performing a 1D DCT on each row followed by a 1D DCT on each column. Initially, the bytes x_i ($0 \leq i \leq 7$) are packed consecutively into one 64-bit quadword, with x_0 being the least significant and x_7 being the most significant bytes. It can be seen that this piece of code exhibits subword DLP, but only 4-way. Furthermore, in order to exploit this parallelism using SIMD instructions, the elements x_0 and x_7, x_1 and x_6, x_2 and x_5, and x_3 and x_4 have to be in corresponding subwords of different registers. This implies that the high and low doublewords of the quadword

$$
\begin{aligned}
s_{10} &= x_0 + x_7 \\
s_{11} &= x_1 + x_6 \\
s_{12} &= x_2 + x_5 \\
s_{13} &= x_3 + x_4 \\
s_{14} &= x_3 - x_4 \\
s_{15} &= x_2 - x_5 \\
s_{16} &= x_1 - x_6 \\
s_{17} &= x_0 - x_7
\end{aligned}
$$

Figure 3.5: *First stage of the LLM algorithm for computing an 8-point DCT.*

have to be split across different registers and that the order of the subwords in one of these registers has to be reversed. An alternative way to realize a 2D DCT is by transposing the matrix so that all the x_i's of different rows are in one register. In other words, it performs several 1D DCTs in parallel rather than trying to exploit the DLP present in a 1D DCT. If the transposition step can be implemented efficiently, this method is more efficient than the first one. Moreover, it allows to exploit 8-way SIMD parallelism provided the subwords can represent the intermediate results.

It should be noted that matrix transposition not only arises in the DCT but also in many other kernels such as the IDCT, vertical padding, and vertical subsampling. Moreover, the matrix has to be transposed *twice* in order to exploit 8-way parallelism in these kernels.

The second example is vectorization of strided data. As was mentioned in the first chapter, SIMD architectures are most efficient when the data which is processed in parallel is stored consecutively in memory. If not, there is a large overhead involving data reorganization instructions. For example, in the case of color space conversion, often the band interleaved format is used where the color components of each pixel are adjacent in memory. This implies that in order to employ SIMD instructions efficiently, the image pixels have to be reorganized so that the red data of different pixels are contained in one register, the green data in another, and the blue data in a third register. In this case, many data reorganization instructions need to be executed.

Figure 3.6 illustrates how the MRF can be used to reorganize the band interleaved RGB data to band separated. With eight load-column instructions (`fldc8u12`) eight red, eight green, and eight blue values are loaded into each register. Each load-column instruction loads eight bytes (three red, three green, and two blue) values as is shown in Figure 3.6 for little endian. To provide correct arrangement of RGB values, an offset which is a multiple of 6 bytes is used for each `fldc8u12` instruction. It is remarked that this also works for other strides. For example, when the stride is 4, an

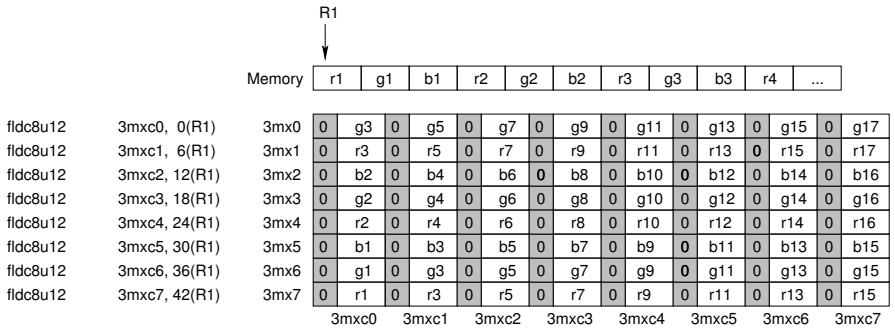


Figure 3.6: *Loading eight red, eight green, and eight blue values into the matrix register file using the* `fldc8u12` *instruction for little endian.*

offset which is a multiple of 8 can be used.

As previously mentioned, the MMMX extension is multimedia oriented. Its instruction set has been designed based on the characteristics of MMAs. An ISA that is application specific has not been considered. For example, SPIs are synthesized using a few general-purpose SIMD instructions. The following section describes the instruction set of the MMMX architecture.

3.3 MMMX Instruction Set Architecture

The main goal of media instruction set is to enable efficient execution of multimedia workloads. In the design of the MMMX instruction set factors such as reduced operation complexity and wider set of available operations have been considered. More details about the available MMMX instruction set is discussed in the following sections.

3.3.1 Load/Store Instructions

The MMMX architecture has different load/store instructions that depicted in Table 3.2. Some of them are explained as follows. The `fld8u12` instruction loads eight unsigned bytes from memory and zero-extends them to a 12-bit format in a 96-bit MMMX register. The `fld8s12` instruction, on the other hand, loads eight signed bytes and sign-extends them to a 12-bit format. These instructions are illustrated in Figure 3.7 for little endian. The `fld16s12` instruction loads eight signed 16-bit, packs them to signed 12-bit format, and writes in a row register. This instruction is useful for those kernels that their input data can be represented by the signed 12-bit, while they use the signed 16-bit storage format. For example, in the DCT kernel,

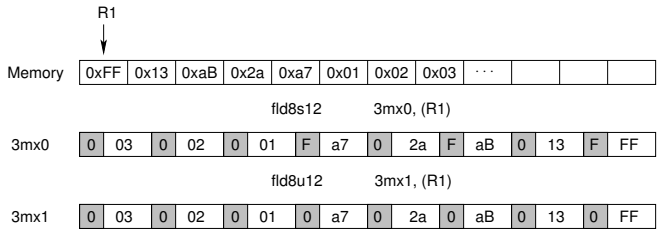


Figure 3.7: *The* `fld8s12` *instruction loads eight signed bytes and sign-extends them to 12-bit values, while the* `fld8u12` *instruction loads eight unsigned bytes and zero-extends them to 12-bit values.*

Instructions	Description
fld8u12	loads eight unsigned bytes, zero-extends them to 12-bit formats, and writes in a row register.
fld8s12	loads eight signed bytes, sign-extends them to signed 12-bit format, and writes in a row register.
fld16s12	loads eight signed 16-bit, packs them to signed 12-bit format, and writes in a row register.
fld16u12	loads eight unsigned 16-bit, packs them to unsigned 12-bit format, and writes in a row register.
fld16s24	loads four signed 16-bit, sign-extends them to signed 24-bit format, and writes in a row register.
fld16u24	loads four unsigned 16-bit, zero-extends them to 24-bit formats, and writes in a row register.
fld32s24	loads four signed 32-bit, packs them to signed 24-bit format, and writes in a row register.
fld32u24	loads four unsigned 32-bit, packs them to unsigned 24-bit format, and writes in a row register.
fld32s48	loads two signed 32-bit, sign-extends them to signed 48-bit format, and writes in a row register.
fldc8u12	loads eight unsigned bytes, zero-extends them to 12-bit format, and writes in a column register.
fldc16s12	loads eight signed 16-bit, packs them to signed 12-bit format, and writes in a column register.
fst12s8	saturates eight signed 12-bit to unsigned 8-bit and store into memory.
fst12s16	unpack eight signed 12-bit to signed 16-bit and store into memory.
fst24s16	saturates four signed 24-bit to signed 16-bit and store into memory.
fst24s32	unpack four signed 24-bit to signed 32-bit and store into memory.
fst48s16	saturates two signed 48-bit to signed 16-bit and store into memory.
fst48s32	saturates two signed 48-bit to signed 32-bit and store into memory.
fst96s16	saturates a signed 96-bit to signed 16-bit and store into memory.
fst96s32	saturates a signed 96-bit to signed 32-bit and store into memory.

Table 3.2: *The load/store instructions of the MMMX architecture.*

the input data is the signed 9-bit format. It uses the signed 16-bit storage format, while it uses the signed 12-bit for computational format. The instruction `fldc8u12` ("load-column 8-bit to 12-bit unsigned") is used to load a column of the MRF.

Load instructions automatically unpack and store instructions automatically pack and saturate, as illustrated for the load instructions in Figure 3.7. Store instructions automatically saturate (clip) and pack the subwords. For example, the instruction `fst12s8` saturates the 12-bit signed subwords to 8-bit unsigned subwords before storing them to memory.

3.3.2 ALU Instructions

Most MMMX ALU instructions are direct counterparts of MMX/SSE instructions. For example, the MMMX instructions `fadd{12,24,48}` (packed ad-

Instructions	Description
fadd{12,24,48}	packed addition of 12-, 24-, or 48-bit subwords.
fsub{12,24,48}	packed subtraction of 12-, 24-, or 48-bit subwords.
fsum{12,24,48}	addition of adjacent elements of 12-, 24-, or 48-bit subwords.
fdiff{12,24,48}	subtraction of adjacent elements of 12-, 24-, or 48-bit subwords.
fneg{12,24,48}	to nagate some or all elements in a packed register.
finc{12, 24, 48}	increment 12-, 24-, or 48- bit subwords.
fdec{12, 24, 48}	decrement 12-, 24-, or 48- bit subwords.
fmin{12, 24, 48}	minimum selection instructions in 12-, 24-, or 48-bit subwords.
fmax{12, 24, 48}	maximum selection instructions in 12-, 24-, or 48-bit subwords.

Table 3.3: *The ALU instructions of the MMMX architecture.*

dition of 12-, 24-, 48-bit subwords) and `fsub{12,24,48}` (packed subtraction of 12-, 24-, 48-bit subwords) correspond to the MMX instructions `padd{b,w,d} mm,mm/mem64` and `psub{b,w,d} mm,mm/mem64`, respectively. MMMX, however, does not support variants of these instructions that automatically saturate the results of the additions to the maximum value representable by the subword data type. They are not needed because as was mentioned the load instructions automatically unpack the subwords and the store instructions automatically pack and saturate. In other words, the MMMX architecture does not support saturation arithmetic. Table 3.3 depicts the ALU instructions.

In the remainder of this section and also the next section some novel MMMX ALU and multiplication instructions are discussed, which are not supported in MMX. In many media kernels all elements packed in a register need to be summed, while in other kernels adjacent elements need to be added. Rather than providing different instructions for summing all elements and adding adjacent elements, it has been decided to support adding adjacent elements only but for every packed data type. Whereas summing all elements would probably translate to a multicycle operation, adding adjacent elements is a very simple operation that can most likely be implemented in a single cycle. Figure 3.8 illustrates how eight 12-bit subwords can be reduced to a single 96-bit sum or 96-bit difference using the instructions `fsum{12,24,48}` and `fdiff{12,24,48}`, respectively. It is remarked that ARM's Neon technology and SSE3 also support pairwise addition instructions.

Another operation that has been found useful in implementing many multimedia kernels such as the (I)DCT kernels is the possibility to negate some or all elements in a packed register. The instructions `fneg{12,24,48} 3mx0, 3mx1, imm8` negate the 12-, 24-, or 48-bit subwords of the source operand if the corresponding bit in the 8-bit immediate `imm8` is set. One example of this instruction is depicted in Figure 3.9. If subwords are 24- or 48-bit, the four or six higher order bits in the 8-bit immediate are ignored.

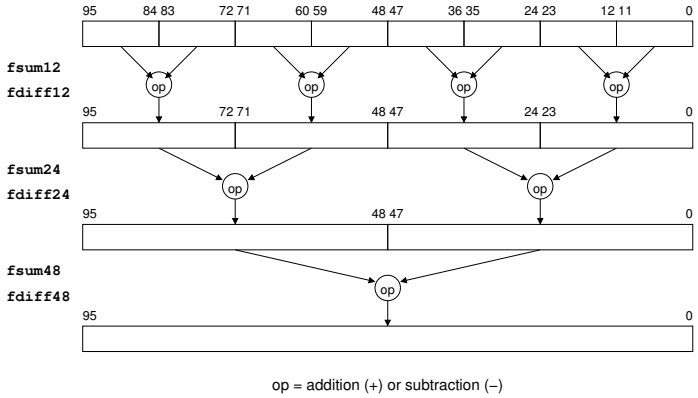


Figure 3.8: *Reducing eight 12-bit subwords to a single 96-bit sum or 96-bit difference using the instructions* `fsum{12,24,48}` *and* `fdiff{12,24,48}`*, respectively.*

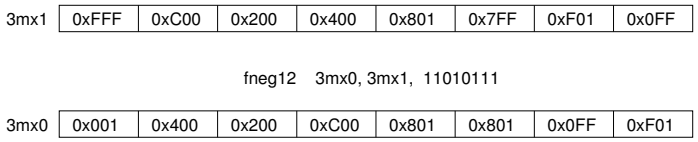


Figure 3.9: *Illustration of the* `fneg12 3mx0, 3mx1, 11010111` *instruction.*

3.3.3 Multiplication Instructions

The MMMX architecture supports three kinds of multiplication instructions that depicted in Table 3.4. The first are full multiplication instructions `fmulf{12,24}`. For example, the `fmulf12` instruction multiplies each 12-bit subword in `3mx0` with the corresponding subwords in `3mx1` and produces eight 24-bit results. This means that each result is larger than a subword. Therefore, the produced results are kept in both registers.

The second kind of multiplication instructions are the partitioned multiply-accumulate instructions `fmadd{12,24}`. These instructions perform the operation on subwords that are either 12- or 24-bit, while the MMX instruction `pmaddwd` performs the MAC operation on subwords that are 16-bit. The MAC operation is an important operation in digital signal processing. Figure 3.10 illustrates the operation of the `fmadd12 3mx0, 3mx1` instruction. This instruction multiplies the

Instructions	Description
fmulf{12,24}	full multiplication instructions.
fmadd{12,24}	partitioned multiply-accumulate instructions.
fmul{12l,12h,24l,24h}	truncation multiplication instructions.

Table 3.4: *The multiplication instructions of the MMMX architecture.*

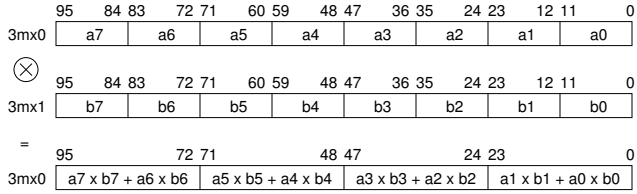


Figure 3.10: *Partitioned multiplication using the* `fmadd12 3mx0, 3mx1` *instruction.*

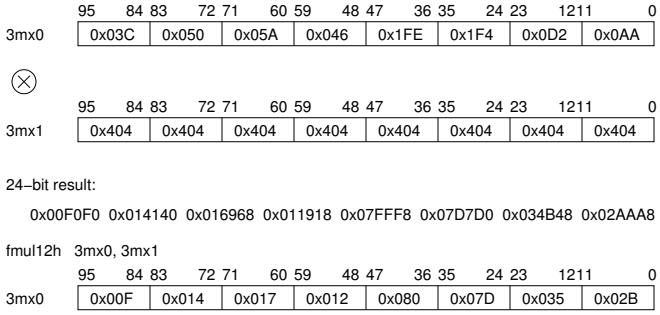


Figure 3.11: *Partitioned multiplication using the* `fmul12h 3mx0, 3mx1` *instruction.*

eight signed 12-bit values of the destination operand by the eight 12-bit values of the source operand. The corresponding odd-numbered and even-numbered subwords are summed and stored in the 24-bit subwords of the destination operand.

The third type of multiplication is truncation. Truncation is performed by the `fmul{12l,12h,24l,24h}` instructions. It means that the high or low bits of the results are discarded. When n-bit fixed point values are multiplied with fractional components, the result should be n-bit of precision. Specifically, the instructions `fmul12{l,h}` multiply the eight corresponding subwords of the source and destination operands and write the low-order (`fmul12l`) or high-order (`fmul12h`) 12 bits of the 24-bit product to the destination operand. This kind of partitioned multiplication can be used in some applications. For example, the `fmul12h` instruction is used in the fixed-point MMMX implementation of color space conversion that is discussed in Chapter 4. Figure 3.11 illustrates one example of the `fmul12h` instruction. In this example, each subword of the `3mx0` register is multiplied by the value 0x404 that has been replicated in the `3mx1` register, and the corresponding 12-bit high values are stored in the destination operand.

There is a loss of precision due to the nature of truncation. In order to reduce the effect of this error, first, the intermediate 24-bit result is internally rounded and after that the 12-bit result will be truncated as depicted in Figure 3.11. For example, the result of the multiplication of the value 0x0AA with the value 0x404 is 0x02AAAB. The final result after truncation with rounding low-order 12 bits is 0x02B, rather

Differences	**MMX/SSE** (integer part)	**MMMX**
Datapath	64-bit	96-bit
Size of register file	8 x 64-bit	8 x 96-bit
Shared with	Floating point registers	Dedicated
Access to register file	row-wise	row-wise + column-wise
Size of the partitioned ALU	64-bit	96-bit
Size of the integer subwords	8-, 16-, and 32-bit	12-, 24-, and 48-bit
Addition instructions	padd{b, w, d}	fadd{12, 24, 48}
Subtract instructions	psub{b, w, d}	fsub{12, 24, 48}
Saturate add instructions	padds{b, w}, paddus{b, w}	No
Saturate subtract instructions	psubs{b, w}, psubus{b, w}	No
Full multiply instruction	No	fmulf{12, 24}
High and low multiply inst.	pmul{hw, lw, huw}	fmul{12l, 12h, 24l, 24h}
The size of MAC unit	16-bit	12- and 24-bit
MAC instructions	pmaddwd	fmadd{12, 24}
Increment instruction	No	finc{12, 24, 48}
Decrement instruction	No	fdec{12, 24, 48}
Negate instruction	No	fneg{12, 24, 48}
Minimum selection instructions	pmin{ub, sw}	fmin{12, 24, 48}
Maximum selection instructions	pmax{ub, sw}	fmax{12, 24, 48}
Adjacent subwords addition	No	fsum{12, 24, 48}
Adjacent subwords subtraction	No	fdiff{12, 24, 48}
Special-purpose instructions	No/pavg{b, w}, psadbw	No
Overhead instructions	packss{wb, dw} packuswb, punpckh{bw, wd, dq} punpckl{bw, wd, dq}, pshufw	funpckl{12, 24} funpckh{12, 24}

Table 3.5: *The main differences between the MMX/SSE and MMMX ISAs.*

than truncated to 0x02A. On the MMX architecture, on the other hand, the `pmulhw` instruction truncates the lower 16-bit rather than rounding it.

3.3.4 Differences Between MMMX and MMX Architectures

The main differences between the MMX/SSE and MMMX ISAs in the integer part are depicted in Table 3.5. The characteristics of the MMMX architecture are the following. First, the MMMX architecture defines a set of eight row and eight column registers, each of them contains 96 bits. Second, each MMMX row register contains eight 12-bit, four 24-bit, or two 48-bit subwords, while its column registers consist of eight 12-bit data elements. This means that the media register file is wider than the data to be loaded into them. The registers are used to hold fixed-point data. In other words, the MMMX architecture is a fixed-point SIMD extension. Third, the MMMX load instructions implicitly unpack data from the storage format to the computation format, and the store instructions implicitly pack and saturate data from the computation format to the storage format. Fourth, media registers can be accessed row-wise as

well as column-wise. In other words, each 12-bit subword $MRF[i,j], 0 \leq i,j \leq 7$ belongs to row register i and column register j.

Additionally, there are some general-purpose SIMD instructions which do not have an MMX equivalent, such as the `fsum{12,24,48}` and `fdiff{12,24,48}` instructions. Due to the extended subwords, the MMMX architecture provides more subword parallelism than MMX as will be shown for some multimedia kernels in Chapter 4. Furthermore, MMMX does not support special-purpose MMX/SSE instructions `psadbw` and `pavg{b,w}` as well as rearrangement instructions such as `pshufw` and `packss{wb,dw,wb}`. Both MMX and MMMX support unpack instructions. For instance, MMMX supports `funpckl{12,24}` and `funpckh{12,24}` instructions as depicted in the last row in Table 3.5 in order to reorder final results and store them in their appropriate places, while MMX supports `punpck{hbw,hwd,hdq,lbw,lwd,ldq}` instructions.

3.3.5 Hardware Cost of the Proposed Techniques

In this section, the overhead hardware cost of the MMMX architecture over the MMX architecture in terms of area and critical path delay is discussed.

The following are differences between the MMX and MMMX architectures from the hardware point of view. First, each MMMX register is 32 bits wider than each MMX register. Second, the MMMX register file is accessible in both directions, while in the MMX architecture it is not. This means that for column-wise access to the MMMX register file, multiplexers, and an additional decoder as well as wiring are required. Third, the MMMX ISA needs to be able to address the column registers. Finally, in the MMMX architecture, a 96-bit partitioned ALU is required to provide eight 12-bit, four 24-bit, and two 48-bit subword parallel processing. In the MMX architecture, on the other hand, a 64-bit partitioned ALU is sufficient. Furthermore, in the MMMX architecture, there are some other SIMD instructions compared to the MMX architecture.

As previously mentioned, in order to reduce the hardware cost of the MMMX architecture, column-wise access on the write port of the register file has been provided. This is because the number of write ports is usually less than the number of read ports. Only load-column instructions can access the column registers, while the other instructions cannot. The number of load-column instructions in the MMMX ISA is two. This is because 8- and 16-bit image data are loaded into column registers. These instructions are used for those kernels that use the MRF technique, such as the RGB-to-YCbCr kernel. A single bit to the instruction format of load instructions is used in order to distinguish between normal load instructions and load-column instructions.

The register file, a 64-bit partitioned ALU, and a multiplication unit of the MMX ar-

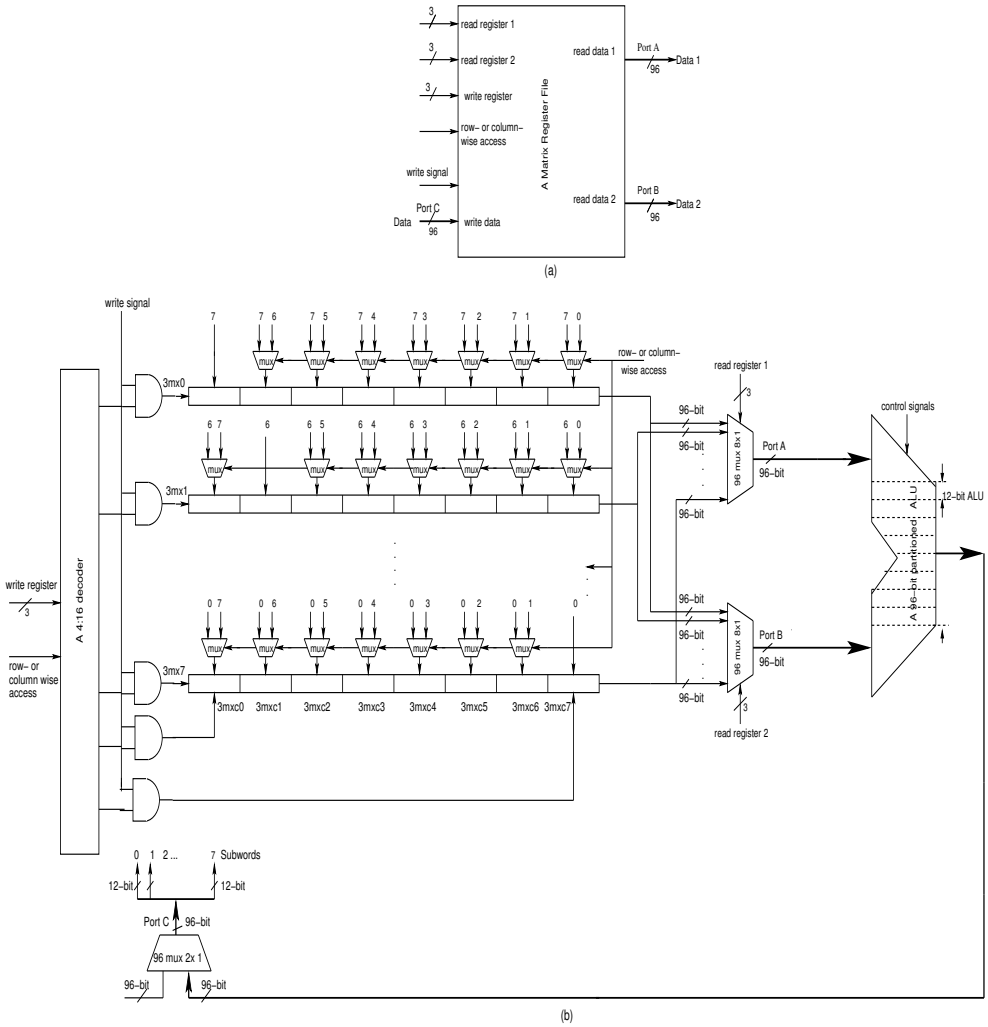


Figure 3.12: *(a) A register file with eight 96-bit registers, 2 read ports, and 1 write port, (b) the implementation of two read ports and one write port for a matrix register file with 8 96-bit registers as well as a partitioned ALU for subword parallel processing.*

chitecture and the MRF, extended subwords, a 96-bit partitioned ALU, and a multiplication unit of the MMMX architecture have been implemented in VHDL. A register file with two read ports and one write port has been considered for both architectures. Figure 3.12(a) depicts a block diagram of a register file with one write port (Port C) and two read ports (Port A and Port B). The input and output of this block diagram is based on eight 96-bit registers. Figure 3.12(b) illustrates the combination of the MRF with a 96-bit partitioned ALU for the MMMX architecture.

The partitioned ALUs have been designed based on the subword adder. Multiplexers

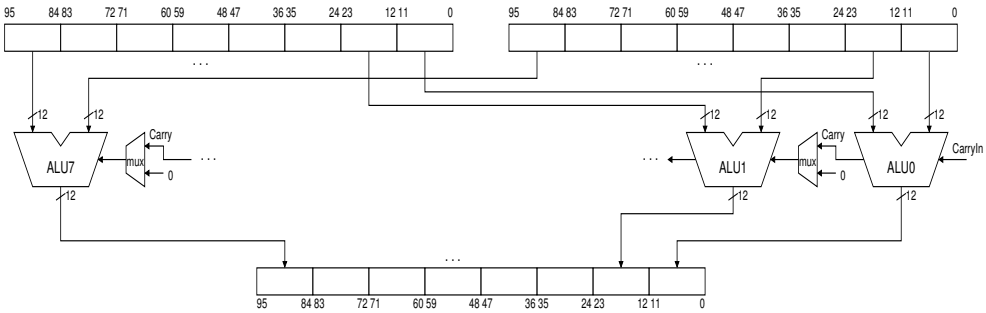


Figure 3.13: *A 96-bit partitioned ALU in the MMMX architecture.*

have been used in subword boundaries to propagate or prevent the subword carries in the carry chain [65]. There are eight 12-bit adders. These adders operate independently for 12-bit data. They can also be coupled to behave an four pairs of two adders to perform four 24-bit operations, or combined into two groups of four adders for two 48-bit format. Figure 3.13 illustrates that a 96-bit ALU can be partitioned into eight 12-bit ALUs. Such a partitioned ALU can perform eight 12-bit, four 24-bit, two 48-bit, and a 96-bit operation.

In addition, all SIMD arithmetic, multiplication, logical, and shift instructions of both architectures have also been implemented in VHDL. In the VHDL implementations of both architectures the same techniques and methods have been used. In order to provide a rough comparison of both architectures in terms of area and speed, the Xilinx FPGA Virtex-II Pro xc2vp30 device has been used. Obviously the hardware cost and speed can be improved by an ASIC implementation. The aim was to derive a rough comparison for both architectures. All designs have been implemented in VHDL and mapped onto the reconfigurable logic such as Look-Up Table (LUT). The dedicated arithmetic resources such as multipliers have not been used. The hardware implementations have been synthesized, placed, and routed using the Xilinx ISE tool. In order to compare both architectures, the ratio of the MMMX area in terms of utilized LUTs and critical path delay over the MMX area and critical path delay are presented.

Table 3.6 illustrates the utilized area and the critical path delay of the MMX and MMMX architectures. In addition, this table shows the ratio of utilized area and the critical path delay of the MMMX over the MMX architecture. As this table shows the area utilization of the register file of the MMMX architecture is 2.89 times larger than the register file of the MMX architecture. This is because in the former eight 96-bit registers, 672 2:1 multiplexers, and one 4:16 decoder is used, while in the latter, eight 64-bit registers and one 3:8 decoder are sufficient. In addition, the timing result shows that the critical path delay of the MRF is 5% larger than the critical path delay of the MMX register file.

Hardware Component		MMX	MMMX	MMMX / MMX
Register File (RF)	Area (# LUTs)	520	1504	2.89
	Delay (ns)	8.08	8.50	1.05
Partitioned ALU	Area (# LUTs)	2424	3429	1.41
	Delay (ns)	20.45	26.50	1.30
Partitioned ALU + 12/16-bit MULT	Area (# LUTs)	5617	6417	1.14
	Delay (ns)	19.27	26.17	1.35
Partitioned ALU + 12,24/16 MULT	Area (# LUTs)	5617	12764	2.27
	Delay (ns)	19.27	27.04	1.40
RF + Partitioned ALU + 12,24/16 MULT	Area (# LUTs)	6.023	14359	2.38
	Delay (ns)	19.98	28.16	1.41

Table 3.6: *The area utilization in terms of LUTs and the critical path delays (ns) of the MMX and MMMX architectures as well as the ratio of utilized area and the critical path delay of MMMX over MMX for their register file architecture, partitioned ALU, and the whole hardware system.*

The partitioned ALU of the MMMX architecture is 1.41 times larger than the partitioned ALU of the MMX architecture. The former ALU is 30% slower than the latter ALU. The critical path delay is because of the subword adder.

The partitioned ALU and multiplication unit of the MMMX architecture are 2.27 times larger than the partitioned ALU and multiplication unit of the MMX architecture. This is because of the following reasons. First, the partitioned ALU of the MMMX architecture is wider than the partitioned ALU of the MMX architecture. Second, there are more general SIMD instructions in the MMMX ISA such as full 12- and 24-bit multiplications. The overhead instructions of the MMX architecture depicted in the last row of Table 3.5 were not considered. The critical path delay of MMMX, which is related to 24-bit multiplication, is 40% longer than the critical path of the MMX ALU. Finally, the whole hardware system of the MMMX architecture is 2.38 times larger than the MMX architecture and its critical path delay is 41% longer than the critical path delay of MMX.

It needs to be mentioned that pipelining of the multiplication operations has not been considered as the aim was to provide a rough comparison between the MMX and MMMX architectures in terms of maximum combinational logic delay.

3.4 Conclusions

This chapter has described the MMMX architecture. The MMMX architecture is based on the MMX architecture but enhanced with extended subwords and the matrix register file. The extended subwords technique extends each media register with 32-bit. This technique alleviates data type conversion instructions and increases the number of subwords that can be processed simultaneously. The matrix register file provides both row-wise and column-wise access to media register file. In the MMMX architecture, column-wise access on the write port of the register file has been provided and this reduces the hardware cost compared to providing column-wise access on the read ports. This is because the number of write ports of the media register file is usually less than the number of read ports. The MRF technique avoids data rearrangement instructions in 2D multimedia kernels. The main reason for this is that this technique can be used to reorganize strided data and to transpose a matrix.

In addition, this chapter has presented new and general SIMD instructions addressing the multimedia domain. It has described different load/store, ALU, and multiplication instructions. There are two kinds of load instructions, namely normal load and load-column instructions. A single bit in the instruction format of load instructions can be used in order to distinguish between normal load instructions and load-column instructions. Only load-column instructions can access the column registers, while the other instructions cannot. Additionally, this chapter has explained different multiplication instructions such as full multiplication, MAC operations, and multiplication with truncation and rounding.

Furthermore, the hardware overhead of the MMMX architecture in terms of area utilization and critical path delay has been discussed. In general, the MMMX architecture is 2.38 times larger than the MMX architecture and the critical path delay of MMMX is 41% larger than the critical path of the MMX architecture.

In the next chapter the performance of the MMMX architecture at kernel-, image-, and application-level will be evaluated. For this, first some MMAs are selected and profiled in order to find the most time consuming kernels. Then, these multimedia kernels are implemented using both MMX and MMMX and these SIMD implementations replace the original scalar version of the applications in order to obtain the application-level speedup.

Chapter 4

Performance Evaluation

This chapter evaluates the MMMX architecture by comparing the performance of MMMX implementations of several kernels and applications to the performance of MMX implementations. For this evaluation, a number of important multimedia benchmarks such as an MPEG-2 encoder/decoder (codec), a JPEG codec, and MJPEG are selected. The performance is obtained for kernel-, image-, and application-level. Kernels are the most time consuming functions and represent a major portion of multimedia applications. For example, motion estimation, color space conversions, 2D (I)DCT, repetitive padding, and 2D DWT are some of the most time consuming kernels in existing MMAs [126, 127]. Some multimedia kernels such as color space conversions are defined using floating-point arithmetic, while as mentioned in previous chapters, the MMMX architecture is a fixed-point SIMD extension. This means that the floating-point kernels are implemented using fixed-point arithmetic. In order to obtain the performance, the `sim-outorder` simulator of the SimpleScalar toolset has been selected as the evaluation environment. The `sim-outorder` is a detailed, execution-driven simulator that supports out-of-order issue and execution.

The chapter is organized as follows. Section 4.1 describes the MMAs and kernels selected for performance evaluation. Section 4.2 presents the algorithms of the media kernels as well as their SIMD implementations using both the MMX and MMMX architectures. This section shows how the proposed techniques, extended subwords and the matrix register file, can be used to reduce the dynamic number of instructions. Section 4.3 presents the experimental methodology and tools. Section 4.4 describes the obtained performance improvement of MMMX over MMX for block-level, image-level, and application-level. Finally, Section 4.5 presents some conclusions.

Multimedia Applications	Description
JPEG encode	JPEG encoder is a DCT-based lossy image compression that is used for digital pictures.
JPEG decode	Decoding digital pictures in JPEG format.
MJPEG	Motion JPEG is used to compress digital video without using block-based motion estimation algorithms.
MPEG-2 encode	MEPG-2 encoding is used to compress digital video using block-based motion estimation algorithms.
MPEG-2 decode	MPEG-2 decoder is used to decompress digital video using block-based motion estimation algorithms.

Table 4.1: *Summary of some multimedia standards.*

4.1 Benchmarks

In this section, some multimedia standards and details of several computationally important kernels are described.

4.1.1 Multimedia Standards

Some multimedia standards have been selected in order to validate the proposed architecture. Table 4.1 summarizes the benchmarks that are used for performance evaluation in this chapter. The Joint Photographic Experts Group (JPEG) and Moving Picture Experts Group (MPEG) established the JPEG image compression and the MPEG video compression standards, respectively. Most image and video compression standards have an encoder and a decoder, which define how an image or video is compressed into a stream of bytes and decompressed back into an image or video. In the image encoder there are usually the following functions that are performed in order: RGB-to-YCbCr color space conversion, downsampling, block splitting, DCT, quantization, and encoding. The decoder performs the inverse of these kernels in reverse order. For example, YCbCr-to-RGB color space conversion is used in the decoder. The block diagram of a JEPG encoder and decoder is shown in Figure 4.1. If the input images are in RGB color space then the RGB-to-YCbCr color space conversion is the first step in the encoder stage. In the encoder $2:1$ horizontal downsampling with $2:1$ vertical downsampling is employed for chrominance information (Cb and Cr) in order to reduce the encoded data with little or no perceptual effect. In order to remove redundant image data, DCT is applied on each block of pixels provided by block splitting. After that each block of DCT coefficients is quantized using

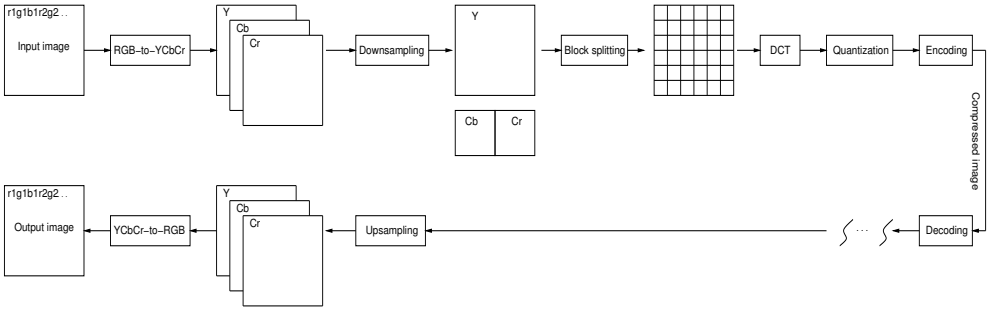


Figure 4.1: *A typical block diagram of an encoder and decoder of the JPEG standard.*

weighting functions. Finally, the resulting coefficients are encoded by some techniques such as Huffman coding in order to remove redundancies in the coefficients.

In the MPEG-2 video codecs, in addition to the mentioned kernels, there are some extra kernels such as motion estimation and motion compensation. Motion JPEG (MJPEG) represents a digital video sequence as a series of JPEG pictures. This means that it does not use motion estimation.

The JPEG and MPEG-2 standards use a block-based DCT transform [117]. The input images are divided into 8×8 disjoint blocks and each of them is transformed with the DCT as depicted in Figure 4.1.

In the following section, some multimedia kernels that take the most computational time of the multimedia standards, are described.

4.1.2 Multimedia Kernels

As mentioned in Chapter 1, most of the execution time of MMAs is spent in multimedia kernels. Therefore, in order to evaluate the proposed techniques, some time consuming kernels of multimedia standards and Content-Based Image and Video Retrieval (CBIVR) systems have been considered. Table 4.2 lists the media kernels along with a small description. In order to clarify which proposed techniques have been used in SIMD implementations of media kernels, the presented kernels are divided into two groups. First, kernels that use both extended subwords and the MRF techniques, for instance, the first seven kernels. Second, kernels that just use extended subwords technique, for example, the rest of the kernels (thirteen kernels).

The 2D transforms such as Discrete Wavelet Tranform (DWT) and (I)DCT are decomposed into two 1D transforms called horizontal and vertical filtering. The horizontal filtering (processing) processes the rows followed by vertical filtering (pro-

Multimedia Kernels	Description
Matrix transpose	Matrix transposition is an important kernel for many 2D media kernels.
Vector/Matrix Multiply	Vector/matrix multiply kernel is used in some multimedia standards.
Repetitive Padding	In this kernel, the pixel values at the boundary of the video object is replicated horizontally as well as vertically.
RGB-to-YCbCr	Color space conversion, which is usually used in the encoder stage.
Horizontal DCT	Horizontal DCT in used in most media standards to process the rows of images in order to remove spatial redundancy.
Horizontal IDCT	Horizontal Inverse DCT is used in the multimedia standards in order to reconstruct the rows of the transformed images.
Horizontal filtering of the DWT	The horizontal filtering of the discrete wavelet transform is used to process the whole rows of an image.
Vertical filtering of the DWT	The vertical filtering of the DWT is used to process the whole columns of an image.
Vertical DCT	Vertical DCT in used in most media standards to process the columns of images in order to remove spatial redundancy.
Vertical IDCT	Vertical IDCT is used in the multimedia standards in order to reconstruct the columns of the transformed images.
Add block	The add block is used in the decoder, during the block reconstruction stage of motion compensation.
2×2 Haar transform	The 2×2 haar transform is used to decompose an image into four different bands.
Inverse 2×2 Haar transform	The inverse 2×2 haar transform is used to reconstruct the original image from different bands.
Paeth prediction	Paeth prediction is used in the PNG standard.
YCbCr-to-RGB	Color space conversion, which is usually used in the decoder stage.
SAD function	The SAD function, which is used in motion estimation kernel to remove temporal redundancies between video frames.
SAD function with interpolation	The SAD function with horizontal and vertical interpolation is used in motion estimation algorithm.
SAD function for image histograms	The SAD function is used for similarity measurements of image histograms.
SSD function	The SSD function, which is used in motion estimation kernel to remove temporal redundancies between video frames.
SSD function with interpolation	The SSD function with horizontal and vertical interpolation is used in motion estimation algorithm.

Table 4.2: *Summary of multimedia kernels.*

cessing) that processes the columns. The efficiency of using SIMD approach for the vertical filtering is better than the horizontal filtering (in a row-major storage format). This is because elements of each loop iteration are adjacent in memory. In order to increase DLP in SIMD implementation of vertical filtering, the extended subwords technique is used, while in SIMD implementation of horizontal filtering both proposed techniques are needed in order to increase DLP and also to avoid data rearrangement instructions. In other words, in the MMMX implementations of both horizontal and vertical processing, the extended subwords technique is employed. In addition to the extended subwords technique, the MRF is also needed in the MMMX implementation of horizontal processing.

The presented kernels in Table 4.2 are responsible for a large portion of execution

time and contain a substantial amount of DLP [95, 50]. For example, the profiling of a JPEG codec shows that RGB-to-YCbCr and YCbCr-to-RGB consume an average of 13.1% and 28.7% of the total execution time, respectively. Other researchers [13, 12] have reported that color space conversion consumes up to 40% of the entire processing time of a highly optimized decoder. Additionally, in [82, 109, 128] it has been indicated that motion estimation takes about 60% to 80% of the encoding time. The sum-of-absolute differences and sum-of-squared differences are usually used in the motion estimation algorithm.

This means that the applications can be accelerated by speeding up the kernels. In order to accelerate a media kernel, but still keep it correct, it is first necessary to understand its algorithm. Therefore, the next section briefly explains the algorithm of multimedia kernels and sketches their MMX and MMMX implementations.

4.2 Algorithm and SIMD Implementation of Kernels

This section briefly describes the algorithms of the selected multimedia kernels as well as their SIMD implementations using both the MMX and MMMX architectures.

4.2.1 Matrix Transpose

Matrix transposition is at the center of many 2D multimedia algorithms. Because of this, it is considered as a kernel benchmark.

The MMMX implementation of the matrix transpose is straightforward. Iteratively, a vector is loaded from memory and written to a column of the register file. Once a sub-matrix has been transposed in this way, it is written back to memory. The MMX implementation, on the other hand, is more difficult and requires many permutation instructions. For example, transposing an 8×8 matrix consisting of single-byte elements requires 56 MMX/SSE instructions to be executed. Figure 4.2 depicts a part of the MMX/SSE implementation to transpose an 8×8 block. After this code, only the first four columns have been transposed. The transposed columns are stored in `mm0`, `mm5`, `mm1`, and `mm2` registers.

In this chapter, the matrix transpose kernel was implemented for two different data types, byte and 12-bit values. For brevity, they will be referred to as Transp. (8) and Transp. (12), respectively. A 12-bit data format arises, for example, in the IDCT. In memory, however, these values are stored as 16-bit.

In order to transpose a matrix larger than 8×8, splitting technique is employed. In this technique, a matrix A of size $N \times N$, where $N = 2^n$, can be transposed by

movq	mm0,	(blk1)	; mm0 =	a07	a06	a05	a04	a03	a02	a01	a00
movq	mm1,	8(blk1)	; mm1 =	a17	a16	a15	a14	a13	a12	a11	a10
movq	mm2,	16(blk1)	; mm2 =	a27	a26	a25	a24	a23	a22	a21	a20
movq	mm3,	24(blk1)	; mm3 =	a37	a36	a35	a34	a33	a32	a31	a30
movq	mm4,	32(blk1)	; mm4 =	a47	a46	a45	a44	a43	a42	a41	a40
movq	mm5,	40(blk1)	; mm5 =	a57	a56	a55	a54	a53	a52	a51	a50
movq	mm6,	48(blk1)	; mm6 =	a67	a66	a65	a64	a63	a62	a61	a60
movq	mm7,	56(blk1)	; mm7 =	a77	a76	a75	a74	a73	a72	a71	a70
punpcklbw	mm0,	mm1	; mm0 =	a13	a03	a12	a02	a11	a01	a10	a00
punpcklbw	mm2,	mm3	; mm2 =	a33	a23	a32	a22	a31	a21	a30	a20
punpcklbw	mm4,	mm5	; mm4 =	a53	a43	a52	a42	a51	a41	a50	a40
punpcklbw	mm6,	mm7	; mm6 =	a73	a63	a72	a62	a71	a61	a70	a60
movq	mm1,	mm0	; mm1 =	a13	a03	a12	a02	a11	a01	a10	a00
movq	mm3,	mm4	; mm3 =	a53	a43	a52	a42	a51	a41	a50	a40
punpcklwd	mm0,	mm2	; mm0 =	a31	a21	a11	a01	a30	a20	a10	a00
punpcklwd	mm4,	mm6	; mm4 =	a71	a61	a51	a41	a70	a60	a50	a40
movq	mm5,	mm0	; mm5 =	a31	a21	a11	a01	a30	a20	a10	a00
punpckldq	mm0,	mm4	; mm0 =	a70	a60	a50	a40	a30	a20	a10	a00
punpckhdq	mm5,	mm4	; mm5 =	a71	a61	a51	a41	a31	a21	a11	a01
punpckhwd	mm1,	mm2	; mm1 =	a33	a23	a13	a03	a32	a22	a12	a02
punpckhwd	mm3,	mm6	; mm3 =	a73	a63	a53	a43	a72	a62	a52	a42
movq	mm2,	mm1	; mm2 =	a33	a23	a13	a03	a32	a22	a12	a02
punpckldq	mm1,	mm3	; mm1 =	a72	a62	a52	a42	a32	a22	a12	a02
punpckhdq	mm2,	mm3	; mm2 =	a73	a63	a53	a43	a33	a23	a13	a03

Figure 4.2: *A part of the MMX/SSE code to transpose an* 8×8 *block.*

splitting the matrix into four sub-matrices A_{ij}, $i, j = 1, 2$, of size $N/2 \times N/2$ and by transposing the sub-matrices recursively as shown in the following equations.

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \qquad A^T = \begin{bmatrix} A_{11}^T & A_{21}^T \\ A_{12}^T & A_{22}^T \end{bmatrix}.$$

4.2.2 Vector/Matrix Multiply

Vector/matrix multiplication is another important kernel with many applications. For example, it has been used to implement FIR filters [26], the discrete wavelet transform [124], and, of course, matrix/matrix multiplication. The naive vector/matrix

multiply algorithm traverses the matrix along the columns. Because in C matrices are typically stored in row-major order, this leaves the columns scattered in memory. This kernel will be referred to as V x M.

In [34] two MMX implementations of vector/matrix multiplication have been presented. In the first implementation the matrix is split into 4×2 sub-matrices and the vector is also split into sub-vectors of two elements. The C algorithm of this method is depicted in Figure 4.3. In addition, the MMX implementation of the inner loop is illustrated in Figure 4.4.

```
int16  Vec[N];
int16  Mat[N][M];
int16  Res[M];
int32  Acc[4];
for (i=0 i<M; i+=4) {
  Acc[0..3] = 0;
  for (j=0; j<N; j+=2)
    Acc[0..3] += Mult4x2(Vec[j], Mat[j][i]);
  Res[i..i+3] = Acc[0..3];
}
```

Figure 4.3: *Pseudo C code for vector matrix multiply.*

```
 mov        eax,  N
 pxor       mm2,  mm2      ; mm2 = |0   |0   |
 pxor       mm3,  mm3      ; mm3 = |0   |0   |
loop2:
 movd       mm7,  (Vec)    ; mm7 = |0   |0   |v1  |v0  |
 punpckldq  mm7,  mm7      ; mm7 = |v1  |v0  |v1  |v0  |
 movq       mm0,  (Mat)    ; mm0 = |a03 |a02 |a01 |a00 |
 movq       mm6,  2*M(Mat) ; mm6 = |a13 |a12 |a11 |a10 |
 movq       mm1,  mm0      ; mm1 = |a03 |a02 |a01 |a00 |
 punpcklwd  mm0,  mm6      ; mm0 = |a11 |a01 |a10 |a00 |
 punpckhwd  mm1,  mm6      ; mm1 = |a13 |a03 |a12 |a02 |
 pmaddwd    mm0,  mm7      ; mm0 = |a11*v1+a01+v0|...  |
 pmaddwd    mm1,  mm7      ; mm1 = |a13*v1+a03+v0|...  |
 paddd      mm2,  mm0      ; mm2 = |0+a11*v1+a01+v0|...|
 paddd      mm3,  mm1      ; mm3 = |0+a13*v1+a03+v0|...|
 add        Mat,  4*M      ; index to row2
 add        Vec,  4        ; index to element v2
 sub        eax,  2
 jnz        loop2
```

Figure 4.4: *The MMX implementation of the inner loop that has been shown in Figure 4.3.*

This algorithm processes four columns of the matrix in parallel (Mult4x2 function)

and accumulates results in a set of four accumulators. However, the algorithm exhibits poor cache utilization. In the second method, the outer loop of the vector/matrix multiply algorithm is unrolled four times. This algorithm processes 16 columns of the matrix in each iteration of the inner loop, and each iteration of the outer loop calculates 16 elements of the output vector. The second algorithm performs better than the first method. In other words, the second algorithm is more optimized than the first technique. For instance, the results presented in [34] indicate that the optimized MMX implementation is up to 4 times faster than the unoptimized MMX implementation. In this thesis the MMMX implementation is compared to the optimized MMX implementation. For this kernel, a matrix size of 8×16 is used and the elements are assumed to be 12 bits wide (stored as 16-bit) as is the case in, for example, the Hadamard transform in image processing and in edge detection algorithm using different masking [55].

MMMX processes 16 columns in two loop iterations. In each loop iteration an 8×8 sub-matrix is processed. In order to process each sub-matrix, MMMX, first transposes each sub-matrix using eight load column instructions after that it employs MAC and `fsum` instructions. In other words, this kernel mainly benefits from the 8×12-bit multiply-add operation provided in MMMX.

4.2.3 Repetitive Padding

An important new feature in MPEG-4 is *padding*. Profiling results reported in [19, 20, 143], indicate that padding is a computationally demanding process. For example, the results presented in [143] indicates that 13% of the execution time in the MPEG-4 decoder is spent in padding kernel.

MPEG-4 defines Video Object Planes (VOPs) as arbitrarily shaped regions of a frame which usually correspond to objects. Motion estimation is defined over VOPs instead of frames. The padding process defines the color values of pixels outside the VOP. It consists of two steps. First, each horizontal line of a block is scanned. If a pixel is outside the VOP and between an end point of the line and an end point of a segment inside the VOP, then it is replaced by the value of the end pixel of the segment inside the VOP. Otherwise, if the pixel is outside the VOP and between two end points of segments inside the VOP, it is replaced by the average of these two end points. In the second step the same procedure is applied to each vertical line of the block. In other words, the repetitive padding kernel consists of horizontal and vertical repetitive padding that their functionality is the same. The vertical repetitive padding is equivalent to performing a transpose operation on the pixel and shape matrices obtained in the horizontal repetitive padding stage, followed by a second iteration of horizontal repetitive padding. Figure 4.5 illustrates horizontal and vertical repetitive padding for an 8×8 pixel block. In this figure VOP boundary pixels are indicated by

a numerical value, interior pixels are denoted by X, and pixels outside the VOP are blank.

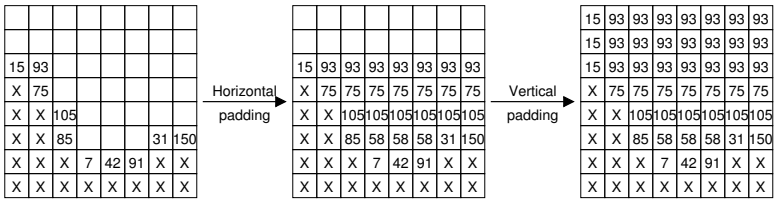


Figure 4.5: *Repetitive padding for VOP boundary blocks.*

The algorithm described in [14] has been used in both the MMX and the MMMX implementations. That algorithm has two horizontal and vertical repetitive padding the same as the previous one. The horizontal repetitive padding of the new algorithm is illustrated in Figure 4.6. It consists of two stages. First, all pixel values are scanned from left to right. In addition, there is a bit flag for each pixel. It indicates if a pixel value is kept from the original value or has to be replaced by its left value. After the first stage, all positions that replaced by a new value have a set flag, while for positions that their values have not been changed have a zero value in their corresponding flags. For example, as Figure 4.6 shows the 85 value is propagated three times from left to right. Second, the intermediate results are scanned from right to left. In other words, pixel values are propagated from the right. If a position has already received a new value in the first stage, the correct value for this position is the average of the present value and the value propagated from the right. For instance, three replaced positions in the first stage are replaced by $(85 + 31)/2 = 58$ value.

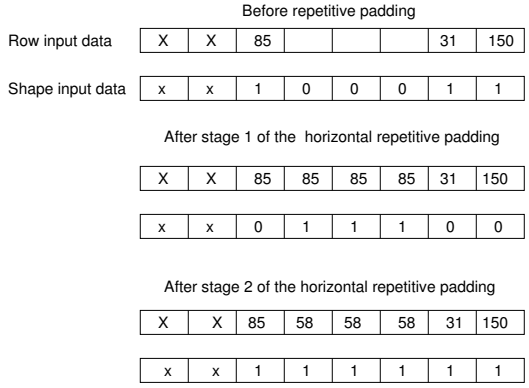


Figure 4.6: *An example of the horizontal repetitive padding using the described algorithm in [14].*

In this algorithm, special instructions have been proposed for both horizontal as well as vertical repetitive padding [14]. In the MMX implementation both input pixel

block and binary shape information have to transposed. In addition, MMX employs the special-purpose `pavgb` instruction which computes the arithmetic average of 8 pairs of bytes. On the other hand, If column-wise access to the register file is supported, then both horizontal and vertical repetitive padding can be performed identically and efficiently using the MMMX instructions. Additionally, MMMX synthesizes the special-purpose `pavgb` instruction using the more general-purpose instructions `fadd12` and `fsar12` (shift arithmetic right on extended subwords).

4.2.4 (Inverse) Discrete Cosine Transform

The discrete cosine transform and its inverse are widely used in several image and video compression applications. JPEG and MPEG partition the input image into 8×8 blocks and perform a 2D DCT on each block. The input elements are often either 8- or 9-bit, and the output is an 8×8 block of 12-bit 2's complement data.

A 2D DCT is efficiently computed by performing a 1D DCT on each row (horizontal DCT) followed by a 1D DCT on each column (vertical DCT). There are many different algorithms to implement the 2D DCT. One algorithm is using a transform matrix. The $M \times M$ transform matrix T, (t_{ij}) $0 \leq i, j \leq M-1$ is given by Equation (4.1).

$$T_{ij} = \begin{cases} \frac{1}{\sqrt{M}}, & \text{if } i = 0, 0 \leq j \leq M-1. \\ \sqrt{\frac{2}{M}} * \cos\frac{\pi(2j+1)i}{2M} & \text{if } 1 \leq i \leq M-1, 0 \leq j \leq M-1. \end{cases} \tag{4.1}$$

For an $M \times M$ matrix A, TA is an $M \times M$ matrix whose columns contain the 1D DCT of the columns of A. The 2D DCT of A can be computed as $B = TAT^{'}$. Since T is a real orthonormal matrix, its inverse is the same as its transpose [98]. Therefore, the inverse 2D DCT of B is given by $T^{'}BT$.

The second algorithm to compute the 2D DCT is the LLM algorithm [96]. This algorithm performs a 1D DCT on each row of the 8×8 block followed by a 1D DCT on each column of the transformed 8×8 block. The algorithm has four stages, the output of each stage is the input of next stage. Figure 4.7 depicts the data flow graph of this algorithm for 8 pixels using fixed-point arithmetic.

Both C and MMX implementations of this algorithm are faster than the C and MMX implementations of the former algorithm that defines the DCT as a matrix multiplication. One main reason for this is that the number of multiplications in the matrix multiplication algorithm is more than the LLM algorithm. Therefore, the LLM algorithm has been used as a reference algorithm in this thesis.

In the MMX implementation of this algorithm, however, 16-bit functionality (4-way parallelism) has been used because the input data is either 8- or 9-bit. This means that

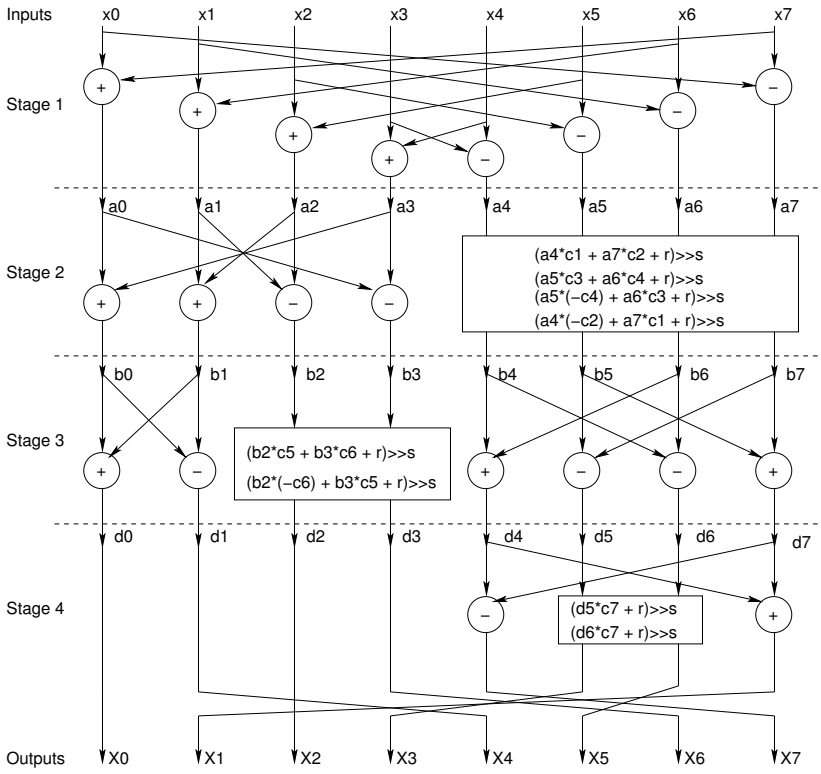


Figure 4.7: *Data flow graph of 8 pixels DCT using LLM [96] algorithm. The constant coefficients of c, r, and s are provided for fixed-point implementation.*

this kernel needs 16-bit storage format, while the intermediate results are smaller than 16-bit. Data type conversion instructions are not needed because four 16-bit can be loaded from memory to the four subwords of a media register. Although many rearrangement instructions are used in this implementation, this implementation exploits 4-way parallelism in all stages. Figure 4.8 depicts the MMX/SSE implementation of the first stage of the LLM algorithm for horizontal DCT. As this figure shows some rearrangement instructions are required in this implementation.

`movq mm0, (dct)`	; mm0 =	x03	x02	x01	x00
`movq mm3, 8(dct)`	; mm3 =	x07	x06	x05	x04
`pshufw mm1, mm0, 27`	; mm1 =	x00	x01	x02	x03
`pshufw mm2, mm3, 27`	; mm2 =	x04	x05	x06	x07
`paddsw mm0, mm2`	; mm0 =	x03+x04	x02+x05	x01+x06	x00+x07
`psubsw mm1, mm3`	; mm1 =	x00-x07	x01-x06	x02-x05	x03-x04

Figure 4.8: *The MMX/SSE code of the first stage of the LLM algorithm for horizontal DCT.*

MMMX processes eight rows in one iteration. A complete 8×8 block is loaded into eight column registers. After that row registers which have eight subwords are processed. Figure 4.9 depicts a part of the MMMX implementation of the LLM algorithm. In this figure, "X" refers to $xi0 \pm xi7$, where $0 \leq i \leq 7$. First, eight load column instructions are used to load a complete 8×8 block into column registers. After that two `fadd12` and `fsub12` instructions are needed to process 16 pixels simultaneously. In MMX, on the other hand, four instructions (two `pshufw` instructions, a `paddsw`, and a `psubsw` instructions) are required to process eight pixels.

`fldc16s12 3mxc0, (dct)` ; 3mxc0 =	x07	x06	x05	x04	x03	x02	x01	x00
`fldc16s12 3mxc1, 16(dct)` ; 3mxc1 =	x17	x16	x15	x14	x13	x12	x11	x10
`fldc16s12 3mxc2, 32(dct)` ; 3mxc2 =	x27	x26	x25	x24	x23	x22	x21	x20
`fldc16s12 3mxc3, 48(dct)` ; 3mxc3 =	x37	x36	x35	x34	x33	x32	x31	x30
`fldc16s12 3mxc4, 64(dct)` ; 3mxc4 =	x47	x46	x45	x44	x43	x42	x41	x40
`fldc16s12 3mxc5, 80(dct)` ; 3mxc5 =	x57	x56	x55	x54	x53	x52	x51	x50
`fldc16s12 3mxc6, 96(dct)` ; 3mxc6 =	x67	x66	x65	x64	x63	x62	x61	x60
`fldc16s12 3mxc7, 112(dct)` ; 3mxc7 =	x77	x76	x75	x74	x73	x72	x71	x70
`fst12s16s 112(dct), 3mx7` ; (mem) =	x77	x67	x57	x47	x37	x27	x17	x07
`fmov 3mx7 , 3mx0` ; 3mx7 =	x70	x60	x50	x40	x30	x20	x10	x00
`fadd12 3mx0 , 112(dct)` ; 3mx0 =	X	X	X	X	X	X	X	x00+x07
`fsub12 3mx7 , 112(dct)` ; 3mx7 =	X	X	X	X	X	X	X	x00-x07

Figure 4.9: *A part of the MMMX implementation for the horizontal DCT algorithm. "X" denotes to* $xi0 \pm xi7$*, where* $0 \leq i \leq 7$*.*

The IDCT is the inverse of the DCT and can be accomplished using the same algorithm except that the stages are reversed. Identical to the 2D DCT, the 2D IDCT is divided into two 1D IDCT, namely horizontal IDCT and vertical IDCT.

4.2.5 Discrete Wavelet Transform

The DWT provides a time-frequency representation of a signal. The DWT is a multiresolution technique that can analyze different frequencies by different resolutions. The DWT is computed by performing lowpass and highpass filtering of the image pixels as shown in Figure 4.10. In this figure, the lowpass and highpass filters are denoted by h and g, respectively. This figure depicts the three levels of the 2D DWT decomposition. At each level, the highpass filter generates detail image pixels information, while the lowpass filter produces the coarse approximations of the input image. At the end of each lowpass and highpass filtering, the outputs are downsampled by two ($\downarrow 2$). As was mentioned in Section 4.1.2, the 2D DWT is separable

and obtains from two 1D DWT. In other words, a 2D DWT can be performed by first performing a 1D DWT on each row, which is referred to horizontal filtering, of the image followed by a 1D DWT on each column, which is called vertical filtering.

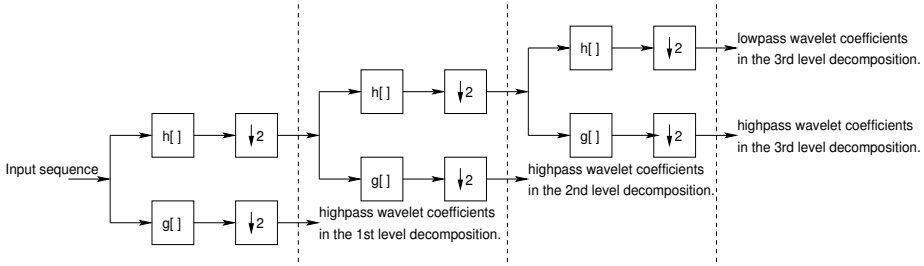


Figure 4.10: *Three level 2D DWT decomposition of an input image using filtering approach. The h and g variables denote the lowpass and highpass filters, respectively. The notation of $(\downarrow 2)$ refers to downsapling of the output coefficients by two.*

There are different approaches to implement 2D DWT such as traditional convolution-based and lifting scheme methods. The convolutional methods apply filtering by multiplying the filter coefficients with the input samples and accumulating the results. Their implementation is similar to the FIR implementation. This kind of implementation needs a large number of computations. The lifting scheme has been proposed for the efficient implementation of the 2D DWT. The lifting approaches need 50% less computations than the FIR approaches. The basic idea of the lifting scheme is to use the correlation in the image pixels values to remove the redundancy [38, 48]. The lifting scheme has three phases, namely: split, predict, and update, as illustrated in Figure 4.11. In the split stage, the input sequence is split into

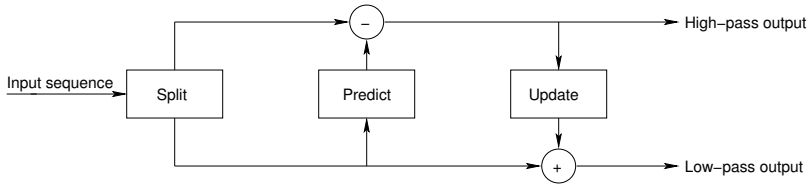


Figure 4.11: *Three different phases in the lifting scheme.*

two sub-sequences consisting of the even and odd samples. In the predict and update stages, the highpass and lowpass values are computed, respectively. One example of this group is the integer-to-integer $(5, 3)$ lifting scheme ($(5, 3)$ *lifting*). Figure 4.12 depicts the C code of the horizontal filtering of the $(5, 3)$ lifting transform for an $N \times M$ image. Since pixel values are assumed 8-bit, in the MMX implementation both data type and rearrangement instructions are needed. This is because the intermediate results are larger than 8-bit and many data permutation instructions are required to put the adjacent pixels in different registers. Figure 4.12 shows that load-

ing the adjacent pixels from (i, j), $(i, j-1)$, and $(i, j+1)$ locations are necessary to calculate a highpass value. On the contrary, in the MMMX implementation due to extended subwords and the MRF techniques those overhead instructions are avoided.

```
void Lifting53_horizontal() {
for (i=0; i< N; i++)
  for (j=1, jj=0; j<M; jj++, j +=2) {
    // calculation of highpass values
    tmp[i][jj+M/2] = img[i][j] - ((img[i][j-1] +
                     img[i][j+1]) >> 1);
    // calculation of lowpass values
    tmp[i][j] = img[i][j-1] + ((img[i][jj+M/2] +
                img[i][jj+M/2-1]+2) >> 2);
  }
}
```

Figure 4.12: *C implementation of the horizontal filtering of the* $(5, 3)$ *lifting scheme.*

4.2.6 Add Block

For encoding a block or macroblock in Intra-coded mode in standards such as MPEG-4, a prediction block is formed based on previously reconstructed blocks. The residual signal between the current block and the prediction is encoded. This residual signal data can be larger than 8-bit. The add block kernel is used in the decoder, during the block reconstruction stage of motion compensation. This kernel requires 9 bits of intermediate precision. Consequently, the MMX implementation needs to unpack the input data from 8- to 16-bit and pack the 16-bit result to 8-bit. In the MMMX implementation this overhead is not required. Figure 4.13 and Figure 4.14 depict the MMX and MMMX implementations of the inner loop of the add block kernel.

4.2.7 2×2 Haar Transform

The 2×2 Haar transform is used in image compression [145]. The 2×2 Haar transform is sometimes referred to as a wavelet. A 2D Haar transform can be performed by first performing a 1D Haar transform on each row (horizontal Haar transform) followed by a 1D Haar transform on each column (vertical Haar transform). This transform is used to decompose an image into four different bands. The 1D 2×2 Haar transform replaces adjacent pixel values with their sums and differences. Figure 4.15 depicts an example of the 2D 2×2 Haar transform for a 4×4 image. This transform generates four different subbands of lowpass and highpass values. A part of the MMX code for the inner loop of the 2D 2×2 Haar transform is depicted in Figure 4.16.

```
  mov       eax   ,  M
  pxor      mm7   ,  mm7
loop2:
  movq      mm0   ,  (Blk1)
  movq      mm1   ,  mm0
  punpcklbw mm0   ,  mm7
  punpckhbw mm1   ,  mm7
  paddsw    mm0   ,  (Blk2)
  paddsw    mm1   ,  (Blk2+8)
  packuswb  mm0   ,  mm1
  movq      (Blk1),  mm0
  add       Blk1  ,  8
  add       Blk2  ,  16
  dec       eax
  jnz       loop2
```

Figure 4.13: *The MMX implementation of inner loop of the add block kernel.*

```
  mov       eax   ,  M
loop2:
  fld8u12   3mx0  ,  (Blk1)
  fld16s12  3mx1  ,  (Blk2)
  fadd12    3mx0  ,  3mx1
  fst12s8   (Blk1),  3mx0
  add       Blk1  ,  8
  add       Blk2  ,  16
  dec       eax
  jnz       loop2
```

Figure 4.14: *The MMMX implementation of inner loop of the add block kernel.*

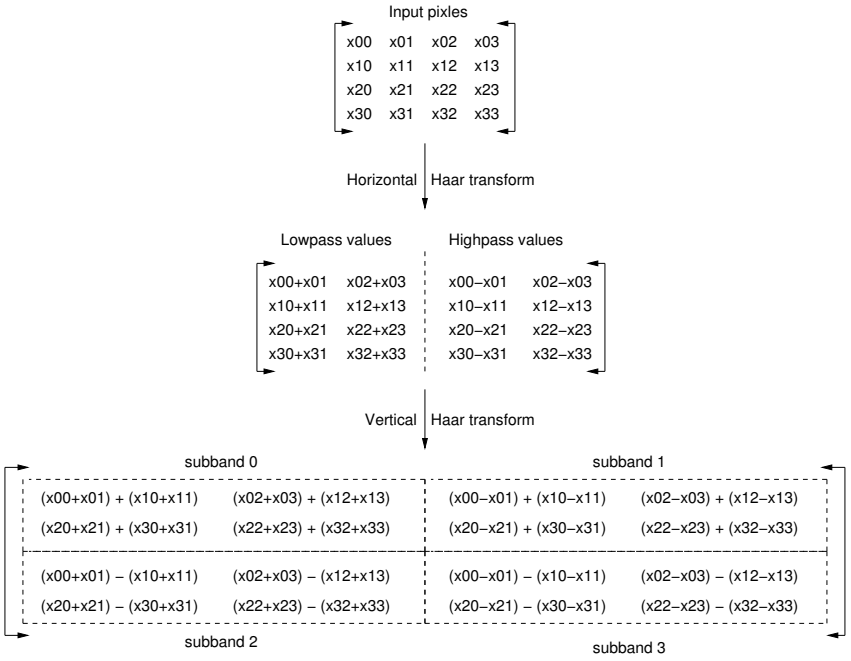


Figure 4.15: *2D 2×2 Haar transform using two 1D horizontal and vertical Haar transform.*

The detail implementation of this kernel has been presented in [68]. Both the horizontal and vertical Haar transform are combined together and they are performed in a single pass. First, sixteen byte values are loaded into mmo and $mm1$ registers from two consecutive rows, $row0$ and $row1$. The `punpcklbw` and `punpckhbw` instructions expand the data to two bytes. The `paddw` and `psubw` instructions are used to calculate the sums and differences of the loaded rows. After this, because both operands are in the same register and because MMX does not have an instruction that adds or subtracts adjacent elements, the instruction `pmaddwd` (MAC operation) with some multiplicands set to 1 and others to -1 (LHB01 and LHB23) is used for the final addition or subtraction.

```
  ; LHB01 and LHB23 are operands
  ; for MAC operation.
  LHB01      dw  1,  1, 1,  1
  LHB23      dw  1, -1, 1, -1
loop2:
  movq       mm0,  (row0)
  movq       mm1,  (row1)
  pxor       mm7,  mm7
  movq       mm4,  mm0
  movq       mm5,  mm1
  punpcklbw  mm0,  mm7
  punpckhbw  mm4,  mm7
  movq       mm2,  mm0
  punpcklbw  mm1,  mm7
  punpckhbw  mm5,  mm7
  paddw      mm0,  mm1
  psubw      mm2,  mm1
  movq       mm1,  mm0
  pmaddwd    mm0,  LHB01
  movq       mm3,  mm2
  pmaddwd    mm2,  LHB01
  movq       mm6,  mm4
  pmaddwd    mm1,  LHB23
  paddw      mm4,  mm5
  pmaddwd    mm3,  LHB23
  .
  .
  jnz       loop2
```

Figure 4.16: *A part of the MMX code for the 2D* 2×2 *Haar Transform.*

```
loop2:
  fld8u12  3mx0,  row0
  fld8u12  3mx1,  row1
  fmov     3mx2,  3mx0
  fadd12   3mx0,  3mx1
  fsub12   3mx2,  3mx1
  fmov     3mx1,  3mx0
  fsum12   3mx0
  fdiff12  3mx1
  fmov     3mx3,  3mx2
  fsum12   3mx2
  fdiff12  3mx3
  .
  .
  jnz       loop2
```

Figure 4.17: *A part of the MMMX code for the 2D* 2×2 *Haar Transform.*

The MMMX code for the 2D 2×2 Haar transform is depicted in Figure 4.17. In the MMMX implementation also both horizontal and vertical Haar transform are combined to each other in a single loop. MMMX calculates the sums and differences

of the pixels that are loaded from the consecutive rows by `fadd12` and `fsub12` instructions. In addition, MMMX has instructions that adds or subtracts adjacent subwords in the same register. Because of these, MMMX reduces the number of instructions in the inner loop by almost a factor of 2 (from 20 to 11) compared to MMX.

Both MMX and MMMX store different subbands in their appropriate places as depicted in Figure 4.15. Because of this, implementation of the inverse 2×2 Haar transform is easier than its forward transform. The inverse transform employs different subband data to construct 2×2 blocks. Figure 4.18 illustrates an example that shows how a 2×2 block is constructed by different subband data.

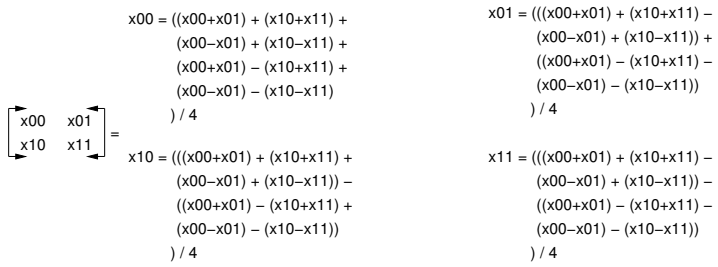


Figure 4.18: *An example of the inverse 2D 2×2 Haar transform that uses subbands data to construct a 2×2 block.*

Both MMX and MMMX can load data from four subbands into four SIMD registers. MMX loads four 16-bit values from a single subband into a register, while MMMX loads eight 16-bit values. There is no need for data permutation instructions in MMX. All sums and differences are performed with normal add and subtract instructions. MMX uses `packuswb` instruction in order to truncate results to one byte, while this operation is performed using store instructions in MMMX. In order to reorder the results and store them in their appropriate places, both architectures use unpack instructions.

4.2.8 Paeth Prediction

Paeth prediction is used in the PNG standard [114]. The Paeth predictor returns the pixel a, b, or c which is closest to the initial prediction $a + b - c$, where a is the element to the east of the current pixel d, b the element to the north, and c the element to the northeast as depicted in Figure 4.19. A pseudo-code description of the Paeth predictor (for one pixel) is given in Figure 4.20.

Because a, b, and c are unsigned bytes, $p = a + b - c$ is in the range $-255 \ldots 510$ (10 bits), $pa = |p - a| = |b - c|$ is in the range $0 \ldots 255$ (8 bits), $pb = |p - b| = |a - c|$

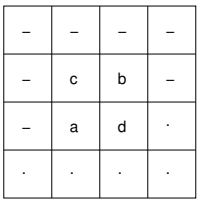


Figure 4.19: *Illustration of the a, b, and c pixels according to PNG specification.*

```
unsigned byte Paeth_predict(a, b, c)
{
   p = a+b-c     /* initial prediction */
   pa = |p-a|
   pb = |p-b|
   pc = |p-c|
   if (pa<=pb AND pa<=pc) return a
   else if (pb<=pc) return b
   else return c
}
```

Figure 4.20: *Pseudo-code description of the Paeth predictor.*

is also in the range $0 \ldots 255$, and $pc = |p - c| = |a + b - 2c|$ is in the range $-510 \ldots 510$ (10 bits). This implies that 12 bits are sufficient to perform this kernel without overflow. MMX, on the other hand, needs to unpack to 16-bit values and performs the main computation calculating the Paeth predictor value for four pixels using 4-way parallelism as depicted in Figure 4.21.

MMMX loads eight pixels values into registers and provides 8-way parallelism. The MMMX implementation computes the prediction for 8 pixels in a single iteration as a part of its implementation is shown in Figure 4.22.

4.2.9 Color Space Conversion

Multimedia standards usually use color space conversions of RGB-to-YCbCr and YCbCr-to-RGB in their encoder and decoder, respectively. Conversion between the YCbCr and RGB formats and vice versa can be represented with the following equations [108].

$$\begin{pmatrix} Y \\ Cb \\ Cr \end{pmatrix} = \begin{pmatrix} 0.256 & 0.502 & 0.098 \\ -0.148 & -0.290 & 0.438 \\ 0.438 & -0.366 & -0.071 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} + \begin{pmatrix} 16.5 \\ 128.5 \\ 128.5 \end{pmatrix} \tag{4.2}$$

```
  mov         eax,  M
  pxor        mm0,  mm0       ; mm0 = |0 |0 |0 |0 |0 |0 |0 |0 |
  pxor        mm7,  mm7       ; mm7 = |0 |0 |0 |0 |0 |0 |0 |0 |
loop1:
  movd        mm1,  (Paeth)   ; mm1 = |0 |0 |0 |0 |c3|c2|c1|c0|
  movd        mm2,  4(Paeth)  ; mm2 = |0 |0 |0 |0 |b3|b2|b1|b0|
  movd        mm3,  M(Paeth)  ; mm3 = |0 |0 |0 |0 |a3|a2|a1|a0|
  punpcklbw   mm1,  mm0       ; mm1 = | c3  | c2  | c1  | c0  |
  punpcklbw   mm2,  mm0       ; mm2 = | b3  | b2  | b1  | b0  |
  punpcklbw   mm3,  mm0       ; mm3 = | a3  | a2  | a1  | a0  |
  movq        mm4,  mm1       ; mm4 = | c3  | c2  | c1  | c0  |
  movq        mm5,  mm2       ; mm5 = | b3  | b2  | b1  | b0  |
  movq        mm6,  mm3       ; mm6 = | a3  | a2  | a1  | a0  |
  psubsw      mm2,  mm1       ; mm2 = |b3-c3|b2-c2|b1-c1|b0-c0|
  psubsw      mm3,  mm1       ; mm3 = |a3-c3|a2-c2|a1-c1|a0-c0|
  movq        mm1,  mm2       ; mm1 = |b3-c3|b2-c2|b1-c1|b0-c0|
  paddsw      mm1,  mm3       ; mm1 = |a3+b3-2c3|...|a0+b0-2c0|
  psubsw      mm7,  mm2       ; mm7 = |c3-b3|c2-b2|c1-b1|c0-b0|
  pmaxsw      mm2,  mm7       ; mm2 = max((bi-ci),(ci-bi))
  pxor        mm7,  mm7       ; mm7 = |0 |0 |0 |0 |0 |0 |0 |0 |
  psubsw      mm7,  mm3       ; mm7 = |c3-a3|c2-a2|c1-a1|c0-a0|
  pmaxsw      mm3,  mm7       ; mm3 = max((ai-ci),(ci-ai))
  .
  .
  sub         eax,  4
  jnz         loop1
```

Figure 4.21: *A part of the MMX code for the Paeth predictor kernel.*

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1.164 & 0.000 & 1.596 \\ 1.164 & -0.392 & -0.813 \\ 1.164 & 2.017 & 0.000 \end{pmatrix} \begin{pmatrix} Y - 16.5 \\ Cb - 128.5 \\ Cr - 128.5 \end{pmatrix} \tag{4.3}$$

In both equations, the coefficients have been rounded to three fractional decimal digits. As these equations show, color space conversions are defined using floating-point arithmetic but here, to avoid using floating-point operations, fixed-point arithmetic is used. Specifically, for MMX implementation, 16-bit fixed-point numbers are used, and for MMMX implementation the color space conversion is approximated by using 12-bit fixed-point arithmetic. To determine the accuracy of these approximations, two different tests have been performed. First, the maximum absolute error has been measured by checking all possible RGB values ($0 \leq R, G, B \leq 255$). For both the MMMX implementation (12-bit) and the MMX implementation (16-bit), the maximum absolute error compared to a single-precision floating-point implementation is 1. Second, the Mean Square Error (MSE) has been measured for real images as well

```
  mov      eax ,  M
loop1:
  fld8u12  3mx1,  (Paeth)    ; 3mx1 = |c7|c6|c5|c4|c3|c2|c1|c0|
  fld8u12  3mx2,  8(Paeth)   ; 3mx2 = |b7|b6|b5|b4|b3|b2|b1|b0|
  fld8u12  3mx3,  M(Paeth)   ; 3mx3 = |a7|a6|a5|a4|a3|a2|a1|a0|
  fmov     3mx4,  3mx1       ; 3mx4 = |c7|c6|c5|c4|c3|c2|c1|c0|
  fmov     3mx5,  3mx2       ; 3mx5 = |b7|b6|b5|b4|b3|b2|b1|b0|
  fmov     3mx6,  3mx3       ; 3mx6 = |a7|a6|a5|a4|a3|a2|a1|a0|
  fsub12   3mx2,  3mx1       ; 3mx2 = |b7-c7|b6-c6|.....|b0-c0|
  fsub12   3mx3,  3mx1       ; 3mx3 = |a7-c7|a6-c6|.....|a0-c0|
  fmov     3mx1,  3mx2       ; 3mx1 = |b7-c7|b6-c6|.....|b0-c0|
  fadd12   3mx1,  3mx3       ; 3mx1 = |a7+b7-2c7|...|a0-b0-2c0|
  fneg12   3mx7,  3mx2, 255  ; 3mx7 = |c7-b7|c6-b6|.....|c0-b0|
  fmax12   3mx2,  3mx7       ; 3mx2 = max((bi-ci),(ci-bi))
  fneg12   3mx7,  3mx3, 255  ; 3mx7 = |c7-a7|c6-a6|.....|c0-a0|
  fmax12   3mx3,  3mx7       ; 3mx3 = max((ai-ci),(ci-ai))
  .
  .
  sub      eax,   8
  jnz      loop1
```

Figure 4.22: *A part of the MMMX code for the Paeth predictor kernel.*

as randomly generated inputs. Figure 4.23 depicts the MSE of the 8-, 12-, and 16-bit implementations as a function of the image size. It shows that MSE of the 12- and 16-bit implementations are very close to each other while the MSE of the 8-bit implementation is much larger.

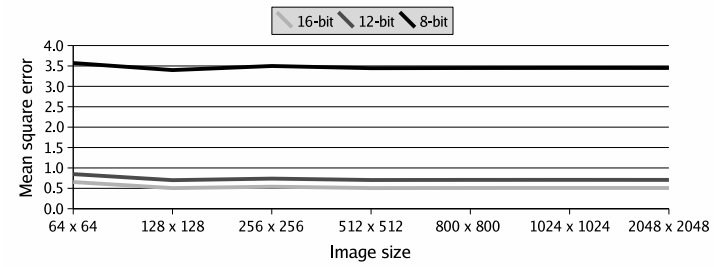


Figure 4.23: *Mean square error in the implementation of color space conversion for different bit widths and image sizes.*

In the remainder of this section, the SIMD implementation of the RGB-to-YCbCr and YCbCr-to-RGB kernels are discussed in detail.

The RGB values are usually in the band interleaved format, i.e., they are stored as $R_1G_1B_1$, $R_2G_2B_2$, $R_3G_3B_3$, etc. Because of this a straightforward MMX imple-

mentation of the RGB-to-YCbCr kernel is not efficient. This is because of the following reasons. First, image pixels must be unpacked from unsigned byte to 16-bit and vice versa because of the mismatch between the storage format and the computational format. This means that many data type conversion instructions are needed. Second, there are four 16-bit subwords in each MMX register and three R, G, and B values for each pixel. This implies that one of the subwords (a quarter of the processing capacity) will be unused. Third, there is no instruction in the MMX ISA that adds adjacent pixels. To synthesize this operation many shift instructions are used and basically perform scalar addition. Additionally, there are unaligned memory accesses because in each loop iteration two pixels are processed and the starting address of them is not necessarily a multiple of 8.

To efficiently implement this kernel compared to straightforward implementation, first the band interleaved format is converted to the band separated format using rearrangement instructions. Experimental results on an actual machine show that the MMX implementation using the band separated format is 4.20 times faster than the straightforward MMX implementation for an image of size 576×768. The faster method is used as the reference.

The MMX implementation using the band separated format consists of the following stages:

1. Load the RGB values of eight pixels into the media register file (3 instructions).
2. Conversion from band interleaved to band separated format using rearrangement instructions (35 instructions).
3. Unpack the packed byte data types to packed 16-bit word data types (6 instructions).
4. Shift the RGB values to the left by 7 bits (6 instructions).
5. Convert from RGB to YCbCr using 16-bit packed multiplication and addition instructions (51 instructions).
6. Truncate the results by shifting them to the right by 6 bits (6 instructions).
7. Pack the unpacked results and store in memory (12 instructions).

This MMX implementation is also not very efficient because many rearrangement and data type conversion instructions are required. For instance, 35 instructions are needed to convert 8 pixels from the band interleaved format to the band separated format. Figure 4.24 shows a part of the MMX code that achieves this rearrangement. It can be seen that many unpack, shift, and data transfer instructions are required to achieve this. As a result, both MMX implementations are inefficient.

`movq mm0, (RGB)`	; mm0 =	g3	r3	b2	g2	r2	b1	g1	r1
`movq mm2,8(RGB)`	; mm2 =	r6	b5	g5	r5	b4	g4	r4	b3
`movq mm1,16(RGB)`	; mm1 =	b8	g8	r8	b7	g7	r7	b6	g6
`movq mm3, mm0`	; mm3 =	g3	r3	b2	g2	r2	b1	g1	r1
`movq mm4, mm0`	; mm4 =	g3	r3	b2	g2	r2	b1	g1	r1
`psrlq mm3, 24`	; mm3 =	0	0	0	g3	r3	b2	g2	r2
`punpcklbw mm4, mm3`	; mm4 =	r3	r2	b2	b1	g2	g1	r2	r1
`movq mm6, mm2`	; mm6 =	r6	b5	g5	r5	b4	g4	r4	b3
`movq mm7, mm2`	; mm7 =	r6	b5	g5	r5	b4	g4	r4	b3
`psrlq mm3, 24`	; mm3 =	0	0	0	0	0	0	g3	r3
`psllq mm6, 16`	; mm6 =	g5	r5	b4	g4	r4	b3	0	0
`psrlq mm7, 8`	; mm7 =	0	r6	b5	g5	r5	b4	g4	r4
`por mm6, mm3`	; mm6 =	g5	r5	b4	g4	r4	b3	g3	r3
`punpcklbw mm6, mm7`	; mm6 =	r5	r4	b4	b3	g4	g3	r4	r3
`movq mm3, mm4`	; mm3 =	r3	r2	b2	b1	g2	g1	r2	r1
`punpcklwd mm4, mm6`	; mm4 =	g4	g3	g2	g1	r4	r3	r2	r1
`punpckhwd mm3, mm6`	; mm3 =	r5	r4	r3	r2	b4	b3	b2	b1
`movq mm0, mm2`	; mm0 =	r6	b5	g5	r5	b4	g4	r4	b3
`movq mm6, mm1`	; mm6 =	b8	g8	r8	b7	g7	r7	b6	g6
`psrlq mm2, 32`	; mm2 =	0	0	0	0	r6	b5	g5	r5
`psrlq mm0, 56`	; mm0 =	0	0	0	0	0	0	0	r6
`psllq mm6, 8`	; mm6 =	g8	r8	b7	g7	r7	b6	g6	0
`por mm0, mm6`	; mm0 =	g8	r8	b7	g7	r7	b6	g6	r6
`punpcklbw mm2, mm0`	; mm2 =	r7	r6	b6	b5	g6	g5	r6	r5
`movq mm0, mm1`	; mm0 =	b8	g8	r8	b7	g7	r7	b6	g6
`psrlq mm1, 16`	; mm1 =	0	0	b8	g8	r8	b7	g7	r7
`psrlq mm0, 40`	; mm0 =	0	0	0	0	0	b8	g8	r8
`punpcklbw mm1, mm0`	; mm1 =	0	r8	b8	b7	g8	g7	r8	r7
`movq mm6, mm2`	; mm6 =	r7	r6	b6	b5	g6	g5	r6	r5
`punpcklwd mm2, mm1`	; mm2 =	g8	g7	g6	g5	r8	r7	r6	r5
`punpckhwd mm6, mm1`	; mm6 =	0	r8	r7	r6	b8	b7	b6	b5
`movq mm1, mm4`	; mm1 =	g4	g3	g2	g1	r4	r3	r2	r1
`punpckldq mm1, mm2`	; mm1 =	r8	r7	r6	r5	r4	r3	r2	r1
`punpckhdq mm4, mm2`	; mm4 =	g8	g7	g6	g5	g4	g3	g2	g1
`punpckldq mm3, mm6`	; mm3 =	b8	b7	b6	b5	b4	b3	b2	b1

Figure 4.24: *The MMX instructions needed to convert RGB values from band interleaved format to band separated format.*

In the MMMX implementation of the RGB-to-YCbCr kernel, due to the MRF technique, changing from the band interleaved format to the band separated format is not needed as was illustrated in Figure 3.6 in the Section 3.2 of Chapter 3. In addition, the data type conversion instructions are avoided and 8-way parallelism is provided using the extended subwords technique. For example, the MMMX architecture needs 115 instructions to process 16 pixels, while the MMX architecture needs 300 instructions.

Figure 4.25 depicts an example that illustrates how the MMMX `fmul12h` instruction provides 8-way parallelism for this kernel. The coefficient value that should be multiplied with the green value of an image pixel is 0.502, as was depicted in Equation (4.2), approximated 0.502 by $1028/2^{11}$.

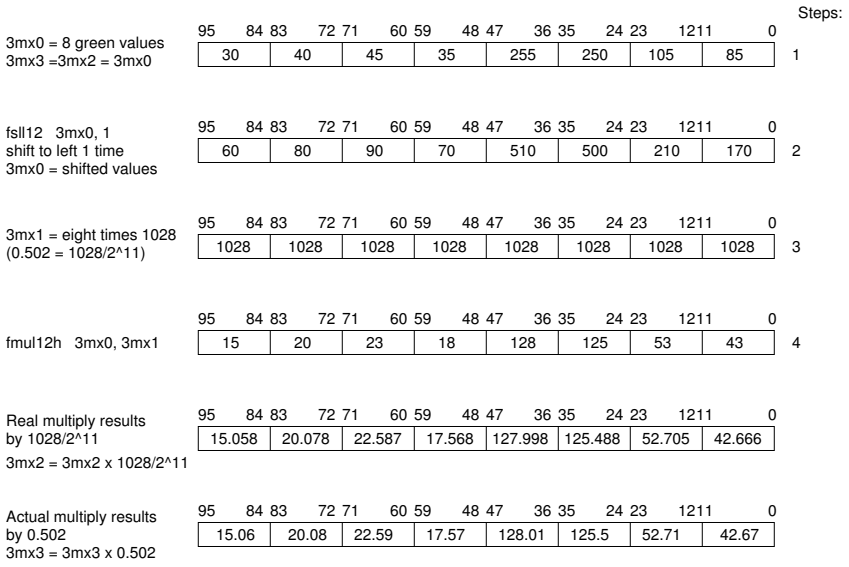


Figure 4.25: *Partitioned multiplication using the* `fmul12h` *instruction.*

This figure shows four steps. In the first step, it loads eight green pixel values into a media register (`3mx0`). In the second step, the subwords are shifted left by one bit. This is accomplished through MMMX's `fsll12` instruction. This is because the `fmul12h` instruction truncates the result between the 11th and 12th bit position of the internal 24-bit result. The lower 12 bits will be discarded. For this need to shift the subwords one bit to the left. The fixed-point coefficient 1028 would exceed the 12-bit signed range if it was shifted left by one bit. Because of this the first operand is shifted left. In the third step, the value 1028 is stored in another media register (3mx1) eight times. Finally, the shifted values are multiplied by the value 1028 using the `fmul12h 3mx0, 3mx1` instruction.

There can be some loss of precision due to this kind of instruction as already discussed in Section 3.3 of Chapter 3. The first error is due to quantizing. The coefficient

is 0.502, while $1028/2^{11} = 0.501953125$. The second reason for loss of precision is due to the nature of truncation. In order to reduce the effect of this error, first, internally round the intermediate 24-bit result and after that truncate the 12-bit result. As a result, if one compares the fixed-point results with the floating-point results shown in the last row of Figure 4.25, one can see there is a small error. Specifically, the rounded result of multiplying the third subword 250 with $1028/2^{11}$ is 125, while the rounded result of multiplying 250 with 0.502 is 126.

Both MMX and MMMX store the calculated Y, Cb, and Cr values in separate arrays. This causes that the SIMD implementation of the YCbCr-to-RGB kernel be almost easier than the SIMD implementation of the RGB-to-YcbCr kernel. MMX and MMMX can load eight Y, eight Cb, and eight Cr values in different registers. They do not need the data permutation instructions and the MRF technique, respectively. MMX unpacks the loaded values to 16-bit by data type conversion instructions, while MMMX does not. This means that MMX employs 4-way parallelism, while MMMX provides 8-way parallelism.

Figure 4.26 and Figure 4.27 show a part of the MMX and MMMX codes for the YCbCr-to-RGB kernel based on the Equation (4.3). The floating-point coefficient 16.5 must be subtracted from Y values and floating-point coefficient 128.5 from Cb and Cr values as well. These coefficients are provided by $1056/2^6 = 16.5$ and $8224/2^6 = 128.5$ in the MMX implementation, while they are provided by $66/2^2 = 16.5$ and $514/2^2 = 128.5$ in the MMMX implementation. This means that MMX and MMMX, first shifts the loaded values to the left by 6 and 2 bits, respectively, and after that the shifted values are subtracted by the fixed-point coefficients. For example, as Figure 4.26 shows, MMX shifts the pixel values by 6-bit to the left and subtracts them by fixed-point coefficients 1056 and 8224 values. The remaining floating-point coefficients in Equation (4.3) are approximated by 10-bit in both architectures. For instance, the floating-point coefficient 1.164 is replaced by $1.164 * 2^{10} = 1192$. Finally, the partitioned multiplication `pmulhw` and `fmul12h` instructions truncates the lower 16- and 12-bit results in MMX and MMMX, respectively.

4.2.10 Similarity Measurements

Among the different similarity measurements, the sum-of-squared differences and the sum-of-absolute differences functions have been found to be the most useful [142, 133, 146, 144]. For example, in [146] eight similarity measurements for image retrieval have been evaluated. Based on the results presented there, in terms of retrieval effectiveness and retrieval efficiency, the SSD and SAD functions are more desirable than other functions.

The SSD and SAD cost functions of two $N \times N$ blocks for motion estimation are de-

```
  Const1     dw    1056, 1056, 1056, 1056  ; 1056 / 2^6 = 16.5
  Const2     dw    8224, 8224, 8224, 8224  ; 8224 / 2^6 = 128.5
  Const3     dw    1192, 1192, 1192, 1192  ; 1192 / 2^6 = 1.164
  mov        eax,  M
loop1:
  pxor       mm0,  mm0    ; mm0 = |0  |0  |0  |0  |0  |0  |0  |0  |
  movq       mm1,  (Y)    ; mm1 = |y7 |y6 |y5 |y4 |y3 |y2 |y1 |y0 |
  movq       mm2,  (Cb)   ; mm2 = |cb7|cb6|cb5|cb4|cb3|cb2|cb1|cb0|
  movq       mm3,  (Cr)   ; mm3 = |cr7|cr6|cr5|cr4|cr3|cr2|cr1|cr0|
  movq       mm5,  mm1    ; mm5 = |y7 |y6 |y5 |y4 |y3 |y2 |y1 |y0 |
  movq       mm6,  mm2    ; mm6 = |cb7|cb6|cb5|cb4|cb3|cb2|cb1|cb0|
  movq       mm7,  mm3    ; mm7 = |cr7|cr6|cr5|cr4|cr3|cr2|cr1|cr0|
  punpcklbw  mm1,  mm0    ; mm1 = |0  |y3 |0  |y2 |0  |y1 |0  |y0 |
  punpcklbw  mm2,  mm0    ; mm2 = |0  |cb3|0  |cb2|0  |cb1|0  |cb0|
  punpcklbw  mm3,  mm0    ; mm3 = |0  |cr3|0  |cr2|0  |cr1|0  |cr0|
  punpckhbw  mm5,  mm0    ; mm5 = |0  |y7 |0  |y6 |0  |y5 |0  |y4 |
  punpckhbw  mm6,  mm0    ; mm6 = |0  |cb7|0  |cb6|0  |cb5|0  |cb4|
  punpckhbw  mm7,  mm0    ; mm7 = |0  |cr7|0  |cr6|0  |cr5|0  |cr4|
  psllw      mm1,  6      ; mm1 = mm1 x 2^6
  psllw      mm2,  6      ; mm2 = mm2 x 2^6
  psllw      mm3,  6      ; mm3 = mm3 x 2^6
  psllw      mm5,  6      ; mm5 = mm5 x 2^6
  psllw      mm6,  6      ; mm6 = mm6 x 2^6
  psllw      mm7,  6      ; mm7 = mm7 x 2^6
  psubw      mm1,  Const1 ; mm1 = mm1 - Const1
  psubw      mm5,  Const1 ; mm5 = mm5 - Const1
  psubw      mm2,  Const2 ; mm2 = mm2 - Const2
  psubw      mm3,  Const2 ; mm3 = mm3 - Const2
  psubw      mm6,  Const2 ; mm6 = mm6 - Const2
  psubw      mm7,  Const2 ; mm7 = mm7 - Const2
  pmulhw     mm1,  Const3 ; mm7 = Y x 1.164
  .
  .
  sub        eax,  8
  jnz        loop1
```

Figure 4.26: *A part of the MMX code for the YCbCr-to-RGB color space conversion.*

fined by Equation (4.4) and Equation (4.5), respectively. In these equations $x(m,n)$ represents the current block of N^2 (usually $N = 16$) pixels, $y(m+i,n+j)$ represents the block in the reference frame, and (i,j) is the motion vector.

$$SSD(i,j) = \sum_{m=1}^{N}\sum_{n=1}^{N}(x(m,n) - y(m+i,n+j))^2. \tag{4.4}$$

$$SAD(i,j) = \sum_{m=1}^{N}\sum_{n=1}^{N}|x(m,n) - y(m+i,n+j)|. \tag{4.5}$$

The SSD and SAD functions are also used in CBIVR systems, where images and

```
  ; 16.5 = 66 / 2^2, 128.5 = 514 / 2^2, and 1.164 = 1192 / 2^10
  Const1   dw    66  , 66  , 66  ,  66 , 66  , 66  , 66  , 66
  Const2   dw    514 , 514 , 514 , 514 , 514 , 514 , 514 , 514
  Const3   dw    1192, 1192, 1192, 1192, 1192, 1192, 1192, 1192
  mov      eax,  M
loop1:
  fld8u12  3mx1,  (Y)    ; 3mx1 = |y7 |y6 |y5 |y4 |y3 |y2 |y1 |y0 |
  fld8u12  3mx2,  (Cb)   ; 3mx2 = |cb7|cb6|cb5|cb4|cb3|cb2|cb1|cb0|
  fld8u12  3mx3,  (Cr)   ; 3mx3 = |cr7|cr6|cr5|cr4|cr3|cr2|cr1|cr0|
  fsll12   3mx1,  2      ; 3mx1 = 3mx1 x 2^2
  fsll12   3mx2,  2      ; 3mx2 = 3mx2 x 2^2
  fsll12   3mx3,  2      ; 3mx3 = 3mx3 x 2^2
  fsub12   3mx1,  Const1 ; 3mx1 = 3mx1 - Const1
  fsub12   3mx2,  Const2 ; 3mx2 = 3mx2 - Const2
  fsub12   3mx3,  Const2 ; 3mx3 = 3mx3 - Const2
  fmul12h  3mx1,  Const3 ; 3mx1 = Y x 1.164
  .
  .
  sub      eax ,  8
  jnz      loop1
```

Figure 4.27: *A part of the MMMX code for the YCbCr-to-RGB color space conversion.*

videos are indexed into a database using a vector of features extracted from the image or video. In the retrieval stage the similarity between the features of the query image and the stored feature vectors is determined. That means that computing the similarity between two images or videos can be transformed into the problem of computing the similarity between two feature vectors [86]. Hence, the large computational cost associated with CBIVR systems is related to matching algorithms for feature vectors, because there are many feature vectors from different images and videos in the feature database.

Histogram Euclidean distance (Equation (4.6)) and bin-to-bin difference (b2b) (Equation (4.7)) are common similarity measurements in CBIVR systems [39]. In these equations h_1 and h_2 represent two histograms, N is the number of pixels in an image, and n is the number of bits in each pixel.

$$d^2(h_1, h_2) = \sum_{i=0}^{2^n-1} (h_1[i] - h_2[i])^2. \tag{4.6}$$

$$fd_{b2b}(h_1, h_2) = \frac{\sum_{i=0}^{2^n-1} |(h_1[i] - h_2[i]|)}{N}. \tag{4.7}$$

The size of a histogram depends on the number of bits in each pixel. If a pixel depth of n bits is supposed, the pixel values will be between 0 and $2^n - 1$, and the histogram will have 2^n elements.

Components of color histograms are unsigned numbers and usually larger than 8- and 16-bit. For instance, if a frame of size 512×512 is completely white or black, the largest element will be 2^{18}.

In the following subsections, the SIMD implementations of the SAD function, the SSD function, and their interpolation are also discussed.

SIMD Implementation of the SAD Function

As mentioned in Chapter 1, the `psadbw` instruction [111] is an SPI specifically targeted at the SAD function. A 64-bit `psadbw` instruction consists of 3 steps: (1) calculate eight 8-bit differences between the elements, (2) calculate the absolute value of the differences, and (3) perform three cascaded summations. These steps are illustrated in Figure 4.28. The C code of the SAD function was depicted in Figure 3.2 in Section 3.1. The code in Figure 4.29 depicts the MMX/SSE implementation of the SAD kernel for two 16×16 blocks using the `psadbw` instruction.

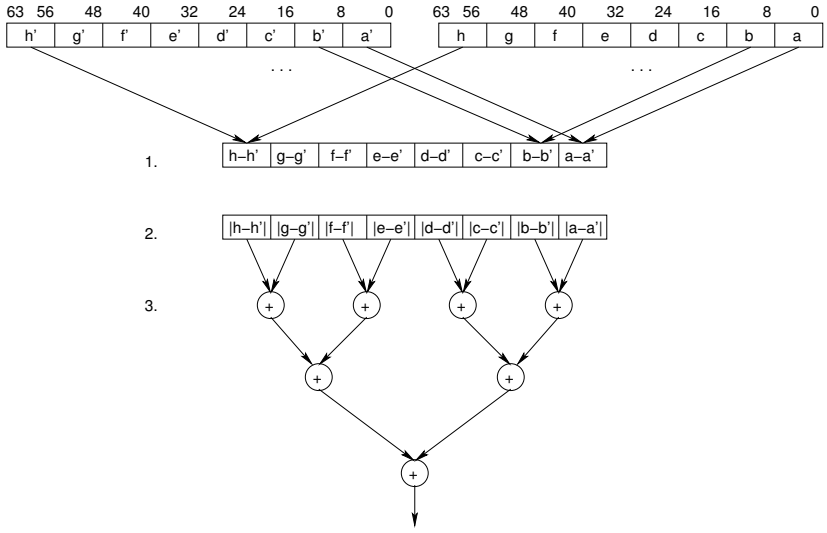


Figure 4.28: *The structure of SAD instruction in multimedia extension.*

One of the reasons why the `psadbw` instruction provides a significant performance benefit is that the 9-bit differences cannot be stored in the 8-bit subwords. Furthermore, there are no instructions to sum all the elements in a register or to add adjacent elements. In the MMMX architecture the 9-bit differences can be stored in the 12-bit subwords. Moreover, it provides instructions to add adjacent elements which can most lightly be performed in a single cycle. In other words, SIMD instructions have been implemented to replace the `psadbw` instruction, which are more general-purpose and can be used in many multimedia kernels and also in other similarity

measurements. The psadbw instruction can be synthesized using a small number of such general-purpose SIMD instructions with only a small performance degradation. Figure 4.30 shows how the SAD function can be implemented using MMMX instructions.

```
  mov     eax ,  16
  pxor    mm5 ,  mm5
loop:
  movq    mm1 ,  (blk1)
  movq    mm2 ,  (blk2)
  movq    mm3 ,  (blk1+8)
  movq    mm4 ,  (blk2+8)
  psadbw  mm1 ,  mm2
  psadbw  mm3 ,  mm4
  paddd   mm1 ,  mm3
  paddd   mm5 ,  mm1
  add     blk1,  16
  add     blk2,  16
  dec     eax
  jnz     loop
```

Figure 4.29: *The MMX/SSE implementation of the SAD function.*

```
  mov       ecx ,  2
loop1:
  fxor      3mx5,  3mx5
  mov       eax ,  8
loop2:
  fld8u12s  3mx1,  (blk1)
  fld8u12s  3mx2,  (blk2)
  fld8u12s  3mx3,  (blk1+8)
  fld8u12s  3mx4,  (blk2+8)
  fsub12    3mx1,  3mx2
  fneg12    3mx7,  3mx1,  255
  fmax12    3mx1,  3mx7
  fsub12    3mx3,  3mx4
  fneg12    3mx7,  3mx3,  255
  fmax12    3mx3,  3mx7
  fadd12    3mx1,  3mx3
  fadd12    3mx5,  3mx1
  add       blk1,  16
  add       blk2,  16
  dec       eax
  jnz       loop2
  fsum12    3mx5
  fsum24    3mx5
  fsum48    3mx5
  dec       ecx
  jnz       loop1
```

Figure 4.30: *The MMMX implementation of the SAD function.*

Since $|a-b| = max(a-b, b-a)$, the absolute difference operation can be synthesized using the MMMX instructions fsub12, fneg12, and fmax12. Thereafter, the sum-of-absolute differences can be computed using the fsum{12,24,48} instructions. These instructions are more general-purpose than the psadbw instruction. Note that the elements in a register are not summed in every iteration. Because each absolute difference is 8 bits and each subword is 12 bits, sixteen absolute differences can be accumulated before there is the risk of overflow. Therefore, in order to provide 8-way parallelism, a 16×16 block was divided into two 8×16 blocks. In the first iteration of the outer loop, the SAD function of the first 8×16 block is calculated and in the next iteration, the SAD function of the other 8×16 block is performed. Finally, the results are accumulated into one register using the fsum{12,24,48}

instructions.

Figure 4.31 and Figure 4.32 depict a part of the MMX and MMMX implementations of the SAD function for similarity measurement of two histograms, respectively.

```
  mov      ecx,  256
  pxor     mm5,  mm5           ; mm5 = 0
loop:
  movq     mm1,  (HCurrent)    ; mm1 = |h2   |h1   |
  movq     mm2,  (HReference)  ; mm2 = |r2   |r1   |
  movq     mm3,  mm1           ; mm3 = |h2   |h1   |
  psubd    mm1,  mm2           ; mm1 = |h2-r2|h1-r1|
  psubd    mm2,  mm3           ; mm2 = |r2-h2|r1-h1|
  movq     mm3,  mm1           ; mm3 = |h2-r2|h1-r1|
  movq     mm4,  mm2           ; mm4 = |r2-h2|r1-h1|
  pcmpgtd  mm1,  mm2           ; mm1 = mm1 > mm2
  pcmpgtd  mm2,  mm3           ; mm2 = mm2 > mm3
  pand     mm1,  mm3           ; mm1 = mm1 & mm3
  pand     mm2,  mm4           ; mm2 = mm2 & mm4
  paddd    mm1,  mm2           ; mm1 = mm1 + mm2
  paddd    mm5,  mm1           ; mm5 = mm5 + mm1
  .
  .
  sub      ecx,  2
  jnz      loop
  movq     mm6,  mm5           ; mm6 = mm5 = |p2|p1|
  psrlq    mm5,  32            ; mm5 = 0  p2
  paddd    mm5,  mm6           ; mm5 = |p2+0 |p2+p1|
```

Figure 4.31: *A part of the MMX implementation of the sum-of-absolute differences for similarity measurement of histograms.*

MMX provides 2-way parallelism as shown in Figure 4.31. This is because the elements of the color histograms are larger than 8- or 16-bit but smaller than 24-bit. The image size determines histogram element size. For example, the largest image size that is used is the High Definition Television (HDTV) standard is 1920×1080 [54] that its histogram element size can be represented by 24-bit. Elements of the histograms are stored in memory as 32-bit data type. Furthermore, there is no special-purpose SAD instruction for 32-bit data types in the MMX ISA. This means that thirteen other SIMD instructions are needed to implement it. The last three instructions in Figure 4.31 are used for addition of two adjacent data elements. MMMX, on the other hand, provides 4-way parallelism as shown in Figure 4.32. This is because 24-bit subwords are sufficient for computational results of the histograms.

```
  mov       ecx,   256
  fxor      3mx5,  3mx5          ; 3mx5 = 0
loop:
  fld32u24  3mx1,  (HCurrent)    ; 3mx1 = |h3  |h2  |h1  |h0   |
  fld32u24  3mx2,  (HReference)  ; 3mx2 = |r3  |r2  |r1  |r0   |
  fsub24    3mx1,  3mx2          ; 3mx1 = |h3-r3|h2-r2|..|h0-r0|
  fneg24    3mx7,  3mx1, 15      ; 3mx7 = |-(h3-r3)|..|-(h0-r0)|
  fmax24    3mx1,  3mx7          ; 3mx1 = max(3mx1, 3mx7)
  fadd24    3mx5,  3mx1          ; 3mx5 = 3mx5+3mx1
  .
  .
  sub       ecx ,  4
  jnz       loop
  fsum24    3mx5
  fsum48    3mx5
```

Figure 4.32: *A part of the MMMX implementation of the sum-of-absolute differences for similarity measurement of histograms.*

SIMD Implementation of the SSD Function

The SSD function for processing a block of size 16×16 has already been shown in Figure 3.1 in Chapter 3. As was discussed in Section 3.1, 24 bits of precision is sufficient for accumulation range of the SSD implementation. This means that an unsigned 24-bit data type is sufficient, which does not map orderly to a general-purpose data type.

Figure 4.33 and Figure 4.34 show how the SSD function can be implemented using MMX and MMMX instructions, respectively. The MMX uses 16-bit subwords for computational format. This is because the difference between two bytes does not fit in 8 bits. In addition, there is an MAC `pmaddwd` instruction for 16-bit data elements. The final result is stored in a 32-bit subword. Because of this, many punpck instructions are used to promote 8-bit to 16-bit data type. This is the reason why the static number of instructions in each iteration of the MMX implementation is 36 compared to 15 in the MMMX code. In the MMMX implementation 12-bit subwords are used for processing and final computational results are stored in 24-bit subwords.

Interpolation

The SAD and SSD similarity measurements are only a summation of the pixel-wise intensity differences and, consequently, small changes may result in a large similarity distance. For example, the Euclidean distance of Figure 4.35(a) and (b) is less than the Euclidean distance of (a) and (c), even though Figure 4.35(a) is more similar to

```
  mov        eax ,  16
  pxor       mm0 ,  mm0    ; mm0 = |0 |0 |0 |0 |0 |0 |0 |0 |
  pxor       mm7 ,  mm7    ; mm7 = |0 |0 |0 |0 |0 |0 |0 |0 |
loop:
  movq       mm1 ,  (blk1) ; mm1 = |a7|a6|a5|a4|a3|a2|a1|a0|
  movq       mm2 ,  (blk2) ; mm2 = |b7|b6|b5|b4|b3|b2|b1|b0|
  movq       mm3 ,  mm1    ; mm3 = |a7|a6|a5|a4|a3|a2|a1|a0|
  movq       mm4 ,  mm2    ; mm4 = |b7|b6|b5|b4|b3|b2|b1|b0|
  punpcklbw  mm1 ,  mm0    ; mm1 = |  a3 |  a2 |  a1 |  a0 |
  punpckhbw  mm3 ,  mm0    ; mm3 = |  a7 |  a6 |  a5 |  a4 |
  punpcklbw  mm2 ,  mm0    ; mm2 = |  b3 |  b2 |  b1 |  b0 |
  punpckhbw  mm4 ,  mm0    ; mm4 = |  b7 |  b6 |  b5 |  b4 |
  psubw      mm1 ,  mm2    ; mm1 = |a3-b3|a2-b2|a1-b1|a0-b0|
  psubw      mm3 ,  mm4    ; mm3 = |a7-b7|a6-b6|a5-b5|a4-b4|
  movq       mm2 ,  mm1    ; mm2 = |a3-b3|a2-b2|a1-b1|a0-b0|
  movq       mm4 ,  mm3    ; mm4 = |a7-b7|a6-b6|a5-b5|a4-b4|
  pmaddwd    mm1 ,  mm2    ; mm1 = |(a3-b3)^2+(a2-b2)^2|...|
  pmaddwd    mm3 ,  mm4    ; mm3 = |(a7-b7)^2+(a6-b6)^2|...|
  paddd      mm1 ,  mm3    ; mm1 = mm1 + mm3
  paddd      mm7 ,  mm1    ; mm7 = mm7 + mm1
  ; for other 8 pixels
  movq       mm1 ,  (blk1+8)
  movq       mm2 ,  (blk2+8)
  ; 13 instructions like
  ; above
  paddd      mm7 ,  mm1
  add        blk1,  16
  add        blk2,  16
  dec        eax
  jnz        loop
  movq       mm6 ,  mm7
  psrlq      mm7 ,  32
  paddd      mm7 ,  mm6
```

Figure 4.33: *The MMX implementation of the sum-of-squared differences function.*

Figure 4.35(c) than to (b).

For images, there are spatial relationships between pixels. There are many ways to consider the relationships between pixels, for example, averaging. Averaging neighboring pixels can be done either on two adjacent pixels horizontally, two adjacent pixels vertically, or four adjacent pixels in both horizontal and vertical dimensions. For instance, the MPEG-2 encoding offers varieties of block matching, involving half-pixel interpolation. The original MPEG-2 code first performs the interpolation, and then computes the sum of absolute differences on the result. To consider relationships between pixels in this thesis, the horizontal and vertical interpolation have

```
\begin{verbatim}
  mov      eax ,  16
  fxor     3mx7,  3mx7     ; 3mx7 = |  0  |  0  |  0  |  0  |
loop:
  fld8u12  3mx1,  (blk1)   ; 3mx1 = |a7|a6|a5|a4|a3|a2|a1|a0|
  fld8u12  3mx2,  (blk2)   ; 3mx2 = |b7|b6|b5|b4|b3|b2|b1|b0|
  fld8u12  3mx3,  (blk1+8) ; 3mx3 = |a15|a14|a13|a12|... |a8|
  fld8u12  3mx4,  (blk2+8) ; 3mx4 = |b15|b14|b13|b12|... |b8|
  fsub12   3mx1,  3mx2     ; 3mx1 = |a7-b7|a6-b6|...  |a0-b0|
  fsub12   3mx3,  3mx4     ; 3mx3 = |a15-b15|a14-b14|.|a8-b8|
  fmov     3mx2,  3mx1     ; 3mx2 = |a7-b7|a6-b6|...  |a0-b0|
  fmov     3mx4,  3mx3     ; 3mx4 = |a15-b15|a14-b14|.|a8-b8|
  fmadd24  3mx1,  3mx2     ; 3mx1 = 3mx1 x 3mx2
  fmadd24  3mx3,  3mx4     ; 3mx3 = 3mx3 x 3mx4
  fadd24   3mx1,  3mx3     ; 3mx1 = 3mx1 + 3mx3
  fadd24   3mx7,  3mx1     ; 3mx7 = 3mx7 + 3mx1
  add      blk1,  16
  add      blk2,  16
  dec      eax
  jnz      loop
  fsum24   3mx7
  fsum48   3mx7
```

Figure 4.34: *The MMMX implementation of the sum-of-squared differences function.*

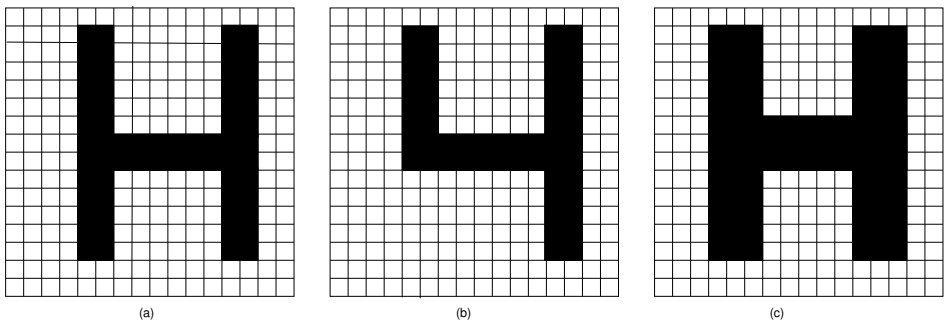


Figure 4.35: *Similar and dissimilar images.*

been implemented that supported by the MPEG-4 standard.

SIMD Implementation of SAD with Interpolation

As previously mentioned, one way to consider the relationship between image pixels is averaging. For this the SSE ISA provides special averaging instructions `pavgb` and `pavgw` for packed unsigned bytes and packed unsigned words, respectively. The former one provides 8-way parallelism, while the latter one supports 4-way parallelism. However, averaging four 8-bit pixels using horizontal and vertical interpolation may produce an error of 1 when performing 3 average operations as follows: $pavgb(x, y, z, t) = pavgb[pavgb(x, y), pavgb(z, t)]$. To avoid this error in the MMX/SSE implementation, 16-bit operations are used and input pixels are converted to 16-bit using pack/unpack instructions. Figure 4.36 and Figure 4.37 show a part of the MMX/SSE and MMMX implementations of the SAD function with horizontal and vertical averaging, respectively.

The sum of four neighboring pixels is larger than 8-bit. First, in the MMX/SSE implementation, the 8-bit data type is unpacked to 16-bit. Then, the averaging of four pixels is performed by 4-way parallelism. After that the 16-bit intermediate results are packed to 8-bit data type. Finally, 8-way parallelism is used by the `psadbw` instruction. In other words, MMX/SSE uses both 4- and 8-way parallelism as can be seen in Figure 4.36. The MMMX implementation, on the other hand, always employs 8-way parallelism because 12-bit is sufficient for the sum of four pixels and it is also sufficient to calculate the sum-of-absolute differences as depicted in Figure 4.37.

The SIMD implementation of SSD with interpolation is almost identical to the SIMD implementation of the SSD function except that it consists of much more load and ALU instructions in order to perform averaging.

Histogram Intersection Distance

In order to show the versatility of the MMMX ISA compared to the MMX/SSE ISA, another distance measurement, which is referred to as histogram intersection has been implemented. The histogram intersection distance between the two histograms h_1 and h_2, $fd_{int}(h_1, h_2)$ was proposed by Swain and Ballard [134] and is used in image and video retrieval [146, 39]. It is defined as:

```
  mov       eax,  16
  pxor      mm0 ,  mm0       ; mm0 = |0 |0 |0 |0 |0 |0 |0 |0|
loop:
  ; Pixels 0..7
  movq      mm1,  (blk1)     ; mm1 = |a0_7|a0_6|a0_5|..|a0_0|
  movq      mm3,  (blk1+16)  ; mm3 = |a1_7|a1_6|a1_5|..|a1_0|
  movq      mm2,  mm1        ; mm2 = |a0_7|a0_6|a0_5|..|a0_0|
  movq      mm4,  mm3        ; mm4 = |a1_7|a1_6|a1_5|..|a1_0|
  punpcklbw mm1,  mm0        ; mm1 = |a0_3|a0_2|a0_1   |a0_0|
  punpcklbw mm3,  mm0        ; mm3 = |a1_3|a1_2|a1_1   |a1_0|
  movq      mm5,  (blk1+1)   ; mm5 = |a0_8|a0_7|a0_6|..|a0_1|
  movq      mm6,  (blk1+17)  ; mm6 = |a1_8|a1_7|a1_6|..|a1_1|
  punpcklbw mm5,  mm0        ; mm5 = |a0_4|a0_3|a0_2|  |a0_1|
  punpcklbw mm6,  mm0        ; mm6 = |a1_4|a1_3|a1_2|  |a1_1|
  .
  .
  packuswb  mm1,  mm2
  psadbw    mm1,  (blk2)
  ; Pixels 8..F
  movq      mm1,  (blk1+8)   ; mm1 = |a0_15|a0_14|...  |a0_8|
  movq      mm3,  (blk1+24)  ; mm3 = |a1_15|a1_14|...  |a1_8|
  movq      mm2,  mm1        ; mm2 = |a0_15|a0_14|...  |a0_8|
  movq      mm4,  mm3        ; mm4 = |a1_15|a1_14|...  |a1_8|
  punpcklbw mm1,  mm0        ; mm1 = |a0_11|a0_10|a0_9 |a0_8|
  punpcklbw mm3,  mm0        ; mm3 = |a1_11|a1_10|a1_9 |a1_8|
  movq      mm5,  (blk1+9)   ; mm5 = |a0_16|a0_15|...  |a0_9|
  movq      mm6,  (blk1+25)  ; mm6 = |a1_16|a1_15|...  |a1_9|
  punpcklbw mm5,  mm0        ; mm5 = |a0_12|a0_11|a0_10|a0_9|
  punpcklbw mm6,  mm0        ; mm6 = |a1_12|a1_11|a1_10|a1_9|
  .
  .
  packuswb  mm3,  mm4
  psadbw    mm3,  (blk2+8)
  .
  .
  add       blk1, 16
  add       blk2, 16
  dec       eax
  jnz       loop
```

Figure 4.36: *The MMX/SSE code of the sum-of-absolute difference function using horizontal and vertical interpolation.*

```
  mov      eax ,  8
loop:
  fld8u12  3mx1,  (blk1)    ; 3mx1 = |a0_7 |a0_6 |...   |a0_0|
  fld8u12  3mx2,  (blk1+8)  ; 3mx2 = |a0_15|a0_14|...   |a0_8|
  fld8u12  3mx3,  (blk1+16) ; 3mx3 = |a1_7 |a1_6 |...   |a1_0|
  fld8u12  3mx4,  (blk1+24) ; 3mx4 = |a1_15|a1_14|...   |a1_8|
  fadd12   3mx1,  3mx3      ; 3mx1 = |a1_7 |a1_6 |...   |a1_0|
  fadd12   3mx2,  3mx4      ; 3mx2 = |a1_15|a1_14|...   |a1_8|
  fld8u12  3mx3,  (blk1+1)  ; 3mx3 = |a0_8 |a0_7 |...   |a0_1|
  fld8u12  3mx4,  (blk1+9)  ; 3mx4 = |a0_16|a0_15|...   |a0_9|
  fld8u12  3mx5,  (blk1+17) ; 3mx5 = |a1_8 |a1_7 |...   |a1_1|
  fld8u12  3mx6,  (blk1+25) ; 3mx6 = |a1_16|a1_15|...   |a1_9|
  fadd12   3mx3,  3mx5      ; 3mx3 = |a0_8+a1_8|.. |a0_1+a1_1|
  fadd12   3mx4,  3mx6      ; 3mx4 = |a0_16+a1_16|.|a0_9+a1_9|
  fadd12   3mx1,  3mx3      ; 3mx1 = |a1_7+a0_8+a1_8|...     |
  fadd12   3mx2,  3mx4      ; 3mx2 = |a1_15+ a0_16+a1_16|... |
  fsra12   3mx1,  2         ; 3mx1 = 3mx1 >> 2
  fsra12   3mx2,  2         ; 3mx2 = 3mx2 >> 2
  fld8u12  3mx3,  (blk2)    ; 3mx3 = |b0_7 |b0_6 |...   |b0_0|
  fld8u12  3mx4,  (blk2+8)  ; 3mx4 = |b0_15|b0_14|...   |b0_8|
  .
  .
  add      blk1,  16
  add      blk2,  16
  dec      eax
  jnz      loop
```

Figure 4.37: *The MMMX implementation of the sum-of-absolute difference function using horizontal and vertical interpolation.*

$$intersection(h_1, h_2) = \frac{\sum_{i=0}^{2^n-1} min(h_1[i], h_2[i])}{N}. \qquad (4.8)$$
$$fd_{int}(h_1, h_2) = 1 - intersection(h_1, h_2).$$

In this equation h_1 and h_2 represent two histograms, N is the number of pixels in an image, and n is the number of bits in each pixel. The elements of the histograms are larger than 16-bit. There are no suitable SIMD instructions such as minimum and maximum selection for data types larger than the *short* data type in the MMX/SSE ISA. As a result, the implementation of this cost function using MMX/SSE is difficult. The above equation shows that SIMD instructions are needed to find the minimum values and addition of adjacent elements. Such SIMD instructions are available in the MMMX ISA. Figure 4.38 depicts part of the MMMX implementation of the histogram intersection for distance measurement of the two histograms.

```
  mov       eax ,  256
  fxor      3mx7,  3mx7 ; 3mx7 = |0 |0 |
loop:
  fld32u24  3mx1,  (h1) ; 3mx1 = |h1_3|h1_2|h1_1|h1_0|
  fld32u24  3mx2,  (h2) ; 3mx2 = |h2_3|h2_2|h2_1|h2_0|
  fmin24    3mx1,  3mx2 ; 3mx1 = |g3  |... |g0  |, gi=min(h1_i & h2_i)
  fsum24    3mx1        ; 3mx1 = |  g3+g2  |  g1+g0  |
  fadd48    3mx7,  3mx1 ; 3mx7 = | 0+g3+g2 | 0+g1+g0 |
  add       h1  ,  16
  add       h2  ,  16
  sub       ecx ,  4
  jnz       loop
  fsum48    3mx7
```

Figure 4.38: *A part of the MMMX code for implementation of the histogram intersection.*

4.3 Evaluation Environment

In this section the experimental methodology and tools that were used for performance evaluation are presented.

In order to evaluate the performance of the proposed techniques, the MMX architecture and the MMMX architecture with extended subwords and the MRF were simulated using the `sim-outorder` simulator of the SimpleScalar toolset [9]. `sim-outorder` is a detailed, execution-driven simulator that supports out-of-order issue and execution. The PISA ISA which consists of 64-bit instructions has been used. It has three following instruction formats. First, register format that is used for computational instructions. Second, immediate format that supports the inclusion of a 16-bit constant. Finally, jump format that supports specification of 24-bit jump targets. These instructions formats are illustrated in Figure 4.39. Each instruction format contains a 16-bit annotate field, which can be used to synthesize new instructions without having to change and recompile the assembler. The MMX/SSE and MMMX instructions have been synthesized using this annotate field.

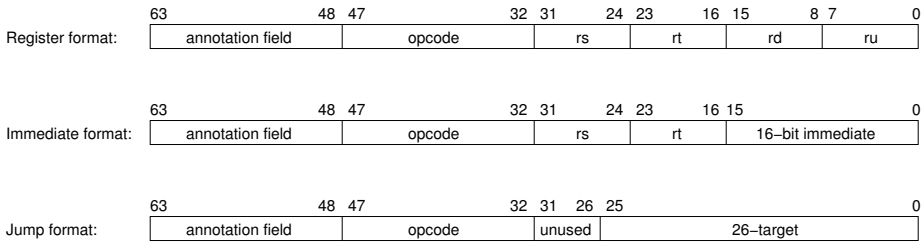


Figure 4.39: *SimpleScalar Portable ISA (PISA) instruction formats.*

As Figure 4.39 shows there are four 8-bit register fields in the register format. Each

of these fields allows to 256 architectural registers. Two of the register fields, rd and ru, can also be used as one 16-bit immediate value in the immediate format. The opcode and annotation fields are 16-bit. These fields are compiled with the opcodes of existing SimpleScalar instructions. The annotation field can be used for synthesizing new instructions in the assembly files as in the following example:

```
add.d/a     $f2,$f4,$f6
```

The annotation `/a` in this example specifies that the first bit of the 16-bit annotation field should be set. The simulator can then be modified to indicate that the instruction above corresponds to, e.g., `paddb` (packed addition of 8 bytes) instead of double-precision floating-point addition. This method, however, is very error-prone.

In order to simplify synthesizing new instructions, two tools have been used: the SimpleScalar Instruction Tool (SSIT) and the SimpleScalar Architecture Tool (SSAT) [74]. SSIT allows to use human-readable instructions such as `paddb` in the assembly files. It processes assembly files containing readable instructions, replaces them with corresponding annotated instructions, and modifies the source code of the `sim-outorder` to support the new instructions. SSAT extends SSIT by providing the possibility to define new registers and to define aliases for existing registers. For example, in MMX as well as in many other media extensions, the media registers correspond to the floating-point registers.

As was mentioned in Chapter 3, the reason why the MMX ISA was selected is that it is a representative SIMD extension. It is remarked that because the SimpleScalar PISA architecture is RISC, MMX and MMMX have not been simulated but rather RISC-like approximations. For example, one operand of many MMX instructions can be a memory location, but load/store architectures have been simulated. This does not affect the validity of the simulations because the main objective in this evaluation is to compare the performance of an SIMD architecture without extended subwords and the MRF to the same architecture with these features. Furthermore, in the Pentium 4 MMX instructions involving memory operands are translated to RISC-like micro-operations (μOPs).

The main parameters of the modeled processors are depicted in Table 4.3. Processors with an issue width of 1, 2, and 4 instructions were modeled. So, when four SIMD instructions are issued simultaneously, up to 32 data operations are executed in parallel. For most parameters the default values were used, except for the size of the Register Update Unit (RUU), which is 16 by default. This, however, is insufficient to find many independent instructions. Therefore, an RUU size of 64 was used.

The RUU is a costly resource, and its size should be minimized. The influence of the RUU on the performance has been studied using the VIS extension in [27]. The presented results in [27] shows that the increasing the RUU has a positive effect on the performance of a VIS-enhanced processor. This means that the VIS-enhanced

superscalar CPU can exploit and execute larger amount of ILP by increasing the RUU. In addition, for each issue width, there is a certain limit for the RUU to yield the maximum performance. For example, the speedup of the 4- and 8-way VIS-enhanced processor saturates when the RUU consists of 64 and 128 entries, respectively. Since, both the MMX and VIS extensions are almost the same, 64 entries for the RUU is realistic. In addition, Alpha has a window size of 80 instructions and the Pentium 4 has a window of 126 instructions.

The latency and throughput of SIMD instructions are set equal to the latency and throughput of the corresponding scalar instructions. This is a conservative assumption given that the SIMD instructions perform the same operation but on narrower data types. In addition, both latency and throughput of the `fsum` instructions are set to 1, while the latency and throughput of the `psadbw` instruction are set to 4 and 1, respectively, the same as in the Pentium 4 processor. The FPGA synthesis results presented in Section 3.3 have not been used, but rather reasonable approximations. This was down because, first FPGA synthesis are also not accurate for ASIC implementation. Second, pipelining SIMD multiplication operations has not been considered. In the simulation, SIMD multiplication operations have a latency of 3 cycles and a throughput of 1 cycle.

Three programs have been implemented by C and assembly languages and simulated using the SimpleScalar simulator for each kernel. Each program consists of three parts. One part is for reading the image, the second part is the computational kernel, and the last part is for storing the transformed image. One program was completely written in C. It was compiled using the gcc compiler targeted to the SimpleScalar PISA with optimization level *-O2*. The reading and storing parts of the other two programs were also written in C, but the second part was implemented by hand us-

Parameter	**Value**
Issue width	1/ 2/ 4
Integer ALU, SIMD ALU	1/ 2/ 4
Integer MULT, SIMD MULT	1/ 2/ 4
L1 Instruction cache	512 set, direct-mapped 64-byte line LRU, 1-cycle hit, total of 32 KB
L1 Data cache	128 set, 4-way, 64-byte line, 1-cycle hit, total of 32 KB
L2 Unified cache	1024 set, 4-way, 64-byte line, 6-cycle hit, total of 256 KB
Main memory latency	18 cycles for first chunk, 2 thereafter
Memory bus width	16 bytes
RUU (register update unit) entries	64
Load-store queue size	8
Execution	out-of-order

Table 4.3: *Processor configuration.*

ing MMX/SSE and MMMX. These programs will be referred to as C, MMX, and MMMX for each kernel. All C, MMX, and MMMX codes use the same algorithms. In addition, the correctness of the MMX and MMMX codes were validated by comparing their output to the output of C programs.

Some of the reasons why the assembly language has been used for the SIMD programming of multimedia kernels in this thesis have already been discussed in Section 1.4. Another reason is that in order to evaluate the performance of both the MMX and MMMX architectures by the SimpleScalar toolset, the assembly language codes of multimedia kernels have been needed. This is because suitable tools were not available yet to generate the assembly codes.

The speedup was measured by the ratio of the total number of cycles for the computational part of each kernel for the MMX implementation to the MMMX implementation. In order to explain the speedup, the ratio of dynamic number of instructions has also been obtained. These metrics formed the basis of the comparative study. Ratio of dynamic number of instructions means the ratio of the number of committed instructions for the MMX implementation to the number of committed instructions for the MMMX implementation. Although based on a rough comparison of both architectures in Chapter 3, the cycle time of MMMX is almost 40% slower than the cycle time of MMX, after considering this MMMX is still faster than MMX.

In addition, a whole image size has been used as input for kernels. For example, a 144×176 image has been used for similarity measurement kernels and a 576×704 image has also been used for the other kernels. The full search algorithm has been implemented for motion estimation on an image size of Quarter Common Intermediate Format (QCIF). The QCIF has a size of 144×176. In order to determine the motion vectors for the reference blocks in the current frame, a macroblock of 8×8 pixel region has been used as the basic block and a search range of ± 16 in the process of motion estimation.

To obtain the application-level speedup, SIMD implementations of the multimedia kernels have been replaced in their original C code in the media applications. In other words, three different versions, namely C, MMX, and MMMX were provided for each application and these versions were simulated by the simulator.

In the following section, the experimental results at block-, image-, and application-level are described.

4.4 Performance Evaluation Results

In this section, the MMMX architecture is evaluated by comparing the performance obtained for the MMMX implementation of a benchmark to the performance of the

MMX implementation of the same benchmark. The performance is compared at block-, image-, and application-level implementations.

4.4.1 Block-level Speedup

Figure 4.40 and Figure 4.41 depict the speedup of MMMX over MMX for media kernels that either use extended subwords technique or use both proposed techniques, respectively. The results have been obtained for one execution of media kernels on a single block on the single issue processor. In addition, these figures show the ratio of committed instructions (MMX implementation over MMMX). Both figures show that MMMX performs better than MMX for all kernels except SAD. The speedup in Figure 4.40 ranges from 0.74 for the SAD kernel to 2.66 for Paeth kernel. MMMX yields a speedup ranging from 1.10 for the 2D IDCT kernel to 4.47 for the Transp.(12) kernel in Figure 4.41. The most important reason why MMMX improves performance is that it needs to execute fewer instructions than MMX. In the SAD kernel, on the other hand, MMMX needs to execute more instructions than MMX. As Figure 4.40 shows, the ratio of committed instructions for the SAD kernel is 0.72.

An SPI has been used in the MMX implementation of the SAD function and the SAD function with interpolation, while in the MMMX implementation this SPI has been synthesized by a few general-purpose SIMD instructions. Both MMX and MMMX employ 8-way parallelism in the SAD function, while MMMX uses more instructions than MMX. MMX employs both 4- and 8-way parallelism in the SAD function with interpolation, which means that it uses many data type conversion instructions. On the contrary, MMMX always employs 8-way parallelism in the SAD function with interpolation kernel. This is the reason that the speedup is almost two for this kernel.

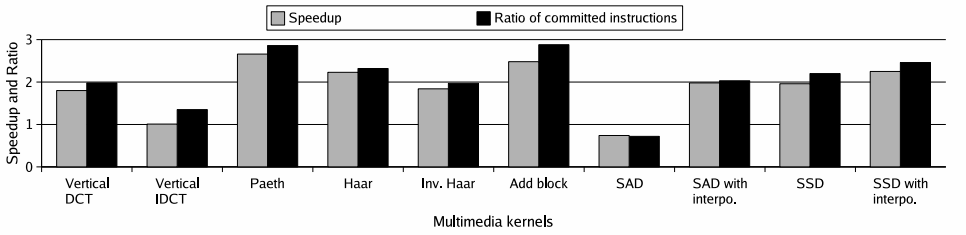


Figure 4.40: *Speedup of MMMX over MMX as well as the ratio of committed instructions (MMX over MMMX) for multimedia kernels, which use extended subwords technique on a single block on the single issue processor.*

The speedup obtained for the Paeth kernel in Figure 4.40 is 2.66. The reason is that intermediate data is at most 10 bits wide and MMMX can, therefore, calculate the prediction for eight pixels in each loop iteration while MMX computes the prediction for four pixels. The speedups of MMMX over MMX for the vertical IDCT and 2D

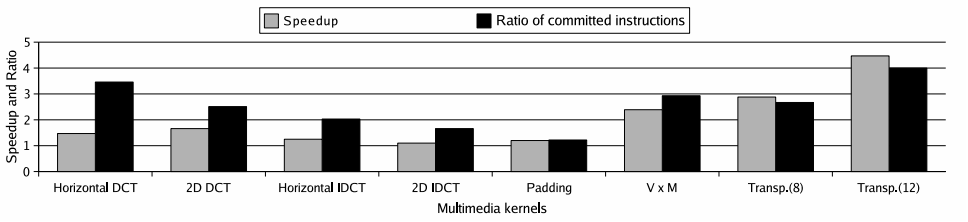


Figure 4.41: *Speedup of MMMX over MMX as well as the ratio of committed instructions (MMX over MMMX) for multimedia kernels, which use both proposed techniques on a single block on the single issue processor.*

IDCT kernels in those figures is less than the speedups for other kernels. This is because the input data of these kernels is 12-bit and some intermediate results are larger than 12-bit. Therefore, the MMMX implementation cannot employ 12-bit functionality (8-way parallel SIMD instructions) all the time but sometimes has to convert to 4×24-bit packed data types. The MMX implementation, on the other hand, is able to use 16-bit functionality all the time.

The reason why MMMX improves performance by just 20% for the Padding kernel in Figure 4.41 is that the MMX implementation employs the special-purpose `pavgb` instruction which computes the arithmetic average of eight pairs of bytes. More precisely, the `pavgb` instruction is supported in the SSE integer extension to MMX. MMMX does not support this instruction because with extended subwords it offers little extra functionality since it can be synthesized using the more general-purpose instructions `fadd12` and `fsar12` (shift arithmetic right on extended subwords). Nevertheless, because the matrix needs to be transposed between horizontal and vertical padding MMMX provides a speedup.

The two kernels for which the highest speedups are obtained are the 8×8 matrix transpose on 8-bit (Transp.(8)) and 12-bit data (Transp.(12)). If the matrix elements are 8-bit, MMMX can use the MRF to transpose the matrix, while MMX requires many pack and unpack instructions to realize a matrix transposition. Furthermore, if the elements are 12-bit (but stored as 16-bit data types), MMMX is able to employ 8-way parallel SIMD instructions, while MMX can only employ 4-way parallel instructions. As a result, MMMX improves performance by more than a factor of 4.47.

The average speedup and ratio of committed instructions for kernels that only use the extended subwords technique are 1.90 and 2.08, respectively, while for the kernels that use both proposed techniques are 2.05 and 2.56. The reduction of the dynamic instruction count in Figure 4.40 is due to extended subwords and in Figure 4.41 it is due to extended subwords and the MRF techniques. As a result, the performance benefits obtained by employing both techniques is higher than just using the extended

subwords technique.

4.4.2 Image-level Speedup

As explained before, the results presented in Figure 4.40 and Figure 4.41 are for one execution on a single block. In most cases, however, the kernels are executed on all blocks of an image or frame. To investigate if this changes the results fundamentally, Figure 4.42 and Figure 4.43 depict the image-level speedups for multimedia kernels that either use the extended subwords technique or use both extended subwords and the MRF techniques, respectively.

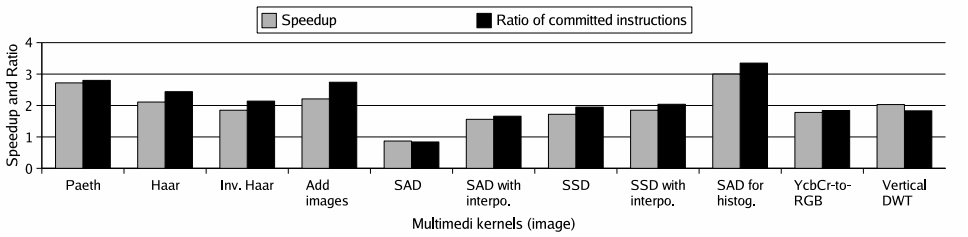


Figure 4.42: *Image-level speedup of MMMX over MMX as well as the ratio of committed instructions for the kernels, which use the extended subwords technique on the single issue processor.*

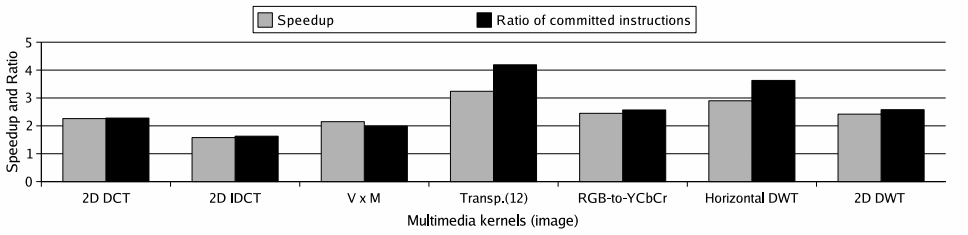


Figure 4.43: *Image-level speedup of MMMX over MMX as well as the ratio of committed instructions for the kernels, which use both proposed techniques on the single issue processor.*

In general, the image-level speedups are higher than the block-level speedups. For example, the block-level speedup for the 2D DCT kernel is 1.66, while the image-level speedup is 2.26. As another example, the block-level speedup for the Paeth kernel is 2.66 whereas the image-level speedup is 2.72. The reason for this behavior is as follows. Although executing the kernels on all blocks of an image does not reduce the number of data cache misses (because the images are too large to be kept in cache), it does reduce the number of instruction cache misses, since the kernels are relatively small and can be kept in the instruction cache. In addition, the image-level speedups for kernels that use both proposed techniques are larger than the image-level

speedups of kernels that only use the extended subwords technique. For example, the average image-level speedup for former kernels is 2.43, while for the latter kernels is 1.97. The main reason for this is that using both extended subwords and the MRF can reduce the dynamic number of instructions more than using the extended subwords technique. The average ratio of committed instructions in Figure 4.43 is 2.70, while it is 2.15 in Figure 4.42.

To summarize, the MMMX architecture improves performance compared to the MMX architecture by executing fewer instructions than MMX. In other words, the large number of instructions that have to be executed by the MMX architecture limits its performance. In many cases the MMMX implementation can employ 8-way parallel SIMD instructions, while MMX can employ only 4- or 2-way parallelism in all kernels except in the SAD function and a part of the SAD function with interpolation. In other words, MMMX can pack more arithmetic and logical operations into a single SIMD instruction. In addition, MMMX avoids the data type conversion and rearrangement instructions.

Figure 4.44 depicts the effect of increasing the issue width. It shows the image-level speedups of the MMMX implementations over the MMX implementations on out-of-order processors with different issue widths. The speedup is relative to the number of cycles taken by the MMX implementation when executed on the processor with the same issue width. In general, it can be observed that the speedup of MMMX over MMX slightly decreases when the issue width is increased. This can be expected because MMMX collapses several MMX instructions into a single instruction, so generally it will decrease the distance between dependent instructions. This means that the MMMX arithmetic and logical instructions allow to execute multiple arithmetic and logical instructions as well as multiple iterations with one MMMX instruction.

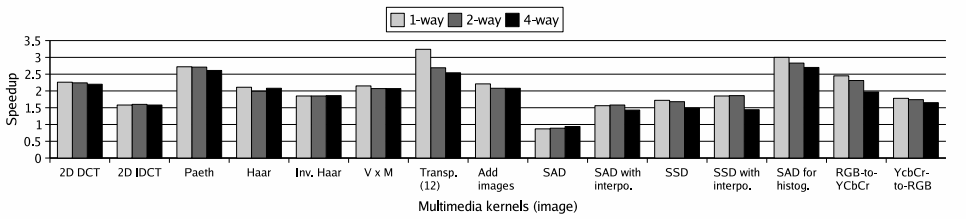


Figure 4.44: *Image-level speedup of MMMX over MMX implementation for different issue widths using out-of-order execution. The speedup is relative to the number of cycles taken by the MMX implementation when executed on the processor with the same issue width.*

The main reason that the MMMX architecture improves performance compared to the MMX architecture is that it executes less instructions than MMX as discussed in previous figures. In order to show where the instruction count reduction comes from, the instructions are divided into three groups, namely SIMD instructions, SIMD ld/st,

and scalar instructions. SIMD instructions consist of SIMD ALU/MULT and SIMD overhead (data type conversions and rearrangement) instructions. The scalar instructions are used for conditional operations, boundary checking, updating the pointers, and incrementing or decrementing index and address values. Figure 4.45 and Figure 4.46 show the ratio of SIMD instructions, scalar, and SIMD ld/st instructions of the MMX implementation to the MMMX implementation for one execution of kernels on a single block that either use extended subwords technique or use both proposed techniques, respectively.

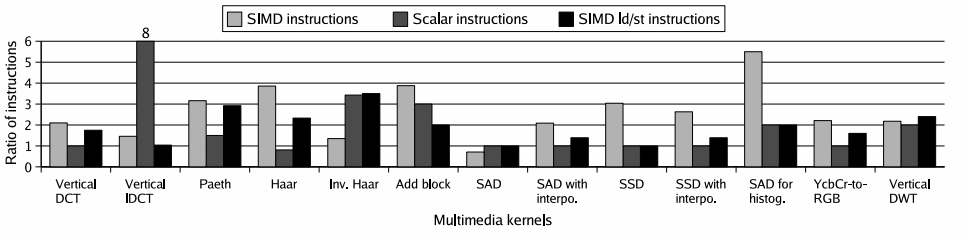


Figure 4.45: *Ratio of SIMD instructions, scalar, and SIMD ld/st instructions of the MMX implementation to the MMMX implementation for one execution of kernels on a single block that use the extended subwords technique.*

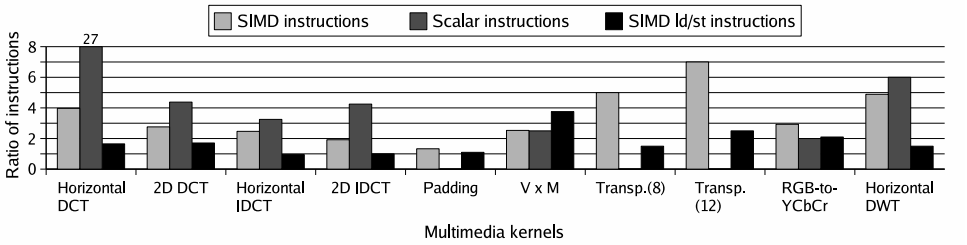


Figure 4.46: *Ratio of SIMD instructions, scalar, and SIMD ld/st instructions of the MMX implementation to the MMMX implementation for one execution of kernels on a single block, which use both extended subwords and the MRF techniques.*

The figures show that most of the reduction is due to the reduction of the number of SIMD instructions and scalar instructions. In Figure 4.45 the average ratios of SIMD instructions, scalar, and SIMD ld/st instructions are 2.63, 2.1, and 1.87, respectively. In Figure 4.46 the average ratios of SIMD instructions, scalar, and SIMD ld/st instructions are 3.48, 4.94, and 1.78, respectively. Figure 4.46 depicts that the reduction of the scalar instructions for some kernels, which use both techniques is slightly higher than the other two parts. In some cases, for instance in the horizontal DCT kernel, the proposed techniques eliminate all loop overhead. In addition, for some kernels, for example, the padding kernel there is no loop overhead in both architectures. In both figures, the reduction of SIMD ld/st instructions is much less than the reduction

of SIMD instructions and scalar instructions. In the MMX implementations, SIMD instructions, SIMD ld/st instructions, and scalar instructions consume an average of 67.17%, 23.88%, and 8.95%, respectively, of the total instructions, while in MMMX, they consume an average of 57.41%, 31.02%, and 11.57%, respectively, of the total instructions. In other words, the percentage usage of SIMD instructions is reduced by MMMX, while the percentage usage of SIMD ld/st and scalar instructions is increased by MMMX. However, the percentage usage of SIMD ld/st instructions is much more than the percentage usage of scalar instructions.

In the following section, the number of SIMD ld/st instructions in MMMX is reduced by increasing the number of registers.

4.4.3 Impact of the Number of Registers

It is well-known that for ISA legacy reasons, MMX has only 8 architectural registers. This is not enough to keep either intermediate results or constants values, which are used by multimedia kernels. For example, some multimedia kernels such as color space conversions use some coefficients that are unmodified through the full execution of the kernel. The number of these coefficients is usually small and cannot be kept in registers but have to be reloaded from memory in each loop iteration. As another example, a complete 8×8 block of the current frame that is used by full search algorithm cannot also be kept in registers but have to be reloaded from memory in each loop iteration. Although the constants and current block will be found in cache most of the times, the number of SIMD ld/st instructions is relatively large compared to the number of arithmetic instructions. In other words, using a larger register file would allow to keep intermediate and constant values during the whole execution of programs in the media registers. In addition, this allows the programmer to take advantage of the spatial and temporal data locality that are in many MMAs. This reduces the required bandwidth for SIMD ld/st instructions and also improves processing efficiency. Therefore, in this section the effect of adding more registers to the MMMX architecture is considered for two multimedia kernels, color space conversions and different similarity measurement algorithms. This is because color space conversions use at most twelve constant coefficients, which can be kept in extra registers. In addition, similarity measurement algorithms use a complete 8×8 block of the current frame that can also be kept in added registers.

It has been found that 13 extra media registers are sufficient to keep the critical data in the register file. In the RGB-to-YCbCr kernel, 11 of these registers are used to hold constants and 2 to hold intermediate results. Since two of the constant coefficients in the YCbCr-to-RGB kernel are zero and three of them are the same, for this kernel only 9 additional registers are needed. For the similarity measurements kernels 8 extra media registers are employed. Thus, an entire 8×8 candidate block can be

stored in these 8 extra media registers, as depicted in Figure 4.47.

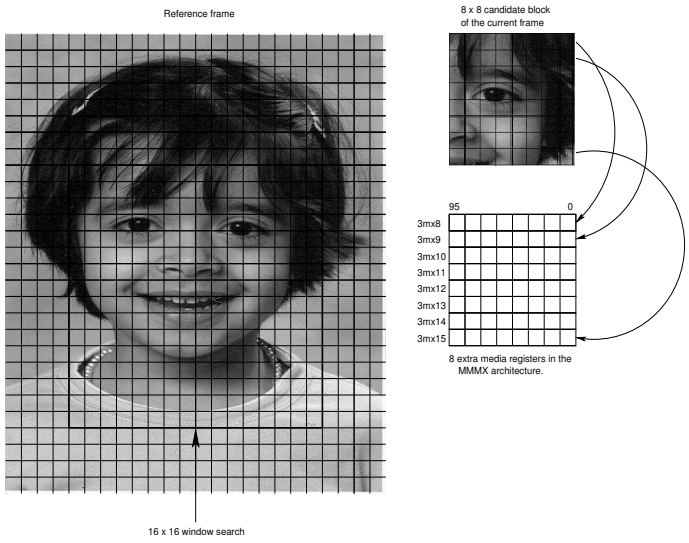


Figure 4.47: *The candidate block of the current frame can be stored in eight media registers to calculate the motion vector at each* 16×16 *window search of the reference frame.*

Figure 4.48 illustrates the speedup of MMMX with 8 registers (MMMX-8) and MMMX with 13 extra registers (MMMX-13) over MMX as well as the ratio of committed instructions (MMX implementation to MMMX) on the single issue processor. MMMX-13 yields speedups ranging from 1.37 to 3.64. Furthermore, the performance improvement of MMMX-13 over MMMX-8 ranges from 1.38 to 1.57, and the ratio of committed instructions (MMMX-8 implementation to MMMX-13) ranges from 1.22 to 1.56. Again, the main reason for these performance improvements is the reduced number of instructions that need to be executed. Because the data that is needed very often can be kept in registers, fewer ld/st instructions need to be executed. The largest performance improvement is achieved for the SAD function. For this kernel the speedup is 0.87 and 1.37 using MMMX-8 and MMMX-13, respectively. Although the MMX code that uses the SAD SPI is faster than the MMMX-8 implementation, the MMMX-13 code yields more performance.

4.4.4 Analysis of each Proposed Technique Separately

As already indicated in Table 4.2, in the SIMD implementations of some kernels such as the horizontal DCT both proposed techniques have been employed. Consequently, a part of the performance benefits is due to extended subwords, which increases DLP and the other part of the performance improvement is due to the MRF that eliminates the data rearrangement instructions. This section discusses some examples, horizon-

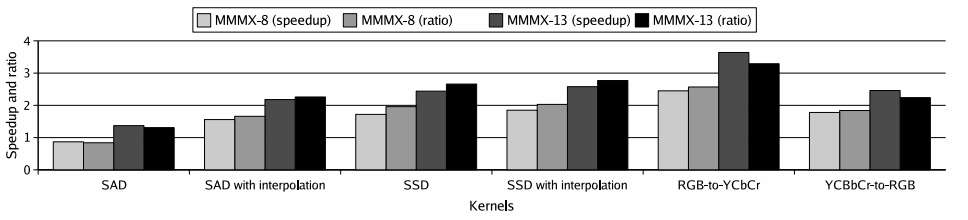


Figure 4.48: *Speedup of MMMX with 8 registers (MMMX-8) and MMMX with 13 extra registers (MMMX-13) over MMX (8 registers) as well as the ratio of committed instructions (MMX implementation to MMMX) on the single issue processor.*

tal DCT and 2D DCT, in order to clarify how much of the performance gain is a result of the additional parallelism provided by extended subwords and how much of it is due to the MRF.

The algorithm of the horizontal DCT has been explained in Section 4.2.4. Two SIMD implementations, one for MMX and one for MMMX have been discussed. In order to determine the performance benefit of each proposed techniques, two extra SIMD implementations, one for MMX enhanced with extended subwords and one for MMX enhanced with the MRF have also been implemented.

MMX Enhanced with Extended Subwords

In MMX enhanced with extended subwords (MMX + ES), there are eight 12-bit subwords in each media register. In order to bring these subwords in a form amenable to SIMD processing, new data permutation instructions such as `fshuflh12`, `fshufhl12`, `fshufhh12`, `fshufll12`, and `frever12` are needed. This is because of the following reasons. First, there is no shuffle instructions in MMX. MMX performs data permutation using pack and unpack instructions, while these instructions are not useful for MMX + ES. Second, there is a `pshufw` (packed shuffle word) instruction in SSE that is used for rearrangement of four subwords within a media register, while MMX + ES has eight subwords.

Figure 4.49 depicts the structure of the `fshuflh12 mm1, mm0, imm8` instruction. This instruction inserts four subwords from the low part of the source operand into the four high part of the destination operand at subword locations selected with the immediate operand *imm8* field. The immediate field has 8 bits. Each 2-bit of this field selects one subword location in the high part of the destination operand. The `fshufhl12` is almost the same as the `fshuflh12` instruction except that it inserts four subwords from the high part of the source operand into the four subwords of the low part of the destination operand.

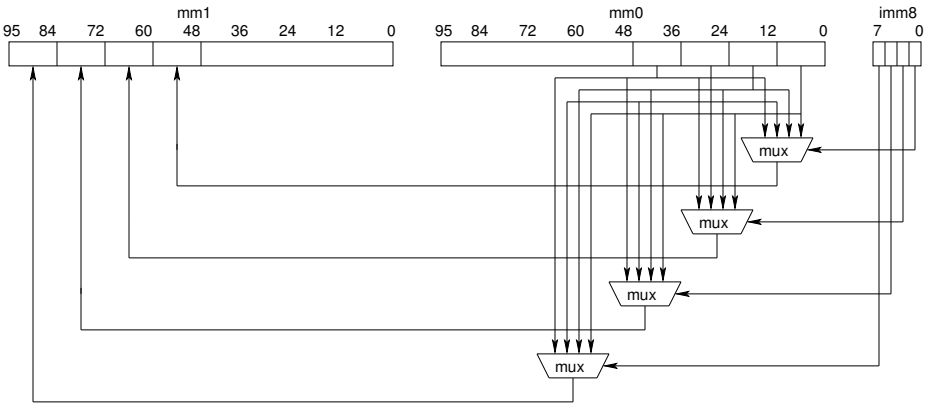


Figure 4.49: *The structure of the* `fshuflh12 mm1, mm0, imm8` *instruction.*

Figure 4.50 depicts the structure of the `fshufll12 mm1, mm0, imm8` instruction. This instruction inserts four subwords from the low part of the source operand into the low part of the destination operand using 8-bit immediate operand. The functionality of the `fshufhh12` instruction is almost the same as `fshufll12` instruction except that it copies subwords from the high part of the source operand and inserts them in the high part of the destination operand. The `fshufll12` and `fshufhh12` instructions are similar to `pshuflw` and `pshufhw` instructions, respectively, which are supported by the Intel SSE2 extension.

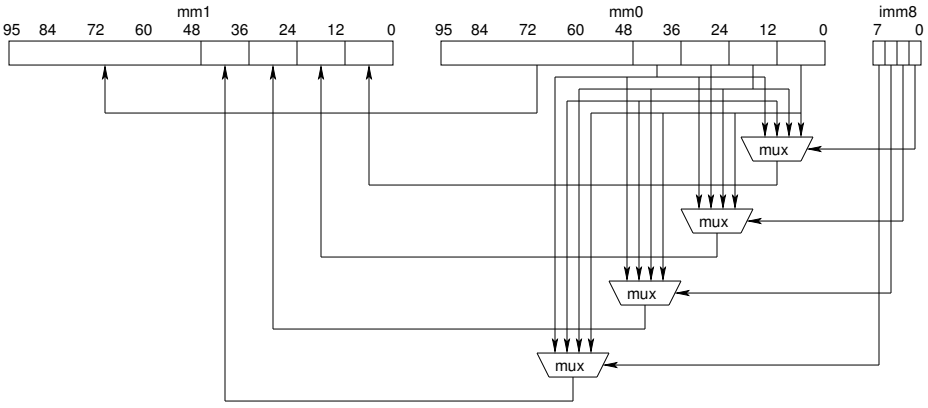


Figure 4.50: *The structure of the* `fshufll12 mm1, mm0, imm8` *instruction.*

Figure 4.51 depicts the operation of the `frever12 mm1, mm0` instruction. This instruction copies subwords from source operand and inserts them in the reverse order in the destination operand.

Figure 4.52 depicts a part of the horizontal DCT code that has been implemented

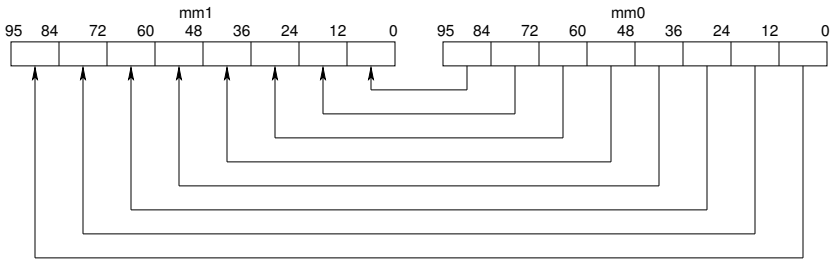


Figure 4.51: *The structure of the* `frever12 mm1, mm0` *instruction.*

by the MMX + ES. In each loop iteration of this implementation eight pixels are processed, the same as the MMX implementation that was discussed in Section 4.2.4.

```
fld16s12   mm0,   (dct)      ; mm0 =
frever12   mm2,   mm0        ; mm2 =
fneg12     mm2,   mm2,   15  ; mm2 =
fadd12     mm0,   mm2        ; mm0 =
```

x7	x6	x5	x4	x3	x2	x1	x0
x0	x1	x2	x3	x4	x5	x6	x7
x0	x1	x2	x3	-x4	-x5	-x6	-x7
x0+x7	x1+x6	x2+x5	x3+x4	x3-x4	x2-x5	x1-x6	x0-x7

Figure 4.52: *A part of the code for horizontal DCT that has been implemented by MMX enhanced by extended subwords.*

MMX Enhanced with an MRF

In MMX enhanced with an MRF, there are four 128-bit column registers and eight 64-bit registers the same as MMX. Each column register has eight 16-bit subword. Each subword in a column register corresponds to a subword in a row register. Each load column instruction can load eight 16-bit pixels into a column register. Figure 4.53 shows how four rows of an 8×8 block can be loaded into four column registers using load column instructions. In addition, Figure 4.54 depicts a part of the MMX + MRF implementation of the horizontal DCT algorithm. There are two loop iterations to process an 8×8 block. This means that in each loop iteration, four rows (32 pixels) are processed.

Experimental Results

Figure 4.55 depicts the speedup of MMX + ES, MMX + MRF, and MMMX over MMX for one execution of an 8×8 horizontal DCT on a single issue processor. In addition, this figure shows the ratio of committed instructions (MMX over the other architectures). The speedup of MMX + ES is 1.15, while the speedup of MMX +

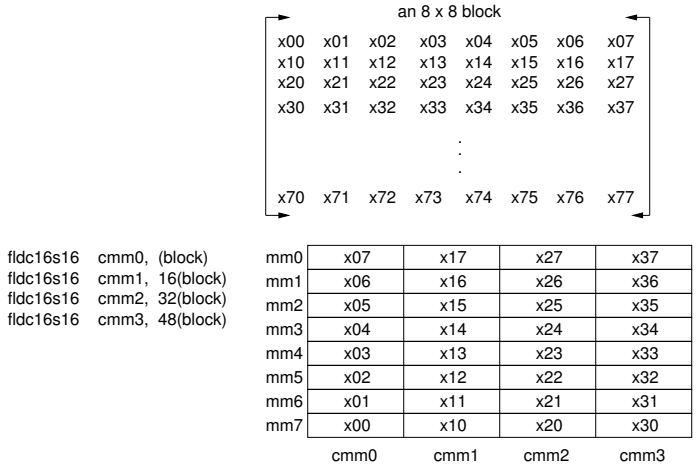


Figure 4.53: *Loading eight consequent stored pixels into a column register by load column instruction for little endian.*

`fldc16s16 cmm0 , (dct)`	; cmm0 =	x07	x06	x05	x04	x03	x02	x01	x00
`fldc16s16 cmm1 , 16(dct)`	; cmm1 =	x17	x16	x15	x14	x13	x12	x11	x10
`fldc16s16 cmm2 , 32(dct)`	; cmm2 =	x27	x26	x25	x24	x23	x22	x21	x20
`fldc16s16 cmm3 , 48(dct)`	; cmm3 =	x37	x36	x35	x34	x33	x32	x31	x30
`movq (dct), mm7`	; (mem) =	x37		x27		x17		x07	
`movq mm7 , mm0`	; mm7 =	x30		x20		x10		x00	
`paddsw mm0 , (dct)`	; mm0 =	x30+x37		x20+x27		x10+x17		x00+x07	
`psubsw mm7 , (dct)`	; mm7 =	x30-x37		x20-x27		x10-x17		x00-x07	

Figure 4.54: *A part of the MMX + MRF implementation of the horizontal DCT algorithm.*

MRF is less than 1. These results indicate that using either extended subwords or the MRF techniques is insufficient to eliminate most pack/unpack and rearrangement overhead instructions. In addition, using the MRF is both unuseful and causes performance loss. The MMMX architecture that employs both proposed techniques, on the other hand, yields much more performance benefits. Its speedup is 1.52.

In order to explain the behavior of Figure 4.55, Figure 4.56 shows the number of SIMD computation, SIMD overhead, SIMD ld/st, and scalar instructions for the four different architectures: MMX, MMX + MRF, MMX + ES, and MMMX for an 8×8 horizontal DCT kernel. As this figure shows, the total number of instructions in the MMX + MRF is almost the same as MMX. This means that the former architecture cannot reduce the total number of instructions. The MMX + MRF reduces the number of SIMD overhead instructions, but it increases the number of SIMD ld/st

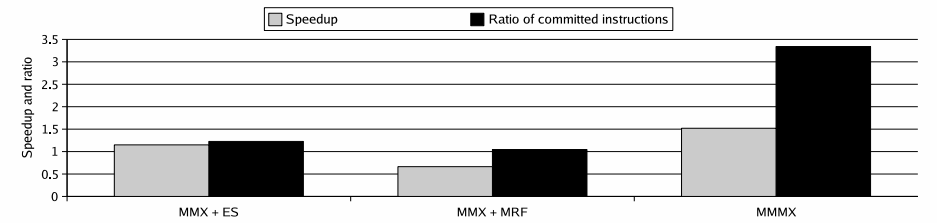


Figure 4.55: *Speedup of the MMX + ES, MMX + MRF, and MMMX over MMX as well as ratio of committed instructions for an* 8×8 *horizontal DCT on a single issue processor.*

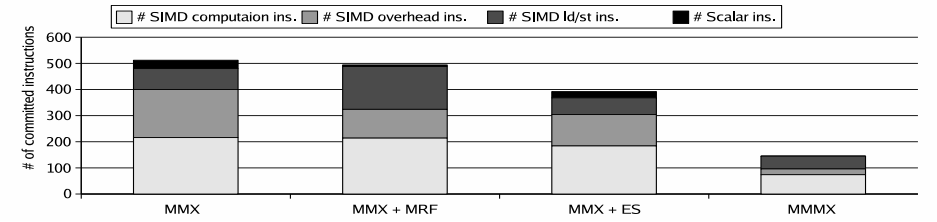


Figure 4.56: *The number of SIMD computation, SIMD overhead, SIMD ld/st, and scalar instructions in four different architectures, MMX, MMX + MRF, MMX + ES, and MMMX for an* 8×8 *horizontal DCT kernel.*

instructions. This is because the MMX + MRF transposes four rows in each iteration and this causes that all eight 64-bit registers are filled. In order to use some of the filled registers for intermediate computations, they are stored and loaded in memory hierarchy and this increases the number of SIMD ld/st instructions. The latency of SIMD ld/st instructions is almost more than the latency of the SIMD overhead instructions. This is the main reason why the MMX + MRF has a performance penalty. MMX + ES, on the other hand, reduces the total number of instructions. The ratio of committed instructions is 1.23 as shown in Figure 4.55.

The extended subwords technique reduces the number of SIMD computation and SIMD ld/st instructions more than the MRF technique, while the latter technique reduces the number of SIMD overhead and scalar instructions more than the former technique. Consequently, these experimental results indicate that using either of these techniques is insufficient to mitigate SIMD computation, SIMD overhead, SIMD ld/st, and scalar instructions. The MMMX architecture that employs both proposed techniques reduces the total number of instructions much more than MMX + MRF and MMX + ES.

The results presented in Figure 4.55 are for one execution of the horizontal DCT kernel on an 8×8 block. The horizontal DCT is a part of the 2D DCT kernel and this kernel is executed on all blocks of an image. Figure 4.57 depicts the image-level speedup of MMX + ES, MMX + MRF, and MMMX over MMX for the 2D DCT

kernel on a single issue processor. The speedup of MMX + ES is 1.39, while the speedup of MMX + MRF is 0.94. MMMX yields a speedup of up to 2.17 due to use of both proposed techniques. These speedups are larger than the block-level speedup. The reasons for this have already explained in Section 4.4.2.

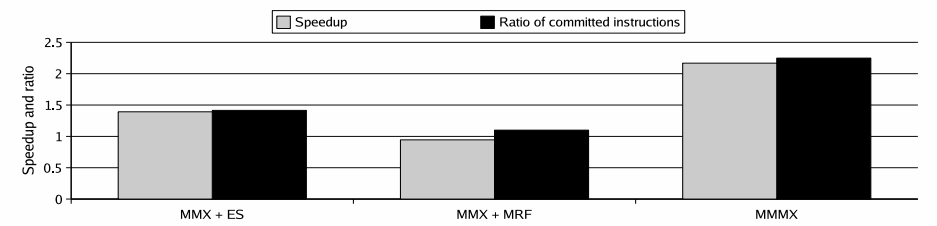


Figure 4.57: *Image-level speedup of MMX + ES, MMX + MRF, and MMMX over MMX as well as the ratio of committed instructions for the 2D DCT kernel on a single issue processor.*

4.4.5 Application-level Speedup

In order to obtain application-level speedup, some MMAs such as MPEG-2, JPEG, and MJPEG standards were studied. For the MPEG-2 encoder, the *walk* bitstream has been used, which consists of three 352×288 frames. For the MPEG-2 decoder, the *vaya* bitstream has been used, which consists of 25 160×112 frames. For the JPEG benchmarks, the *fallsbig* input was used, which is a 3000×2000 pixel image. Finally, for the MJPEG benchmark the *tenis* input was used, which is a frame of size 800×800.

In order to find the most time consuming kernels, these standards were profiled using the GNU profiler, *gprof* command. In the MPEG-2 encoder, *dist1* (sum-of-absolute differences) is the most compute-intensive kernel. In this standard, four different similarity measurements, SAD, SAD with interpolation, SSD, and SSD with interpolation, have been separately used. In other words, there are four MPEG-2 encoder versions, namely MPEG-2 encoder with SAD, MPEG-2 encoder with the SAD interpolation, MPEG-2 encoder with SSD, and MPEG-2 encoder with the SSD interpolation. Those similarity measurements consume an average of 87.2%, 91.2%, 90.2%, and 91.3%, respectively. In the MPEG-2 decoder the 2D IDCT kernel consumes an average of 26.7% of the total execution time. In the JPEG encoder (cjpeg), the 2D DCT and RGB-to-YCbCr color space conversion kernels consume an average of 35.7% and 25.0% of the total execution time, respectively. In the JPEG decoder (djpeg), the YCbCr-to-RGB color space conversion and 2D IDCT kernels take the most of the total execution time. They take 64.3% and 21.4%, respectively. Finally, the 2D DCT kernel consumes an average of 32.2% of the total execution time in the MJPEG application.

As already discussed these kernels have been accelerated by MMX and MMMX. For example, Table 4.4 depicts the image-level speedups of the MMX and MMMX implementations for different multimedia kernels, which have been used in the application-level speedup, over the scalar implementations on a single issue processor. In addition, the third column shows the speedup of MMMX over MMX. As this table shows, both architectures improve the performance of all media kernels. The MMMX architecture yields more speedups than the MMX architecture for all kernels except the SAD kernel. These performance improvements can be used to improve the performance of the whole applications. To obtain the application-level speedup, SIMD implementations of the multimedia kernels have been replaced in their original C code in the media applications.

The speedups of MMMX over MMX for complete applications on the single issue processor are depicted in Figure 4.58. The MMMX architecture improves the performance of all applications except the MPEG-2 encoder with the SAD kernel. MMMX achieves speedups of 0.91, 1.67, 1.43, and 1.90 for MPEG-2 encoder with SAD, MPEG-2 encoder with the SAD interpolation, MPEG-2 encoder with SSD, and MPEG-2 encoder with the SSD interpolation, respectively. In the MPEG-2 encoder that uses the SAD kernel, there is a slightly performance degradation. As mentioned previously, this is because the MMX code uses a special-purpose `psadbw` instruction, while MMMX does not. The largest performance improvement that is 1.90 occurs for the MPEG-2 encoder that uses SSD with interpolation. In addition, the MMMX architecture improves performance by a factor 1.15, 1.17, 1.37, and 1.16 for the MPEG-2 decode, JPEG encode, JPEG decode, and MJPEG, respectively.

The total of improvement depends on fraction that is improved and amount of improvement for that fraction. Since MMMX improves the performance of multimedia kernels more than MMX, it yields more application-level speedups than MMX.

Multimedia Kernel	MMX	MMMX	MMMX/MMX
2D DCT	2.7	6.1	2.3
2D IDCT	3.2	5.1	1.6
RGB-to-YCbCr	4.8	11.8	2.5
YcbCr-to-RGB	6.2	11.0	1.8
SAD	15.3	13.3	0.9
SAD with interpolation	6.3	9.8	1.6
SSD	7.1	12.2	1.7
SSD with interpolation	4.6	8.5	1.9

Table 4.4: *Image-level speedup of the MMX and MMMX implementations for different multimedia kernels, which have been used in the application-level speedup, over the scalar implementations on a single issue processor.*

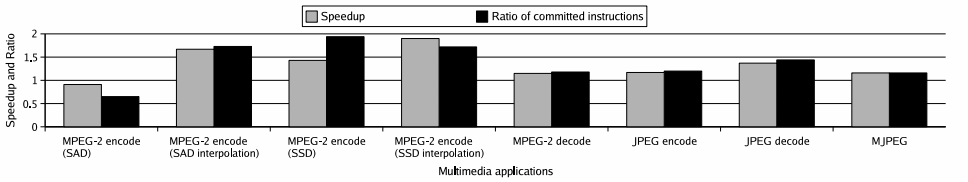


Figure 4.58: *Application-level speedup of MMMX over MMX as well as ratio of committed instructions for multimedia applications on the single issue processor.*

4.5 Conclusions

In this chapter, the MMX architecture enhanced with the extended subwords and the matrix register file techniques was evaluated. In order to evaluate the effectiveness of the MMMX architecture, a number of multimedia benchmarks such as MPEG-2 codecs, JPEG codecs, and MJPEG were selected. These applications were accelerated by accelerating their kernels. This is because the media kernels are the most time consuming functions and represent a major portion of MMAs. This chapter also presented the SIMD implementations of the media kernels using both the MMX and MMMX architectures. The floating-point kernels were implemented by fixed-point arithmetic. This chapter showed how the extended subwords and the matrix register file techniques can be used to reduce the dynamic number of instructions by decreasing data type conversion and data permutation overhead.

Simulation results were obtained by extending the *sim-outorder* simulator of the SimpleScalar tool set. The performance was obtained at the kernel- image-, and application-level. The presented results showed that MMMX improves performance significantly compared to MMX. In other words, the large number of instructions that had to be executed by the MMX architecture limits its performance. In addition, the experimental results showed that using either of extended subwords or the MRF technique is insufficient to mitigate data rearrangement instructions. The MMMX architecture that employs both proposed techniques reduces the total number of instructions much more than MMX enhanced with the MRF and MMX enhanced with extended subwords.

In the next chapter the Discrete Wavelet Transform (DWT) will be discussed in more detail. The DWT is used in the JPEG2000 standard and processes whole images, while the DCT is used in the MPEG standards and processes 8×8 blocks of pixels. Three issues related to the efficient computation of the 2D DWT on general-purpose processors, in particular the Pentium 4, are discussed, which are 64K aliasing, cache conflict misses, and SIMD vectorization.

Chapter 5

Optimizing the Discrete Wavelet Transform

The JPEG2000 standard employs the Discrete Wavelet Transform (DWT) instead of the DCT that is used in the MPEG-2 compression standard. The reason for this is that the image quality is much higher at lower bit rates with a wavelet based transform. In addition, the DCT based compression standards partition an image into discrete 8×8 pixel blocks. This generates blocking artifacts in the output image. The DWT operates on a complete image or a large part of an image and this avoids the artifact problem. Consequently, it needs more memory than the DCT. The DWT is much more computationally intensive than other functions of the JPEG2000 standard. For example, the obtained results in [126] show that the DWT consumes on average 46% of the encoding time for lossless compression. For lossy compression, the DWT even requires 68% of the total encoding time on average. Therefore, it is of great importance to enhance the performance of the DWT. In order to improve performance, several researchers [35, 48] have proposed hardware implementations of the DWT. Programmable processors, however, are preferred to special-purpose hardware because they are more flexible, enable different transforms to be employed, and allow various filter bank lengths and various transform levels.

In this chapter, three issues related to the efficient computation of the 2D DWT on SIMD-enhanced GPPs are described, which are 64K aliasing, cache conflict misses, and SIMD vectorization. The 64K aliasing is a phenomenon that it happens on the Pentium 4, and it can degrade performance by an order of magnitude. It occurs if two or more data input whose addresses differ by a multiple of 64K need to be cached simultaneously. In addition, there are many cache conflict misses in the implementation of vertical filtering, if the filter length exceeds the number of cache ways.

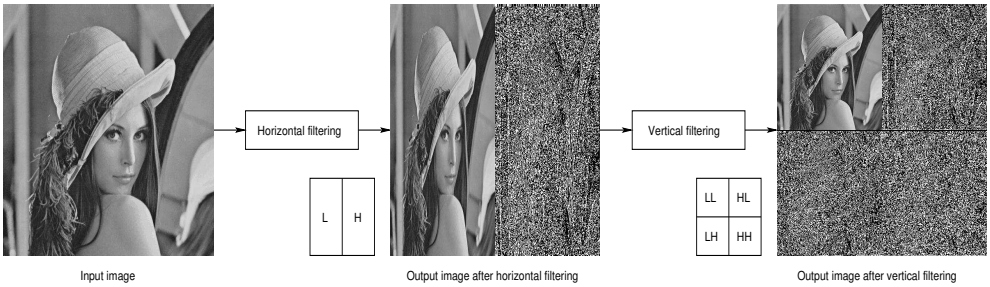


Figure 5.1: *Different sub-bands after first decomposition level.*

Additionally, the performance of the 2D DWT is improved by exploiting the DLP using the SIMD instructions supported by most GPPs.

This chapter is organized as follows. Section 5.1 describes the discrete wavelet transform and different approaches to traverse an image to implement the 2D DWT. The problems related to implementing the 2D DWT efficiently on GPPs are discussed in Section 5.2. Section 5.3 presents the experimental environment. Section 5.4 proposes and evaluates two techniques to circumvent 64K aliasing. Section 5.5 addresses the cache behavior of transforms with long filters and presents two techniques to avoid conflict misses. SIMD implementations of the 2D DWT are described in Section 5.6. Finally, conclusions are given in Section 5.7.

5.1 2D Discrete Wavelet Transform

The wavelet representation of a discrete signal X consisting of N samples can be computed by convolving X with the lowpass and highpass filters and down-sampling the output signal by 2, so that the two frequency bands each contains $N/2$ samples. With the correct choice of filters, this operation is reversible. This process decomposes the original image into two sub-bands: the lower and the higher band [132]. This transform can be extended to multiple dimensions by using separable filters. A 2D DWT can be performed by first performing a 1D DWT on each row (*horizontal filtering*) of the image followed by a 1D DWT on each column (*vertical filtering*).

Figure 5.1 illustrates the first decomposition level ($d = 1$). In this level the original image is decomposed into four sub-bands that carry the frequency information in both the horizontal and vertical directions. In order to form multiple decomposition levels, the algorithm is applied recursively to the LL sub-band. Figure 5.2 illustrates the second ($d = 2$) and third ($d = 3$) decomposition levels as well as the layout of the different bands.

As was mentioned in previous chapter, there are different approaches to implement

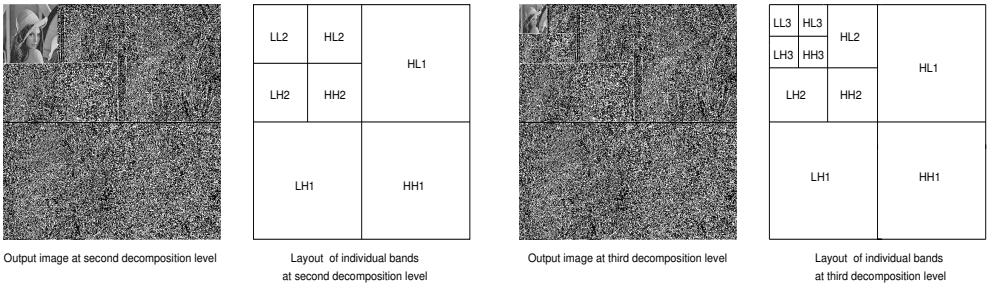


Figure 5.2: *Sub-bands after second and third decomposition levels.*

the 2D DWT such as traditional convolution-based and lifting scheme methods. The convolutional methods apply filtering by multiplying the filter coefficients with the input samples and accumulating the results. The Daubechies' transform with four coefficients [141] (Daub-4) and the Cohen, Daubechies and Feauveau 9/7 filter [31] (CDF-9/7) are examples of this category. For instance, the CDF-9/7 transform has 9 lowpass filter coefficients $h = \{h_{-4}, h_{-3}, h_{-2}, h_{-1}, h_0, h_1, h_2, h_3, h_4\}$ and 7 high-pass filter coefficients $g = \{g_{-2}, g_{-1}, g_0, g_1, g_2, g_3, g_4\}$. Both filters are symmetric, i.e., $h_{-i} = h_i$. The lifting scheme has been proposed for the efficient implementation of the 2D DWT. This approach has three phases, namely: split, predict, and update, which has been discussed in previous chapter. One example of this group is the integer-to-integer $(5, 3)$ lifting scheme ($(5, 3)$ *lifting*).

In this chapter, three different transforms, Daub-4, CDF-9/7, and $(5, 3)$ lifting, selected from both discussed groups are considered. These filters are considered for various reasons. First, the $(5, 3)$ lifting and CDF-9/7 transforms are included in Part 1 of the JPEG2000 standard [110]. Second, these transforms have been considered in many recent papers (e.g., [15, 59, 121, 25, 141, 29]). Finally, the $(5, 3)$ lifting scheme has low computational complexity and performs reasonably well for lossy as well as lossless compression compared to other lifting filters [1]. Additionally, the $(5, 3)$ filter has only one lifting step. Transforms with fewer lifting steps tend to perform better than transforms with more lifting steps in terms of speed as well as accuracy [1].

There are different algorithms to traverse an image to implement these transforms, namely *Row-Column Wavelet Transform* (RCWT) and *Line-Based Wavelet Transform* (LBWT) [3, 4, 5, 6, 30]. These approaches are discussed in the following sections.

5.1.1 Row-Column Wavelet Transform

In the RCWT approach, the 2D DWT is divided into two 1D DWTs, namely horizontal and vertical filtering. The horizontal filtering processes the rows of the original

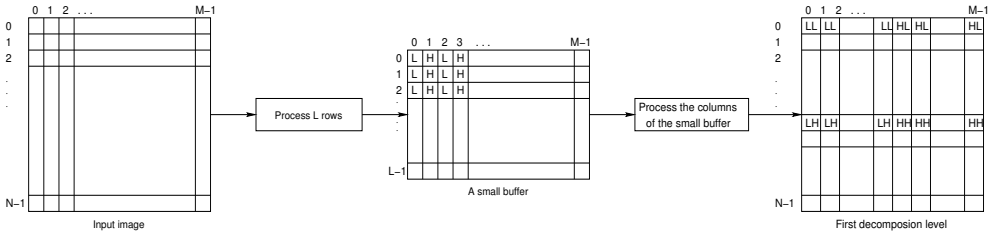


Figure 5.3: *The line-based wavelet transform approach processes both rows and columns in a single loop.*

image and stores the wavelet coefficients in an auxiliary matrix. Thereafter, the vertical filtering phase processes the columns of the auxiliary matrix and stores the results back in the original matrix. In other words, this algorithm requires that all lines are horizontally filtered before the vertical filtering starts. The computational complexity of both horizontal and vertical filtering is the same. Figure 5.1 depicts both horizontal and vertical filtering. Each of these filtering is applied separately. Each of these $N \times M$ matrices requires NMc bytes of memory, where c denotes the number of bytes required to represent one wavelet coefficient. The RCWT traversal technique has been used to implement the 2D DWT in this chapter.

5.1.2 Line-Based Wavelet Transform

In the line-based wavelet transform approach, the vertical filtering starts as soon as a sufficient number of lines, as determined by the filter length, has been horizontally filtered. In other words, the LBWT algorithm uses a single loop to process both rows and columns together. This technique computes the 2D DWT of an $N \times M$ image by the following stages. First, the horizontal filtering filters L rows, where L is the filter length, and stores the lowpass and highpass values interleaved in an $L \times M$ buffer. Thereafter, the columns of this small buffer are filtered. This produces two wavelet coefficients rows, which are stored in different subbands in an auxiliary matrix in the order expected by the quantization step. Finally, these stages are repeated to process all rows and columns. Figure 5.3 illustrates the LBWT algorithm.

5.2 Issues Related to the 2D DWT on the GPPs

As previously mentioned, a 2D DWT consists of horizontal filtering along the rows followed by vertical filtering along the columns. In order to develop high-performance implementations of the 2D DWT on GPPs in general and the P4 in particular, the following issues need to be addressed:

```
void Lifting53_vertical(){
for (i=0, ii=1; ii<N; i++, ii+=2)
  for (j=0; j<M; j++) {
    img[i+N/2][j] = tmp[ii][j] - ((tmp[ii-1][j]
                  + tmp[ii+1][j])>>1);
    img[i][j] = tmp[ii-1][j] + ((img[i+N/2][j]
              + img[i+N/2-1][j]+2)>>2);
  }
}
```

Figure 5.4: *C implementation of vertical filtering using the* $(5,3)$ *lifting scheme with loop interchange technique.*

First, the P4 suffers from a problem known as 64K aliasing, which can degrade performance by an order of magnitude. It occurs when two data blocks need to be cached simultaneously whose addresses differ by a multiple of 64K [69]. In implementation of the 2D DWT, there is a 64K alias between the lowpass and the highpass values when the image size is a large power of two. This phenomena and the proposed techniques to circumvent it will be discussed in more detail in Section 5.4.

Second, the straightforward way of performing vertical filtering is by processing each column entirely before advancing to the next column. This method, however, results in excessive cache misses because it is unable to exploit spatial locality, since the cache blocks corresponding to the first rows will have been evicted from the cache when the algorithm advances to the next column. In order to improve spatial locality, *loop interchange* has been applied, which is a well-known compiler technique. Figure 5.4 depicts the C implementation of vertical filtering using the $(5,3)$ lifting scheme for an $N \times M$ image with loop interchange. As this figure shows, the loop interchange technique places the loop with index `j` after the loop with index `i` allowing to process the same rows successively, thereby helping to reduce cache misses. Figure 5.5 depicts the effectiveness of loop interchange for vertical filtering. It depicts the speedup of vertical filtering with interchanged loops over the straightforward implementation, which processes each column entirely before advancing to the next column for the $(5,3)$ lifting and Daub-4 transforms. Clearly, the implementations with interchanged loops are much more efficient than the straightforward implementations, especially when the image is large.

Loop interchange, however, does not solve all cache and memory problems. This is because there are still many conflict misses if the filter length exceeds the number of cache ways, in particular, if the image size is a multiple of the cache size. Therefore, two techniques have been proposed to reduce the number of conflict misses that will be explained in Section 5.5. Although 64K aliasing is a problem specific to the P4, the conflict avoidance methods are general and can be applied to other processors as well. To show this, results are also presented for the P3 and AMD Opteron processors.

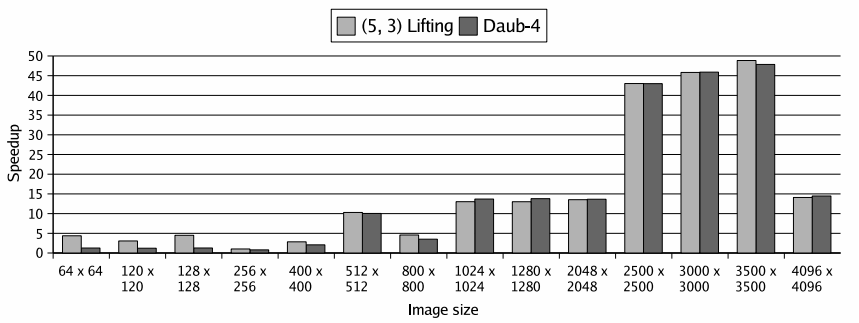


Figure 5.5: *Effectiveness of loop interchange on the Pentium 4. This figure depicts the speedup of vertical filtering with interchanged loops over the straightforward implementation, which processes each column entirely before advancing to the next column for the lifting and Daub-4 transforms.*

As shown in Figure 5.5, the loop interchange technique improves the performance of the vertical filtering significantly. For this reason the performance of the proposed techniques will be compared to the performance attained by the algorithms after loop interchange. In other words, the implementations with interchanged loops will be used as reference implementations.

Third, high-performance implementations of the 2D DWT must exploit the DLP using the SIMD instructions supported by most GPPs. Section 5.6 describes how the 2D DWT can be vectorized using MMX and SSE instructions. Vertical filtering is relatively straightforward to vectorize. Horizontal filtering is more difficult to vectorize, however, since it requires the data to be reorganized.

5.3 Experimental Setup

All C programs and SIMD implementations have been executed on the P4 processor. In addition, to show generality of the proposed techniques to avoid cache conflict misses, the P3 and AMD Opteron processors have also been used. The main architectural parameters of these systems are summarized in Table 5.1. All versions were compiled using *gcc* with optimization level *-O2* and executed on a lightly loaded system. The speedup was measured by the ratio of execution cycle count. The total number of cycles has been obtained by the IA-32 cycle counter [70], which has been explained in Section 1.4 of Chapter 1. To eliminate the effects of context switching and compulsory cache misses, the *K-best* measurement scheme and a *warmed up* cache have been used.

Processor	Intel Pentium 3	AMD Opteron	Intel Pentium 4
CPU Clock Speed	451MHz	2.0GHz	3.0GHz
L1 Data Cache	16 KBytes 2-way set associativity 32 Bytes line size	64 KBytes 2-way set associativity 64 Bytes line size	8 KBytes 4-way set associativity 64 Bytes line size
L2 Cache	512 KBytes 8-way set associativity 32 Bytes line size	1 MBytes 8-way set associativity 32 Bytes line size	512 KBytes 8-way set associativity 64 Bytes line size
Memory	384 MBytes	1 GBytes	1 GBytes

Table 5.1: *Parameters of the experimental platforms.*

5.4 Avoiding 64K Aliasing

In the Pentium 4 there is a phenomenon known as *64K aliasing* [69]. It occurs if two or more data blocks whose addresses differ by a multiple of 64K need to be cached simultaneously. If it occurs, the associativity of the cache is useless and the effectiveness of the cache is greatly reduced. The reasons for the 64K aliasing problem are not well documented. Some sources [64] say it is due to incomplete tag encoding. More precisely, only 16 bits are used for the cache lookup: 6 bits for the block offset, 5 bits for the index, and bits 11 to 15 for the tag [97]. The remaining tag bits come from the Dynamic Translation Look-aside Buffer (DTLB). Because of this, references to addresses with the same 16 lower-order bits (i.e., addresses that are 2^{16} bytes or a multiple thereof apart) are not resolvable in the L1 data cache. According to the Intel documentation [69], the instruction that accesses the second 64K aliasing data item has to wait until the first one is written from the cache. This clearly obstructs out-of-order processing.

For some image sizes, the 2D DWT suffers from 64K aliasing. To illustrate this, Figure 5.6 depicts the slowdown of the reference implementation of vertical filtering over horizontal filtering on the P4. Even though they perform the same number of operations, for some image sizes vertical filtering is substantially slower (up to a factor of 4.27) than horizontal filtering. One reason for this could be the cache behavior. To analyze if this is the case, Figure 5.7 shows the ratio of the number of cache misses incurred by vertical filtering to the number of cache misses incurred by horizontal filtering. These results have been obtained using a trace-driven cache simulator with the cache configured as the L1 data cache of the P4.

It can be seen that the slowdown of vertical filtering over horizontal filtering cannot be explained by the cache miss behavior. For example, when the image size is

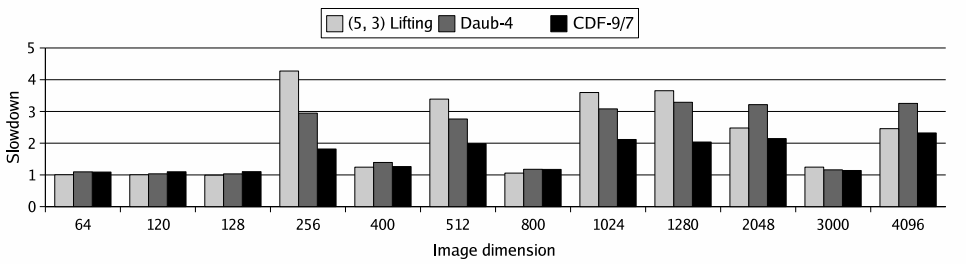


Figure 5.6: *Slowdown of vertical filtering over horizontal filtering on the P4.*

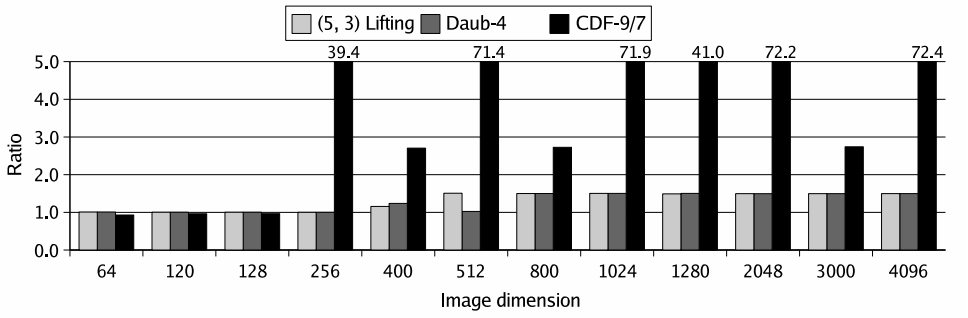


Figure 5.7: *Ratio of the number of cache misses incurred by vertical filtering to the number of cache misses incurred by horizontal filtering for an 8KB 4-way set-associative L1 data cache with a line size of 64 bytes.*

256×256, vertical filtering is slower than horizontal filtering by a factor of 4.27 for the lifting transform, and by a factor of 2.95 for the Daub-4 transform. For both transforms, however, they incur about the same number of cache misses. For the CDF-9/7 transform, on the other hand, vertical filtering is slower by a factor of 1.82 but generates more than 39 times as many cache misses as horizontal filtering. Similar behavior can be observed for other image sizes. Hence the large slowdown of vertical versus horizontal filtering should not (only) be attributed to cache misses but mainly to 64K aliasing.

To further explain why and when 64K aliasing occurs, Figure 5.8 depicts a C implementation of vertical filtering using the Daub-4 transform. It can be seen that one iteration of the inner loop accesses `img[i][j]` and `img[i+N/2][j]`. Hence 64K aliasing occurs if $cN^2/2$ is a multiple of 2^{16}, where c is the number of bytes needed to represent one wavelet coefficient. Since c is 2 for the lifting and 4 for the Daub-4 and CDF-9/7 transforms, for square $N \times N$ images 64K aliasing occurs if $N = 256$ (since $2 \cdot 256^2/2 = 2^{16}$), for powers of 2 larger than 256, and for $N = 1280$ (since $2 \cdot 1280^2/2 = 25 \cdot 2^{16}$). Although it is focused on square images in this chapter, 64K aliasing may also occur for non-square images. For $N \times M$ images, it occurs

```
void Daub_4_vertical() {
int i, j, jj;
float low[] ={-0.1294, 0.2241, 0.8365 , 0.4830};
float high[]={-0.4830, 0.8365, -0.2241, -0.1294};
for (i=0, ii=0; ii<N; i++, ii +=2)
  for (j=0; j<M; j++) {
    img[i][j]= tmp[ii][j]   *low[0]+tmp[ii+1][j]*low[1] +
               tmp[ii+2][j] *low[2]+tmp[ii+3][j]*low[3];

    img[i+N/2][j]= tmp[ii][j]  *high[0]+tmp[ii+1][j]*high[1] +
                   tmp[ii+2][j]*high[2]+tmp[ii+3][j]*high[3];
  }
}
```

Figure 5.8: *C implementation of vertical filtering using the Daub-4 transform. Note that the loops have been interchanged w.r.t. the straightforward implementation.*

when $cNM/2$ is a multiple of 2^{16}.

To circumvent 64K aliasing, two techniques are proposed and evaluated. The first idea is to split the inner loop so that the lowpass (`img[i][j]`) and highpass values (`img[i+N/2][j]`) are calculated in separate loops. In this way the 64K alias between them is removed. This is actually a well-known compiler technique called *loop fission*. Loop fission, however, is usually applied to enable other transformations such as loop interchange and vectorization, while here it is applied to avoid 64K aliasing.

Figure 5.9 depicts the speedup resulting from this program transformation. For those image sizes that suffer from 64K aliasing (as explained above, powers of two larger than 256×256 and 1280×1280), loop fission indeed improves performance significantly. In these cases the speedup ranges from 1.97 to 2.94 for the lifting transform, from 2.36 to 3.31 for Daub-4, and from 1.27 to 1.75 for CDF-9/7. For CDF-9/7, the performance improvements are smaller than for the other two transforms, because it also suffers from many cache conflict misses. The CDF-9/7 transform has a filter length of 9, which is larger than the filter length of the other transforms. In other words, the filter length of the CDF-9/7 exceeds the number of cache ways that it is 4. However, for those image sizes that do not suffer from 64K aliasing, loop fission reduces performance by up to 20%. This is due to the following reasons. First and most importantly, loop fission removes the temporal reuse that exists between the calculation of the highpass and lowpass values. As can be seen in Figure 5.8, the first statement in the loop body accesses `tmp[ii][j]` and so does the second statement. After loop fission has been applied, the two statements are in different loops and this temporal reuse has been removed. Second, loop fission increases loop overhead but this overhead could be reduced by unrolling the loop.

The second proposed technique is to offset the memory address of the highpass value by one or two rows depending on the transform. In other words, instead of storing the

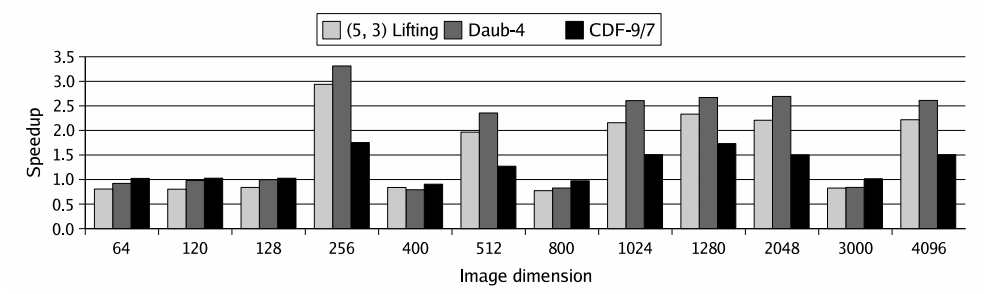


Figure 5.9: *Speedup of vertical filtering over the reference implementation achieved by loop fission.*

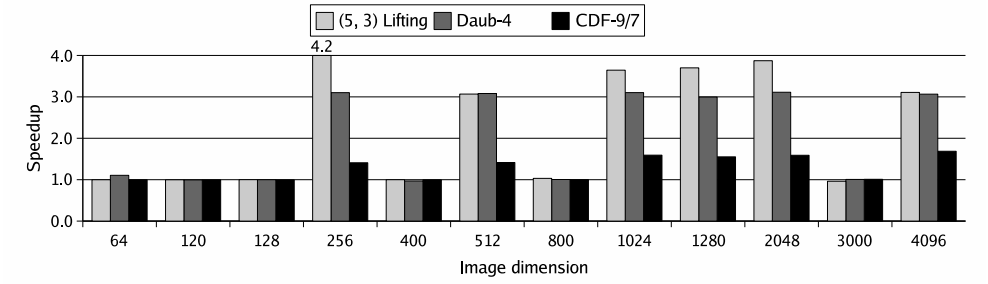


Figure 5.10: *Performance improvement achieved by the offsetting technique.*

highpass value in `img[i+N/2][j]`, it is stored in `img[i+N/2+1][j]`. By applying this offsetting technique, the distance between the two addresses is no longer a multiple of 64K, but to apply this method, the matrices have to be extended with one or two rows.

Figure 5.10 depicts the speedup achieved by the offsetting technique. For those image sizes that suffer from 64K aliasing, it improves performance by a factor ranging from 3.07 to 4.20 for the lifting transform, from 2.99 to 3.11 for Daub-4, and from 1.41 to 1.69 for CDF-9/7. Moreover, the offsetting technique does *not* incur a performance penalty for image sizes that do not suffer from 64K aliasing. This is because this technique does not destroy the temporal locality between the calculation of the lowpass and highpass values. Concluding, the offsetting technique is better than loop fission.

5.5 Cache Optimization

Figure 5.7 shows that for small images (up to 128×128), vertical filtering does not produce more cache misses than horizontal filtering, regardless of the transform em-

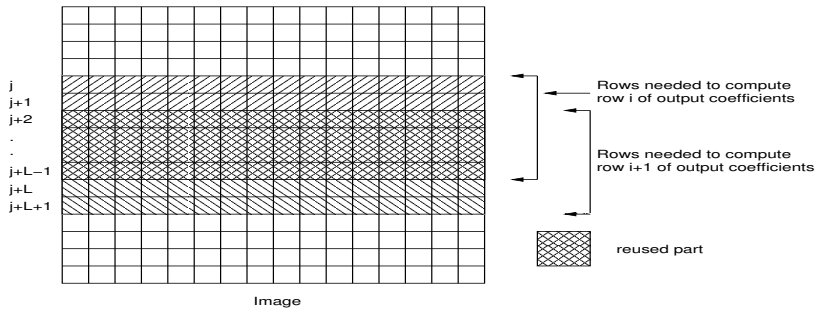


Figure 5.11: *Reuse in vertical filtering.*

ployed. For images larger than 800×800, however, vertical filtering generates about 50% more misses than horizontal filtering for the lifting and Daub-4 transforms. For the Daub-4 transform (Figure 5.8) this can be explained as follows. To compute row i of `img`, it uses rows $2i, 2i+1, 2i+2, 2i+3$ of `tmp`. Hence to compute row $i+1$, it uses rows $2i+2, 2i+3, 2i+4, 2i+5$. This implies that rows $2i+2$ and $2i+3$ are reused, provided 4 rows can be kept in cache. When $N \geq 800$, however, they cannot (since $4 \times 4 \times 800 > 8$KB), which is why vertical filtering generates more misses than horizontal filtering. For the lifting transform this actually already occurs for $N = 512$, because this transform does not access 4 consecutive rows of input data. In general, if the rows of input image needed to compute row i of output data are the rows j to $j+L-1$, then rows $j+2$ to $j+L-1$ are reused to compute the next row $i+1$, provided L rows can be kept in cache. This is illustrated in Figure 5.11. More serious behavior, however, is exhibited by the CDF-9/7 transform.

For example, when N is (a multiple of) a large power of two, vertical filtering with this transform generates up to more than 72 times as many cache misses as horizontal filtering. This can be explained as follows. When $N = 512$, each row is $512 \times 4 =$ 2KB of data. Since each way of the P4 L1 data cache is also 2KB, corresponding blocks in different rows map to the same cache set. Consequently, since 9 blocks are needed to compute one block of output data and this exceeds the number of cache ways, many conflict misses are generated. In other words, the reuse that exist between the computation of `img[i][j]` and `img[i][j+1]` (provided they are in the same cache block) is destroyed. The same holds when N is a multiple of 512. When $N = 256$ or $N = 1280$, 5 blocks map to the same cache set, causing also many cache misses but fewer than when all 9 blocks map to the same set. For the lifting and Daub-4 transforms, this problem does not exist because their filter lengths are equal to the number of cache ways.

To solve this problem, it needs to be ensured that at most n rows are accessed before advancing to the next output data, where n is the number of cache ways. In the remainder of this section, two methods are presented for doing so. Since they are

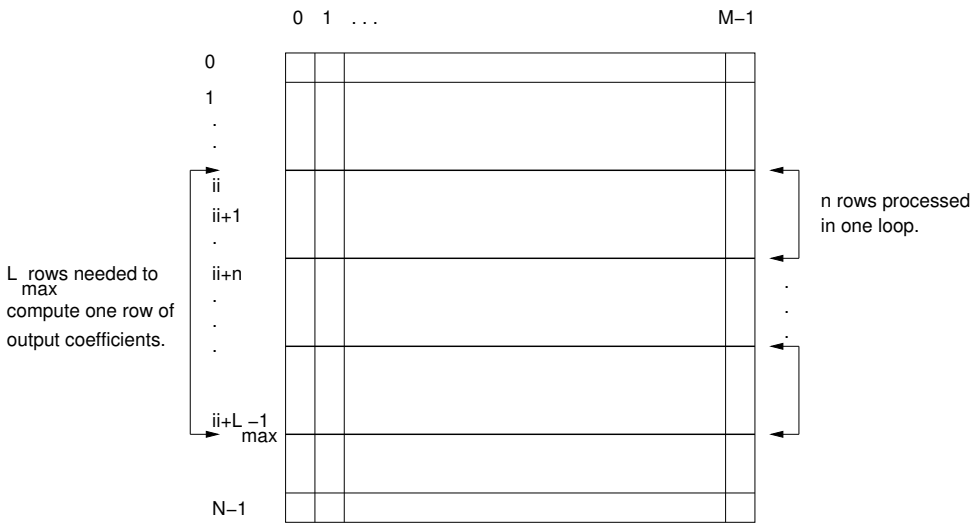


Figure 5.12: *Associativity-conscious loop splitting.*

parameterized by the number of cache ways and the filter length, they can also be applied to other cache organizations and transforms. In other words, both techniques are general and architecture independent.

5.5.1 Associativity-Conscious Loop Fission Technique

The first method is referred to as *associativity-conscious loop fission* (ACLF). The idea is to split the loop that computes one row of wavelet output into multiple loops so that each loop accesses at most n rows. Each loop computes the partial results that can be computed by accessing the first n rows of input data. The remaining loops add their results to these partial results.

Specifically, let $L_{\text{max}} = \max\{L_{\text{low}}, L_{\text{high}}\}$, where L_{low} and L_{high} are the lengths of the lowpass and highpass filters of the DWT. One row of wavelet coefficients is calculated using L_{max}/n loops, and each loop accesses n rows of input coefficients. This transformation is illustrated in Figure 5.12 and pseudo-code that illustrates the transformation is depicted in Figure 5.13. For brevity and simplicity, start-up and clean-up code has been omitted.

5.5.2 Lookahead Technique

In the second scheme, which is called *lookahead*, the rows are processed in a skewed manner. There is only one loop, as in the original algorithm. In it-

```
for (i=0, ii=0; ii<N; i++, ii+=2)
  for (j=0; j<M; j++) {
    img[i][j] = tmp[ii][j] * low[0] + tmp[ii+1][j] * low[1]
              + . . .   + tmp[ii+L_max-1][j] * low[L_max-1];
    . . .
  }
```

(a)

```
for (i=0, ii=0; ii<N; i++, ii+=2) {
  for (L=0; L<L_max ; L+=n)
    for (j=0; j<M; j++) {
      img[i][j] += tmp[ii+L][j]   * low[L]
                +  tmp[ii+L+1][j] * low[L+1]
                +  . . . + tmp[ii+L+n-1] * low[L+n-1];
      . . .
    }
}
```

(b)

Figure 5.13: *(a) reference implementation and (b) associativity-conscious loop splitting technique.*

eration j ($0 \leq j < N$) a partial results is computed for the output element `img[i][j]` but, in the same iteration, a partial result is computed for the output element `img[i][(j+B/c) mod N]` that is located B/c columns ahead, for the element `img[i][(j+2*B/c) mod N]`, and so on. Here B is the cache line size in bytes and c is the number of bytes per wavelet coefficient, as before. To compute a partial result for `img[i][j]`, n input elements `tmp[ii][j]`, `tmp[ii+1][j]`, ..., `tmp[ii+n-1][j]` are processed. A partial result for `img[i][(j+B/c) mod N]` is computed using the elements `tmp[ii+n][(j+B/c) mod N]`, ..., `tmp[ii+2n-1][(j+B/c) mod N]`, and so on. So in each iteration, L/n partial results are computed, where L is the filter length. In later iterations, partial results corresponding to the same column are accumulated. This scheme ensures that no more than n input coefficients accessed in one loop iteration map to the same cache set. This algorithm is illustrated in Figure 5.14 and pseudo-code that illustrates the transformation is given in Figure 5.15. For brevity and simplicity, start-up and clean-up code has been omitted.

5.5.3 Performance Results

Figure 5.16 compares the performance improvements obtained by applying ACLF or lookahead in addition to offsetting to the speedup achieved by applying offsetting alone. Results are presented only for CDF-9/7, since only this transform suffers from both 64K aliasing as well as excessive cache misses. For image sizes that experience

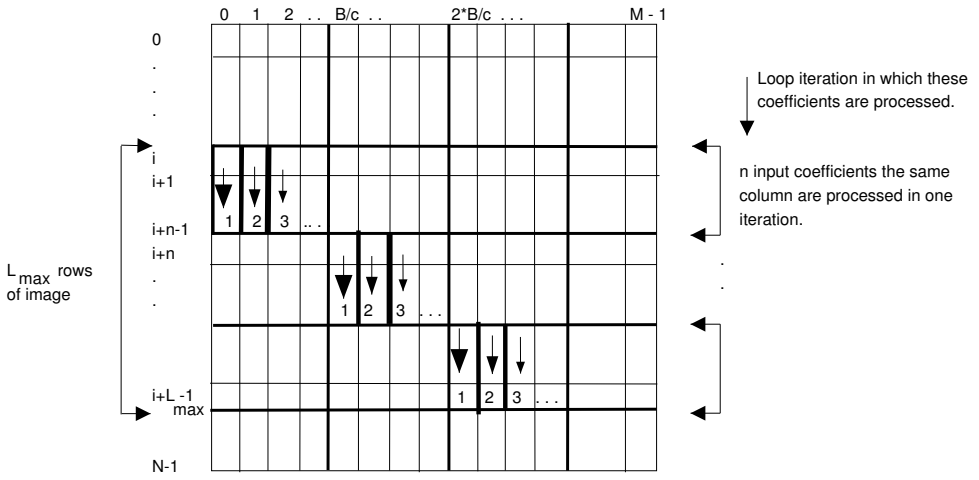


Figure 5.14: *Illustration of the lookahead algorithm for vertical filtering.*

```
for (i=0, ii=0; ii<N; i++, ii+=2)
  for (j=0; j<M; j++) {
    img[i][j] = tmp[ii][j] * low[0] + tmp[ii+1][j] * low[1]
              + . . .  + tmp[ii+L_max-1][j] * low[L_max-1];
    . . .
  }
```

(a)

```
for (i=0, ii=0; ii<N; i++, ii+=2)
  for (j=0; j<M - L_max/n*B/c; j++) {
    img[i][j] += tmp[ii][j] * low[0]
              +  tmp[ii+1][j] * low[1]
              + . . . + tmp[ii+n-1] * low[n-1];

    img[i][j+B/c] += tmp[ii+n][j+B/c]   * low[n]
                  +  tmp[ii+n+1][j+B/c] * low[n+1]
                  + . . . + tmp[ii+2*n-1][j+B/c] * low[2*n-1];
    . . .
    img[i][j+L_max/n*B/c] +=tmp[ii+L_max-1-n][j+L_max/n*B/c]
                 * low[L_max-1-n]
                 + tmp[ii+L_max-1-n+1][j+L_max/n*B/c]
                 * low[L_max-1-n+1] + . . .
                 + tmp[ii+L_max-1][j+L_max/n*B/c]*low[L_max-1];
  }
```

(b)

Figure 5.15: *(a) reference implementation and (b) lookahead technique.*

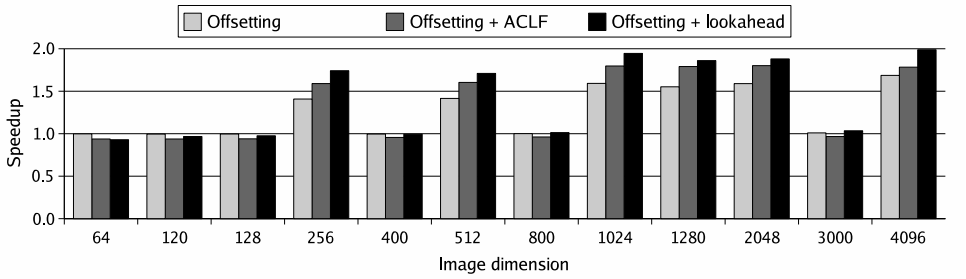


Figure 5.16: *Comparison of the speedups obtained by applying offsetting alone to the speedups achieved by applying associativity-conscious loop fission or lookahead in addition to offsetting for the CDF-9/7 transform.*

many cache conflicts (as explained in Section 5.5, $N = 256$ and multiples thereof), avoiding them provides additional performance improvements. For example, applying offsetting alone provides a speedup of up to 1.69, while combining it with the lookahead technique yields a speedup of up to 1.99. In general, the lookahead technique performs slightly better than ACLF. This is because it incurs less loop overhead than ACLF. For image sizes that do not generate many conflict misses, both schemes generally slightly decrease performance (by at most 7%). This is due to overhead needed for managing loop and index variables and address calculations.

As mentioned before, both ACLF and the lookahead technique are general and architecture independent. This means that, although results have been measured for the CDF-9/7 transform and on the Pentium 4, they can also be applied to other transforms and processors with different cache configurations. For example, for certain image sizes, the $(5, 3)$ lifting and Daub-4 transforms would incur many cache conflict misses for a 2-way set-associative cache. But in these cases the same techniques can be applied with the parameters $L = 4$ and $n = 2$. To validate this claim, Figure 5.17 depicts the speedup obtained by applying ACLF and the lookahead techniques on the P3 and the Opteron processors. Analytically it can be determined that on the P3 many conflict misses occur when $N = 1024, 2048, 4096$ and on the Opteron when $N = 2048, 4096$ and, to a lesser extent, for $N = 1280$. Figure 5.17 shows that for these image sizes ACLF provides a performance improvement ranging from 14% to 18% on the P3 and from 2% to 95% on the Opteron. On the other hand, the lookahead technique improves performance by 34% to 41% on the P3 and by 15% to 126% on the Opteron. However, for image sizes that do not generate many cache conflict misses, those techniques reduce performance by up to 20%. This shows that it is necessary to provide different versions of the code and, depending on the image size and the cache organization of the target platform, to call the most efficient version.

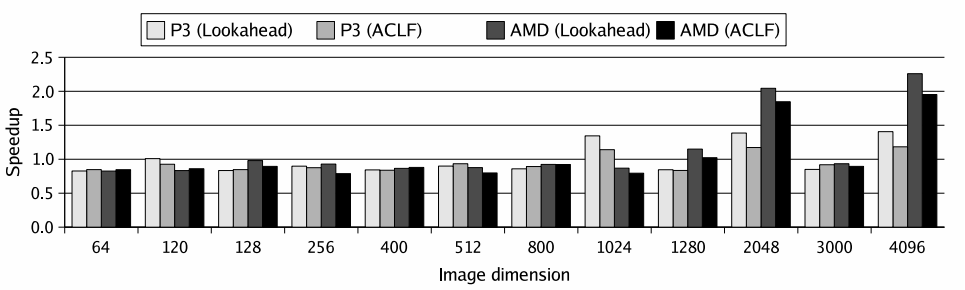


Figure 5.17: *Speedups obtained by applying ACLF and the lookahead technique over the reference implementation of the CDF-9/7 transform on the P3 and Opteron.*

5.6 SIMD Vectorization

An efficient implementation of the DWT on the P4 as well as other GPPs must exploit the SIMD extensions provided by these processors. This section presents MMX/SSE implementations of the DWT and also discusses performance results on the Intel P4 processor. This section is organized as follows. The SIMD implementations of the convolutional methods (Daub-4 and CDF-9/7) are discussed in Section 5.6.1. Thereafter, an SIMD implementation of the lifting scheme is presented in Section 5.6.2. Performance results are provided in Section 5.6.3. In Section 5.6.4, the limitations of the SIMD implementations that restrict the performance improvements are discussed. Section 5.6.5 discusses the possible solutions to improve the performance of SIMD implementations of the 2D DWT. Finally, experimental results are presented in Section 5.6.6.

5.6.1 SIMD Implementations of Convolutional Methods

The SIMD implementations of Daub-4 and CDF-9/7 are very similar. Both process single-precision floating-point values and apply filtering by multiplying the filter coefficients with the input samples and accumulating the results. They will therefore be discussed together.

Under a row-major image layout, it is relatively straightforward to vectorize vertical filtering using SSE instructions. This is because the elements that can be processed simultaneously are stored consecutively in memory. Consider, for example, the Daub-4 transform and let $x_{i,j}$ be the input samples, let $c_0, \ldots, c_3$ denote the lowpass filter coefficients, and let $L_{i,j}$ be the lowpass values. Then vertical filtering of the lowpass values is given by:

$$
\begin{aligned}
&(L_{i,j}\ L_{i,j+1}\ L_{i,j+2}\ L_{i,j+3}) = \\
&\quad (c_0\ c_0\ c_0\ c_0) \times (x_{2i,j}\ x_{2i,j+1}\ x_{2i,j+2}\ x_{2i,j+3}) + \\
&\quad (c_1\ c_1\ c_1\ c_1) \times (x_{2i+1,j}\ x_{2i+1,j+1}\ x_{2i+1,j+2}\ x_{2i+1,j+3}) + \\
&\quad (c_2\ c_2\ c_2\ c_2) \times (x_{2i+2,j}\ x_{2i+2,j+1}\ x_{2i+2,j+2}\ x_{2i+2,j+3}) + \\
&\quad (c_3\ c_3\ c_3\ c_3) \times (x_{2i+3,j}\ x_{2i+3,j+1}\ x_{2i+3,j+2}\ x_{2i+3,j+3})
\end{aligned} \tag{5.1}
$$

In this equation, the operator $\times$ denotes elementwise vector multiplication. Similar equations can be drawn for the highpass values and other convolutional filters. This equation can be mapped almost one-to-one to SSE instructions. The only technical detail is that each coefficient needs to be replicated four times. Figure 5.18 illustrates the data flow graph of the vertical filtering. As this figure shows four different input samples of each row are multiplied with one filter coefficient simultaneously. Each filter coefficient should be replicated in each of the four different subwords of a media register. After four multiplications of four consecutive rows with different coefficients, the results of each column are added to each other. Finally, four wavelet coefficients are calculated simultaneously.

Horizontal filtering is more difficult to vectorize, however. In this case, the lowpass values can be calculated using the equation:

$$
\begin{aligned}
&(L_{i,j}\ L_{i,j+1}\ L_{i,j+2}\ L_{i,j+3}) = \\
&\quad (c_0\ c_0\ c_0\ c_0) \times (x_{i,2j}\ x_{i,2j+2}\ x_{2i,2j+4}\ x_{2i,2j+6}) + \\
&\quad (c_1\ c_1\ c_1\ c_1) \times (x_{i,2j+1}\ x_{i,2j+3}\ x_{2i,2j+5}\ x_{2i,2j+7}) + \\
&\quad (c_2\ c_2\ c_2\ c_2) \times (x_{i,2j+2}\ x_{i,2j+4}\ x_{2i,2j+6}\ x_{2i,2j+8}) + \\
&\quad (c_3\ c_3\ c_3\ c_3) \times (x_{i,2j+3}\ x_{i,2j+5}\ x_{2i,2j+7}\ x_{2i,2j+9})
\end{aligned} \tag{5.2}
$$

In addition, Figure 5.19 depicts the data flow graph of the horizontal filtering. As this figure shows four different input samples are multiplied with four different coefficients. The intermediate results are accumulated into one destination operand. In other words, to map this figure to SIMD instructions, a vector-vector multiplication (dot product) instruction would have been useful, but since SSE does not provide such an instruction, the elements should be rearranged so that, for example, the input samples $x_{i,2j}$, $x_{i,2j+2}$, $x_{2i,2j+4}$, and $x_{2i,2j+6}$ are stored consecutively in an SSE register. Figure 5.20 shows the SSE code that computes four lowpass values. It can be seen that many overhead (unpack) instructions are needed.

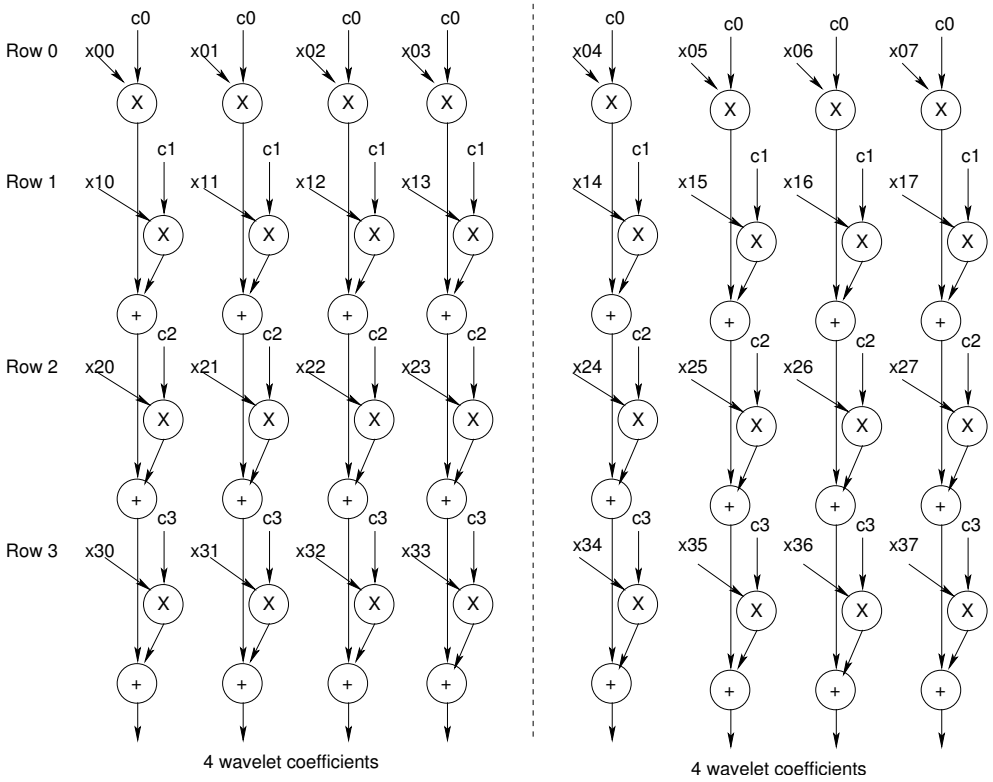


Figure 5.18: *Data flow graph of the vertical filtering of the Daub-4 transform.*

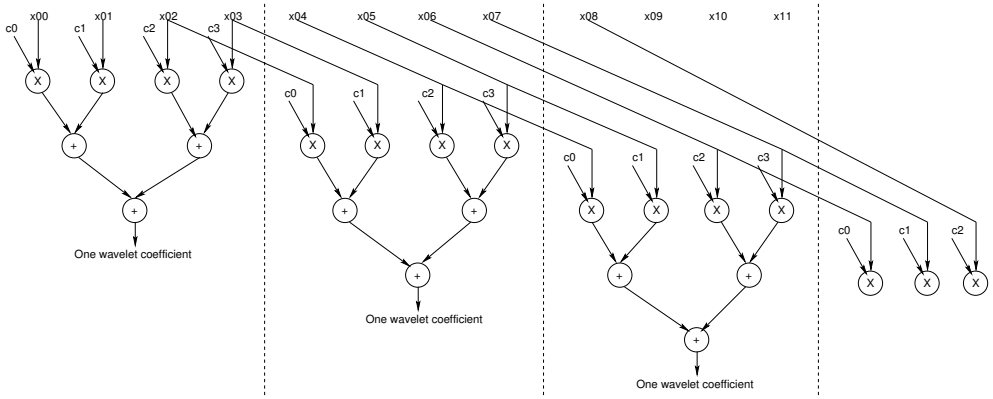


Figure 5.19: *Data flow graph of the horizontal filtering of the Daub-4 transform.*

`movaps xmm0, (esi)` ; xmm0 =	a_3	a_2	a_1	a_0
`movaps xmm1,16(esi)` ; xmm1 =	a_7	a_6	a_5	a_4
`movaps xmm2, xmm0` ; xmm2 =	a_3	a_2	a_1	a_0
`unpcklps xmm0, xmm1` ; xmm0 =	a_5	a_1	a_4	a_0
`unpckhps xmm2, xmm1` ; xmm2 =	a_7	a_3	a_6	a_2
`movaps xmm1, xmm0` ; xmm1 =	a_5	a_1	a_4	a_0
`unpcklps xmm0, xmm2` ; xmm0 =	a_6	a_4	a_2	a_0
`unpckhps xmm1, xmm2` ; xmm1 =	a_7	a_5	a_3	a_1
`movups xmm2,8(esi)` ; xmm2 =	a_5	a_4	a_3	a_2
`movups xmm3,24(esi)` ; xmm3 =	a_9	a_8	a_7	a_6
`movaps xmm4, xmm2` ; xmm4 =	a_5	a_4	a_3	a_2
`unpcklps xmm2, xmm3` ; xmm2 =	a_7	a_3	a_6	a_2
`unpckhps xmm4, xmm3` ; xmm4 =	a_9	a_5	a_8	a_4
`movaps xmm3, xmm2` ; xmm3 =	a_7	a_3	a_6	a_2
`unpcklps xmm2, xmm4` ; xmm2 =	a_8	a_6	a_4	a_2
`unpckhps xmm3, xmm4` ; xmm3 =	a_9	a_7	a_5	a_3
`movaps xmm4, xmm0` ; xmm4 =	a_6	a_4	a_2	a_0
`movaps xmm5, xmm1` ; xmm5 =	a_7	a_5	a_3	a_1
`movaps xmm6, xmm2` ; xmm6 =	a_8	a_6	a_4	a_2
`movaps xmm7, xmm3` ; xmm7 =	a_9	a_7	a_5	a_3

Figure 5.20: *Computing four lowpass values for horizontal filtering using SSE instructions (Daub-4 transform).*

5.6.2 MMX Implementation of the Lifting Scheme

The SIMD implementation of the $(5, 3)$ lifting scheme is significantly different from the SSE implementations of Daub-4 and CDF-9/7 for the following reasons. First, the $(5, 3)$ lifting transform uses integer arithmetic and hence its SIMD implementation employs MMX instructions. Second, in the MMX implementation there are no multiplication operations, since the input values need to be divided by powers of 2 which can be accomplished using shift operations. Third, because of its structure, the $(5, 3)$ lifting scheme is vectorized in a completely different way than the convolutional transforms.

As previously mentioned, the lifting operation consists of several stages. First, the original 1D input signal is split into a subsequence consisting of the even-numbered input values $\{s_i^0\}$ and a subsequence containing the odd-numbered input values $\{d_i^0\}$. Thereafter, the prediction stage produces the highpass output values $\{d_i^1\}$ and the update stage generates the lowpass output values $\{s_i^1\}$.

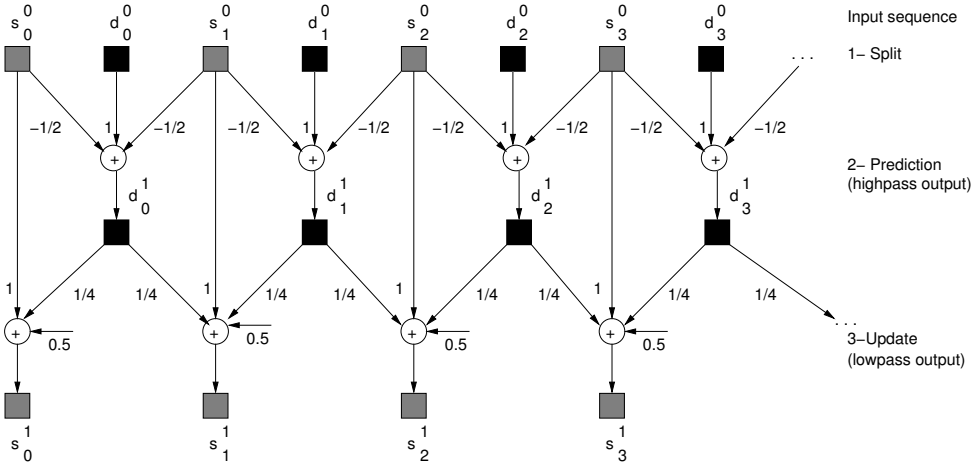


Figure 5.21: *One prediction and update stage in the lifting scheme of the* $(5, 3)$ *lifting transform.*

The lifting operation for the $(5, 3)$ filter bank is depicted in Figure 5.21. The sequences $\{s_i^0\}$ and $\{d_i^0\}$ denote the even and odd input sequence and the outputs $\{s_i^1\}$ and $\{d_i^1\}$ are the lowpass and highpass output coefficients of the DWT filter, respectively.

The equations that are used in the prediction and update stage of the $(5, 3)$ lifting transform are given by:

$$d_i^1 = d_i^0 - \lfloor \frac{s_i^0 + s_{i+1}^0}{2} \rfloor \tag{5.3}$$

$$s_i^1 = s_i^0 + \lfloor \frac{d_{i-1}^1 + d_i^1 + 2}{4} \rfloor \tag{5.4}$$

Figure 5.22 depicts a part of the data flow graph of the $(5, 3)$ lifting scheme based on Equation (5.3) and Equation (5.4). In order to vectorize horizontal filtering, the data needs to be rearranged so that the even and odd subsequences are placed in different registers. Furthermore, because s_i^0 and s_{i+1}^0 have to be added, two copies of the even subsequence are required, one that starts with s_0^0 and one that starts with s_1^0. Figure 5.23 shows the MMX code that achieves this rearrangement. It can be seen that many unpack instructions are required to achieve this. After the code has been executed, the first four highpass output values can be computed by adding `mm0` with `mm3`, shifting the results to the right by 1 bit position, and adding these results to `mm4`.

As was the case for the convolutional transforms, vertical filtering is easier to vectorize. In this case, the even and odd subsequences do not have to be split because they correspond to different rows. A drawback of vertical filtering compared to

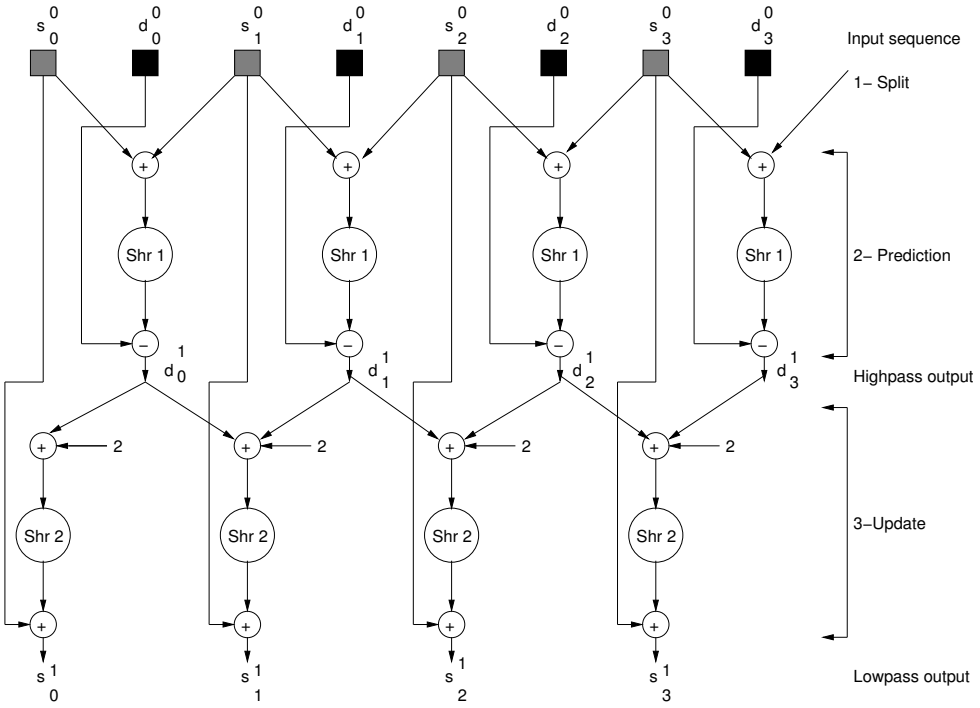


Figure 5.22: *Part of the data flow graph of the forward integer-to-integer lifting transform using the* $(5,3)$ *filter bank (Shr = Shift right).*

`movq mm0, (esi)` ; mm0 =	s_0^0	d_0^0	s_1^0	d_1^0	s_2^0	d_2^0	s_3^0	d_3^0
`movq mm1,8(esi)` ; mm1 =	s_4^0	d_4^0	s_5^0	d_5^0	s_6^0	d_6^0	s_7^0	d_7^0
`pxor mm7, mm7` ; mm7 =	0	0	0	0	0	0	0	0
`movq mm2, mm0` ; mm2 =	s_0^0	d_0^0	s_1^0	d_1^0	s_2^0	d_2^0	s_3^0	d_3^0
`punpcklbw mm0, mm7` ; mm0 =	s_0^0		d_0^0		s_1^0		d_1^0	
`punpckhbw mm2, mm7` ; mm2 =	s_2^0		d_2^0		s_3^0		d_3^0	
`punpcklbw mm1, mm7` ; mm1 =	s_4^0		d_4^0		s_5^0		d_5^0	
`movq mm3, mm0` ; mm3 =	s_0^0		d_0^0		s_1^0		d_1^0	
`punpcklwd mm0, mm2` ; mm0 =	s_0^0		s_2^0		d_0^0		d_2^0	
`punpckhwd mm3, mm2` ; mm3 =	s_1^0		s_3^0		d_1^0		d_3^0	
`movq mm4, mm0` ; mm4 =	s_0^0		s_2^0		d_0^0		d_2^0	
`punpcklwd mm0, mm3` ; mm0 =	s_0^0		s_1^0		s_2^0		s_3^0	
`punpckhwd mm4, mm3` ; mm4 =	d_0^0		d_1^0		d_2^0		d_3^0	
`punpcklwd mm2, mm1` ; mm2 =	s_2^0		s_4^0		d_2^0		d_4^0	
`punpcklwd mm3, mm2` ; mm3 =	s_1^0		s_2^0		s_3^0		s_4^0	

Figure 5.23: *MMX instructions needed to rearrange the elements for the* $(5,3)$ *lifting scheme.*

horizontal filtering is, however, that the previous highpass output values which are needed for the update stage cannot be kept in a register, while in horizontal filtering they can. For example, in vertical filtering, after calculating the highpass values $\{d^1_{i+1,j}, d^1_{i+1,j+1}, d^1_{i+1,j+2}, d^1_{i+1,j+3}\}$, the computation of the lowpass values $\{s^1_{i+1,j}, s^1_{i+1,j+1}, s^1_{i+1,j+2}, s^1_{i+1,j+3}\}$ should start. For this, access to the previous row to load the four calculated highpass values $\{d^1_{i,j}, d^1_{i,j+1}, d^1_{i,j+2}, d^1_{i,j+3}\}$ is necessary. Consequently, the access pattern needed in vertical filtering is more complex than the access pattern needed in horizontal filtering.

5.6.3 Performance Results

First, the offsetting technique has been applied to the SIMD implementations of all three transforms and, in addition, the lookahead technique to CDF-9/7. The resulting speedups are depicted in Figure 5.24. It can be seen that applying offsetting to the MMX implementation of the $(5, 3)$ lifting transform improves performance significantly. For those image sizes that suffer from 64K aliasing, it improves performance by factors ranging from 1.68 to 6.74. For the Daub-4 and CDF-9/7 transforms, however, the attained speedups are comparatively small. Applying offsetting to Daub-4 provides a speedup of 1.78 for images of size 256×256. For the other image sizes that suffer from 64K aliasing, the speedups are smaller than 1.10. Applying both offsetting and lookahead to CDF-9/7 improves performance by factors ranging from 1.14 to 1.45 when 64K aliasing as well as excessive cache conflict misses occur. The reason for this behavior is that vectorization already (partially) eliminates 64K aliasing. The SSE implementations of the convolutional methods, load 32 bytes of data (half a cache line) into registers before accessing a different cache line that could conflict with the current one. The MMX implementation of the $(5, 3)$ lifting scheme loads 16 bytes of data into registers before accessing a different cache line (see the first two lines of the code given in Figure 5.20). Hence in these implementations 64K aliasing still occurs, but to a lesser extent than in the scalar version.

Figure 5.25 depicts the speedups of the SIMD implementations of horizontal filtering over the corresponding scalar version. The largest speedups are obtained for the $(5, 3)$ lifting scheme. For this transform the speedup ranges from 1.69 to 3.39, while the speedups for Daub-4 and CDF-9/7 range from 1.10 to 1.79 and from 1.25 to 1.44, respectively. There are three main reasons why the speedups for the $(5, 3)$ lifting scheme are higher than for the Daub-4 and CDF-9/7 transforms. First, there are no misaligned memory accesses in the MMX implementation of horizontal filtering using the $(5, 3)$ lifting scheme, while in the SSE implementations there are as shown in Table 5.2. Although SSE permits misaligned memory accesses, they are much slower than aligned memory accesses. Second, there are more MMX execution units than SSE units. This implies that more MMX instructions can be executed in parallel.

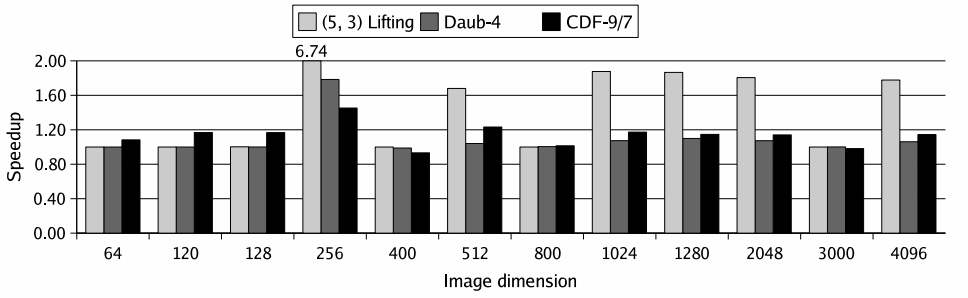


Figure 5.24: *Performance improvements achieved by applying the offsetting technique to the SIMD implementations of all three transforms and, in addition, the lookahead technique to CDF-9/7.*

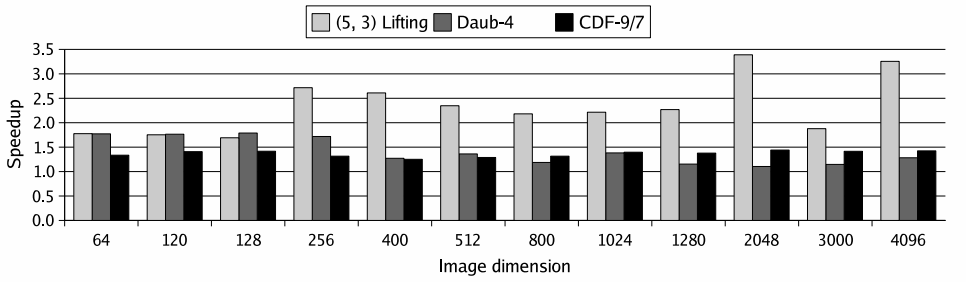


Figure 5.25: *Speedup of the SIMD implementations of horizontal filtering over the scalar versions.*

Third, the MMX implementation of the $(5, 3)$ lifting scheme performs more arithmetic operations per wavelet sample than the SSE implementations of Daub-4 and CDF-9/7. Because SIMD vectorization significantly reduces the CPU component of the execution time, horizontal filtering using Daub-4 and CDF-9/7 has become memory-bound. As Table 5.2 shows the number of load and store instructions in each loop iteration of the horizontal filtering of the $(5, 3)$ lifting scheme is less than the Daub-4 and CDF-9/7 transforms.

Transforms	**Horizontal filtering**		
	# load/store for input/output data	# load for coefficients	# misaligned accesses
$(5, 3)$ Lifting	5	0	0
Daub-4	6	8	2
CDF-9/7	14	19	4

Table 5.2: *Number of load/store instructions and misaligned accesses in each loop iteration of horizontal filtering in the (5, 3) lifting, Daub-4, and CDF-9/7 transforms.*

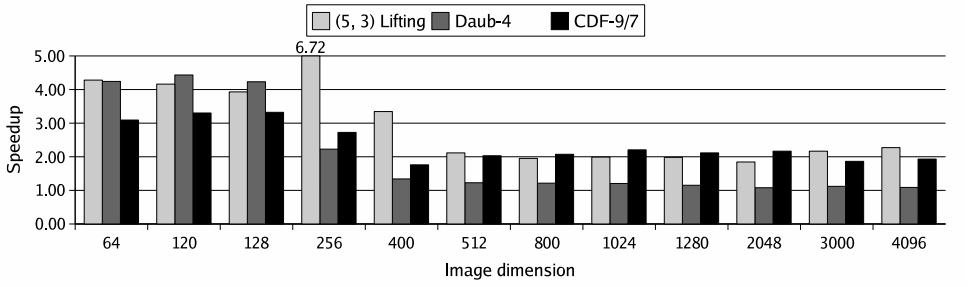


Figure 5.26: *Speedup of the SIMD implementation of vertical filtering over scalar version.*

Two other important observations can be drawn from Figure 5.25. First, the speedups for the $(5, 3)$ lifting scheme are lower for small images ($N \leq 128$) than for larger images, while for Daub-4 the opposite behavior can be observed. Second, since all SIMD implementations perform four operations in one instruction, the expected maximum speedup is 4, but the attained speedups are smaller. The first behavior can be explained as follows. When $N \leq 128$, almost all reads hit the L1 data cache (except for compulsory misses). Hence the speedups obtained for these image sizes are the speedups resulting from SIMD vectorization. When $N > 128$, however, other factors start to play a role. For $(5, 3)$ lifting, the speedup increases because the MMX implementation incurs fewer cache conflicts and hence fewer memory stall cycles than the scalar implementation. For Daub-4, on the other hand, the speedup decreases because this implementation has become memory-bound. The fact that the obtained speedups are smaller than 4 even when $N \leq 128$ is mainly due to the overhead instructions required to vectorize horizontal filtering.

Figure 5.26 depicts the speedups for vertical filtering. As explained in the previous sections, vertical filtering is easier to vectorize and incurs less overhead than horizontal filtering. This explains why the obtained speedups are about the maximum speedup of 4 for images smaller than 256×256. For the lifting and Daub-4 transforms the speedups for these image sizes are even larger than 4 in all but one case due to reduction of loop overhead. For CDF-9/7, the speedups are slightly smaller (around 3.32), because due to the small number of SSE registers, this implementation needs to spill registers to memory. When $N > 256$, however, the obtained speedups are smaller. For the lifting and CDF-9/7 transforms they are around 2 in most cases, but for Daub-4 the average speedup for images larger than 256×256 is only 1.26. Again this should be attributed to a memory bandwidth bottleneck.

Transforms	**Horizontal filtering** # dynamic instructions	**Vertical filtering** # dynamic instructions	**Ratio** (Col. 2 / Col. 3
(5, 3) Lifting	$5 + (5 + \frac{31*M}{8}) * N$	$6 + (5 + \frac{22*M}{4}) * \frac{N}{2}$	1.40
Daub-4	$4 + (4 + \frac{48*M}{8}) * N$	$4 + (4 + \frac{37*M}{4}) * \frac{N}{2}$	1.30

Table 5.3: *The dynamic number of instructions of the SIMD implementations of the horizontal and vertical filtering and also their ratio for different transforms for an $N \times M$ image.*

5.6.4 Discussion

In this section the limitations of the SIMD implementations that restrict the performance and possible solutions are discussed. From the convolution-based transforms, Daub-4 and CDF-9/7, only Daub-4 transform is considered in the following sections. This is because their problems are almost the same.

First, as previously mentioned, the horizontal filtering is not easy to vectorize, while the vertical filtering is. To vectorize the horizontal filtering, overhead instructions are needed. Table 5.3 shows the dynamic number of instructions of the horizontal and vertical filtering and their ratio for the $(5, 3)$ lifting and Daub-4 transforms for an $N \times M$ image. As this table illustrates the number of executed instructions of the vertical filtering is 1.40 and 1.30 times less than the number of executed instructions of horizontal filtering for the $(5, 3)$ lifting scheme and Daub-4 transforms, respectively. Second, in the MMX implementation of the $(5, 3)$ lifting, there is a mismatch between the storage and the computational formats. For instance, about 12.7% of the dynamic number of instructions are needed to convert the pixels to 16-bit values in the MMX implementation. Third, there are some misaligned accesses in the SIMD implementation of horizontal filtering of the Daub-4 transform.

Therefore, some architectural enhancements such as the MAC operation and the MRF technique to aid the efficient SIMD vectorization of horizontal filtering are proposed. A packed MAC instruction is used for Daub-4 transform. A MAC operation can perform a four 32-bit single-precision floating-point multiplication with accumulation. The SSE/SSE2/SSE3 ISAs do not provide the MAC operation for floating-point numbers. Only a packed multiply and add instruction for fixed-point numbers is supported. The proposed MRF technique, which was discussed in the previous chapters, can also be used for all transforms.

In order to evaluate if extended subwords can be used to improve the performance of the $(5, 3)$ lifting transform, the minimum and maximum wavelet coefficient and intermediate result for a 5-level decomposition have been determined. As input, the well-known "Lena" image as well as randomly generated images with 7 to 10 bits per pixel (bpp) have been employed. The results are depicted in Table 5.4. The first column shows the range of the input image pixels, the second and third columns the minimum resp. maximum wavelet coefficient/intermediate result, and the last col-

Image Data Between	Min. value	Max. value	# bits
0, 127	-238	239	9
-128, 127	-472	477	10
0, 255	-475	478	10
-256, 255	-950	955	11
0, 511	-953	957	11
-512, 511	-1904	1915	12
0, 1023	-1906	1916	12

Table 5.4: *Minimum and maximum wavelet coefficients and intermediate results for a 5-level decomposition using the* $(5, 3)$ *lifting scheme for 7- to 10-bit per pixel images.*

umn shows the number of bits required to represent each coefficient and intermediate result. The table shows that a 12-bit data format is sufficient for a 5-level decomposition of images of up to 10 bpp. This means that the extended subwords technique can be employed in order to exploit more DLP in the $(5, 3)$ lifting transform.

The following section describes how the proposed techniques can be used for the efficient SIMD implementation of the DWT.

5.6.5 MAC Operation, Extended Subwords and the MRF

The first proposed technique is to design and to implement a MAC operation. The SSE ISA includes a packed multiply and add (`pmaddwd`) instruction for integers, while it does not provide such instruction for floating-point numbers. As previously mentioned, providing a MAC unit that can perform a 32-bit single-precision floating-point multiplication with accumulation is a good solution to vectorize horizontal filtering. As Figure 5.27 shows, multiplication of coefficients and input samples is possible without using overhead instructions and replication of coefficients. The `pmaddsd` (parallel multiply and add single-precision values to double-precision) performs an SIMD multiply of the four single-precision floating-point values in the source operand by the four single-precision floating-point values in the destination operand. The two high-order words are summed and stored in the upper doubleword of the destination operand, and the two low-order words are summed and stored in the lower doubleword of the destination operand.

The second possible solution for the efficient SIMD implementation of DWT is using extended subwords and the MRF. First, extended subwords and the MRF are used to vectorize the horizontal filtering of the $(5, 3)$ lifting transform. Then, the idea of the MRF is extended to floating-point numbers, and it is used for efficient implementation of the horizontal filtering of the Daub-4 transform.

The extended subwords and the MRF techniques have been discussed in Chapter 3. Both techniques are used to vectorize the horizontal filtering of the $(5, 3)$ lifting, while the extended subword is employed for the implementation of vertical filter-

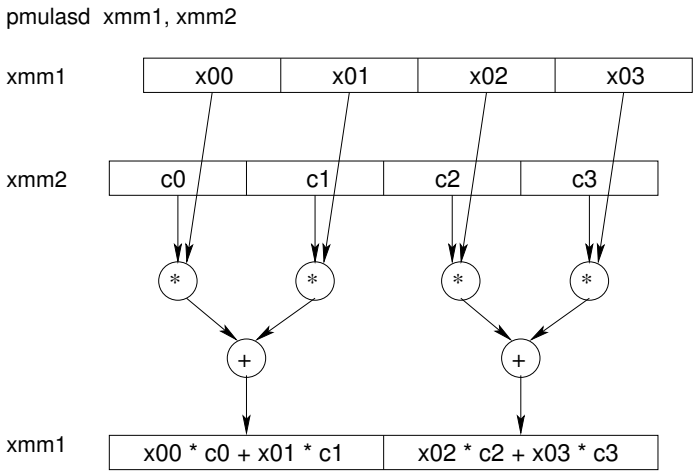


Figure 5.27: *The structure of the* `pmaddsd` *instruction.*

ing. Figure 5.28 illustrates how the MRF technique can be used to reorder the input data. Eight load-column instructions are used to load input sequence into $3mxc0, 3mxc1, ..., 3mxc7$ column registers. To provide correct arrangement of even and odd values according to Equation (5.3) and Equation (5.4), an offset, which is a multiple of 6 bytes for each load-column instruction, is used. After eight load-column instructions, each row register consists of either even ($\{s_i^0\}$) or odd ($\{d_i^0\}$) values. Thereafter, the SIMD ALU instructions can be used to process the row registers. The extended subwords technique provides 8-way parallelism in both horizontal and vertical filtering.

In addition, the idea of the MRF is applied to floating-point numbers using the SSE register file. The SSE register file has eight 128-bit floating-point registers $xmm0, ..., xmm7$. Each register contains four single-precision floating-point values. This register file is modified by the MRF technique. The modified register file has eight row registers, the same as the normal register file, and four column registers $xcmm0, ..., xcmm3$. Figure 5.29 depicts the architecture of the modified register file with 32-bit subwords. This modified register file has two read ports and one write port that has been connected to a 128-bit partitioned floating-point ALU. As this figure shows four registers can be accessed in both horizontal and vertical directions. Data loaded from memory can be written to a row register as well as to a column register. Three 2:1 32-bit multiplexers are needed in each row to select between row-wise and column-wise accesses. Only load-column instructions can write to a column register, the same as in the MMMX register file. Therefore, a transposition of a block stored in the memory can be performed using column-wise load instructions followed by normal store instructions. Four load-column instructions and four normal store instructions are needed to transpose a 4×4 block of single-point floating-point values using the modified register file, while 20 SSE instructions are required as was dis-

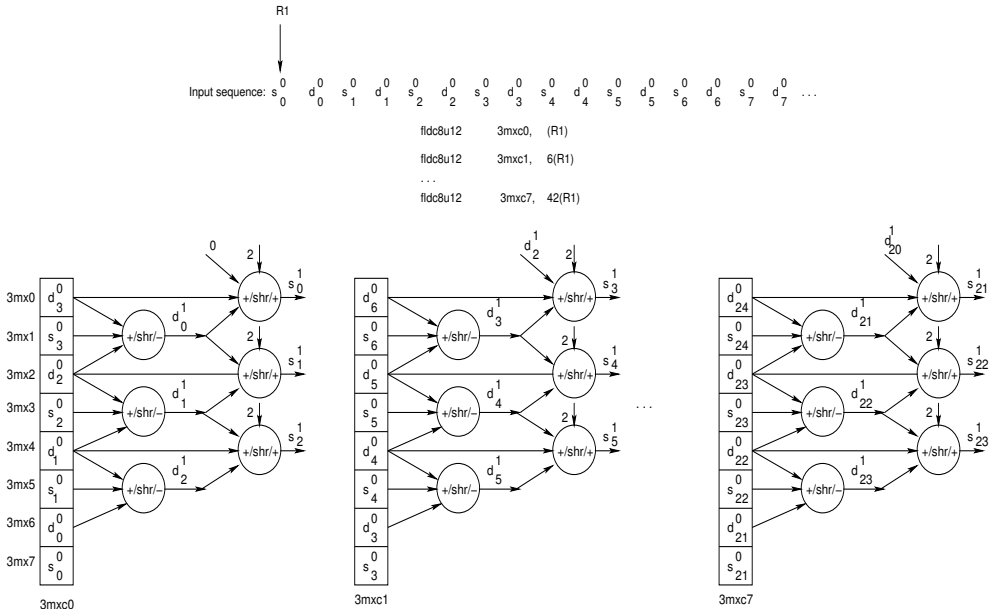


Figure 5.28: *Vectorization of the horizontal filtering of the* $(5, 3)$ *lifting scheme using the matrix register file and extended subwords techniques.*

cussed in Chapter 1. Each load-column instruction can load 128 bits from memory to a column register the same as the normal load and store instructions that are supported by the SSE extensions. For instance, the `movaps` instruction transfers 128 bits of packed data from memory to a row register. The modified register file is used for vectorization of the horizontal filtering of the Daub-4 transform.

5.6.6 Experimental Results

In this section the proposed techniques are evaluated.

Experimental Setup

In order to evaluate the proposed techniques, the `sim-outorder` simulator of the SimpleScalar toolset has been used that has been explained in Chapter 4. The horizontal and vertical filtering of the $(5, 3)$ lifting have been implemented using the MMMX architecture. This architecture has been discussed in Chapter 3. For each horizontal and vertical filtering of the $(5, 3)$ lifting transform, two SIMD implementations using MMX and MMMX instructions have been implemented and simulated. In the MMMX implementation of horizontal filtering, both extended subwords and

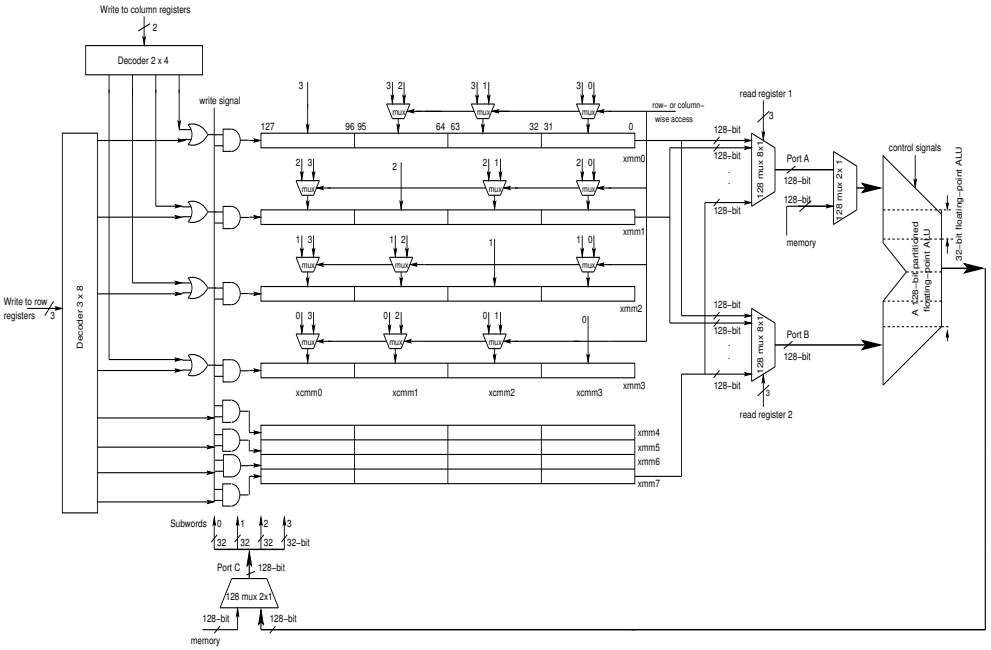


Figure 5.29: *A matrix register file with eight 128-bit registers, two read ports, and one write port. Four registers can be accessed in row-wise as well as column-wise. The modified register file is connected to a 128-bit partitioned floating-point ALU for subword parallel processing.*

the MRF techniques have been used, while in the vertical filtering only the extended subwords technique has been used.

In addition, three SIMD implementations, namely SSE, SSE-MAC, and SSE-MRF for horizontal filtering of the Daub-4 transform have been implemented and simulated. The SSE version is the implementation that has been discussed in Section 5.6.1. In the SSE-MAC implementation the proposed MAC operation has been used, while in the SSE-MRF implementation the modified SSE register file has been employed.

The main parameters of the simulated processor are the same as the parameters that have been discussed in Section 4.3 in Chapter 4. The performance obtained by the MMMX and SSE-MAC as well as SSE-MRF implementations is compared to the performance attained by the MMX and SSE implementations, respectively. In other words, the SIMD implementations of the MMX and SSE are used as reference implementations.

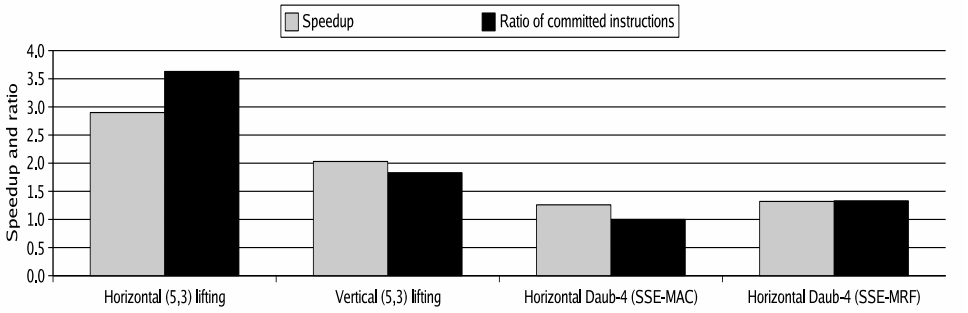


Figure 5.30: *speedups of the MMMX implementation of the horizontal and vertical filtering of the* $(5, 3)$ *lifting, SSE-MAC, and SSE-MRF implementations of the horizontal filtering of the Daub-4 transform over MMX and SSE, respectively, as well as the ratio of committed instructions for an image size of* 480×480 *on a single issue processor.*

Performance Evaluation Results

Figure 5.30 depicts the speedups of the MMMX implementation of the horizontal and vertical filtering of the $(5, 3)$ lifting, SSE-MAC, and SSE-MRF implementations of the horizontal filtering of the Daub-4 transform over MMX and SSE, respectively, as well as the ratio of committed instructions for an image size of 480×480 on a single issue processor. The MMMX implementation of the horizontal filtering of the $(5, 3)$ lifting transform is 2.90 times faster than the MMX implementation, while the MMMX implementation of the vertical filtering is 2.03 times faster than the MMX implementation. The reason why the speedup of the horizontal filtering is larger than the vertical filtering is that in the MMMX implementation of the horizontal filtering both techniques, extended subwords and the MRF, have been used, but in the MMMX implementation of the vertical filtering only extended subwords has been used.

The speedup of SSE-MAC and SSE-MRF is 1.26 and 1.32, respectively. The ratio of committed instructions is 1.00 and 1.33. This means that the MAC operation does not reduce the number of committed instructions. SSE-MAC executes eight SIMD and four scalar instructions in the inner loop to calculate two wavelet coefficients in each iteration. SSE executes 44 SIMD and four scalar instructions in the inner loop to calculate eight wavelet coefficients in each iteration. This means that in the SSE-MAC implementation, four SIMD and two scalar instructions are needed to calculate one wavelet coefficient, while in the SSE implementation, 5.5 SIMD and 0.5 scalar instructions are required to calculate one wavelet coefficient. As a result, SSE-MAC reduces the number of SIMD instructions, while it increases the number of scalar instructions. The latency of the scalar instructions, which are used for incrementing or decrementing index and address values, is less than the latency of the SIMD multiplication instructions. This is the reason why SSE-MAC yields a speedup of 1.26.

Transforms	**Horizontal filtering** # Dynamic instructions	**Vertical filtering** # Dynamic instructions	**Ratio** (Col. 2 / Col. 3
$(5,3)$ Lifting	$6+(4+\frac{51*M}{48})*N$	$5+(4+\frac{20*M}{8})*\frac{N}{2}$	0.85
Daub-4	$4+(4+\frac{36*M}{8})*N$	$4+(4+\frac{37*M}{4})*\frac{N}{2}$	0.97

Table 5.5: *Number of dynamic instructions of the SIMD implementation of both horizontal and vertical filtering of the $(5,3)$ lifting and Daub-4 transforms after using the proposed techniques for an $N \times M$ image.*

In order to reduce the number of the scalar instructions in the SSE-MAC implementation, the inner loop has been unrolled four times. This unrolled version yields a speedup of 1.28 over SSE and its ratio of committed instructions is 1.33.

Table 5.5 shows the dynamic number of instructions for both horizontal and vertical filtering and their ratio for $(5,3)$ lifting and Daub-4 transforms after using the proposed techniques for an $N \times M$ image. The ratio of the number of executed instructions of the horizontal filtering over the executed instructions of the vertical filtering is 0.85 and 0.97 for $(5,3)$ lifting and Daub-4 transforms, respectively. From comparison of Table 5.3 and Table 5.5, it can be understood that after applying the proposed techniques, the number of executed instructions is reduced from 1.40 and 1.30 to 0.85 and 0.97, respectively.

5.7 Conclusions

In this chapter efficient implementation of the two dimensional discrete wavelet transform on SIMD-enhanced GPPs have been discussed. Three different transforms, $(5,3)$ lifting, Daub-4, and CDF-9/7, have been considered. Three issues related to the efficient implementation of the 2D DWT on general-purpose, programmable processors, in particular the Pentium 4, have been addressed. First, a simple and effective technique to improve the cache locality of vertical filtering is the loop interchange. It has been identified, however, that for certain image sizes the resulting implementation suffers from a phenomenon known as 64K aliasing. To avoid this problem, two techniques have been applied: loop fission and offsetting. For image sizes that suffer from 64K aliasing, loop fission provides a speedup that ranges from 1.27 to 3.31, depending on the transform applied , while offsetting achieves speedups between 1.41 and 4.20. Loop fission, however, incurs more loop overhead and, more seriously, destroys the temporal locality between the lowpass and highpass values. Consequently, for image sizes that do not suffer from 64K aliasing it reduces performance by up to 20%. Because offsetting does not destroy the temporal reuse, it is concluded that offsetting is better than loop fission.

It has also been shown that for certain image sizes, vertical filtering (with inter-

changed loops) still generates many more misses than horizontal filtering. On the P4, this happens in particular for the CDF-9/7 transform. The reason is that the filter length exceeds the number of cache ways. Because of this, conflicts occur if the input coefficients needed to compute one output coefficient map to the same cache set. To avoid these conflicts two techniques have been applied: associativity-conscious loop fission and lookahead. For image sizes that experience many cache conflict misses ACLF improves performance by a factor that ranges from 1.59 to 1.80, while the lookahead technique provides a speedup between 1.71 and 1.99. For other image sizes, both schemes generally decrease performance slightly, due to the increased loop overhead. Except for two image sizes, the lookahead technique performs slightly better than ACLF, because it incurs less loop overhead. Both schemes are general because they can also be applied to other cache organizations and/or filter lengths. To show this, results for the P3 and Opteron have also been presented.

To further enhance the performance of the 2D DWT, the SIMD instructions provided by most general-purpose programmable processors must be exploited. MMX implementations of the lifting transform and SSE implementations of the convolutional transforms have been presented . While vertical filtering is relatively straightforward to vectorize, horizontal filtering requires to rearrange the elements (sub-words) within a register. Mainly because of this overhead, the speedups obtained for horizontal filtering are relatively small, ranging from 1.69 to 3.39 for the lifting transform and from 1.10 to 1.79 for the convolutional transforms. Because vertical filtering does not incur this overhead, the speedups approach the ideal speedup of 4 when most reads hit the L1 data cache. For larger images, however, the obtained speedups are smaller, because the computation becomes memory-bound. This is especially the case for the Daub-4 transform which has a smaller computation-to-communication ratio than the other two transforms.

Amongst others, this work has shown that it is difficult to obtain a single implementation of the 2D DWT that works well for all image sizes, because most techniques incur some overhead. This indicates that in order to obtain the fastest implementation of this important kernel, a parameterizable implementation is needed that takes into account factors such as the cache organization of the target processor, the image size, the filter length, etc. Specifically, focusing on the cache conflict problem, if the cache organization, image size, and filter length are such that the number of input blocks needed to compute one output block exceeds the number of cache ways, then the lookahead technique should be applied. Otherwise, the reference implementation should be called.

Additionally, the proposed extended subwords, the MRF, and MAC operation have been used to improve the performance of the SIMD implementations. The extended subwords and the MRF techniques have been used in the horizontal filtering of the $(5, 3)$ lifting transform, while only the extended subwords technique has been em-

ployed in the vertical filtering. These techniques provided speedups of 2.90 and 2.03 for horizontal and vertical filtering, respectively. The register file of the SSE extension has been modified by the idea of the matrix register file. The SSE modified register file improves the performance of the horizontal filtering of the Daub-4 transform by a factor of 1.32, while the MAC operation yields a speedup of 1.26.

Chapter 6

Conclusions and Future Work

In this dissertation, performance bottlenecks of multimedia extensions have been addressed. Some of these bottlenecks are as follows. First, in many media kernels, there is a mismatch between the computational format and the storage format in the processing of multimedia applications. This is because the precision of intermediate results is usually larger than the storage format. Therefore, data type conversion (unpack) instructions are required before the operations are performed and the results also have to be packed before they can be stored back to memory. Second, existing SIMD computational instructions cannot efficiently exploit DLP of the 2D multimedia data. The main reason is that the 2D multimedia algorithms process the input pixels in both the horizontal and vertical directions, while the media register file is just accessed in the horizontal direction. Consequently, data rearrangement instructions are needed to efficiently implement 2D media kernels. As a solution, a novel SIMD extension called MMMX has been proposed. The MMMX extension enhances the MMX architecture with the extended subwords and matrix register file techniques. While, these techniques have been applied to the MMX architecture, they could also be applied to other SIMD architectures. The proposed architecture has been evaluated using the `sim-outorder` simulator of the SimpleScalar toolset on several MMAs and kernels. In addition, three issues related to the efficient implementation of the 2D DWT on SIMD-enhanced GPPs have been addressed. These issues are 64K aliasing, cache conflict misses, and SIMD vectorization.

This chapter summarizes the contents of the dissertation, outlines its contributions, and proposes future research directions. It is organized in three separate sections. Section 6.1 presents a summary of the thesis. Section 6.2 highlights the main contributions of this work and finally, some open research directions are given in Section 6.3.

6.1 Summary

The work presented in this thesis can be summarized as follows.

In Chapter 2, the background information about data type conversion, data permutation instructions, SIMD vectorization, and cache optimization were described. First, data type conversion instructions were discussed. The proposed techniques to avoid these instructions were also described. For example, one way to avoid data type conversion is by using wider media registers that has been considered in the design of some DSPs. Second, three different techniques for data rearrangement were explained. Data reordering can be performed using either explicit instructions, memory operations, or via register file structures. There are some explicit instructions such as the packed shuffle word instruction in the SSE extension for data reorganization. Strided load and store instructions can also reorder the input data. The structure of the media register file can also be used to reorganize data. Third, some SIMD vectorization techniques were described. This technique improves the performance of MMAs. This is because SIMD vectorization transforms sequential codes to parallel codes. The vectorized programs can be executed on GPPs enhanced with SIMD extensions. GPPs typically employ cache memory. Hence, some cache optimization techniques were also discussed at the end of Chapter 2.

Chapter 3 described the Modified MMX (MMMX) architecture which features the extended subwords and matrix register file techniques. The MMMX architecture is an SIMD extension for the multimedia domain. It focuses on maintaining programmability and accelerates MMAs by exploiting DLP. The MMMX architecture alleviates the data type conversion and data rearrangement instructions. Its instructions are applicable in multiple domain. It did not consider an ISA that is application specific. First, extended subwords and the matrix register file have been discussed in detail. Second, novel SIMD load/store instructions and SIMD ALU operations were presented. Third, the main differences between the MMMX and MMX architectures were shown. Thereafter, it was discussed how the MMMX architecture can reduce the dynamic number of instructions and improve performance significantly compared to MMX architecture. Finally, the hardware cost of the MMMX architecture was evaluated.

Chapter 4 presented the performance evaluation of the proposed architecture. Its performance was compared to the performance of the MMX architecture at the kernel-, image-, and application-level. Several important MMAs were selected. The most time consuming kernels were implemented using the MMMX and MMX/SSE architectures. In order to obtain the application-level speedup, the SIMD implementations were replaced in the original scalar version of the applications.

Chapter 5 discussed one kernel, the discrete wavelet transform, in more detail. The

DWT which is used in the JPEG2000 standard processes whole image size, while the DCT which is used in the MPEG standards processes 8×8 blocks of pixels. Three issues related to the efficient computation of the 2D DWT on general-purpose processors, in particular the Pentium 4, have been discussed. These issues are 64K aliasing, cache conflict misses, and SIMD vectorization. 64K aliasing is a Pentium 4 phenomenon that can degrade performance by an order of magnitude. In addition, there are many cache conflict misses in the straightforward implementation of vertical filtering, if the filter length exceeds the number of cache ways. Two techniques were proposed to avoid 64K aliasing as well as two techniques to alleviate cache conflict misses. Additionally, the performance of the 2D DWT was improved by exploiting the DLP using the SIMD instructions supported by most GPPs. Finally, some techniques were proposed and evaluated to improve the performance of SIMD implementations of the 2D DWT, such as multiply-accumulate operation for single-precision floating-point data types and the use of extended subwords and the MRF.

6.2 Major Contributions

The main contributions of this thesis are highlighted below.

- Detailed examination of the limitations of MMX data movement operations led to the proposal of the matrix register file, which supports efficient matrix transpose operations. The matrix register file allows both row-wise as well as column-wise accesses to the register file. This technique reduces the amount of explicit data reorganization instructions required by many SIMD calculations.
- Detailed examination of the amount of data type conversion in MMX and the cause of these conversions led to the proposal of extended subwords. The extended subwords technique uses four extra bits for each byte of the media register file. Load instructions implicitly unpack data from the storage format to the computation format, and the store instructions implicitly pack and saturate data from the computation format to the storage format. This design eliminates the need for explicit pack and unpack instructions.
- A novel SIMD extension called MMMX has been proposed. The MMMX extension enhances the MMX architecture with the extended subwords and matrix register file techniques. The MMMX architecture reduces the total number of instructions much more than MMX, MMX enhanced with extended subwords, and MMX enhanced with the MRF. MMMX achieves an image-level speedup of up to 3.24 over MMX. In addition, a set of new general-purpose SIMD instructions have been proposed for multimedia computing.
- An initial examination of how many registers are needed to efficiently process

multimedia workloads on an SIMD processor. For example, MMMX with 13 extra registers yields speedups ranging from 1.37 to 3.64 over MMX.

- In order to develop high-performance implementations of the 2D DWT on general-purpose processors in general and the P4 in particular, three issues were addressed.
  - The P4 suffers from a problem known as 64K aliasing, which can degrade performance by an order of magnitude. Two techniques were proposed and evaluated to avoid 64K aliasing. The first technique is loop fission that split the inner loop so that the lowpass and highpass values are calculated in separate loops. The loop fission technique provides a speedup of up to 3.31, but for image sizes that do not suffer from 64K aliasing it reduces performance by up to 20%. The second technique is to offset the memory address of the highpass value by one or two rows depending on the transform. This offsetting technique achieves a speedup of up to 4.20 and incurs no performance penalty for image sizes that do not suffer from 64K aliasing.
  - There are many cache conflict misses in the implementation of vertical filtering. Two techniques, namely associativity-conscious loop fission and lookahead, were proposed to reduce the number of conflict misses. The associativity-conscious loop fission splits the loop that computes one row of wavelet output into multiple loops so that each loop accesses at most n rows, where n is the number of ways. Each loop computes the partial results that can be computed by accessing the first n rows of input data. The remaining loops add their results to these partial results. The lookahead technique processes the rows in a skewed manner. There is only one loop, as in the original algorithm. The former technique improves performance by up to 80% and the latter by up to 99%. For image sizes that do not generate many conflict misses, both techniques slightly decrease performance due to the overhead introduced by applying these techniques.
  - High-performance implementations of the 2D DWT must exploit the DLP using the SIMD instructions supported by most GPPs. It was described how the 2D DWT can be vectorized using MMX and SSE instructions. Vertical filtering is relatively straightforward to vectorize. Horizontal filtering is more difficult to vectorize, however, since it requires the data to be reorganized. The maximum speedups of the SIMD implementations of horizontal and vertical filtering over the corresponding scalar versions are 3.39 and 6.72, respectively. In addition, some techniques were proposed and evaluated to improve the performance of SIMD implementations of the 2D DWT, such as a MAC operation for

single-precision floating-point data types and the use of extended subwords and the MRF. The MAC operation achieves a speedup of up to 1.26 and extended subwords and the MRF yield a speedup of up to 2.90.

6.3 Future Proposed Research Directions

In spite of a comprehensive description and experimental validation of the SIMD architectures that have been provided in this dissertation, there exist a number of interesting issues which can be addressed in future. The following directions for future improvements are proposed.

- One item related to the performance of multimedia extensions is the memory alignment. Memory references are more efficient when accessing aligned regions. A memory operation is aligned if its address is a multiple of the data width. In other words, an n-byte transfer must be set on an n-byte boundary. In most SIMD architectures, unaligned memory accesses have a large performance penalty or are even disallowed. For example, our results show that for the addition of two arrays of size 1024×1024, whose addresses are either aligned or unaligned, the aligned implementation is 1.47 times faster than the unaligned implementation using SSE instructions. An interesting research direction could be to investigate hardware techniques to mitigate the effects of misaligned accesses.

- A complete investigation of register utilization in multimedia applications has not been considered in this thesis. If the media registers are effectively used, they can reduce memory traffic by removing load and store instructions and decrease power consumption by eliminating operations on the memory hierarchy. In addition, the SIMD implementations of multimedia kernels can use a lot of wide registers that cause a large increase in the number of registers used. A larger register file allows coefficient values to be allocated to media registers. The benefit is that the value is loaded once at the start of a function and stored once at the end of the execution. However, determining the exact number of registers is a future work to still consider. For example, the Cell processor provides 128 128-bit registers and the register file of the TM3270 media processor consists of 128 32-bit registers. Are 128 registers enough? Both VMX and AltiVec extensions need a number of registers to keep the data for rearrangement instructions, while the number of registers can be reduced by some micro-architectural modifications such as the MRF technique.

- Another challenge in SIMD architectures is to determine the number of subwords that can be processed simultaneously. If the number of subwords is too less, it limits the ability to exploit DLP, whereas if the number of subwords

is too large, it can reduce performance improvement because of insufficient DLP and using many more overhead instructions. This means that there is no guarantee to obtain a larger speedup with wider vectors compared to shorter vectors.

- The issue of the compiler support for SIMD vectorization has also not been considered in this work. This issue is very important because writing code for SIMD processors using assembly language usually macro-like intrinsics is more tedious and error prone than compilers that automatically identify vectorizable parts of the program and generate the appropriate SIMD instructions. Currently available compilers cannot exploit efficiently SIMD vectorization automatically.
- The MMMX architecture improves the performance of multimedia kernels, whereas its power dissipation has not been considered. As a result, the power consumption of the proposed techniques can be the future research. For example, considering the proposed techniques in the design of low-power media processors.
- An interesting issue to investigate is how much multimedia applications can benefit from multi-core architectures.

Bibliography

[1] D. M. Adams and F. Kossentini. “Reversible Integer-to-Integer Wavelet Transforms for Image Compression: Performance Evaluation and Analysis”. *IEEE Trans. on Image Processing*, 9(6):1010–1024, June 2000.

[2] Advanced Micro Devices Inc. *“3DNow Technology Manual”*, 2000.

[3] Y. Andreopoulos, K. Masselos, P. Schelkens, G. Lafruit, and J. Cornelis. “Cache Misses and Energy Dissipation Results for JPEG-2000 Filtering”. In *Proc. 14th IEEE Int. Conf. on Digital Signal Processing*, pages 201–209, 2002.

[4] Y. Andreopoulos, P. Schelkens, and J. Cornelis. “Analysis of Wavelet Transform Implementations for Image and Texture Coding Applications in Programmable Platforms”. In *Proc. IEEE Signal Processing Systems*, pages 273–284, 2001.

[5] Y. Andreopoulos, P. Schelkens, G. Lafruit, K. Masselos, and J. Cornelis. “High-Level Cache Modeling for 2-D Discrete Wavelet Transform Implementations”. *Journal of VLSI Signal Processing*, 34:209–226, 2003.

[6] Y. Andreopoulos, N. D. Zervas, G. Lafruit, P. Schelkens, T. Stouraitis, C. E. Goutis, and J. Cornelis. “A Local Wavelet Transform Implementation Versus an Optimal Row-Column Algorithm for the 2D Multilevel Decomposition”. In *Proc. IEEE Int. Conf. on Image Processing*, volume 3, pages 330–333, 2001.

[7] N. Aron, H. Wijaya, A. Singh, and V. Malhotra. “Study of Multimedia Application Characteristics”. http://www.stanford.edu/class/ee392c/handouts/apps/media-long.pdf, 2003.

[8] R. Asokan and S. Nazareth. “Processor Architectures for Multimedia”. In *Proc. 14th Annual Workshop on Architecture and System Design*, pages 589–

594, November 2001.

[9] T. Austin, E. Larson, and D. Ernst. "SimpleScalar: An Infrastructure for Computer System Modeling". *IEEE Computer*, 35(2):59–67, February 2002.

[10] P. Bannon and Y. Saito. "The Alpha 21164PC Microprocessor". In *IEEE Proc. Compcon 97*, pages 20–27, February 1997.

[11] M. Baron. "Cortex-A8: High Speed, Low Power". *Microprocessor Report*, 11(14):1–6, 2005.

[12] M. Bartkowiak. "Optimizations of Color Transformation for Real Time Video Decoding". In *Proc. EURASIP Conf. on Digital Signal Processing for Multimedia Communications and Services*, September 2001.

[13] F. Bensaali and A. Amira. "Accelerating Colour Space Conversion on Reconfigurable Hardware". *Image and Vision Computing*, 23:935–942, 2005.

[14] M. Berekovic, H. J. Stolberg, M. B. Kulaczewski, and P. Pirsch. "Instruction Set Extensions for MPEG-4 Video". *Journal of VLSI Signal Processing*, 23:27–49, 1999.

[15] G. Bernabe, J. M. Garcia, and J. Gonzales. "Reducing 3D Wavelet Transform Execution Time Through the Streaming SIMD Extensions". In *Proc. 11th Euromicro Conf. on Parallel Distributed and Network based Processing*, February 2003.

[16] C. Bobda. *"Introduction to Reconfigurable Computing Architectures, Algorithms and Applications"*. Springer, 2007.

[17] R. E. Bryant and D. R. O'Hallaron. *"Computer Systems: A Programmer's Perspective"*. Prentice Hall, 2003.

[18] A. Chamas, A. Dalal, P. Dedood, P. Ferolito, B. Frederick, O. Geva, D. Greenhill, H. Hingarh, J. Kaku, L. Kohn, L. Lev, M. Levitt, R. Melanson, S. Mitra, R. Sundar, M. Tamjidi, P. Wang, D. Wendell, R. Yu, and G. Zyner. "A 64-b Microprocessor with Multimedia Support ". In *Proc. IEEE Conf. on Solid State Circuits*, pages 178–179, February 1995.

[19] H. C. Chang, L. G. Chen, M. Y. Hsu, and Y. C. Chang. "Performance Analysis and Architecture Evaluation of MPEG-4 Video Codec System". In *Proc. IEEE Int. Symp. on Cicuits and Systems*, volume 2, pages 449–452, May 2000.

[20] H. C. Chang, Y. C. Wang, M. Y. Hsu, and L. G. Chen. "Efficient Algorithms and Architectures for MPEG-4 Object-Based Vodeo Coding". In *IEEE Workshop on Signal Processing Systems*, pages 13–22, 2000.

[21] S. Chatterjee and C. D. Brooks. "Cache Efficient Wavelet Lifting in JPEG

2000". In *Proc. IEEE Int. Conf. on Multimedia*, pages 797–800, August 2002.

[22] S. Chatterji, M. Narayanan, J. Duell, and L. Oliker. "Performance Evaluation of Two Emerging Media Processors: VIRAM and Imagine". In *Proc. 14th IEEE Int. Symp. on Parallel and Distributed Processing*, April 2003.

[23] D. Chaver, M. Prieto, L. Pinuel, and F. Tirado. "Parallel Wavelet Transform for Large Scale Image Processing". In *Proc. IEEE Int. Symp. on Parallel and Distributed Processing*, pages 4–9, April 2002.

[24] D. Chaver, C. Tenllado, L. Pinuel, M. Prieto, and F. Tirado. "2-D Wavelet Transform Enhancement on General-Purpose Microprocessors: Memory Hierarchy and SIMD Parallelism Exploitation". In *Proc. Int. Conf. on the High Performance Computing*, December 2002.

[25] D. Chaver, C. Tenllado, L. Pinuel, M. Prieto, and F. Tirado. "Vectorization of the 2D Wavelet Lifting Transform Using SIMD Extensions". In *Proc. 17th IEEE Int. Symp. on Parallel and Distributed Image Processing and Multimedia*, 2003.

[26] W. Chen, H. J. Reekie, S. Bhave, and E. A. Lee. "Native Signal Processing on the Ultrasparc in the Ptolemy Environment". In *IEEE Conf. on Signals Systems and Computers*, volume 2, pages 1368–1372, November 1996.

[27] D. Cheresiz, B. Juurlink, S. Vassiliadis, and H. A. G. Wijshoff. "Performance Scalability of Multimedia Instruction Set Extensions". In *Proc. Euro-Par Parallel processing*, pages 849–861, 2002.

[28] D. Cheresiz, B. Juurlink, S. Vassiliadis, and H. A. G. Wijshoff. "The CSI Multimedia Architecture". *IEEE Trans. on VLSI Systems*, 13(1):1–13, January 2005.

[29] B. D. Choi, K. S. Choi, M. C. Hwang, J. K. Cho, and S. J. Ko. "Real-time DSP Implementation of Motion-JPEG2000 Using Overlapped Block Transferring and Parallel-Pass Methods". *Real-Time Imaging*, 10:277–284, 2004.

[30] C. Chrysafis and A. Ortega. "Line-Based, Reduced Memory, Wavelet Image Compression". *IEEE Trans. on Image Processing*, 9(3):378–389, March 2000.

[31] A. Cohen, I. Daubechies, and J. C. F. Eauveau. "Biorthogonal Bases of Compactly Supported Wavelets". *Communications on Pure and Appl. Math.*, 45(5):485–560, June 1992.

[32] J. Corbal, R. Espasa, and M. Valero. "MOM: a Matrix SIMD Instruction Set Architecture for Multimedia Applications". In *Proc. IEEE/ACM Conf. on Supercomputing*, pages 1–10, November 1999.

[33] J. Corbal, M. Valero, and R. Espasa. "Exploiting a New Level of DLP in

Multimedia Applications". In *Proc. Int. Symp. on Microarchitecture*, 1999.

[34] Intel Corporation. "An Efficient Vector/Matrix Multiply Routine using MMX Technology". Technical report, Intel Developer Services, 2004.

[35] P. P. Dang and P. M. Chau. "Reduce Complexity Hardware Implementation of Discrete Wavelet Transform for JPEG 2000 Standard". In *Proc. IEEE Int. Conf. on Multimedia and Expo*, pages 321–324, August 2002.

[36] A. Dasu and S. Panchanathan. "A Survey of Media Processing Approaches". *IEEE Trans. on Circuits and Systems for Video Technology*, 12(8):633–644, August 2002.

[37] A. Dasu and S. Panchanathan. "Reconfigurable Media Processing". *Parallel Computing*, 28(7):1111–1139, August 2002.

[38] I. Daubechies and W. Sweldens. "Factoring Wavelet Transforms into Lifting Steps". *Journal of Fourier Analysis and Applications*, 4(3):247–269, 1998.

[39] S. Deb. *"Video Data Management and Information Retrieval"*. IRM Press, 2005.

[40] J. H. Derby and J. H. Moreno. "A High-Performance Embedded DSP Core With Novel SIMD Features". In *Proc. IEEE Int. Conf. on Acoustics Speech and Signal Processing*, pages 301–304, April 2003.

[41] K. Diefendorff and P. K. Dubey. "How Multimedia Workloads Will Change Processor Design". *IEEE Computer*, 30(9):43–45, September 1997.

[42] K. Diefendorff, P. K. Dubey, R. Hochsprung, and H. Scales. "AltiVec Extension to PowerPC Accelerates Media Processing". *IEEE Micro*, 20(2):85–95, March-April 2000.

[43] J. T. J. Van Eijndhoven, F. W. Sijstermans, K. A. Vissers, E. J. D. Pol, and M. J. A. Tromp. "Trimedia CPU64 Architecture". In *Proc. Int. Conf. on Computer Design*, pages 1–7, October 1999.

[44] R. Espasa and M. Valero. "Exploiting Instruction- and Data-Level Parallelism". *IEEE Micro*, 17(5):20–27, September-October 1997.

[45] P. Faraboschi, G. Desoli, and J. A. Fisher. "The Latest Word in Digital and Media Processing". *IEEE Signal Processing Magazine*, pages 59–85, March 1998.

[46] M. Feil, R. Kutil, P. Meerwald, and A. Uhl. "Wavelet Image and Video Coding on Parallel Architectures". In *Proc. 2nd IEEE - EURASIP Symp. on Image and Signal Processing and Analysis*, 2001.

[47] F. Ferrand. "Optimization and Code Parallelization for Processors with Multi-

media SIMD Instructions". Master's thesis, ENST Bretagne, 2003.

[48] M. Ferretti and D. Rizzo. "A Parallel Architecture for the 2-D Discrete Wavelet Transform with Integer Lifting Scheme". *Journal of VLSI Signal Processing*, 28:165–185, 2001.

[49] 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, T. Vandung, and E. Iwata. "The Microarchitecture of the Synergistic Processor for a Cell Processor". *IEEE Journal of Solid-State Circuits*, 41:63–70, January 2006.

[50] J. Fritts. *"Architecture and Compiler Design Issues in Programmable Media Processors"*. PhD thesis, University of Princeton, 2000.

[51] J. Fritts and W. Wolf. "Instruction Fetch Characteristics of Media Processing". In *Proc. of SPIE Photonics West Media Processing*, 2002.

[52] J. Fritts, W. Wolf, and B. Liu. "Understanding Multimedia Application Characteristics for Designing Programmable Media Processors". In *Proc. SPIE Photonics West Media Processors*, pages 2–13, January 1999.

[53] B. Furht. "Processor Architectures for Multimedia: A Survey". In *Proc. Conf. on Multimedia Modeling*, pages 89–109, November 1997.

[54] M. Ghanbari, D. Crawford, M. Fleury, E. Khan, J. Woods, H. Lu, and R. Razavi. "Future Performance of Video Codecs". Technical Report SES2006-7-13, Department of Electonic System Engineering University of Essex, November 2006.

[55] R. C. Gonzalez and R. E. Woods. *"Digital Image Processing"*. Pearson/Prentice Hall, 3rd edition, 2008.

[56] J. Goodacre and A. N. Sloss. "Parallelism and the ARM Instruction Set Architecture". *IEEE Computer*, 38(7):42–50, 2005.

[57] L. Gwennap. "Digital, MIPS Add Multimedia Extensions". *Microprocessor Report*, 10(15):24–28, November 1996.

[58] B. Hanounik and X. Hu. "Linear-Time Matrix Transpose Algorithms Using Vector Register File with Diagonal Registers". In *Proc. 15th Int. Conf. on Parallel and Distributed Processing*, April 2001.

[59] D. He and W. Zhang. "The Parallel Algorithm of 2-D Discrete Wavelet Transform". In *Proc. 4th IEEE Int. Conf. on Parallel and Distributed Computing Applications and Technologies*, pages 738–741, August 2003.

[60] Hu. Yu Hen. *"Programmable Digital Signal Processors : Architecture, Programming, and Applications"*. New York Dekker, 2002.

[61] J. L. Hennessy and D. A. Patterson. *"Computer Architecture: A Quantitative Approach"*. Morgan Kaufmann, 2002. 3rd edition.

[62] H. P. Hofstee. "Power Efficient Processor Architecture and the Cell Processor". In *Proc. 11th IEEE Int. Symp. on High-Performance Computer Architectur*, pages 258–262, February 2005.

[63] J. Y. F. Hsieh, A. Avoird, R. P. Kleihorst, and T. H. Y. Meng. "Transpose Memory for Video Rate JPEG Compression on Highly Parallel Single-chip Digital CMOS Imager". In *Proc. IEEE Int. Conf. on Image Processing*, volume 3, September 2000.

[64] http://www.virtualdub.org/blog/pivot/entry.php?id=18. *Does Hyperthreading Technology speed up VirtualDub*.

[65] L. Huang, M. Lai, K. Dai, H. Yue, and L. Shen. "Hardware Support for Arithmetic Units of Processor with Multimedia Extension". In *Proc. IEEE Int. Conf. on Multimedia and Ubiquitous Engineering*, pages 633–637, April 2007.

[66] H. C. Hunter and J. H. Moreno. "A New Look at Exploiting Data Parallelism in Embedded Systems". In *Proc. IEEE Int. Conf. on Compilers Architectures and Synthesis for Embedded Systems*, pages 159–169, 2003.

[67] IBM. *"Synergistic Processor Unit Instruction Set Architecture"*, January 2007. Version 1.2.

[68] Intel. "Using MMX Instructions to Compute the 2x2 Haar Transform". Technical report, Intel Developer Services, 2004.

[69] Intel Corporation. *"IA-32 Intel Architecture Optimization"*, 2004. Order Number: 248966-011.

[70] Intel Corporation. *"The IA-32 Intel Architecture Software Developer's Manual Volume 3 System Programming Guide"*, 2004. Order Number: 253668.

[71] N. Jayasena, M. Erez, J. Ahn, and W. Dally. "Stream Register Files With Indexed Access". In *Proc. 10th IEEE Int. Symp. on High Performance Computer Architecture*, February 2004.

[72] M. D. Jennings and T. M. Conte. "Subword Extensions for Video Processing on Mobile Systems". *IEEE Concurrency*, 6(3):13–16, July-September 1998.

[73] Y. Jung, S. G. Berg, D. Kim, and Y. Kim. " A Register File with Transposed Access Mode". In *Proc. Int. Conf. on Computer Design*, pages 559–560, September 2000.

[74] B. Juurlink, D. Borodin, R. J. Meeuws, G. T. Aalbers, and H. Leisink. "The SimpleScalar Instruction Tool (SSIT) and the SimpleScalar Architecture Tool (SSAT)". Available via `http://ce.et.tudelft.nl/~shahbahrami/`, 2007.

[75] B. Khailany, W. J. Dally, S. Rixner, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, and A. Chang. "Imagine: Media Processing with Streams". *IEEE Micro*, 21(2):35–46, March-April 2001.

[76] H. Komi and A. Ortega. "Analysis of Cache Efficiency in 2D Wavelet Transform". In *Proc. IEEE Int. Conf. on Multimedia and Expo*, pages 465–468, 2001.

[77] K. Konstantinides. "VLIW Architectures for Media Processing". *IEEE Signal Processing Magazine*, 15(2):16–19, March 1998.

[78] C. Kozyrakis, J. Gebis, D. Martin, S. Williams, I. Mavroidis, S. Pope, D. Jones, D. Patterson, and K. Yelick. "Vector IRAM: A Media-oriented Vector Processor with Embedded DRAM". In *Proc. 12th Int. Conf. on Hot Chips*, August 2000.

[79] C. Kozyrakis and D. Patterson. "Vector Vs. Superscalar and VLIW Architectures for Embedded Multimedia Benchmarks". In *Proc. 35th Int. Symp. on Microarchitecture*, November 2002.

[80] C. E. Kozyrakis and D. Patterson. "A New Direction for Computer Architecture Research". *IEEE Computer*, 31(11):24–32, November 1998.

[81] A. Kudriavtsev and P. Kogge. "Generation of Permutations for SIMD Processors". In *Proc. ACM Conf. on Language, Compiler and Tool Support for Embedded Systems*, volume 40/7, pages 147–156, July 2005.

[82] P. Kuhn. *"Algorithms, Complexity Analysis and VLSI Architectures for MPEG-4 Motion Estimation"*. Kluwer Academic Publishers, 1999.

[83] I. Kuroda and T. Nishitani. "Multimedia Processors". *Proc. IEEE*, 86(6):1203–1221, June 1998.

[84] R. Kutil. "A Single-Loop Approach to SIMD Parallelization of 2D Wavelet Lifting". In *Proc. 14th Euromicro Int. Conf. on Parallel, Distributed, and Netwrok-Based Processing*, pages 413–420, 2006.

[85] S. Larsen and S. Amarasinghe. "Exploiting Superword Level Parallelism with Multimedia Instruction Sets". In *Proc. ACM Conf. on Programming Language Design and Implementation*, pages 145–156, 2000.

[86] A. J. T. Lee, R. W. Hong, and M. F. Chang. "An Approach to Content-based Video Retrieval". In *Proc. IEEE Int. Conf. on Multimedia and Expo*, volume 1,

pages 273–276, June 2004.

[87] C. G. Lee and D. J. DeVries. "Initial Results on the Performance and Cost of Vector Microprocessors". In *Proc. 13th Annual IEEE/ACM Int. Symp. on Microarchitecture*, pages 171–182, December 1997.

[88] R. B. Lee. "Accelerating Multimedia With Enhanced Microprocessors". *IEEE Micro*, 15(2):22–32, April 1995.

[89] R. B. Lee. "Subword Parallelism with MAX-2". *IEEE Micro*, 16(4):51–59, August 1996.

[90] R. B. Lee. "Multimedia Extensions for General-Purpose Processors". In *Proc. IEEE Workshop on Signal Processing Systems*, pages 9–23, November 1997.

[91] R. B. Lee. "Efficiency of MicroSIMD Architectures and Index-Mapped Data for Media Processors". In *Proc. Symp. on Media Processors IS&T/SPIE and Electric Imaging*, pages 34–46, January 1999.

[92] R. B. Lee. "Subword Permutation Instructions for Two-Dimensional Multimedia Processing in MicroSIMD Architectures". In *Proc. IEEE Int. Conf. on Application-specific Systems Architectures and Processors*, pages 9–23, July 2000.

[93] R. B. Lee and M. D. Smith. "Media Processing: A New Design Target. *IEEE Micro*, 16(4):6–9, August 1996.

[94] Y. D. Lee, B. D.Choi, J. K. Cho, and S. J. Ko. "Cache Management for Wavelet Lifting in JPEG 2000 Running on DSP". *Electronics Letters*, 40(6), March 2004.

[95] H. Liao and A. Wolfe. "Available Parallelism in Video Applications". In *Proc. 13th IEEE.ACM Int. Symp. on Microarchitecture*, pages 321–329, December 1997.

[96] C. Loeffler, A. Ligtenberg, and G. S. Moschytz. "Practical Fast 1-D DCT Algorithms With 11 Multiplications". In *Proc. Int. Conf. on Acoustical and Speech and Signal Processing*, pages 988–991, May 1989.

[97] A. A. Lopez-Estrada. "Reduction of Address Aliasing". Technical Report Pub. No.: US 2005/0188172 A1, August 2005.

[98] H. Lütkepohl. *"Handbook of matrices"*. John Wiley & Sons, Chichester, 1996.

[99] P. Meerwald, R. Norcen, and A. Uhl. "Cache Issues with JPEG2000 Wavelet Lifting". In *Proc. of Visual Communications and Image Processing*, January 2002.

[100] J. H. Moreno, V. Zyuban, U. Shvadron, F. D. Neeser, J. H. Derby, M. S. Ware, K. Kailas, A. Zaks, A. Geva, S. Ben-David, S. W. Asaad, T. W. Fox, D. Littrell, M. Biberstein, D. Naishlos, and H. Hunter. "An Innovative Low-power High-performance Programmable Signal Processor for Digital Communications". *IBM Journal of Research and Development*, 47(2/3):299–326, March/May 2003.

[101] D. Naishlos, M. Biberstein, S. B. David, and A. Zaks. "Vectorizing for a SIMdD DSP Architecture". In *Proc. Int. Conf. on Compilers Architectures and Synthesis for Embedded Systems*, volume 2, pages 2–11, November 2003.

[102] D. Nuzman, I. Rosen, and A. Zaks. "Auto-vectorization of Interleaved Data for SIMD". In *Proc. ACM Conf. on Programming Language Design and Implementation*, volume 41/6, pages 132–143, June 2006.

[103] J. D. Owens, S. Rixner, U. Kapasi, P. Mattson, and B. Towles. "Media Processing Applications on the Imagine Stream Processor". In *Proc. IEEE Int. Conf. on Computer Design*, September 2002.

[104] S. Panchanathan. "Architectural Approaches for Multimedia Processing". In *Proc. 4th Int. Conf. on Parallel Numerics and Parallel Computing in Image Processing Video Processing and Multimedia*, pages 196–210, 1999.

[105] A. Peleg, , and U. Weiser. "MMX Technology Extension to the Intel Architecture". *IEEE Micro*, 16(4):42–50, August 1996.

[106] A. Peleg, S. Wiljie, and U. Weiser. "Intel MMX for Multimedia PCs". *Communications of the ACM*, 40(1):24–38, January 1997.

[107] P. Pirsch, A. Freimann C. Klar, and J. P. Wittenburg. "Processor Architectures for Multimedia Applications". In *Proc. Workshop on Embedded Processor Design Challenges: Systems Architectures, Modeling and Simulation*, pages 188–206, February 2002.

[108] C. Poynton. *"A Technical Introduction to Digital Video"*. John Wiley and Sons, Inc., 1996.

[109] M. Rabbani and P. W. Jones. *"Digital Image Compression Techniques"*. Bellinghan, 1991.

[110] M. Rabbani and R. Joshi. "An Overview of the JPEG2000 Still Image Compression Standard". *Signal Processing: Image Communication*, 17(1):3–48, January 2002.

[111] S. K. Raman, V. Pentkovski, and J. Keshava. "Implementing Streaming SIMD Extensions on the Pentium 3 Processor". *IEEE Micro*, 20(4):47–57, July-August 2000.

[112] 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. on Computer Architecture*, pages 124–135, 1999.

[113] G. Ren, P. Wu, and D. Padua. "Optimizing Data Permutations for SIMD Devices". In *Proc. ACM Conf. on Programming Language Design and Implementation*, pages 118–131, June 2006.

[114] G. Roelofs. *"PNG: The Definitive Guide"*. O'Reilly, 1999.

[115] K. Ronner and J. Kneip. "Architecture and Applications of the HiPAR Video Signal Processor". *IEEE Trans. on Circuits and Systems for Video Technology*, 6(1):56–66, February 1996.

[116] H. Sasaki. "Multimedia Complex on a Chip". In *IEEE Int. Conf. on Solid-State Circuits*, pages 16–19, 1996.

[117] R. Schafer and T. Sikora. "Digital Video Coding Standards and Their Role in Video Communications". *Proceedings of the IEEE*, 83(6):907–924, June 1995.

[118] Freescale Semiconductor. "AltiVec Technology Programming Environments Manual", 2002.

[119] Philips Semiconductors. "TriMedia TM-1000: Programmable Media Processor", 1998.

[120] N. Seshan. "High VelociTI Processing". *IEEE Signal Processing Magazine*, 15(2):86–101, March 1998.

[121] J. A. Shafer. "Embedded Vector Processor Architecture for Real-Time Wavelet Video Compression". Master's thesis, Department of Electrical and Computer Eng. University of Dayton, 2004.

[122] A. Shahbahrami, B. Juurlink, D. Borodin, and S. Vassiliadis. "Avoiding Conversion and Rearrangement Overhead in SIMD Architectures". *International Journal of Parallel Programming*, 34(3):237–260, June 2006.

[123] A. Shahbahrami, B. Juurlink, and S. Vassiliadis. "A Comparison Between Processor Architectures for Multimedia Applications". In *Proc. 15th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC2004)*, pages 138–152, November 2004.

[124] A. Shahbahrami, B. Juurlink, and S. Vassiliadis. "Performance Comparison of SIMD Implementations of the Discrete Wavelet Transform". In *Proc. 16th IEEE Int. Conf. on Application-Specific Systems Architectures and Processors (ASAP)*, July 2005.

[125] A. Shahbahrami, B. Juurlink, and S. Vassiliadis. "Accelerating Color Space Conversion Using Extended Subwords and the Matrix Register File". In *Proc. 8th IEEE Int. Symp. on Multimedia*, December 2006.

[126] A. Shahbahrami, B. Juurlink, and S. Vassiliadis. "Implementing the 2D Wavelet Transform on SIMD-Enhanced General-Purpose Processors". *IEEE Trans. on Multimedia*, 10(1):43–51, January 2008.

[127] A. Shahbahrami, B. Juurlink, and S. Vassiliadis. "Versatility of Extended Subwords and the Matrix Register File". *ACM Transactions on Architecture and Code Optimization (TACO)*, 5(1), May 2008.

[128] T. Shanableh and M. Ghanbari. "Heterogeneous Video Transcoding to Lower Spatio-Temporal Resolutions and Different Encoding Formats". *IEEE Trans. on Multimedia*, 2(2):101–110, June 2000.

[129] N. T. Slingerland and A. J. Smith. "Multimedia Instruction Sets for General Purpose Microprocessors: A Survey". Technical Report UCB//CSD-00-1124, University of California, December 2000.

[130] N. T. Slingerland and A. J. Smith. "Measuring the Performance of Multimedia Instruction Sets". *IEEE Trans. on Computers*, 51(11):1317–1332, November 2002.

[131] D. B. Stewart. "Measuring Execution Time and Real-Time Performance". In *Proc. Conf. on Embedded Systems*, pages 1–15, September 2006.

[132] E. J. Stollnitz, T. D. Derose, and D. H. Salesin. *"Wavelets for Computer Graphics: Theory and Applications"*. Morgan Kaufmann, 1996.

[133] S. R. Subramanya, H. Patel, and I. Ersoy. "Performance Evaluation of Block-based Motion Estimation Algorithms and Distortion Measures". In *Proc. IEEE Int. Conf. on Information Technology: Coding and Computing*, volume 2, pages 2–7, 2004.

[134] M. Swain and D. Ballard. "Color Indexing". *International Journal of Computer Vision*, 7(1):11–32, 1991.

[135] W. Sweldens. "The Lifting Scheme: A Custom-Design Construction of Biorthogonal Wavelets". *Journal of Applied and Computational Harmonic Analysis*, 3(2):186–200, 1996.

[136] A. Tamhankar and K. R. Rao. "An Overview of H.264/MPEG-4 Part 10". In *Proc. 4th Int. Conf. on Video and Image Processing and Multimedia Communications*, pages 1–51, July 2003.

[137] R. Tessier and W. Burleson. "Reconfigurable Computing for Digital Signal Processing: A Survey". *Journal of VLSI Signal Processing*, 28(1):7–27, June

2001.

[138] Texas Instruments. *"TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide"*, July 2007. Literature Number: SPRU732D.

[139] S. Thakkar and T. Huff. "The Internet Streaming SIMD Extensions". *Intel Technology Journal*, pages 1–8, 1999.

[140] M. Tremblay, J. Michael 0'Connor, V. Narayanan, and L. He. "VIS Speeds New Media Processing". *IEEE Micro*, 16(4):10–20, August 1996.

[141] M. A. Trenas, J. Lopez, E. L. Zapata, and F. Arguello. "A Memory System Supporting the Efficient SIMD Computation of the Two Dimensional DWT". In *Proc. IEEE Int. Conf. on Acoustics Speech and Signal Processing*, volume 3, pages 1521–1524, May 1998.

[142] S. M. Vajdic and A. R. Downing. "Similarity Measures for Image Matching Architectures a Review with Classification". In *Proc. IEEE Symp. on Data Fusion*, pages 165–170, November 1996.

[143] S. Vassiliadis, G. Kuzmanov, and S. Wong. "MPEG-4 and the New Multimedia Architectural Challenges". In *15th Int. Conf. SAER*, September 2001.

[144] L. Wang, Y. Zhang, and J. Feng. "On the Euclidean Distance of Images". *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 27(8):1334–1339, August 2005.

[145] P. Xiao. "Image Compression By Wavelet Transform". Master's thesis, East Tennessee State University, 2001.

[146] D. Zhang and G. Lu. "Evaluation of Similarity Measurement for Image Retrieval". In *Proc. IEEE Int. Conf. on Neural Networks and Signal Processing*, volume 2, pages 928–931, December 2003.

List of Publications

Journal

- **A. Shahbahrami**, B. Juurlink, and S. Vassiliadis. "Implementing the 2D Wavelet Transform on SIMD-Enhanced General-Purpose Processors". IEEE Transactions on Multimedia, Vol. 10, No. 1, pages: 43-51, January 2008.

- **A. Shahbahrami**, B. Juurlink, and S. Vassiliadis. "Versatility of Extended Subwords and the Matrix Register File". ACM Transactions on Architecture and Code Optimization (TACO), Vol. 5, No. 1, May 2008.

- **A. Shahbahrami**, B. Juurlink, D. Borodin, and S. Vassiliadis. "Avoiding Conversion and Rearrangement Overhead in SIMD Architectures". International Journal of Parallel Programming, Vol. 34, No. 3, Pages 237-260, June 2006.

Proceedings

- **A. Shahbahrami**, J. Y. Hur, B. Juurlink, and S. Wong. "FPGA Implementaion of Parallel Histogram Computation". Proc. 2nd HiPEAC Workshop on Reconfigurable Computing, pp. 63-72, January 2008, Gothenburg, Sweden.

- **A. Shahbahrami**, B. Juurlink, and S. Vassiliadis. "SIMD Vectorization of Histogram Functions". 18th IEEE Int. Conf. on Application-Specific Systems Architectures and Processors (ASAP) pp. 174-179, July 2007, Montreal, Canada.

- J. Tao, **A. Shahbahrami**, B. Juurlink, R. Buchty, W. Karl, and S. Vassiliadis. "Optimizing Cache Performance of the Discrete Wavelet Transform Using a Visualization Tool". Proc. 9th IEEE Int. Symp. on Multimedia, December

2007, Taichung, Taiwan.

- **A. Shahbahrami**, B. Juurlink, and S. Vassiliadis. "Accelerating Color Space Conversion Using Extended Subwords and the Matrix Register File". Proc. 8th IEEE Int. Symp. on Multimedia, pp. 37-46, December 2006, San Diego, California, The USA.

- **A. Shahbahrami**, B. Juurlink, and S. Vassiliadis. "Limitation of Special-Purpose Instructions for Similarity Measurements in Media SIMD Extensions". Proc. ACM/IEEE Int. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems, pp. 293-303, October 2006, Soeul, South Korea.

- **A. Shahbahrami**, B. Juurlink, and S. Vassiliadis. "Improving the Memory Behavior of Vertical Filtering in the Discrete Wavelet Transform". Proc. 3rd ACM Int. Conf. on Computing Frontiers pp. 253-260, May 2006, Ischia, Italy.

- **A. Shahbahrami**, B. Juurlink, and S. Vassiliadis. "Performance Comparison of SIMD Implementations of the Discrete Wavelet Transform". Proc. 16th IEEE Int. Conf. on Application-Specific Systems Architectures and Processors, July 2005, Samos, Greece.

- **A. Shahbahrami**, B. Juurlink, and S. Vassiliadis. " Matrix Register File and Extended Subwords: Two Techniques for Embedded Media Processors". Proc. 2th ACM Int. Conf. on Computing Frontiers, May 2005, Ischia, Italy.

- B. Juurlink, **A. Shahbahrami**, and S. Vassiliadis. "Avoiding Data Conversions in Embedded Media Processors". The 20th Annual ACM Symposium on Applied Computing Santa Fe, March 2005, New Mexico, The USA.

- **A. Shahbahrami**. "The Determination of Initial Condition in Constraint Satisfaction Neural Networks for Medical Images Segmentation". The 6th Int. Conf. on CSI Computer (CSICS 2001) 20-22 Feb. 2001 University of Isfahan, Iran.

- **A. Shahbahrami**. "A Comparison of Nonparametric Algorithms in Selection of Thresholding Based on Entropy". Proc. 1st Int. Iranian Conf. on Machine Vision, Image Processing and Applications (MVIP2001), 2001, Birjand, Iran.

ProceedingsLocal

- **A. Shahbahrami** and B. Juurlink. "A Comparison of Two SIMD Implementations of the 2D Discrete Wavelet Transform". Proc. 18th Annual Workshop

on Circuits, Systems and Signal Processing (ProRISC2007), pp.169-177 , November 2007, The Netherlands.

- **A. Shahbahrami**, B. Juurlink, and S. Vassiliadis. "Performance Impact of Misaligned Accesses in SIMD Extensions". Proc. 17th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC2006) , pp. 334-342, November 2006, The Netherlands.

- **A. Shahbahrami**, B. Juurlink, and S. Vassiliadis. "Efficient Vectorization of the FIR Filter". Proc. 16th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC2005), November 2005, The Netherlands.

- **A. Shahbahrami**, B. Juurlink, and S. Vassiliadis. "A Comparison Between Processor Architectures for Multimedia Applications". Proc. 15th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC2004), November 2004, The Netherlands.

- **A. Shahbahrami**. "The Selection of Multiple Threshold Scheme for Image Segmentation Based on Co-Occurrence Matrix". The 6th Annual Scientific and Research Conference in the University of Guilan, March 2000.

- **A. Shahbahrami**. "Simulation of a Environment for Comparison of Different Page Replacement Algorithms in Structural Memory Hierarchy". The 7th Annual Scientific and Research Conference in the University of Guilan, March 2001.

Samenvatting

In deze dissertatie wordt een vernuftige SIMD uitbreiding, genaamd geModificeerde MMX (MMMX), voor multimedia berekeningen gepresenteerd. De MMX architectuur is verbeterd door toevoeging van de vergrote deelwoorden techniek en de matrix registerstructuur techniek. De vergrote deelwoorden techniek gebruikt SIMD registers die breder zijn dan het formaat waarin de data in het geheugen wordt opgeslagen. Het gebruikt 32 bits meer voor elk 64-bits register. De vergrote deelwoorden techniek vermijdt het converteren van data types en verhoogt het parallellisme in SIMD architecturen. Dit is omdat het promoveren van deelwoorden voordat ze gebruikt worden in berekeningen, en het degraderen van het resultaat voordat het kan worden opgeslagen, kosten met zich meebrengt. De matrix registerstructuur staat toe dat data die is opgeslagen in aaneenliggend geheugen in een kolom van de registerstructuur kan worden geladen. Een kolom komt overeen met de corresponderende deelwoorden van de verschillende registers. Met andere woorden, deze techniek staat toe dat de registerstructuur zowel rij- als kolomsgewijs gelezen en geschreven kan worden. Het is bruikbaar voor matrix operaties die gebruikelijk zijn in media toepassingen. Daarnaast worden in dit werk nieuwe en algenene SIMD instructies onderzocht die bedoeld zijn voor multimedia toepassingen. Toepassingsspecifieke instructies worden niet beschouwd. Speciale instructies worden gesynthetiseerd door een aantal algemene instructies. De prestatie van de MMMX architectuur wordt vergeleken met de prestatie van de MMX/SSE architectuur voor verschillende multimedia toepassingen en toepassingskernen gebruikmakend van de sim-outorder simulator, die onderdeel is van de SimpleScalar simulatieomgeving. Daarnaast worden drie zaken bediscussieerd die gerelateerd zijn aan het efficient implementeren van de 2D Discrete Wavelet Transform (DWT) op processoren voor algemene doeleinden, met name de Pentium 4. Deze zijn 64K aliasing, cache mis vanwege conflict, en vectorisatie SIMD. 64K aliasing is een verschijnsel dat kan voorkomen in de Pentium 4 en welke de prestatie aanzienlijk kan verminderen. Het vindt plaats als twee of meer data elementen, wiens adressen een veelvoud van 64K verschillen, tegelijkertijd in de cache moeten worden opgeslagen. Er is ook vaak een cache mis vanwege conflict in de implementatie van het verticale filter van de DWT als de filter lengte het aantal cache paden overschrijdt. In deze dissertatie worden technieken voorgesteld om 64K aliasing te voorkomen en de effecten van een cache mis vanwege conflict te verminderen. Verder wordt de prestatie van de 2D DWT verbeterd door data parallellisme middels SIMD instructies te exploiteren, die ondersteund worden door de meeste algemene processoren.

Curriculum Vitae

Asadollah Shahbahrami was born in Kelardasht, Chaloos, Mazandaran, Iran on 21^{st} of September 1968. He graduated from Bahonar high school in 1989 and at the same year, he was accepted to study Computer Engineering in Iran University of Science & Technology in Tehran. He got his B.Sc degree in 1993 and at the same year he was accepted to study his master study in Shiraz University, Shiraz, Iran. He received the M.Sc degree in Computer Engineering-Machine Intelligence in 1996. At the same year he was offered a permanent position at the University of Guilan, Rasht, Iran. He has worked there from 1996 to 2003 as a lecturer.

In 2003, he was entitled to an overseas Ph.D scholarship from the Iranian Ministry of Science, Research and Technology. In January 2004, he joined the Faculty of Electrical Engineering, Mathematics, and Computer Science (EEMCS), Delft University of Technology, Delft, The Netherlands, as a full-time Ph.D student under advisors Prof. Stamatis Vassiliadis and Dr. Ben Juurlink. This thesis covers his Ph.D study. When he finalized his thesis in the beginning of the 2008 year, he has started working as a research associate at the same university.

His research interests include computer architecture, image and video processing, multimedia instructions set design, and SIMD programming. He is a member of IEEE and ACM.

Acknowledgments

My first thank goes to my first promotor, Prof. Stamatis Vassiliadis who was a really great person in all aspects. He was that kind of person hard to find. His personality went well beyond the scope of nationalities. He had the skill to gather people from differnet cultures to create the best environment for work and living. He was a person that affected me to be a better person both in life and in science. It has been my fortune to choose his group amongst the many choices that I had. I remember all his guidance, support, kindness, friendliness, advice, and confidence to my family and I. He will always reside in my heart. God bless him.

I am especially grateful for the countless contributions of my supervisor, Dr. Ben Juurlink who is a smart person and always straight to the point. During my Ph.D study in Computer Engineering Laboratory, he has helped to challenge me towards critical academic reasoning and improved technical writing. I specifically thank him for his endless guidance, suggestions, attention, and support.

I acknowledge Prof. Dr. K. G. W. Goossens, my new promotor at CE group. His support in the set up of the final thesis defense was important in concluding my research at TU Delft.

I would like to thank Dr. Stephon Wong for the time and help he gave me during my study. I would also like to thank Dr. Koen Bertels, Dr. Georgi Gaydadjiev, Dr. Sorin Cotofana, Dr. Said Hamdioui, Dr. Zaid Al-Ars, Dr. Georgi Kuzmanov, and Dr. Reza Hassanpour for their time, help, and nice discussions.

Many thanks go to my former roommates, Dmitry Cheresiz, Pyrros Stathis, Bayu Kanigoro, Demid Borodin, and Ricardo Chaves who have endured my presence. I thank all my cheerful friends in CE group. I also thank Ijeoma Sandra Irobi for her time spent reading my thesis and Cor Meenderinck for his help on translating the abstract and propositions to Dutch. I am thankful to Lidwina and Bert for the help and time they gave to me. I also acknowledge S. Kaneman and V. A. C. E. van Der

Burg for designing the cover page.

Financial support for this thesis was provided by the Iranian Ministry of Science, Research, and Technology, University of Guilan, and a part by the Netherlands Organization for Scientific Research (NWO). They are gratefully acknowledged for their support to pursue my PhD research.

Our stay in Delft would have been very boring without our friends who made this time unforgettable. My special thanks goes to Helen Skinner, our first landlady in Delft, who has been helpful to my family, even doing some shopping for us when we arrived in Delft. She also helped with Dutch translation of our letters and babysitting my daughter sometimes. I always remember her help, time, support, and our "Chakochoneh". I would also like to thank all my Iranian friends in Delft warmly.

I want to express my gratitude to my literature teacher, Mr. Delfan in high school in Kelardasht, who gave me a force and self confidence to do everything that I like and to continue my study at the University. My gratitude is also extended to my aunts Golambar, Mahnaz, and their respectful family Ali Cavooci and Salim Teimornejad. I always remember their help and kindness.

Last but not least, I am eternally grateful to my mother Gohar, father Saadi, and my eldest brother Zabiullah for their unlimited love and support in all aspects of my life. I would also like to thank my mother-in-law Mehr Mah Nekoeemehr who encouraged me to continue my studies and my wife, Mitra for her unlimited love, understanding, and patience and as the mother of our lovely daughter Yasamin. Finally, I express my deepest love to Yasamin for her understanding when I was really busy with my work and for her patience all those evenings that I was away.

Delft, *Asadollah Shahbahrami*
September 2008