Efficient Execution of Video Applications

on Heterogeneous Multi- and Many-Core Processors

Efficient Execution of Video Applications

on Heterogeneous Multi- and Many-Core Processors

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus Prof.ir. K.C.A.M Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op maandag 6 juni 2011 om 15:00 uur
door
Arnaldo PEREIRA DE AZEVEDO FILHO
Master in Computer Science
Universidade Federal do Rio Grande do Sul
geboren te Natal, Rio Grande do Norte, Brazil.

Dit proefschrift is goedgekeurd door de promotor:
Prof.dr. B.H.H. Juurlink

Copromotor:
Dr. K.L.M. Bertels

Samenstelling promotiecommissie:

Rector Magnificus,	voorzitter
Prof.dr. B.H.H. Juurlink,	Technische Universität Berlin, promotor
Dr. K.L.M. Bertels,	Technische Universiteit Delft, copromotor
Prof.dr. J. van Leeuwen,	U-Utrecht
Prof.dr. L.A. Sousa,	U. Tecnica de Lisboa
Prof.dr. H.A.G. Wijshoff,	U-Leiden
Prof.dr.ir. H.J. Sips,	Technische Universiteit Delft
Prof.dr.ir. R.L. Langendijk,	Technische Universiteit Delft
Prof.dr. C. Witteveen,	Technische Universiteit Delft, reservelid

ISBN: 978-90-72298-17-1

Printed in the Netherlands

Dedicated to the memory of my father.

Efficient Execution of Video Applications

on Heterogeneous Multi- and Many-Core Processors

Abstract

In this dissertation we present methodologies and evaluations aiming at increasing the efficiency of video coding applications for heterogeneous many-core processors composed of SIMD-only, scratchpad memory based cores. Our contributions are spread in three different fronts: thread-level parallelism strategies for many-cores, identification of bottlenecks for SIMD-only cores, and software cache for scratchpad memory based cores.

First, we present the 3D-Wave parallelization strategy for video decoding that scales for many-core processors. It is based on the observation that dependencies between frames are related with the motion compensation kernel and motion vectors are usually within a small range. The 3D-Wave strategy combines macroblock-level parallelism with frame- and slice-level parallelism by overlapping the decoding of frames while dynamically managing macroblock dependencies. The 3D-Wave was implemented and evaluated in a simulated many-core embedded processor consisting of 64 cores. Policies for reducing memory footprint and latency are presented. The effects of memory latency, cache size, and synchronization latency are studied.

The assessment of SIMD-only cores for the increasing complexity of current multimedia kernels is our second contribution. We evaluate the suitability of SIMD-only cores for the increasing divergent branching in video processing algorithms. The H.264 Deblocking Filter is used as test case. Also, the overhead imposed by the lack of a scalar processing unit for SIMD-only cores is measured using two methodologies. Low area overhead solutions are proposed to add scalar support to SIMD-only cores.

Finally, we focus on the memory hierarchy and we propose a new software cache organization to increase the efficiency and efficacy of scratchpad memories for unpredictable and indirect memory accesses. The proposed Multidimensional Software Cache reduces software cache overhead by allowing the programmer to exploit known access behavior in order to reduce the number of accesses to the software cache and by grouping memory requests. An instruction to accelerate MDSC lookup is also presented and analyzed.

Acknowledgments

This thesis is a result of an effort performed with several direct and indirect collaborators that I would like to thank. First, I thank God for the gift of life and for constantly being watching over me. I want to thank my family: my father for this example and my mother for her support and understanding; my wife, Ozana, for her love and for being at my side in this process; and my son, Alex, that knows, like no one else, how to bring a smile to my face. I also thank my extended "family", my friends, which from far away or close by always had time to share with me the ups and downs.

I would like to thank my supervisor, Ben Juurlink, for his guidance and patience throughout the PhD. process. I also would like to thank my former master and undergrad advisors Sergio Bampi and Ivan Saraiva, which paved my initial scientific career. I thank also Cor Meenderinck, Mauricio Alvarez, Alex Ramirez, Andrei Terechko, Jan Hoogerbrugge, Zdravko Popovic and Roberto Giorgi for their collaboration on the development of this work.

I thank also the support staff of our faculty, Lidwina Tromp, Monique Tromp, Eef Hartman, and Erik de Vries, for their effort to keeps things working properly so we can concentrate on our work.

During the years in the group, some people arrived and others left and many of them I can consider friends, not just colleagues. I will not name them to avoid injustices, but I thank each of you for the time we spent together. I also thank my other colleagues that, for a reason or another, did not share more time together, but also offered great input with their presence.

For all of you, please keep in mind that my gratitude is not directly proportional with the length of this acknowledgments.

A. Pereira de Azevedo Filho Delft, The Netherlands, 2011

Contents

List of Figures

List of Tables

Abbreviations

2DW	2D-Wave
3DW	3D-Wave
ABT	Adaptive Block size Transform
AMDSC	Accelerated Multidimensional Software Cache
AML	Average Memory Latency
API	Application Programming Interface
AVC	Advanced Video Coding
BLAS	Basic Linear Algebra Subprograms
BP	Baseline Profile
BS	Boundary Strength
CABAC	Context Adaptive Binary Arithmetic Coding
CAVLC	Context Adaptive Variable Length Coding
CIF	Common Intermediate Format (352×288 pixels)
DCT	Discrete Cosine Transform
DI	Deblocking Info
DF	Deblocking Filter
DLP	Data Level Parallelism
DMA	Direct Memory Access
ED	Entropy Decoding
FHD	Full High Definition (1920×1080 pixels)
FIFO	First-In-First-Out
FIR	Finite Impulse Response
FMO	Flexible Macroblock Ordering
GLCM	Gray-Level Co-occurrence Matrix
GOP	Group of Pictures
HD	High Definition (1280×720 pixels)
HD-DVD	High Definition Digital Video Disc
HiP	High Profile
I	Intra predicted frame, slice, or macroblock
IDCT	Inverse Discrete Cosine Transform
IEC	International Electrotechnical Commission

ILP	Instruction Level Parallelism
IQ	Inverse Quantization
ISO	International Organization for Standardization
KoL	Kick-off List
LDT	Large-Data-Type
LRU	Least Recently Used
LS	Local Store
MAP	Maximum A Posteriori
MB	MacroBlock
MC	Motion Compensation
MDSC	Multidimensional Software Cache
ME	Motion Estimation
MFC	Memory Flow Controller
MIC	Memory Interface Controller
MIMD	Multiple Instruction Multiple Data
MP	Main Profile
MPEG	Moving Picture Experts Group
MV	Motion Vector
MVP	Motion Vector Prediction
NoC	Network on Chip
P	Predicted frame, slice, or macroblock
POC	Picture Order Count
PP	Picture Prediction
PPE	Power Processing Element
PPU	Power Processing Unit
PS3	PlayStation 3
PSNR	Peak Signal-to-Noise Ratio
QCIF	Quarter Common Intermediate Format (176×144 pixels)
QVGA	Quarter Video Graphics Array (320×240 pixels)
RISC	Reduced Instruction Set Computer
RMB	reference MB
SARC	Scalable computer Architecture
SC	Software Cache
SD	Standard Definition (720×576 pixels)
SIMD	Single Instruction Multiple Data
SMT	Simultaneous Multi-Threading
SPE	Synergistic Processing Element
SPMD	Single Program Multiple Data
SPU	Synergistic Processing Unit

SSE	Streaming SIMD Extensions
STI	Sony-Toshiba-IBM
TAGP	Tag Array in General Purpose
TARF	Tag Array Register File
TLP	Thread Level Parallelism
TP	Task Pool
UHD	Ultra High Definition
UVLC	Universal Variable Length Coding
VA	Viterbi Algorithm
VCEG	Video Coding Experts Group
VLC	Variable Length Coding
VLIW	Very Large Instruction Word
VP	vector prediction
XP	Extended Profile
ZT	Zero-Tail

1 Introduction

In the past, increasing computational demand was mainly satisfied by increasing the clock frequency and by exploiting more instruction-level parallelism (ILP). Due to the inability to increase the clock frequency much further because of thermal constraints and because it is difficult to exploit more ILP, multi-core architectures have appeared on the market. As the number of transistors in a die keep increasing as the fabrication technology continuously evolve, it is expected that the number of cores on a chip will double every three years [91]. Current Intel processor Westmere features 6 high performance cores in a single die [54]. AMD also produces a 6 core processor, the Phenom II [30]. The power budget for the processor, however, does not increase, as heat releasing capacity does not have changed. This limits the number of computations that can be performed simultaneously on the processor.

On the other hand, the consumer market pushes higher quality and feature rich media experience. Audio is shifting from lossy (such as MP3) to lossless compression (FLAC [23], DTS-HD Master Audio [31], Dolby TrueHD [28]), even in multichannel formats. High Definition video playback and recording are already present in mobile devices [70] and the next devices capable of decoding 4K (4096×3072) video resolution [73] are being developed [27]. 4K resolution video is even already supported by the internet streaming video service YouTube [84].

Another feature being pushed by the market is the 3D video in stereoscopic (at the time of writing) and soon in freeview (glasses free 3D) formats. Stereoscopic 3D requires 2 different images, one for each eye, to be displayed simultaneously. It relies on glasses that filter the correct image to each eye. Freeview experience, however, is already being advertised [69] and requires up to 9 frames to be displayed simultaneously. These features are covered with amendments in the H.264 standard [92], called Multiview Video Coding (MVC) [19]. In order to decrease the bitstream size, the extra frames depend on the simultaneous and previous ones. This results in an even higher computational complexity than running several independent H.264 streams.

To enable the processing of this new media contents within the limits of the power budget, an increase in power efficiency is needed. One option to increase power efficiency is to parallelize the application on a heterogeneous multi-/many-core processor composed of power efficient cores in tune with the target applications characteristics [53, 52]. This, however, requires thread-level parallelism (TLP) in applications. TPL is the case when different threads or processes can execute simultaneously, either on the same or different data. Moreover, the amount of thread-level parallelism should be sufficient to scale the processing to a large number of cores to provide the required computational power by increasing the number of cores. Fortunately, multimedia processing usually exhibits TLP but current approaches do not scales to large number of cores [66].

Multimedia applications typically also exhibit significant amounts of data-level parallelism (DLP). DLP is the case when the same operation or task can be applied simultaneously on different pieces of data. DLP can be exploited in a power-efficient manner by means of Single-Instruction Multiple-Data (SIMD) operations. SIMD units have been widely used by high-end processors to accelerate multimedia applications. The Sony-Toshiba-IBM (STI) Cell processor [50] brought this concept further. The Cell processor is an heterogeneous multi-core processor that features a general purpose core and 8 SIMD-only cores. SIMD-only cores are cores which instructions operate exclusively on a SIMD fashion, i.e., on all elements of the input vectors simultaneously. Its design has been driven by power efficiency [45].

The shift towards multi-core architectures also brings new challenges for the memory hierarchy. One of these challenges is the design of the memory hierarchy tuned for power efficiency. Scratchpad memories reappeared as a solution because they can be very efficient in terms of power and performance for applications with predictable memory access patterns [12]. Furthermore,

multimedia applications feature mostly predictable data sets and access patterns making it possible to transfer the necessary data before the computation. These data transfers usually need to be explicitly exposed by the programmer. This structure also makes possible to overlap computation and data transfers by means of double buffering techniques. Scratchpad memories also have predictable latencies. However, some multimedia applications do not feature predictable data sets and access patterns. These applications present two significant problems for scratchpad memory based processors. The first problem is that the data transfer cannot be overlapped with the computation. The process has to wait for the data to be transferred to the scratchpad memory. The second problem is that data locality cannot easily be exploited. It is difficult to keep track of the memory area present in the scratchpad memory and new data must be requested for each access.

The remainder of this chapter is organized as follows. Section 1.1 presents the processor architecture template that will be used as the baseline for improvements. The objectives of this thesis are detailed in Section 1.2. The organization and contributions of this thesis are presented in Section 1.3.

1.1 SARC Architecture

This thesis is part of the collaborative Scalable computer ARChitecture (SARC) project [83, 78]. The SARC project aimed at designing a scalable many-core architecture for a wide range of applications. The SARC architecture is a heterogeneous many-core architecture template that is based on a master-worker programming model. It targets a new class of task-based data-flow programming models that includes StarSs [77], Cilk [14, 39], RapidMind [65], Sequoia [36], and OpenMP 3.0 [74, 25]. These programming models allow programmers to write efficient parallel programs by identifying candidate functions to be off-loaded to processing cores.

A SARC processor instance consists of Master cores, target applications or application domain accelerators, Network on Chip (NoC), a banked level-2 (L2) cache, and Memory Interface Controllers (MICs). Figure 1.1 depicts the block diagram of a general SARC processor instance. The type and number of the application accelerators as well as the number of master cores, L2 banks, and MICs are implementation dependent. A brief description of each architectural component is given below.

The Master cores are responsible for executing the master threads. The Master cores start the applications and create the threads that will be executed

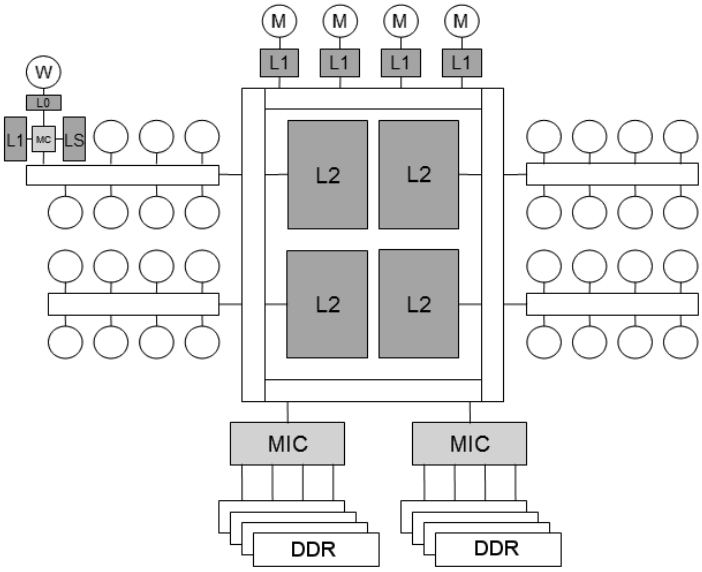


Figure 1.1: Block diagram of a generic SARC instance.

by the Worker cores. It runs the runtime system, which manages data and task scheduling. A Master core can also create threads for other Master cores that will also spawn Worker threads. Because the system performance depends on the tasks being sequentially spawned by the master core, their single-thread performance is critical. Therefore, they are high-performance out-of-order cores. Master cores rely on data and instruction caches to exploit locality as their working set is not predictable.

Application accelerators consist of a set of locally connected Worker cores. The Worker cores are specialized application or application domain cores. They execute the tasks offloaded from the Master cores. A common feature of the Worker cores is the presence of a scratchpad memory. The scratchpad memory is mapped in the logical address space, so Workers can access scratchpad memories of other Workers. Workers feature a DMA controller to transfer data from/to their scratchpad memories to/from main memory while processing. In this thesis, we will focus on the development of the SARC multimedia instance, including the microarchitecture of the Worker cores.

The SARC NoC consists of a hierarchy of K-buses. A K-bus is a collection of buses. For instance, a 4-bus allows for up to 4 simultaneous data transfers, given the fact that the destination ports are mutually exclusive.

A banked L2 cache is implemented to eliminate the need for maintaining cache coherence. This is possible because data is mapped to a cache bank based on its physical address. The L2 cache in the SARC architecture template captures both misses from L1 caches and DMA transfers. The L2 cache accesses the external memory via the MICs. Each MIC supports several Dynamic Random-Access Memory (DRAM) channels.

The above description of the SARC architecture template is the result of the research realized through the period of the project, from 2006 to 2010, and were not finalized during the development of the work described in this thesis. Because of this, the used architecture template in this thesis is slightly different. The differences are mainly restricted to the memory hierarchy. While in the described architecture template the Worker cores features a cache hierarchy, the version used in this thesis features only a scratchpad memory. Also, the described L2 cache is not featured on the experiments and, therefore, the DMA unit accesses the external memory directly.

Thought this thesis, we use the STI Cell processor and its Synergistic Processing Elements (SPEs) [43] as an architectural emulator for the SARC media instance and the Worker core, respectively. The Cell was chosen because it was, at the beginning of our work, an already available heterogeneous multi-core multimedia accelerator that shares several characteristics with the SARC accelerators. The Cell processor uses a ring buses to connect the SPEs, similarly to the SARC K-buses local connection, and the SPEs are scratchpad memory based cores that transfer data via DMA requests. The Cell processor and the SPE cores are described in detail in Section 4.3.

1.2 Objectives

In the design of the SARC multimedia instance, we focus on three topics to achieve efficient execution of video coding applications on many-cores processors. TLP, DLP, and memory hierarchy need to be efficiently exploited in order to scale the computational throughput with the number of cores. Improvements on each of these topics correspond to the main objectives of this thesis.

Current parallelization techniques for high definition (from 1280×720 to 1920×1080 pixels) video processing do not properly scale over 8 cores [66, 22]. Therefore, the first objective of this thesis is to leverage video coding applications to the forthcoming many-core era. This is key for the development of many-core processors because if there is not sufficient parallelism, most of the

cores will be idle or under utilized. The solution should exhibit high scalability with increasing number of cores in order to be able to provide the required performance for future applications. The solution should also be feasible in terms of processing latency and memory requirements, among others.

Identifying and eliminating bottlenecks in the execution on SIMD-only cores is the second objective of this thesis. SIMD processing has well documented processing overheads [80, 89, 8], such as data transpositions, data (un)packing, and alignment. Transposition is the operation of writing the columns of a matrix A as the rows of a matrix A^T. This operation is required when data store in row-wise form (in single memory word) need to be computed in a column-wise fashion (in separated memory words). Data packing is the selection of data scattered over memory words into a SIMD word while unpacking is the merging of the contents of the SIMD word back with the original content of the memory words. Alignment issues happen when the data to be processed are split in two consecutive SIMD words, as they need to be combined in one single word to be processed. SIMD-only cores introduces new aspects, in contrast with cores with SIMD units, that lead to overheads. These specific SIMD-only overheads and how they could be eliminated are still an open topic.

Finally, the third objective of this thesis is to increase the effectiveness and efficiency of scratchpad memories for unpredictable memory accesses. In general, scratchpad memories are power efficient and fit well with the characteristics of video coding applications, as well as other multimedia applications. Some multimedia kernels, however, present access behaviors that do not allow the request of required data in advance. In this case, the core will have to stall while waiting for the data. One solution for such kernels with irregular and indirect data accesses is to use software caches. Software caches, however, incur significant overhead with the cache access being the dominant overhead, as will be demonstrated later in this thesis. Software and hardware alternatives should be analyzed in order to reduce software cache overhead.

1.3 Organization and Contributions

As mentioned in the previous section, we focus in three topics: TLP, DLP, and memory hierarchy. Each of these topics is addressed in a set of two chapters as follows. Chapters 2 and 3 deals with the exploitation of TLP in video processing for many-core architectures. It is followed, in Chapters 4 and 5, by the evaluation of SIMD-only cores for divergent branching kernels and scalar pro-

cessing. Solutions for increasing efficiency and efficacy of scratchpad memories for unpredictable and indirect memory accesses are presented in Chapters 6 and 7. The contributions of this thesis and the contents of each chapter are presented below.

The first main contribution of this thesis is the 3D-Wave video (de)coding parallelization strategy that scales to a large number of cores. This strategy breaks frame dependencies in a novel way by overlapping the (de)coding of several inter-dependent frames and stressing the MB parallelism that can be exploited. We choose the H.264 as our target video decoding standard due to its high computational demands and wide utilization. In order to evaluate the potential of the 3D-Wave strategy, the Static 3D-Wave evaluation methodology is introduced in Chapter 2. The Static 3D-Wave calculates the number of macroblocks (MBs) that can be processed in parallel given a maximum motion vector length, where each MB has a fixed decoding time. A second evaluation methodology, called Dynamic 3D-Wave, is also presented. It computes the number of MBs that can be processed in parallel by evaluating the MB dependencies chain in a given video sequence. A fixed MB decoding time is also assumed. Chapter 2 also briefly presents the H.264 standard and review previous parallelization techniques for video (de)coding.

Implementing the 3D-Wave in a simulated many-core processor turned out to be quite challenging and led to several contributions. A mechanism that guarantees that MBs are not processed before their intra and inter frame dependencies are satisfied is presented. Furthermore, a decoding policy is introduced to reduce the latency of the technique given the number of frames decoded simultaneously. In addition, in order to control the size of the working memory, a frame scheduling policy is introduced to control the number of frames in flight. Memory latency, impact of L1 cache size, impact of task management latency, and entropy decoding acceleration are also evaluated. The implementation of the 3D-Wave and the above contributions and evaluations are described in Chapter 3.

The second main contribution of this thesis is the assessment of SIMD-only cores for divergent branching multimedia kernels. As a case study, the highly adaptive H.264 Deblocking Filter (DF) kernel is vectorized with SIMD instructions on the Worker core. The vectorized DF execution is analyzed and its overheads are classified into SIMD and divergent branching related. For comparison, the DF is also vectorized with SIMD instructions in a similar general purpose core with a SIMD processing unit. The described assessment is presented in Chapter 4. Chapter 4 also presents a brief description of the

DF kernel, a literature review on SIMD overhead, and a description of the Cell processor with focus on the SPE (used to emulate the Worker core).

The quantification of the scalar processing overhead on SIMD-only cores is the third main contribution of this thesis. The scalar processing in current SIMD-only cores is handled by the compiler. The compiler introduces a number of operations to guarantee the correct program semantic. These operations, however, introduce overhead in terms of extra instructions to be executed and pipeline stalls. Two evaluation methodologies are developed in order to quantify the overhead. The first methodology, called Large-Data-Type, eliminates the sources of scalar processing overhead in SIMD-only cores by increasing the size of the scalar word to 128 bits. Without the sources of overhead, the program execution in the SIMD-only core is equal to a program execution in a core with scalar support. The second methodology compares the execution times of kernels in the SIMD-only core with a similar core with scalar support. Additionally, special load and store instructions that minimize the scalar overhead are presented. The scalar overhead evaluation and the proposed instructions are presented in Chapter 5.

In Chapter 6, we contribute by presenting a novel Multidimensional Software Cache (MDSC) organization for scratchpad memory based cores that reduces software cache overhead. The MDSC reduces the software cache overhead on two fronts. First, it allows the exploitation of known access behavior to minimize the number of cache accesses. This exploitation is enabled by the use of matrix indices instead of linear addresses combined with the multidimensional cache blocks. For instance, if the programmer (or the compiler) identifies access patterns in the program that are restricted to a small number of cache blocks, the first access to the cache block is kept while the subsequent accesses can be replaced by pointer arithmetic. Second, the MDSC groups multiple external memory requests, which reduces latency when compared with the sum of individual transfers.

The last main contribution of this thesis is an instruction to accelerate the MDSC. The proposed instruction, called LookUp_SC, performs the tag formation, cache look-up, and address calculation. These steps are the main body of the MDSC access functions that are the predominant overhead of the MDSC, as will be presented on Chapter 6. Nevertheless, these steps consist of operations that are sufficiently simple to be implemented in hardware and many steps can be performed in parallel. The LookUp_SC instruction resulted in speedups ranging from 1.28 to 2.1 in the evaluated kernels. The description and evaluation of the LookUp_SC instruction is presented in Chapter 7.

Chapter 8 concludes this dissertation. A summary of the presented content and contributions are given. Chapter 8 also includes directions on how the work can be extended in the future, based on the acquired results.

Because the contributions of this thesis relate to different topics, there is no separated related work chapter. Instead, the related work will be presented per chapter or per group of chapters, in order to provide easiness of reference and to obtain a more consistent structure of the thesis.

2

A Scalable Parallel Algorithm for H.264 Decoding

As argued in Chapter 1, an important question is whether emerging and future applications exhibit sufficient parallelism, in particular thread-level parallelism (TLP), to exploit the large numbers of cores future many-core processors are expected to contain. In this chapter, we investigate the amount of TLP available in video coders/decoders (codecs), an important application domain now and in the future. Specifically, we analyze and present a method to enhance the parallel scalability of the H.264 [92] decoding process.

This chapter is organized as follows. Section 2.1 briefly introduces the need and applicability of scaling video processing to many-core architectures. Section 2.2 presents a short overview of the H.264 standard. Possible parallelization techniques for H.264 and related work are reviewed in the Section 2.3. In Section 2.4, we present a novel parallelization strategy called 3D-Wave and analyze the amount of TLP it exhibits using a static approach. Section 2.5 discusses the effects of limiting resources available to the 3D-Wave strategy. A dynamic 3D-Wave evaluation that takes real macroblock (MB) dependencies into account is briefly presented in Section 2.6. Section 2.7 concludes this chapter.

2.1 Introduction

The demand for computational power increases continuously in the consumer market as new applications such as Ultra High Definition (UHD) video [73], 3D TV [29], and real-time High Definition (HD) video encoding are forecasted. In the past this demand was mainly satisfied by increasing the clock frequency and by exploiting more instruction-level parallelism (ILP). As noted in Chapter 1, however, due to the inability to increase the clock frequency much further because of thermal constraints and because it is difficult to exploit more ILP, multi-core architectures have appeared on the market.

This new paradigm relies on the existence of sufficient thread-level parallelism (TLP) to exploit large number of cores. Techniques to extract TLP from applications will be crucial to the success of multi-cores. This chapter investigates the exploitation of the TLP available in an H.264 video decoder on an multi-core processor H.264 was chosen due to its high computational demands, wide utilization, and development maturity. Furthermore, even more demanding applications such as 3D TV are based on current video coding methods [68]. Although a 64-core processor is not required to decode a Full High Definition (FHD) video in real-time, real-time encoding remains a problem, and decoding, furthermore, is part of encoding.

In this chapter we propose a novel parallelization strategy, called 3D-Wave, for H.264 decoding which is scalable to a large number of cores. It is mainly based on the observation that inter-frame dependencies have a limited spatial range. Because of this, certain MBs of consecutive frames can be decoded in parallel. In this chapter, we analyze the available MB-level parallelism using a static approach, called Static 3D-Wave. In this static approach, the decoding of the next frame is started as soon as the reference window of the upper-left MB is decoded, given an arbitrary maximum motion vector (MV) length. We use arbitrary MV lengths because the actual MV lengths on real sequences are much smaller than the maximum MV length allowed by the H.264 standard, as we will present in Section 2.6. In Section 2.6 we also consider the actual MV length and analyze how this affects the scalability. This will be used as basis for the implementation described in Chapter 3.

Although our focus in this chapter is on the decoder, the proposed technique is also suitable for parallelizing the encoder. Because in the encoder the dependencies are known a priori, however, it is easier to apply the 3D-Wave to the encoder than to the decoder. Due to the restrictions and challenges in the decoder, it was chosen as the target of this study.

This chapter is the result of a close collaboration effort with two other PhD students, Cor Meenderinck and Mauricio Alvarez. Their work is presented mainly in the introductory sections of this chapter. Credit will be given where it is due. Broadly, we made the observation which forms the basis of the 3D-Wave strategy and performed an initial evaluation using the static approach, while Cor Meenderinck performed the dynamic analysis, and Mauricio Alvarez reviewed the related literature.

2.2 Overview of the H.264 Standard *

Currently, one of the leading video coding standard, in terms of compression and quality is H.264 [2, 72]. It is used in Blu-Ray Disc, internet streaming, and many countries are using or will use it for terrestrial television broadcast, satellite broadcast, and mobile television services. It has a compression improvement of over two times compared to previous standards such as MPEG-4 ASP and H.262/MPEG-2 [98]. The H.264 standard was designed to serve a broad range of application domains ranging from low to high bitrates, from low to high resolutions, and a variety of networks and systems, e.g., internet streams, mobile streams, disc storage, and broadcast. The H.264 standard was jointly developed by ITU-T Video Coding Experts Group (VCEG) and ISO/IEC Moving Picture Experts Group (MPEG). It is also called MPEG-4 part 10 or AVC (Advanced Video Coding).

Figures 2.1 and 2.2 depict block diagrams of the decoding and the encoding process of H.264, respectively. The main kernels are Prediction (intra prediction, Motion Estimation (ME), and Motion Compensation (MC)), Discrete Cosine Transform (DCT), Quantization, Deblocking filter, and Entropy Coding. They operate on macroblocks (MBs), which are blocks of 16×16 pixels, although the standard allows some kernels to operate on smaller blocks, down to 4×4. H.264 uses the YCbCr color space with a 4:2:0 subsampling.

Advances in ME/MC is one of the major contributors to the compression improvement of H.264. The standard allows variable block sizes ranging from 16×16 down to 4×4, and each block has its own MV(s). The MV is quarter sample accurate. Furthermore, multiple reference frames can be used in a weighted fashion. This significantly improves coding occlusion areas where an accurate prediction can only be made from a frame further in the past. The MC kernel will be further detailed in Chapter 6.

*This section is based on text written mainly by Cor Meenderinck.

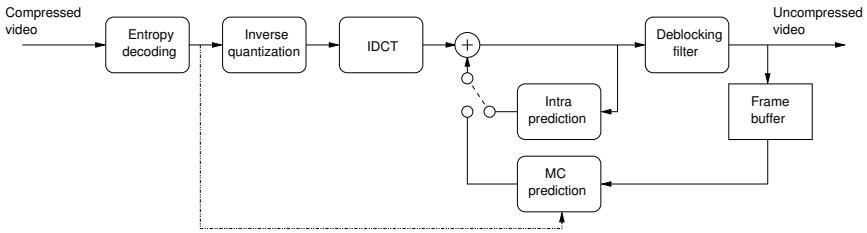


Figure 2.1: Block diagram of the decoding process.

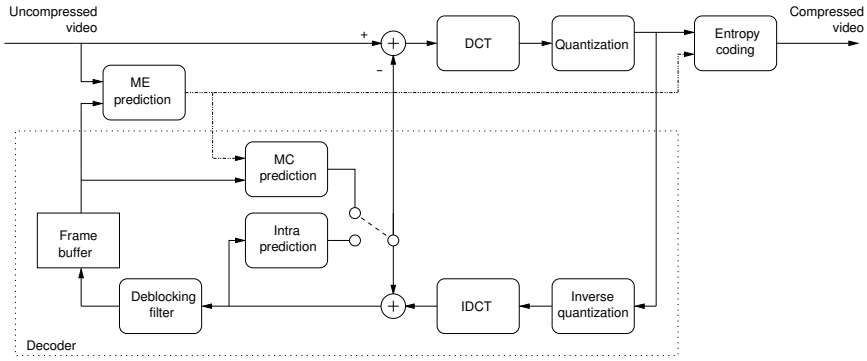


Figure 2.2: Block diagram of the encoding process.

Two types of intra coding are supported, which are denoted as Intra_4×4 and Intra_16×16. The first type uses spatial prediction on each 4×4 luminance block. Eight modes of directional prediction are available, among them horizontal, vertical, and diagonal. This mode is well suited for MBs with small details. For smooth image areas the Intra_16×16 type is more suitable, for which four prediction modes are available. Chroma components are estimated for whole MBs using one specialized prediction mode.

MPEG-2 and MPEG-4 part 2 employed an 8×8 floating point transform. However, due to the decreased granularity of the motion estimation, there is less spatial correlation in the residual signal. Thus, a standard 4×4 (that means 2×2 for chrominance) transform is used, which is as efficient as a larger transform [63]. Moreover, a smaller block size reduces artifacts known as ringing and blocking. An optional feature of H.264 is Adaptive Block size Transform (ABT), which adapts the block size used for DCT to the size used in motion estimation [100]. Furthermore, to prevent rounding errors that occur in floating point implementations, an integer transform was chosen.

Processing a frame in MBs can produce blocking artifacts, generally considered the most visible artifact in prior standards. This effect can be resolved by applying a deblocking filter around the edges of a block. The strength of the filter is adaptable through several syntax elements [59]. While in H.263+ this feature was optional, in H.264 it is standard and it is placed within the motion compensated prediction loop (see Figure 2.1) to improve the motion estimation. The Deblocking Filter will be further detailed in Chapter 4.

There are two classes of entropy coding available in H.264: Variable Length Coding (VLC) and Context Adaptive Binary Arithmetic Coding (CABAC). The latter achieves up to 10% better compression but at the cost of large computational complexity [64]. The VLC class consists of Context Adaptive VLC (CAVLC) for the transform coefficients, and Universal VLC (UVLC) for the small remaining part. CAVLC achieves large improvements over simple VLC, used in prior standards, without the full computational cost of CABAC.

Figure 2.3 depicts how the video data is structured in H.264. A video sequence is composed out of Group of Pictures (GOPs) which are independent sections of the video sequence. GOPs are used for synchronization purposes because there are no temporal dependencies between them. Each GOP is composed by a set of frames, which can have temporal dependencies when motion prediction is used. Each frame can be composed of one or more slices. The slice is the basic unit for encoding and decoding. Each slice is a set of MBs and there are no temporal or spatial dependencies between slices. Further, there are MBs, which are the basic units of prediction. MBs are composed of luma and chroma blocks of variable size. Finally each block is composed of picture samples. Data-level parallelism can be exploited at each level of the data structure, each one having different constraints and requiring different parallelization methodologies.

H.264 defines three main types of slices and macroblocks: I, P, and B. An I-slice uses intra prediction MBs (I MBs) and is independent of other slices. In intra prediction a MB is predicted based on adjacent blocks. A P-slice is composed of I and P MBs. A P-MB uses motion estimation and depends on one or more previous slices, either I or P. Motion estimation is used to exploit temporal correlation between slices. Finally, B-slices are composed of I, P and B MBs. A B-MB uses bidirectional motion estimation and depends on slices from past and future [38]. Figure 2.4 depicts a typical slice order and the dependencies, assuming each frame consists of one slice only. The standard

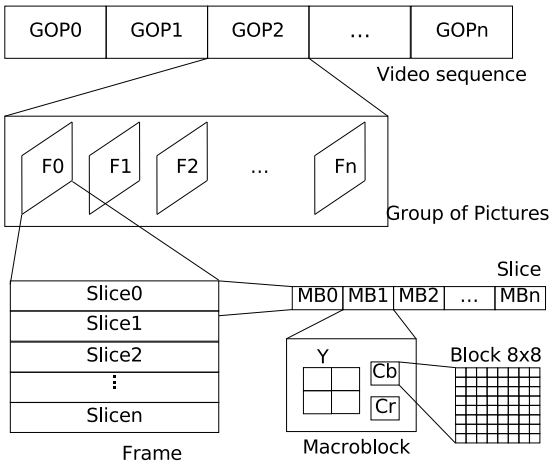


Figure 2.3: H.264 data structure.

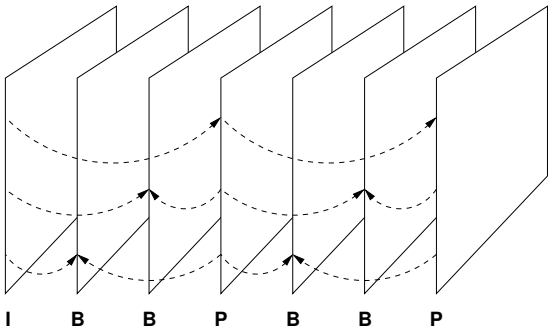


Figure 2.4: A typical slice/frame sequence and its dependencies.

also defines SI and SP slices that are slightly different from the ones mentioned before and which are targeted at mobile and internet streaming applications.

The standard was designed to suite a broad range of video application domains. Each domain, however, is expected to use only a subset of the available options. For this reason profiles and levels were specified to mark conformance points. Encoders and decoders that conform to the same profile are guaranteed to inter-operate correctly. Profiles define sets of coding tools and algorithms that can be used while levels place constraints on the parameters of the bitstream.

The standard initially defined three profiles, but has since then been extended to a total of 11 profiles, including three main profiles, four high profiles,

and four all-intra profiles. The three main profiles and the most important high profile are:

- **Baseline Profile (BP):** the simplest profile mainly used for video conferencing and mobile video.
- **Main Profile (MP):** intended to be used for consumer broadcast and storage applications, has been overtaken by the high profile.
- **Extended Profile (XP):** intended for streaming video and includes special capabilities to improve robustness.
- **High Profile (HiP)** intended for high definition broadcast and disc storage, and is used in HD DVD and Blu-ray.

Besides HiP there are three other high profiles that support up to 14 bits per sample, 4:2:2 and 4:4:4 sampling, and other features [92]. The all-intra profiles are similar to the high profiles and are mainly used in professional camera and editing systems.

In addition 16 levels are currently defined which are used for all profiles. A level specifies, for example, the upper limit for the picture size, the decoder processing rate, the size of the multi-picture buffers, and the video bitrate. Levels have profile independent parameters as well as profile specific ones. The H.264 standard has many options. For more details the interested reader is referred to [99, 93].

2.3 Parallelizing H.264 †

The coding efficiency gains of advanced video codecs such as H.264 come at the price of increased computational requirements. The demands for computing power increases also with the shift towards high definition resolutions. As a result, current high performance uniprocessor architectures are not capable of providing the required performance for real-time processing [5, 76, 55, 47].

In order to obtain the required performance for real-time operation at high definition, it is necessary to exploit parallelism. The H.264 codec can be parallelized either by a task-level or a data-level decomposition. Each technique has advantages and disadvantages. In this section we examine both and compare them in terms of communication and synchronization requirements, load balancing, and scalability.

†This section is based on text written mainly by Mauricio Alvarez.

2.3.1 Task-Level Decomposition

In a task-level decomposition the functional partitions of the algorithm are assigned to different processors. As depicted in Figure 2.1, the process of decoding H.264 consists of performing a series of operations on the coded input bitstream. Some of these tasks can be done in parallel and the processing can be pipelined where the results of one task are streamed as input to the next.

Because of this streaming behavior, task-level decomposition requires a significant amount of communication between the different tasks in order to send the data from one processing stage to the other, and this may become a bottleneck. Additionally, synchronization between the modules is required.

The main drawbacks, however, of task-level decomposition are poor load balancing and limited scalability. Balancing the load is difficult because the time to execute each task is not known before hand and depends on the data being processed. Scalability is also difficult to achieve because increasing the number of processors requires to redistribute the tasks and a new load balancing of the pipeline. Finally, from the software optimization perspective, the task-level decomposition requires that each task/processor implements a specific software optimization strategy.

Gulati et al. [44] describe a system for encoding and decoding H.264 on a multiprocessor architecture using a task-level decomposition approach. The employed multiprocessor includes eight DSPs and four control processors. This system achieves real-time operation for low resolution video inputs, using the baseline profile which is a limited set of the H.264 standard features (e.g., no CABAC, no B-frames).

2.3.2 Data-Level Decomposition

In a data-level decomposition the work (data) is divided into smaller parts and each of the parts is assigned to a different processor. Each processor runs the same program but on different (multiple) data elements (SPMD). In H.264 data decomposition can be applied at different levels of the data structure.

2.3.2.1 GOP-Level Parallelism

The coarsest grained parallelism is at the GOP-level (see Figure 2.3.2). H.264 can be parallelized at the GOP-level by defining a GOP size of N frames and assigning each GOP to a different processor. GOP-level parallelism requires

a significant amount of memory for storing all the frames, and therefore this technique maps well to multicomputers in which each processing node has a lot of computational and memory resources. Additionally, parallelization at the GOP-level results in a very high latency that cannot be tolerated in some applications. This scheme is not well suited for multi-core architectures, in which the memory is shared by all the processors, because the working set is likely larger than the last-level of shared cache.

Rodriguez et al. [81] implemented the H.264 *encoder* using GOP- (and slice-) level parallelism on a cluster of workstations using MPI. Although real-time operation can be achieved with such an approach, the latency is very high.

2.3.2.2 Frame-Level Parallelism

The next level after GOP-level is frame-level parallelism. As shown in Figure 2.4, in an I-P-B-B frame sequence inside a GOP, some frames are used as reference frames for other frames (such as I and P frames), while some other frames (B frames) are not used as reference frames. That means that different B frames can be processed in parallel. In this case, a control processor can assign independent frames to different processors. Frame-level parallelism achieves good load balancing but has scalability problems. This is due to the fact that usually there are no more than two or three B frames between P frames. This limits the amount of TLP to a few threads. The main disadvantage of frame-level parallelism, however, is that, unlike previous video standards, in H.264 B frames can be used as reference frames. In that case, the encoder cannot use B frames as reference if the decoder wants to exploit frame-level parallelism. This might increase the bitrate, but more importantly, encoding and decoding are usually completely separated and there is no way for a decoder to enforce its preferences on the encoder.

Frame-level parallelism has been implemented in the open-source encoder x264 [102] and is also described in the work of Chen et al. [20], where a combination of frame-level and slice-level parallelism is proposed. To obtain frame-level parallelism they do not allow to use B-frames as reference in the encoder and use a static I-P-B-B-P-B-B frame sequence. They obtain a 3.8× speedup on a machine with 4 cores. Their approach, however, does not scale to more processors.

2.3.2.3 Slice-Level Parallelism

In H.264 as well as in most current hybrid video coding standards, each picture is partitioned into one or more slices. Slices have been included in order to add robustness to the encoded bitstream in the presence of network transmission errors and losses. In order to accomplish this, slices in a frame should be completely independent from each other. That means that no content of a slice is used to predict elements of other slices (in the same frame) [99, 92]. Although support for slices have been designed for error resilience, it can be used for exploiting TLP because slices in a frame can be encoded or decoded in parallel. The main advantage of slices is that they do not have dependency or ordering constraints. This allows exploiting slice-level parallelism without making significant changes to the code.

There are a number of disadvantages associated with exploiting TLP at the slice-level, however. First, in H.264 the number of slices per frame is determined by the encoder. That poses a scalability problem for parallelization at the decoder level as the encoder can produce frames with only one slice. Second, H.264 includes a deblocking filter that can be applied across slice boundaries and thus making them dependent. Finally, the main disadvantage of slices is that an increase of the number of slices per frame increases the bitrate for the same quality level (or, equivalently, it reduces quality for the same bitrate level). Meenderinck et al. [66] report that when the number of slices increases to 32 slices the bitrate increase ranges from 3% to 24%, and when going to 64 slices the increase ranges from 4% to 34%. For some applications this bitrate increase is unacceptable and thus using a large number of slices to obtain high scalability is not feasible.

Several works have proposed to utilize slice-level parallelism in order to exploit TLP in the H.264 encoder and/or decoder. In [21, 20] a combination of frame-level and slice-level parallelism is proposed for IA32 Pentium processors with Simultaneous Multi-Threading (SMT) and CMP multi-threading capabilities. The proposed algorithm first exploits frame-level parallelism and when the limit of independent frames is reached, slice-level parallelism is additionally exploited. This scheme cannot scale to a large number of processors because of the limited frame-level parallelism and the coding efficiency limitations of having a large number of slices. In [82] a scheme is proposed for exploiting slice-level parallelism in the H.264 decoder by modifying the encoder. The main idea is to overcome the load balancing disadvantage by developing an encoder that produces slices that are not balanced in the number of MBs, but in their decoding time. The main disadvantages of this approach

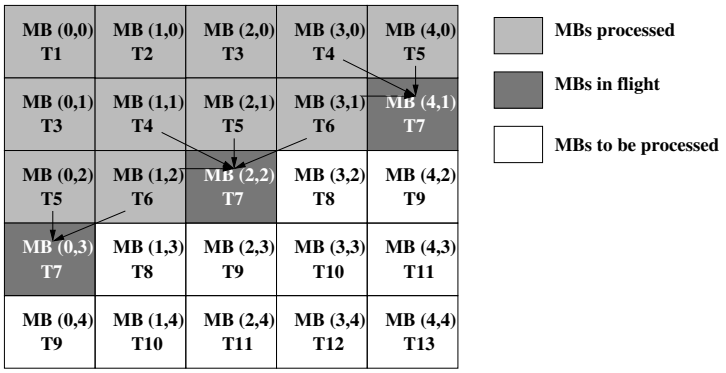


Figure 2.5: 2D-Wave approach for exploiting MB parallelism. The arrows indicate dependencies.

is that it requires modifications to the encoder in order to exploit parallelism at the decoder, and the inherent loss of coding efficiency due to having a large number of slices.

2.3.2.4 Macroblock-Level Parallelism

To exploit parallelism between macroblocks (MBs) it is necessary to take into account the dependencies between them. In H.264, motion vector prediction, intra prediction, and the deblocking filter use data from left, upper-left, upper-right, and upper neighboring MBs. MBs can be processed out of scan order provided these dependencies are satisfied. Processing MBs in a diagonal wave-front manner satisfies all the dependencies and therefore allows to exploit parallelism between MBs. We refer to this parallelization technique as 2D-Wave, to distinguish it from the 3D-Wave proposed in this chapter.

Figure 2.5 depicts an example for a 5×5 MBs image frame (80×80 pixels). Assuming it takes one time slot to decode a MB, at time slot T7 three independent MBs can be processed: MB (4,1), MB (2,2), and MB (0,3). The figure also shows the dependencies that need to be satisfied in order to process each of these MBs. The maximum number of independent MBs in a frame depends on the resolution. Figure 2.6 depicts the available MB-level parallelism over time for Standard Definition (SD), High Definition (HD), and Full High Definition (FHD) resolutions, assuming that the time to decode a MB is constant. In other words, it shows the number of MBs that can be decoded in parallel at each time slot.

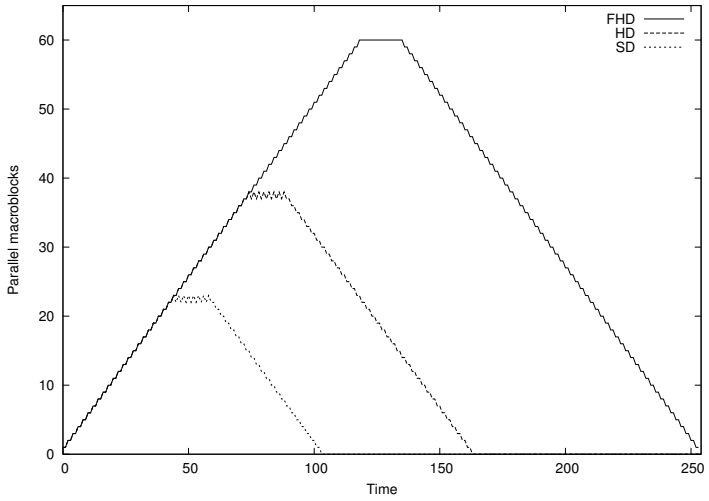


Figure 2.6: MB parallelism for a single SD, HD, and FHD frame using the 2D-Wave approach.

MB-level parallelism has several advantages over other parallelization schemes for H.264. First, this scheme potentially has good scalability since, as shown in Figure 2.6, the number of independent MBs increases with the resolution of the image. Second, it is possible to achieve good load balancing provided a dynamic scheduling system is used. A dynamic scheduling system is a system that dynamicaly evaluates dependency resolutions and is needed because the time to decode a MB is not constant and depends on the data being processed. The load would be balanced if a centralized scheduler would dynamically assign a MB to a processor once all the dependencies of the MB have been satisfied. Additionally, because in MB-level parallelization all the processors/threads run the same program, the same set of software optimizations (for exploiting ILP and SIMD) can be applied to all processing elements.

MB-level parallelism has also some disadvantages, however. The first one is that the entropy decoding cannot be parallelized using data decomposition, due to the fact that the lowest level of data that can be parsed from the bitstream are slices. Individual MBs cannot be identified without performing entropy decoding. That means that in order to decode independent MBs, they should be entropy decoded first, in sequential order. This disadvantage can be overcome by using special purpose instructions or hardware accelerators for the entropy decoding process, such as the hardware accelerator described in [75].

The second disadvantage is that the number of independent MBs does not remain constant during the decoding of a frame, as can be seen in Figure 2.6. Therefore, it is not possible to sustain a certain processing rate during the decoding of a frame. This problem is solved by using the parallelization strategy proposed in Section 2.4.

MB-level parallelism has been proposed in previous work. Van der Tol et al. [95] proposed the exploitation of MB-level parallelism for optimizing the H.264 decoding. The analysis was performed for a multiprocessor system consisting of 8 Trimedia processors with private L1 caches and a shared L2 cache. The paper also suggested the combination of MB-level with frame-level parallelism to increase the number of independent MBs, but did not analyze it thoughtfully. The use of frame-level parallelism is determined statically by the length of the MVs. Chen et al. [20] evaluated a similar approach for a H.264 *encoder*: a combination of MB- frame-level parallelism on Pentium machines with SMT and CMP capabilities. In the above mentioned works the exploitation of frame-level parallelism is limited to two consecutive frames and the identification of independent MBs is done statically by taking into account the maximum allowed MV length, which is roughly half the vertical resolution of the frame.

2.3.2.5 Block-Level Parallelism

The finest grain of data-level parallelism is at the block level. Most computations of the H.264 basic modules are performed at the block level. This applies, for example, to the interpolations performed at the motion compensation stage, to the IDCT, and to the deblocking filter. This level of data parallelism maps well to SIMD instructions [104, 90, 5, 56]. SIMD parallelism is orthogonal to the other levels of parallelism described above and because of that it can be combined, for example, with MB- and frame-level parallelism to increase the performance of each thread.

2.4 3D-Wave Strategy

None of the approaches described above scale to future many-core architectures consisting of 100 cores or more. Figure 2.6 shows that in the 2D-Wave, there is a considerable amount of MB-level parallelism, but at the beginning and at the end of processing a frame, there are only a few MBs that can be processed in parallel. In this work we propose to start decoding a MB as soon

as its reference area in the reference frame has been decoded. In other words, we combine MB-level parallelism with frame-level parallelism. We refer to our strategy as the 3D-Wave.

2.4.1 Parallelization Strategy

In the decoding process there are dependencies between frames only in the Motion Compensation (MC) module. MC can be regarded as copying an area, called the reference area, from the reference frame, and then adding this predicted area to the residual MB to reconstruct the MB in the current frame.

The reference area is pointed to by a Motion Vector (MV). The MV length is limited by two factors: the H.264 level and the motion estimation algorithm. The H.264 standard has levels which define the maximum length of the vertical component of the MV and a fixed horizontal component for all levels from -2048 to 2047.75, inclusive [2]. In the encoding process the motion estimation maximum search range is usually restricted to only dozens of pixels, because increasing the search range is very computationally demanding and provides only a small benefit [61]. Most motion estimation algorithms use some kind of strategy to decrease the search area, since exhaustive searching the full range is computationally too expensive.

When the reference area has been decoded it can be used by the referencing frame. Thus it is not necessary to wait until the entire frame, or maximum reference area, is completely decoded before decoding the next frame, as in Van der Tol et al. [95]. The decoding process of the next frame can start after the reference area of the reference frame(s) has been decoded. Figure 2.7 illustrates the 3D-Wave parallel decoding of frames, each of which is decoded using the 2D-Wave strategy.

2.4.2 3D-Wave Static Evaluation

In order to obtain a first approximation of the potential of combining MB-level parallelism with frame-level parallelism, we developed a static analysis method. This method computes the number of MBs that could be processed in parallel when decoding a video sequence, for a given MV range. In this analysis a MB can be decoded after its reference range has been decoded in the reference frame and its intra-frame dependencies has been satisfied. Figure 2.8 illustrates the reference range concept, assuming a MV range of 32 pixels. The hashed MB in Frame 1 is the MB under consideration. Its reference range is

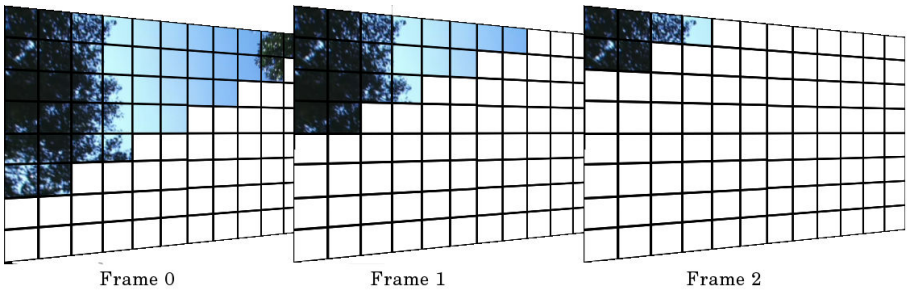


Figure 2.7: 3D-Wave strategy: Frames can be decoded partially in parallel because inter-frame dependencies have a limited range.

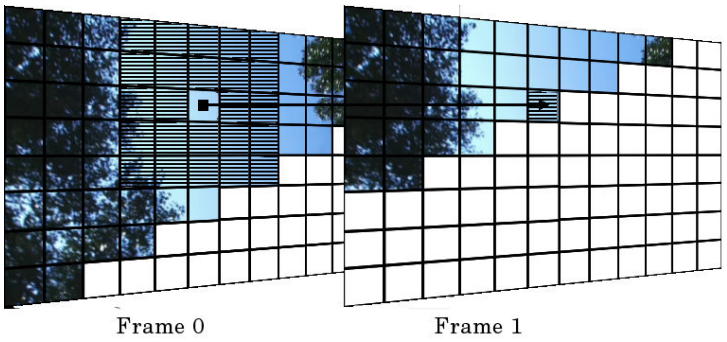


Figure 2.8: Reference range example: The hashed area in Frame 0 is the reference range of the hashed MB in Frame 1.

represented by the hashed area in Frame 0. The MV can point to any area a the range of [-32,+32] pixels, representing an offset in the vertical and horizontal directions from the MB under consideration. In the same way, every MB in the wavefront of Frame 1 has a reference range similar to the presented one, with its respective displacement.

In this analysis, we assume that all available MBs are processed in parallel and have the same processing time. Once the reference range of the first (upper-left) MB is decoded, the 2D-Wave decoding of the frame can start.

In this analysis the following conservative assumptions are made to calculate the amount of MB-level parallelism. First, B frames are used as reference frames, since in the H.264 standard, B frames can be used as reference frames. This also makes the evaluation valid for the Baseline profile of the standard, where B frames are not used. When B frames are not used as refer-

ence frames they can be decoded in parallel with other B frames and with the next P frame in the decoding order.

Second, the reference frame is always the previous one. H.264 allows multiple reference frames. In this evaluation the worst case is assumed, since having the previous frame as the reference frame limits the progression of the wave decoding of the current frame. In other words, the older the reference frame is, the further it has been decoded.

Third, only the first frame of the sequence is an I frame. As I frames do not have inter frame dependencies, they can be decoded in parallel with their preceding frames. The number of frames between I frames, however, is defined by the encoder and can be arbitrary large, e.g., 60 frames or more. Because of that, only the first frame of the video sequence is considered to be an I frame.

The static analysis methodology of the 3D-Wave strategy can be described as follows: Given a MV range, it is possible to determine the number of time steps that have to elapse between the decoding of two consecutive frames. For example, for MVs with a maximum length of 16 pixels, it is possible to start the decoding of the second frame when the MBs (0,0), (0,1), (1,0), and (1,1) of the first frame have been decoded. Of these MBs, (1,1) is the last one decoded in 2D-Wave order, namely at time $t = 3$. So when the maximum MV length is 16 pixels, we can start decoding the second frame at time $t = 4$, and the third frame at time $t = 8$, etc. Similarly, we find that for a maximum MV length of 32 pixels, we can start decoding the second frame at time $t = 7$ and the third frame at time $t = 14$. For maximum MV lengths of 64, 128, 256, and 512, we can start decoding the second frame at time $t = 13$, 25, 49, and 97, respectively. In general, for a MV range length of n pixels ($\lceil n/16 \rceil$ MBs), the decoding of the second frame can start at time $t = 1 + 3 \times \lceil n/16 \rceil$.

The number of parallel MBs available in a video sequence is calculated based on the overlap between consecutive frames (each one decoded using the 2D-Wave). The number of available parallel MBs in each time step. Let $N_{2D}(t)$ be the number of parallel MBs at time step t using the 2D-Wave. Then the number of parallel MBs $N_{3D}(t)$ at time step t of the 3D-Wave is given by

$$N_{3D}(t) = \sum_{i=0}^{t/offset} N_{2D}(t - i \times offset) \tag{2.1}$$

where $offset$ is the number of time steps that have to elapse between the decoding of two consecutive frames.

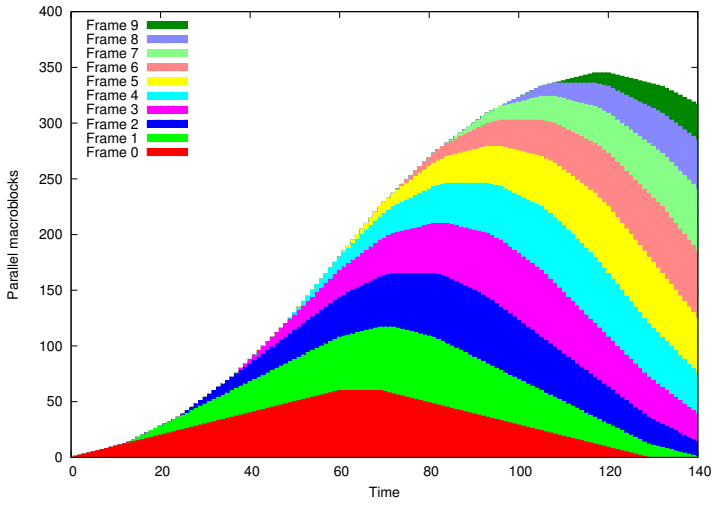


Figure 2.9: Stacking of frames of a FHD sequence with a 128 pixels maximum MV length.

Figure 2.9 depicts this function for the first 10 frames of a FHD sequence with a maximum MV length of 128 pixels. It also shows to which frame each MB belongs. It can be seen that a stacking effect occurs.

The videos from HD-VideoBench [7] benchmark was chosen because of their resolution and the content of the scenes. The benchmark contains the following test sequences:

- **rush hour:** rush-hour in Munich city; static background, slowly moving objects.
- **riverbed:** riverbed seen through waving water; abrupt and stochastic changes.
- **pedestrian area:** shot of a pedestrian area in city center; static background, fast moving objects.
- **blue sky:** top of two trees against blue sky; static objects, sliding camera.

All movies are available in three formats: 720×576 (SD), 1280×720 (HD), 1920×1088 (FHD). Each movie has a frame rate of 25 frames per second and has a length of 100 frames.

Table 2.1: Static 3D-Wave results of available parallel MBs and number of frames in flight.

Offset	Max # Parallel MBs			Max # Frames in Flight		
	SD	HD	FHD	SD	HD	FHD
512	-	40	91	-	2	3
256	34	76	169	3	4	6
128	66	145	328	5	7	11
64	126	277	629	9	13	20
32	232	515	1166	17	24	37
16	414	900	2040	29	42	64

2.4.2.1 Maximum Parallelism

Using the static analysis we determined the number of parallel MBs in the 3D-Wave for each video resolution in the HD-VideoBench, SD, HD, and FHD with MV ranges of 16, 32, 64, 128, 256, and 512 pixels. 512 pixels is the maximum vertical MV length allowed for level 4.0 in the H.264 standard.

Figure 2.10 depicts the MB-level parallelism (i.e., the number of MBs that can be processed in parallel) obtained by the static analysis of the 3D-Wave strategy for each point in time for an FHD sequence using different maximum MVs ranges. The results show that the 3D-Wave strategy is much more scalable than the 2D-Wave strategy. For example, while the 2D approach can process at most 60 MBs in parallel for FHD resolution, the 3D-Wave strategy achieves 1150 parallel MBs for a MV range of 32 pixels and more than 2000 parallel MBs for a MV range of 16 pixels. Furthermore, the 3D-Wave strategy can process more than 80 MBs in parallel even when the MV range is 512 pixels (almost half the frame height). An additional advantage of the 3D-Wave is that after a certain time, the number of parallel MBs stays constantly high. This is in contrast to the 2D-Wave strategy, which achieves little parallelism at the beginning and at the end of processing each frame.

Figure 2.11 depicts the number of frames in flight for different MV ranges. The time unit is the time required to process one MB (which is assumed to be constant in this evaluation). The shapes of the SD and HD resolutions curves are similar and are therefore not presented for brevity. Instead, Table 2.1 presents the maximum MB parallelism and number of frames in flight for SD, HD and FHD video sequences.

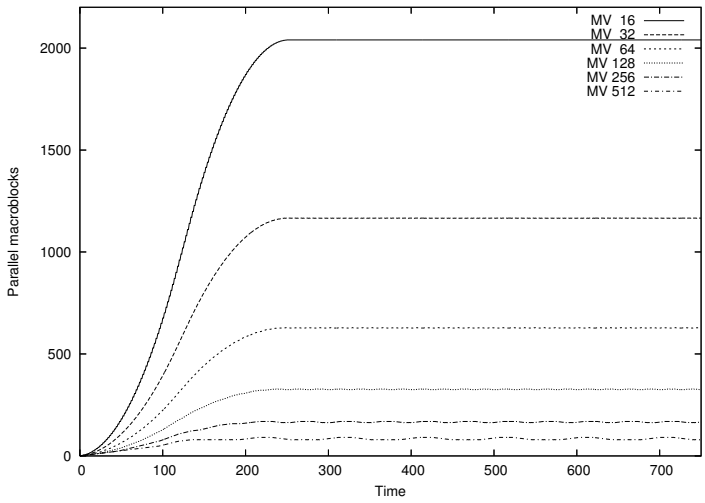


Figure 2.10: Number of parallel MBs in the 3D-Wave for FHD frames with different MV ranges.

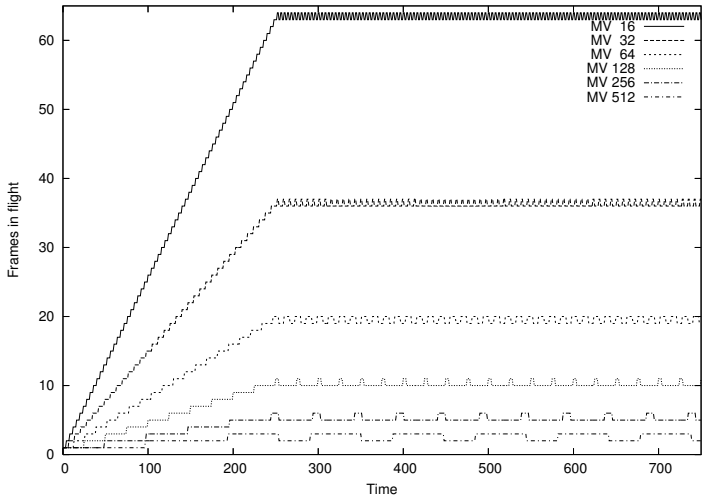


Figure 2.11: Frames in flight for FHD frames with different MV ranges.

2.5 Limited Resources

A potential problem of the 3D-Wave strategy is, however, the large number of frames that are decoded simultaneously. Figure 2.11 depicts the number of frames in flight at each point in time for an FHD sequence and different MV ranges. The figure shows that when, for example, the MV range is 16 pixels, there can be up to 64 frames in flight simultaneously. This can be a problem in systems with limited memory since each active frame has to be stored in memory. To investigate this limitation, we have also analyzed the tradeoff between the number of active frames and the amount of MB-level parallelism.

In this analysis the number of active frames is limited. The decoding of the next frame is started only if the number of active frames is less than or equal to a specified maximum number of frames in flight. If this is not the case, the decoding of the next frame has to wait until a frame is decoded.

The analysis focuses on FHD resolution with a MV range of 16 pixels. This MV range has been chosen because it clearly shows the effect of limiting the number of frames in flight, as it has the maximum amount of MB-level parallelism. The maximum number of frames in flight was set to 4, 8, 16, and 32. Figure 2.12 presents the results. For reference, the curve for the case that does not limit the number of active frames is also shown.

Limiting the number of frames in flight has two different effects on the amount of MB-level parallelism. The first one is a proportional decrease of the average number of parallel MBs. After the ramp up part of the MB parallelism curve, there are 2040 MBs available when there is no limit on the number of frames in flight in which case there are up to 64 frames in flight. Limiting the number of active frames to 32, the average number of MBs available to be processed in parallel is reduced to 1020. For 16, 8, and 4 active frames, the results show an average of 508, 253, and 126 parallel MBs, respectively.

The second effect of limiting the number of frames in flight is that the number of parallel MBs fluctuates over time, as can be seen in Figure 2.12. While there is just a very small (or no) fluctuation when there is no limit on the number of active frames, the fluctuation is large when the number of active frames is limited. This fluctuation also increases when the number of active frames decreases. For 32 active frames, after the ramp up, at most 1528 and at least 512 parallel MBs are available. Limiting the number of active frames to 16, 8, and 4, the maximum number of available MBs becomes 888, 472, 240 and the minimum is 128, 32, 8 MBs, respectively. This fluctuation is the result of the superposition of peaks in the 2D-Wave MB parallelism curves.

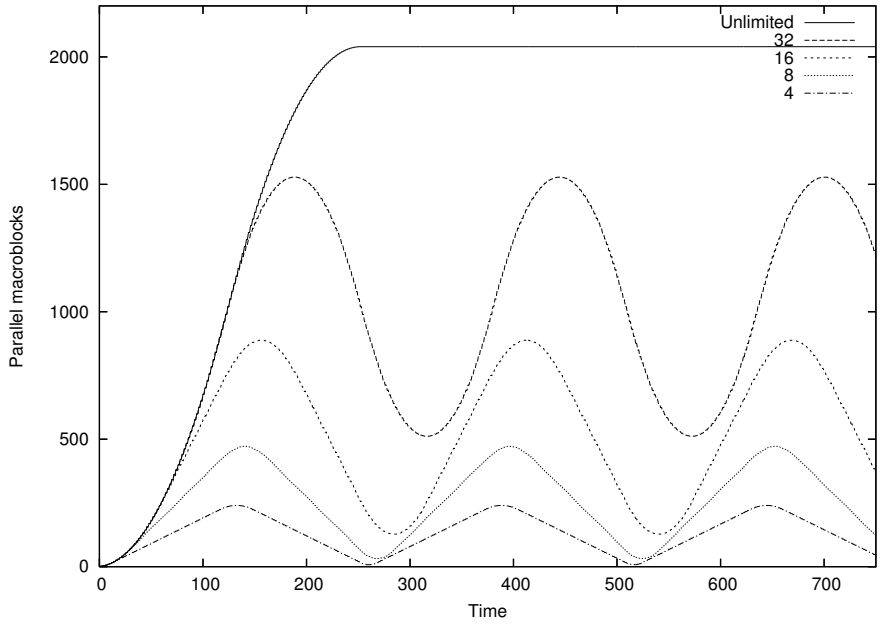


Figure 2.12: Macroblock parallelism with limited frames in flight.

The fluctuation of the number of available parallel MBs can result in underutilization of computational resources. This can happen, for instance, when the number of cores available is larger than the number of available parallel MBs at a given time.

A scheduling technique can be used to reduce the fluctuation of the amount of MB-level parallelism. To minimize the fluctuation, we should start decoding the next frame $T_{\text{frame}}/n_{\text{active}}$ time units after the time we have started decoding the current frame, where T_{frame} is the processing time of a frame and n_{active} is the maximum number of active frames. Figure 2.13 shows the curves resulting from this scheduling strategy. This strategy has a small effect on the average amount of MB-level parallelism, compared with limiting the frame parallelism without the scheduling. Table 2.2 presents the maximum, minimum, and average MB parallelism with and without scheduling. As can be seen in this table, the amount of MB-level parallelism is proportional to the number of allowed frames in parallel.

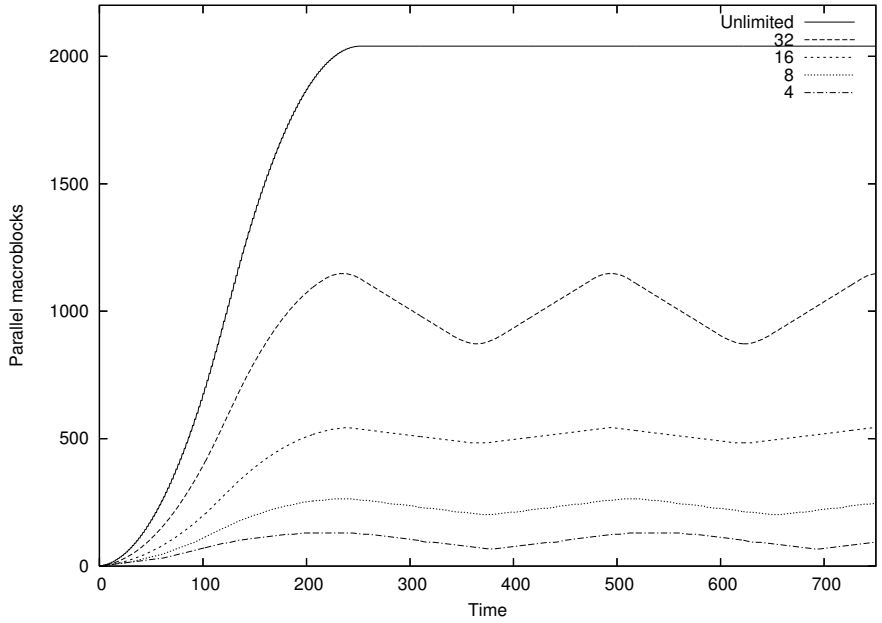


Figure 2.13: Scheduled macroblock parallelism with limited frames in flight.

Table 2.2: Number of available parallel MBs with and without the proposed scheduling technique, for FHD.

Frames in Flight	Regular				With Scheduling			
	Max	Min	Fluctuation	Avg	Max	Min	Fluctuation	Avg
32	1528	512	66%	1020	1148	872	24%	1010
16	888	128	86%	508	544	484	11%	514
8	472	32	93%	252	264	202	23%	233
4	240	8	97%	124	127	67	47%	97

2.6 Dynamic Analysis of the 3D-Wave [‡]

In the previous section we have analyzed the 3D-Wave using a static approach, considering that the MV length of all MBs is equal to a given maximum. In reality, however, the MV length of each MB may differ. For completeness and because the implementation described in the next chapter have similar behavior, in this section we briefly present a dynamic analysis of the 3D-Wave. Meenderinck et al. [66] evaluate the 3D-Wave technique using a dynamic approach. This dynamic approach computes the number of available parallel MBs considering their actual dependencies in the evaluated video sequence. As before, it is assumed that it takes one time unit to decode a MB.

To determine the amount of parallelism, the FFmpeg H.264 decoder [37] was modified to analyze the MB dependencies in real video sequences. Each MB has its dependencies analyzed and a timestamp assigned as follows. The timestamp of a MB is simply the maximum of the timestamps of all MBs upon which it depends (in the same frame as well as in the reference frames) plus one. Because the frames are processed in decoding order, and within a frame the MBs are processed from left to right and from top to bottom, the MB dependencies are observed and it is assured that the MBs on which a MB B depends have been assigned their correct timestamps by the time the timestamp of MB B is calculated.

The video sequences are from the HD-VideoBench. However, because the sequences have only 100 frames, the used input sequences are composed of the original sequences replicated four times. The encoding is performed by the X264 encoder using the following options: 2 B-frames between I and P frames (B frames cannot be reference frames), 16 reference frames, weighted prediction, hexagonal motion estimation algorithm (hex) with a maximum search range of 24, one slice per frame, and adaptive block size transform. Movies encoded with this set of options represent the typical case.

Figure 2.14 depicts the MB-level parallelism time curve for FHD, while Figure 2.15 depicts the number of frames in flight in each time slot. The results show peaks of 4.000 to 7.000 parallel MBs. The maximum number of frames in flight is just above 200 for all sequences.

These results show that much more parallelism is available in actual video sequences than indicated by the static analysis. For example, the static analysis reported a maximum of 2.040 parallel MBs for a FHD sequence while the dynamic analysis reports between 2.000 to 4.000 parallel MBs, after ramp-

[‡]This section is based on work performed by Cor Meenderinck.

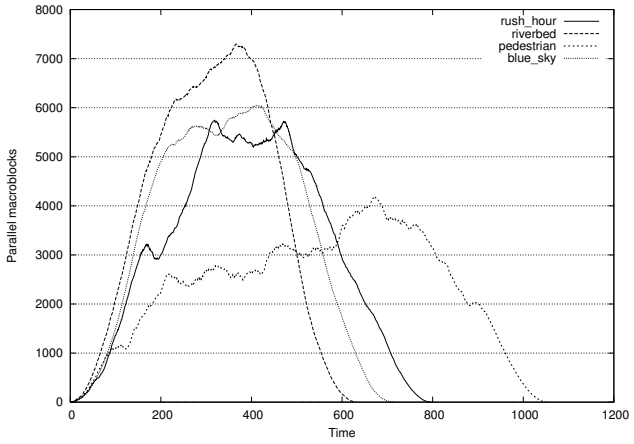


Figure 2.14: Number of parallel MBs in each time slot for the 3D-Wave, using the dynamic analysis, for FHD resolution using a 400-frame sequence.

up, for the pedestrian sequence, which is the sequence with the lowest number of parallel MBs among the tested sequences. This is due to two factors. First, the static evaluation only considers inter-dependent frames while the dynamic evaluation uses 2 B frames between P frames. As the 2 B frames are independent from each other, they can be processed in parallel as well as the next P frame. This is a three fold increase in the number of MBs and frames in parallel. Second, the smallest range for the static evaluation considers a maximum MV length of 16 pixels, while the average MV length in the evaluated sequences for the dynamic evaluation is about 5 pixels. As reported in the previous section, the number of parallel MB increases when the MV length decreases.

This study adds more precision to the static analysis presented earlier and confirms our findings. However, it still presents limitations. The MB decoding time is assumed to be fixed while in reality this is not the case. By itself this reduces the available MB parallelism by 33% [3]. Another limitation is that our analysis does not include the synchronization overhead. Synchronization overhead can be a major problem in a parallel implementation of the decoder [4]. Moreover, an implementation mechanism is not proposed. These limitations will be addressed in our implementation of the 3D-Wave on a many-core processor in the next chapter.

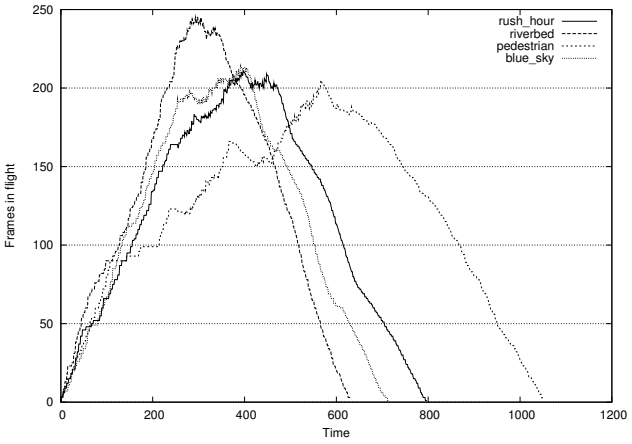


Figure 2.15: Number of frames in flight in each time slot for the 3D-Wave, using the dynamic analysis, for FHD resolution using a 400-frame sequence.

2.7 Conclusions

In this chapter we have investigated if contemporary and perhaps future video applications exhibit sufficient parallelism to exploit the large number of cores expected in future many-core processors. As a case study, we have analyzed the parallel scalability of the H.264 video decoder, currently the leading video coding standard.

First, we have discussed different ways to parallelize H.264 and showed that slice-level parallelism has two main limitations. First, using many slices increases the bitrate and, second, not all sequences contain sufficient slices to scale to a large number of cores, since the encoder determines the number of slices per frame. It was also shown that frame-level parallelism, which exploits the fact that some frames (B frames) are not used as reference frames and can therefore be processed in parallel, is also not very scalable, because usually there are no more than three B frames between consecutive P frames and, furthermore, in H.264 B frames can be used as reference frames.

More promising is MB-level parallelism, which can be exploited inside a frame in a diagonal wavefront manner. Intra-frame MB-level parallelism, however, exhibits at most 60 parallel MBs, which is not sufficient for a many-core. It was observed that MBs in different frames are only dependent through MVs which have a limited range. We therefore proposed a novel parallelization strategy, called 3D-Wave, which combines MB-level parallelism with frame-

level parallelism. The MB-level parallelism available in the 3D-Wave was analyzed using a static and a dynamic approach. In the static approach, for FHD resolution, the results show that when the MV range is limited to 16 pixels, up to 2040 MBs can be processed in parallel. In the worst case, when the MV is at most 512 pixels, at most 91 parallel MBs are available, which is still much more than the number of parallel MBs other parallelization approaches reach. Furthermore, we showed that the fluctuation of the number of parallel MBs can be reduced by applying a simple scheduling strategy. Nevertheless, this static analysis is conservative since it is based on the assumption that the MV length of each MB is equal to the maximum MV length. A dynamic evaluation of the 3D-Wave showed that in real video sequences with regular coding options, there is even more MB-level parallelism than the static evaluation suggests.

3

3D-Wave Implementation

In the previous chapter we have proposed the 3D-Wave parallelization strategy for H.264 video decoding. It has been shown that the 3D-Wave strategy potentially scales to a much larger number of cores than previous strategies do. However, we have done so using a very high-level evaluation methodology, assuming constant macroblock (MB) decoding time and zero communication and synchronization overhead. In this chapter we remedy these limitations by presenting an implementation of 3D-Wave parallelization strategy on an embedded many-core with up to 64 cores.

This chapter is organized as follows. Section 3.1 motivates the presented implementation and details the contributions of this chapter. In Section 3.2 the simulation environment and the experimental methodology used to evaluate the proposed 3D-Wave implementation are presented. In Section 3.3 the implementation of the 3D-Wave on the embedded many-core is detailed. This section also presents a frame scheduling policy used to limit the number of frames in flight and the priority policy used to reduce latency. Additionally, a 3D-Wave visualization tool is also described in Section 3.3. Section 3.4 presents extensive experimental results analyzing the following aspects: scalability and performance of the 3D-Wave, the policies for frame scheduling and priority, the impacts of the memory latency, L1 cache size, parallelization overhead, and entropy decoding. Conclusions are drawn in Section 3.5.

3.1 Introduction

In the previous chapter we have proposed the 3D-Wave parallelization strategy for H.264 video decoding. It has been shown to potentially scale to a much larger number of cores than existing strategies do. However, the previous chapter presented only analytical results. It analyzed how many macroblocks could be processed in parallel assuming infinite resources, no communication delay, infinite bandwidth, and a constant MB decoding time.

In this chapter we present an implementation of the 3D-Wave strategy on an embedded many-core consisting of up to 64 cores. Implementing the 3D-Wave turned out to be quite challenging. First, it requires to dynamically identify inter-frame MB dependencies and handle their thread synchronization, in addition to intra-frame dependencies and synchronization. In our previous analytical analysis thread synchronization was not an issue because the MBs were visited sequentially and assigned a timestamp representing the earliest time slot in which they could be processed. For the 3D-Wave strategy both intra- and inter-frame synchronization have to be taken into account in a way that the synchronization overhead does not limit the scalability, for the target number of cores. This led to the development of a subscription mechanism where the current MB checks if the MBs it depends on are already processed. If one of the MBs is not yet processed, the current MB subscribes itself to a so-called *Kick-off List* (KoL) associated with the MB it depend on and halts. The processing of halted MBs is restarted when the current MB, after finishing its processing, resumes the processing of MBs subscribed to its KoL.

A potential drawback of the 3D-Wave strategy is that the decoder latency may become unbounded because many frames are decoded simultaneously. A policy is presented that gives priority to the oldest frame so that newer frames are only decoded when there are idle cores. Another potential drawback of the 3D-Wave strategy is that the memory requirements might increase because of the large number of frames in flight. To overcome this drawback we present a frame scheduling policy to control the number of frames in flight.

The effect of memory system, thread synchronization, and entropy decoding in the 3D-Wave scalability are also analyzed in this Chapter. Memory requirements are an issue when dealing with parallel applications. We analyze the impact of the memory latency and the L1 cache size on the scalability and performance of the 3D-Wave strategy. The experimental platform features hardware support for thread management and synchronization, making it relatively light weight to submit/retrieve a task to/from a task pool. We analyze the

importance of this hardware support by artificially increasing the time it takes to submit/retrieve a task. The 3D-Wave focuses on the MB decoding part of the H.264 decoding and assumes an accelerator for entropy decoding. Finally, we analyze the performance required from the entropy decoding accelerator to avoid affecting the 3D-Wave scalability.

3.2 Experimental Methodology

In this section the tools and methodology used to implement and evaluate the 3D-Wave technique are detailed. The components of the many-core system simulator used to evaluate the technique are also presented.

An NXP proprietary simulator based on SystemC is used to run the application and collect performance data. Computations on the cores are modeled cycle-accurate. The memory system is modeled using average transfer times with channel and bank contention. When channel or bank contention is detected, the traffic latency is increased. NoC contention is also modeled. The simulator is capable of simulating systems with up to 64 TM3270 [94] cores with shared memory and their cache coherence protocol. The operating system is not simulated.

The TM3270 is a Very Large Instruction Word (VLIW) media-processor based on the Trimedia architecture. It is targeted at the requirements of multi-standard video processing at standard resolution and the associated audio processing requirements for the consumer market. The architecture supports VLIW instructions with five guarded issue slots. The pipeline depth varies from 7 to 12 stages. Address and data words are 32 bits wide. The unified register file has 128 32-bit registers. 2×16-bit and 4×8-bit SIMD instructions are supported. To produce code for the TM3270 the state-of-the-art highly optimizing NXP TriMedia C/C++ compiler version 5.1 is used. For our experiments, the TM3270 processor clock frequency was set to 300 MHz.

The modeled system features a shared memory using the MESI cache coherence protocol. Each core has a private L1 data cache and can copy data from other L1 caches through 4 channels. The 64KB L1 data cache has 64-byte lines and is 4-way set-associative with LRU replacement and write allocate policies. The instruction cache is not modeled. It is not considered to be a limitation because as the code is concise and highly optimized, it would fit in a regular sized instruction cache. The cores share a distributed L2 cache with 8 banks and an average access time of 40 cycles. The average access time takes into account L2 hits, misses, and interconnect delays. L2 bank

contention is modeled in the simulator, thus two cores cannot access the same bank simultaneously.

The multi-core programming model follows a task pool model. A Task Pool (TP) library implements submissions and requests of tasks to/from the task pool, synchronization mechanisms, and the task pool itself. Each core runs a thread by requesting a task from the TP, executing it, possibly identifying and submitting tasks to the TP, and requesting another task. The task execution overhead is low. The time to request a task is less than 2% of the MB decoding time.

The experiments focus on the baseline profile of the H.264 standard. This profile only supports I and P frames and every frame can be used as a reference frame. This feature prevents the exploitation of frame-level parallelization techniques such as the one described in [20]. This profile, however, highlights the advantages of the 3D-Wave, since the scalability gains come purely from the application of the 3D-Wave technique. Encoding was done with the X264 encoder [102] using the following options: no B-frames, at most 16 reference frames, weighted prediction, hexagonal motion estimation algorithm with a maximum search range of 24, and one slice per frame. The experiments use all four videos from the HD-VideoBench [7].

The 3D-Wave technique focuses on the TLP available in the MB processing kernels of the decoder. The entropy decoder is known to be challenging to parallelize. To isolate the influence of the entropy decoder, its output has been buffered and its decoding time is not taken into account. Although it is not the main objective, as a side effect the 3D-Wave also eases the entropy decoding challenge. Since entropy decoding dependencies do not cross slice/frame borders, multiple entropy decoders can be used. We analyze the performance required from an entropy decoder accelerator in Section 3.4.7.

3.3 Implementation

Our work is based on the NXP H.264 decoder. The 2D-Wave parallelization strategy has already been implemented in this decoder [46], making it a perfect starting point for the implementation of the 3D-Wave. The NXP H.264 decoder is highly optimized, including both machine-dependent optimizations (e.g. SIMD operations) and machine-independent optimizations (e.g. code restructuring).

The 3D-Wave implementation serves as a proof of concept and thus the implementation of all features of H.264 is not necessary. Intra prediction inputs are deblock filtered samples instead of unfiltered samples as specified in the standard. This does not, however, add perceptive visual artifacts to the decoded frames or change the MB dependencies.

This section describes the 2D-Wave implementation used as the starting point, the 3D-Wave implementation, and the frame scheduling and priority policies.

3.3.1 2D-Wave Implementation

The MB processing tasks of the 2D-Wave implementation consists of four kernels that are grouped slightly different than previously presented. The kernels are vector prediction (VP), picture prediction (PP), deblocking info (DI), and deblocking filter (DF). VP calculates the motion vectors (MVs) based on the predicted motion vectors of the neighbor MBs and the differential motion vector present in the bitstream. PP performs the reconstruction of the MB based on neighboring pixel information (Intra Prediction) or on reference frame areas (Motion Compensation). Inverse quantization and the inverse DCT are also part of this kernel. DI calculates the strength of the DF based on MB data, such as the MBs type and MVs. Finally, DF smooths block edges to reduce blocking artifacts.

Each kernel is parallelized using the 2D-Wave strategy. As shown in Figure 2.5 on page 21, within a frame each MB depends on at most four other MBs. These dependencies are covered by the dependencies from the left MB to the current MB and from the upper-right MB to the current MB. In other words, if the left and the upper-right dependencies are satisfied then all dependencies are satisfied. Therefore, in the implementation each MB is associated with a reference count between 0 and 2 representing the number of MBs it depends on. For example, the upper-left MB has a reference count of 0, the other MBs at the top edge and at the left edge have a reference count of 1. When a MB has been processed, the reference counts of the MBs that depend on it are decremented. When one of these counts reaches zero, a thread that will process the associated MB is submitted to the TP. Figure 3.1 depicts pseudo C-code for applying the DF on a frame and on a MB. `tp_atomic_decrement` atomically decrements the counter and returns its value. `tp_submit` submits a function and its parameters to the TP.

```
int deblock_ready[w][h]; // matrix of reference counts

void deblock_frame() {
   for (x=0; x<w; x++)
      for (y=0; y<h; y++)
         deblock_ready[x][y] = initial reference count;
                                          // 0, 1, or 2
   tp_submit(deblock_mb, 0, 0); //start 1st task MB<0,0>
   tp_wait();
}

void deblock_mb(int x, int y){
   // ... the actual work

   if (x!=0 && y!=h-1){
      new_value =
       tp_atomic_decrement(&deblock_ready[x-1][y+1], 1);
      if (new_value==0)
         tp_submit(deblock_mb, x-1, y+1);
   }
   if (x!=w-1){
      new_value =
         tp_atomic_decrement(&deblock_ready[x+1][y], 1);
      if (new_value==0)
         tp_submit(deblock_mb, x+1, y);
   }
}
```

Figure 3.1: Pseudo-code for deblocking a frame and a MB.

When a core loads a MB in its cache, it also fetches the data of neighboring MBs. Therefore, locality can be improved if the same core also processes the right MB. To increase locality and reduce task submission and acquisition overhead, the baseline 2D-Wave implementation features an optimization called *tail submit*. After the MB is processed, the reference counts of the MB candidates are checked. If both MB candidates are ready to execute, the core processes the right MB and submits the other one to the task pool. If only one MB is ready, the core starts its processing without submitting or acquiring tasks to/from the TP. In case there is no neighboring MB ready to be processed, the task finishes and the core requests another one from the TP. Figure 3.2 depicts pseudo-code for MB decoding with the tail submit optimization. If the `deblock_ready` counter reaches zero, the MB dependencies are met.

```
void deblock_mb(int x, int y){
again:
   // ... the actual work

   ready1 = x>=1 && y!=h-1 &&
      tp_atomic_decrement(&deblock_ready[x-1][y+1])==0;
   ready2 = x!=w-1 &&
        tp_atomic_decrement(&deblock_ready[x+1][y])==0;

   if (ready1 && ready2){
      tp_submit(deblock_mb, x-1, y+1);
                                // submit left-down block
      x++;
      goto again;               // goto right block
   }
   else if (ready1){
      x--; y++;
      goto again;               // goto left-down block
   }
   else if (ready2){
      x++;
      goto again;               // goto right block
   }
}
```

Figure 3.2: Tail submit.

3.3.2 3D-Wave Implementation

In this section the 3D-Wave implementation is described. First we note that the original per-kernel structure of the decoder is not suitable for the 3D-Wave strategy, because inter-frame dependencies are satisfied only after the DF is applied. To implement the 3D-Wave, it is necessary to develop a version in which the kernels are applied on a MB basis rather than on a slice/frame basis. In other words, we need a `decode_mb` function in which the MB kernels are fused.

Since the 3D-Wave implementation decodes multiple frames concurrently, modifications to the Reference Frame Buffer (RFB) are required. The RFB stores the decoded frames that are going to be used as reference. As it can serve only one frame in flight, the 3D-Wave would require multiple RFBs. In this proof of concept implementation, the RFB was modified such that a

```
void decode_mb(int x, int y, int skip, int RMB_start){
    if (!skip) {
        Vector_Prediction(x,y);
        RMB_List = RMB_Calculation(x,y);
    }
    RMB = RMB_List.table[RMB_start];
    while (RMB != RMB_List.table[RMB_last]) {
        if (!RMB.Ready) {
            RMB.Subscribe(x, y);
            return;
        }
        RMB = RMB.next();
    }
    Picture_Prediction(x,y);
    Deblocking_Info(x,y);
    Deblocking_Filter(x,y);
    Ready[x][y] = true;

    MB = KoL.start;
    while (MB != KoL.last){
       tp_submit(
             decode_mb, MB.x, MB.y, true, MB.RMB_start);
       MB = MB.next;
    }
    //TAIL_SUBMIT
}
```

Figure 3.3: Pseudo-code for the decode_mb function of the 3D-Wave.

single instance can serve all frames in flight. In the new RFB all the decoded frames are stored. The mapping of the reference frame index to RFB index was changed accordingly.

Figure 3.3 depicts pseudo-code for the `decode_mb` function of the 3D-Wave implementation. It relies on the ability to test if the reference MBs (RMBs) of the current MB have already been decoded or not. The RMB is defined as the MB in the bottom right corner of the reference area, including the extra samples for fractional motion compensation. To be able to test this, first the RMBs have to be calculated. If an RMB has not been processed yet, a method is needed to resume the execution of this MB after the RMB is ready.

The RMBs can only be calculated after motion vector prediction, which also defines the reference frames. Each MB can be partitioned in up to four

8×8 pixel areas and each one of them can be partitioned in up to four 4×4 pixel blocks. The 4×4 blocks in an 8×8 partition share the reference frame. With the MVs and the reference frames information, it is possible to calculate the RMB of each MB partition. This is done by adding the MV, the size of the partition, the position of the current MB, and the additional area for fractional motion compensation, and by dividing the result by 16, the size of the MB. The RMB results of each partition is added to a list associated with the MB data structure, called the `RMB-list`. To reduce the number of RMBs to be tested, the reference frame of each RMB is checked. If two RMBs are in the same reference frame, only the one with the larger 2D-Wave decoding order (see Figure 2.5 on page 21) is added to the list.

To reduce the overhead, the first time `decode_mb` is called for a specific MB it is called with the parameter `skip` set to `false` and `RMB_start` set to `0`. If the decoding of this MB is resumed, it is called with the parameter `skip` set to `true`. Also `RMB_start` carries the position of the MB in the RMB-list to be tested next.

Once the RMB-list of the current MB is computed, it is verified if each RMB in the list has already been decoded or not. Each frame is associated with a MB ready matrix, similar to the `deblock_ready` matrix in Figure 3.1. The corresponding MB position in the ready matrix associated with the reference frame is atomically checked. If all RMBs are decoded, the decoding of this MB can continue.

To handle the cases where an RMB is not ready, an RMB subscription technique has been developed. The technique was motivated by the specifics of the TP library, such as low thread creation overhead and no sleep/wake up capabilities. Each MB data structure has a second list called the *Kick-off List* (KoL) that contains the parameters of the MBs subscribed to this RMB. When an RMB test fails (i.e., it has not been decoded yet), the current MB subscribes itself to the KoL of the RMB and finishes its execution. Furthermore, each MB, after finishing its processing, indicates that it is ready in the ready matrix and verifies its KoL. A new task is submitted to the TP for each MB in the KoL. The subscription process is repeated until all RMBs are ready. Finally, the intra-frame MBs that depend on this MB are submitted to the TP using tail submit, identical to Figure 3.2.

Figure 3.4 illustrates this subscription mechanism. Light gray boxes represent decoded MBs and dark gray boxes MBs that are currently being processed. Hatched boxes represent MBs available to be decoded, while white boxes represent MBs whose dependencies have not yet been resolved. In this

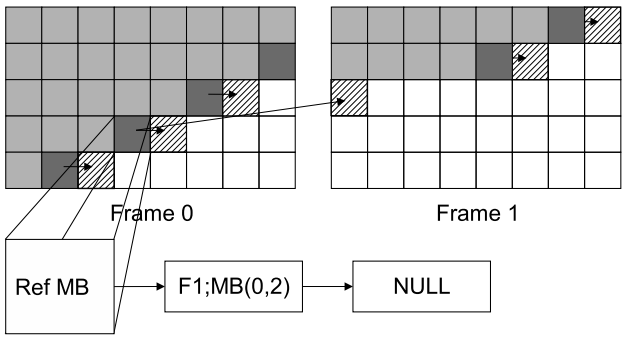


Figure 3.4: Illustration of the 3D-Wave and the subscription mechanism.

example MB(0,2) of frame 1 depends on MB(3,3) of frame 0 and is subscribed to the KoL of the latter. When MB(3,3) is decoded it submits MB(0,2) to the task pool.

3.3.3 Frame Scheduling Policy

To achieve the highest speedup, all frames of the sequence are scheduled to run as soon as their dependencies are met. This, however, can lead to a large number of frames in flight and large memory requirements, since every frame must be kept in memory. Mostly, it is not necessary to decode a frame as soon as possible to keep all cores busy. The frame scheduling technique presented in Section 2.5, was adapted to keep the working set to its minimum.

Frame scheduling uses the RMB subscription mechanism to define the moment when the processing of the next frame should be started. In this policy, the first MB of the next frame can be subscribed to start after a specific MB of the current frame. With this simple mechanism it is possible to control the number of frames in flight. Adjusting the number of frames in flight is done by selecting an earlier or later MB with which the first MB of the next frame will be subscribed.

3.3.4 Frame Priority

Latency is an important quality characteristic of video decoding systems. The frame scheduling policy described in the previous section reduces the frame latency compared to the baseline 3D-Wave implementation, since the next frame

is scheduled only when a part of the current frame has been decoded. When a new frame is scheduled to be decoded, however, the available cores are distributed equally among the frames in flight. A priority mechanism was added to the TP library in order to reduce the frame latency even further.

The TP library was modified to support two levels of priority. An extra task buffer was implemented to store high priority tasks. When the TP receives a request for a task, it first checks if there is a task in the high priority buffer. If so this task is selected, otherwise a task in the low priority buffer is selected. With this simple mechanism it is possible to give priority to the tasks belonging to the frame "next in line". Before submitting a new task, the process checks if its frame is the frame "next in line". If so the task is submitted with high priority. Otherwise it is submitted with low priority. This mechanism does not lead to starvation because if there is insufficient parallelism in the frame "next in line", the low priority tasks are selected.

3.3.5 3D-Wave Viewer

A tool to visualize the 3D-Wave strategy, called 3D-Wave Viewer, was developed in order to help with the development of the 3D-Wave itself as well as the frame scheduling and frame priority policies. The 3D-Wave Viewer enables a more in depth analysis of the temporal behavior of the 3D-Wave decoding process and the identification of bugs in the frame scheduling and priority policies.

The 3D-Wave Viewer uses trace information from the execution of the 3D-Wave H.264 decoder. The trace contains the starting and finishing time of the decoding of each MB of the input sequence as well as the MB frame number and its vertical and horizontal coordinates. The tool reads the trace file and orders the MBs by their decoding start time. Following the decoding order, the tool draws a black square to represent a decoded MB at adjustable intervals. Rectangular white areas are used to depict frames. A screenshot of the 3D-Wave Viewer is presented in Figure 3.5. The figure shows the decoding of the first 12 frames of the Rush Hour sequence by the 3D-Wave strategy on the 64-core processor without frame scheduling or priority. The screenshot shows that at this point in time 10 frames are decoded in parallel (frame 3 to frame 12), and that the decoding of each frame stays a little "behind" the decoding of the previous frame, as expected.

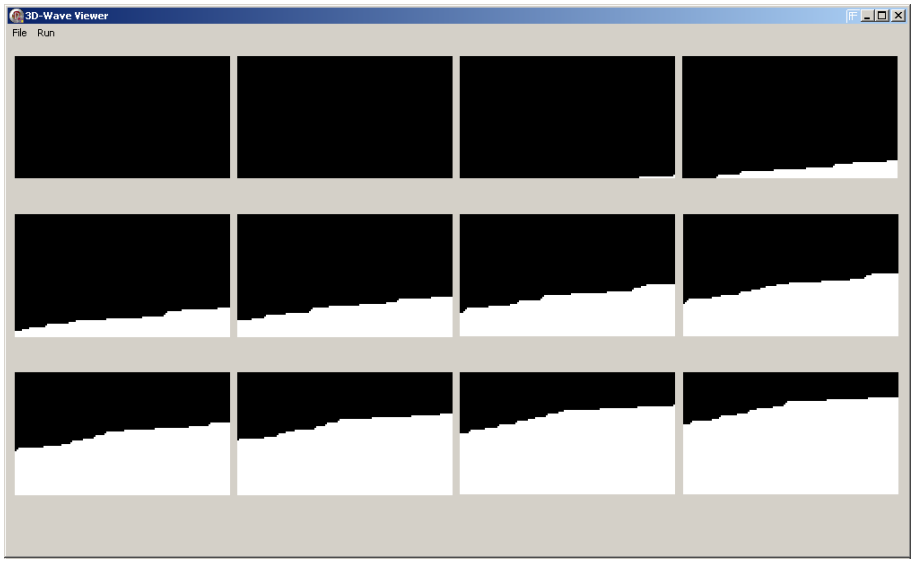


Figure 3.5: Screenshot of the 3D-Wave Viewer.

3.4 Experimental Results

In this section, the experimental results are presented. The results include the scalability results of the 3D-Wave (Section 3.4.1), results of the frame scheduling and priority policies (Section 3.4.2), memory and bandwidth requirements (Section 3.4.3), influence of memory latency (Section 3.4.4), influence of L1 data cache size on scalability and performance (Section 3.4.5), the impact of parallelism overhead on scalability (Section 3.4.6), and the requirements for the CABAC accelerator to leverage a 64-core system (Section 3.4.7).

To evaluate the 3D-Wave, one second (25 frames) of each sequence was decoded. Longer sequences could not be used due to the maximum amount of memory the simulator could allocate was insufficient. The four sequences of the HD-VideoBench using three resolutions were evaluated.

3.4.1 Scalability

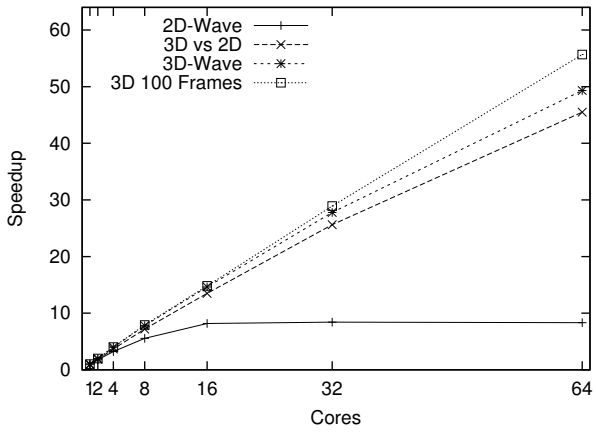
The scalability results are presented for 1 to 64 cores. Initially, the simulator supported up to 32 cores. It was extended to support 64 cores but more cores could not be simulated due to the required modifications in the compiler to support larger number of cores. Figures 3.6(a), 3.6(b), and 3.6(c) depict for

SD, HD, as well as FHD, the scalability of the 3D-Wave implementation on p processors ($T_{3D}(1)/T_{3D}(p)$), 2D-Wave scalability ($T_{2D}(1)/T_{2D}(p)$), and speedup of the 3D-Wave over the 2D-Wave on a single core ($T_{2D}(1)/T_{3D}(p)$), labeled as 3D vs 2D. Figure 3.6(c) depicts the scalability of the 3D-Wave for each of the HD-VideoBench sequences, Rush Hour (labeled RH), Riverbed (RB), Blue Sky (BS), and Pedestrian Area (PA). Since the result for the Rush Hour sequence are close to the average and other sequences differ less than 5%, only its results are analyzed and depicted in the remaining figures, including Figures 3.6(a) and 3.6(b). On a single core, the 2D-Wave can decode 39 SD, 18 HD, and 8 FHD frames per second.

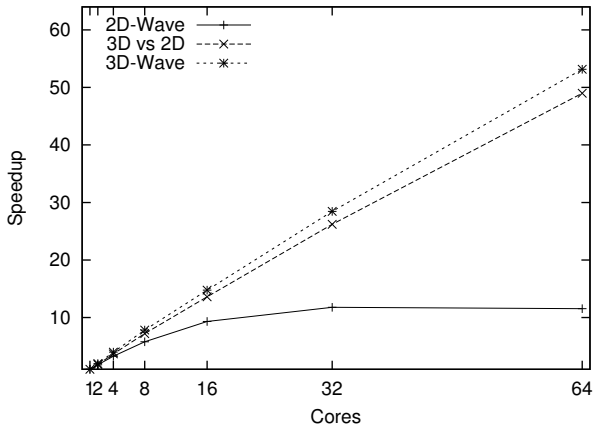
On a single core the 3D-Wave implementation takes about 8% more time than the 2D-Wave implementation due to the additional administrative overhead. The 3D-Wave scales almost perfectly up to 8 cores, while the 2D-Wave incurs an 11% efficiency drop already for 2 cores due to the following reason. The tail submit optimization attempts to assign a line of MBs to a single core. At the end of processing a frame, however, when a core finishes its line and there is no other line to be decoded, in the 2D-Wave the core remains idle until all cores have finished their line. If the last line happens to be slow, the other cores wait for a long time, and the core utilization is low. In the 3D-Wave, cores that finish their line, when there is no new line to be decoded in the current frame, will be assigned a line of the next frame. Therefore, the core utilization as well as the scalability of the 3D-Wave is higher. Another advantage of the 3D-Wave over the 2D-Wave is that it increases the efficiency of the Tail Submit optimization. In the 2D-Wave the relatively low growth of thread-level parallelism causes the cores to stall more often due to unsolved intra-frame dependencies. In the 3D-Wave, the available parallelism is much larger, which increases the distance between the MBs being decoded (as depicted in Figure 3.5), minimizing intra-frame dependency stalls.

For SD sequences, the 2D-Wave technique saturates at 16 cores, with a speedup of only 8.4 (Figure 3.6(a)). This happens because of the limited amount of MB-level parallelism inside a frame and the dominant ramp up and ramp down of the number of parallel MBs (see Figure 2.6 on page 22). The 3D-Wave technique for the same resolution continuously scales up to 64 cores, with almost 80% of the linear scalability. For the FHD sequence, the saturation of the 2D-Wave occurs at 32 cores with a speedup of 14.4, while the 3D-Wave continuously scales up to 64 cores, with 85% of the linear scalability.

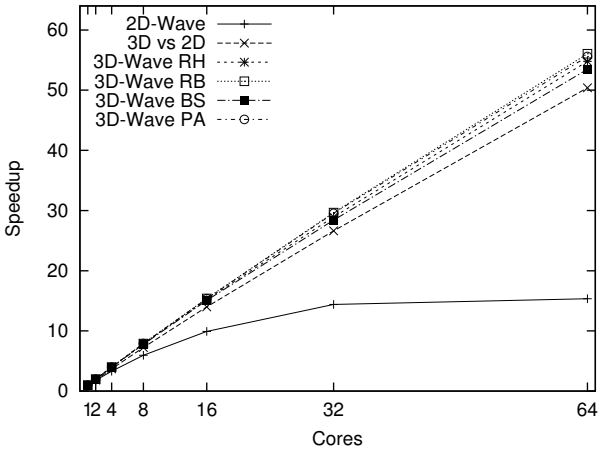
The scalability of the 3D-Wave increases slightly for higher resolutions, the 2D-Wave implementation, on the other hand, achieves higher speedups



(a) SD.



(b) HD.



(c) FHD.

Figure 3.6: 2D-Wave and 3D-Wave speedups for the 25-frame sequence Rush Hour for different resolutions.

for higher resolutions, since the amount of MB-level parallelism increases. It would take, however, an extremely large resolution for the 2D-Wave to leverage 64 cores, and the 3D-Wave implementation would still be more efficient.

The slightly less than perfect speedup of the 3D-Wave for larger numbers of cores has two reasons. First, cache trashing occurs for large numbers of cores, leading to a significant number of memory stalls, as will be shown in the next section. Second, at the start and at the end of a sequence, not all cores can be used because little parallelism is available. The more cores are used, the more cycles are relatively wasted during these two periods. The ramp up and ramp down effects on the 3D-Wave would be negligible in a real sequence with many frames. To demonstrate this Figure 3.6(a) also shows the scalability results for 100 frames of the Rush Hour SD sequence. Simulations with HD or FHD sequences with more than 25 frames are not possible because the simulator cannot allocate the required data structures.

For 64 cores the scalability grows from 49.3 to 55.7 when processing 100 instead of 25 SD frames. The effects of ramp up and ramp down times are minimized when more frames are used. In this case, the scalability results are closer to the results that would be achieved in a real life situation.

The overall performance of the 64-core system is sufficient to process FHD sequences about 16 times faster than real-time performance in frames per second. This achieved performance is important for processing future video applications. For instance, the processing of stereoscopic 4K resolution sequences would require such computational power [27].

As previously mentioned, this study focuses on the baseline profile sequence where all frames can be used as reference frames. Other H.264 profiles use B frames which are not commonly used as reference frames. B frames can be started simultaneously together with the following I or P frames for cases where B frames are not marked as references. Depending on the number of B frames, the available MB parallelism can be multiplied by two or three, for IBPBP and IBBPBB sequences, respectively.

3.4.2 Frame Scheduling and Priority

In this section, experimental results for the frame scheduling and priority policies are presented. The effectiveness of these policies is presented first, followed by their impact on the 3D-Wave efficiency.

Figure 3.7 presents the results of the frame scheduling technique applied to the FHD Rush Hour sequence using a 16-core system. This figure presents

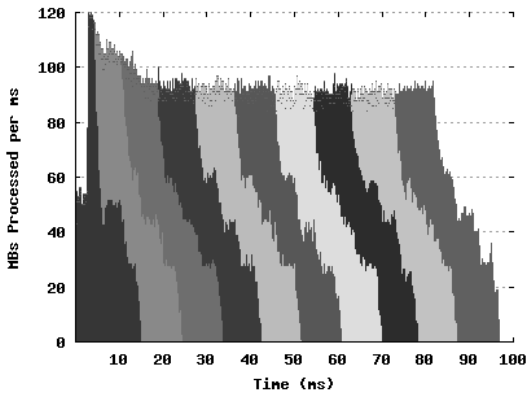


Figure 3.7: Number of MBs processed per ms using frame scheduling for FHD Rush Hour on a 16-core processor. Different gray scales represent different frames.

the number of MBs processed per ms. Furthermore, the different colors show to which frame the processed MBs belong. In this particular case, processing of the next frame is started after the last MB on the MB line that is at 1/3rd of the frame vertical resolution is decoded. For this configuration there are at most 3 frames in flight. Currently, the selection of the MB that will kick-off the processing of the next frame must be done statically by the programmer. A methodology to dynamically fire new frames based on core utilization remains to be developed.

In the original 3D-Wave implementation, the latency of the first frame is 58.5 ms, using the FHD Rush Hour sequence with 16 cores. Using the frame scheduling policy, the latency drops to 15.1 ms. This latency is further reduced to 9.2 ms when the priority policy, presented in Section 3.3.4, is applied together with frame scheduling. This is only 0.1 ms longer than the latency of the 2D-Wave, which decodes frames one-by-one. Figure 3.8 depicts the number of MBs processed per ms when this feature is used.

Four configurations are used to analyze the impact of frame scheduling and priority on the scalability. These configurations use 3 and 6 frames in flight, with and without frame priority. Figures 3.9(a), 3.9(b), and 3.9(c) depict the impact of the presented techniques on the scalability for SD, HD, and FHD resolutions, respectively. The baseline 2D-Wave (labeled 2DW) and 3D-Wave (labeled 3DW) scalability results are presented for reference. In Figure 3.9, FS refers to the frame scheduling. The addition of frame priority has no significant

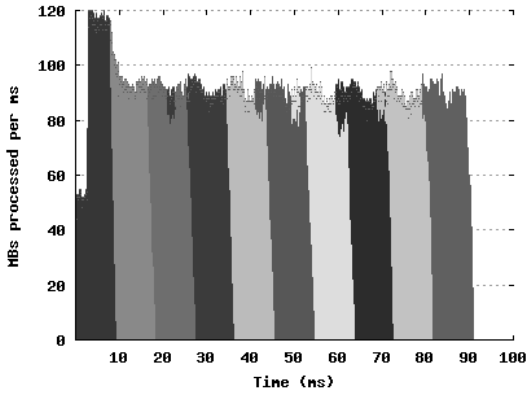


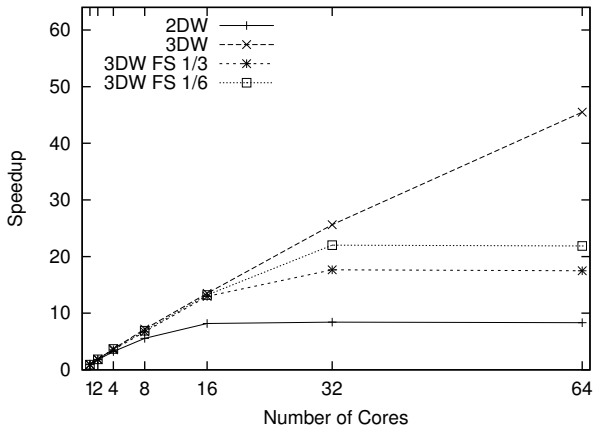
Figure 3.8: Number of MBs processed per ms using frame scheduling and priority policy for FHD Rush Hour on a 16-core processor. Different gray scales represent different frames.

impact on the scalability and is only show in Figure 3.9(b), as it decreases legibility. The reported scalability is based on the 2D-Wave execution time on a single core.

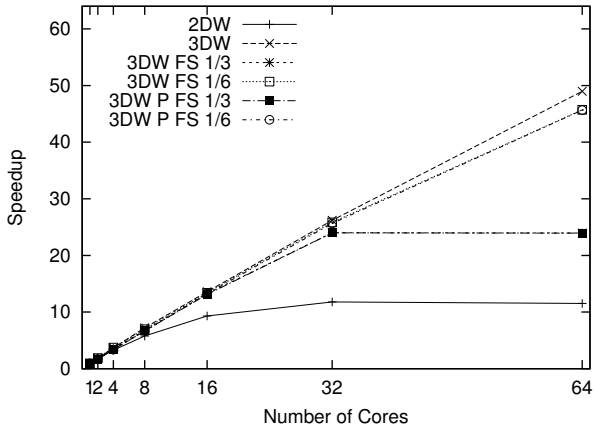
Figure 3.9(a) shows that 3 or 6 frames in flight are not sufficient to leverage a 64-core system when decoding an SD sequence. The maximum speedup of 22.0 is the result of the relatively low amount of MB-level parallelism in SD frames. As presented in Figure 3.6(a), for SD the 2D-Wave has a maximum speedup of 8.4. For HD, the performance when 6 frames in flight are allowed is already close to the performance of the original 3D-Wave, as depicted in Figure 3.9(b). The maximum speedups are 24.0 and 45.7, for 3 and 6 frames in flight, respectively. The latter is 92% of the maximum 3D-Wave speedup. For FHD, depicted in Figure 3.9(c), allowing three frames in flight provides a speedup of 45.9. When 6 frames are used, the difference between the performance of the 3D-Wave with the frame scheduler enabled and the original 3D-Wave is only 1%.

3.4.3 Bandwidth Requirements

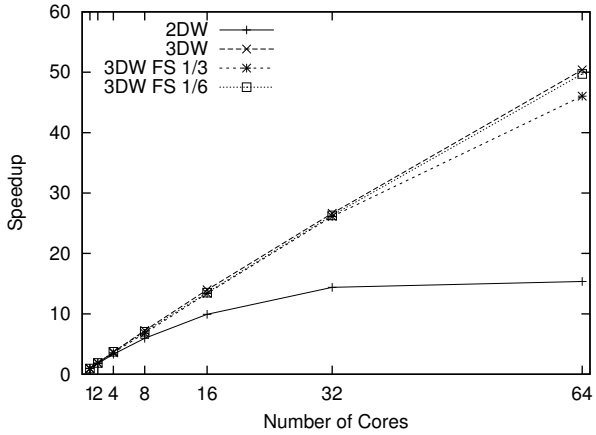
In this section, the intra-chip bandwidth requirements for the 3D-Wave and its frame scheduling and priority policies are reported. The amount of data traffic between L2 and L1 data caches is measured. Accesses to the main memory are not reported by the simulator.



(a) Scalability for SD



(b) Scalability for HD



(c) Scalability for FHD

Figure 3.9: Frame scheduling and priority scalability results of the Rush Hour 25-frame sequence

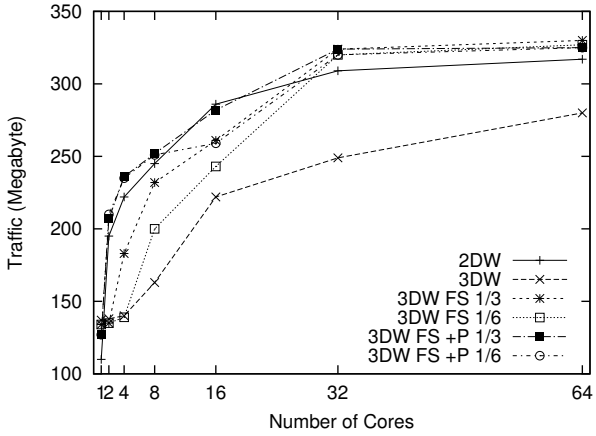
The effects of frame scheduling and priority policies on data traffic between L2 and L1 data caches are depicted in Figures 3.10(a), 3.10(b), and 3.10(c). The figures depict the data traffic for SD, HD, and FHD resolutions, respectively. In the figures, FS refers to the frame scheduling while P refers to the use of frame priority.

Data locality decreases as the number of cores increases, because the task scheduler does not take data locality into account when assigning a task to a core (except with the tail submit strategy). This decrease in the locality contributes to traffic increase. Due to these effects, the 3D-Wave increases the data traffic by approximately 104%, 82%, and 68% when going from 1 to 64 cores, for SD, HD, and FHD, respectively.

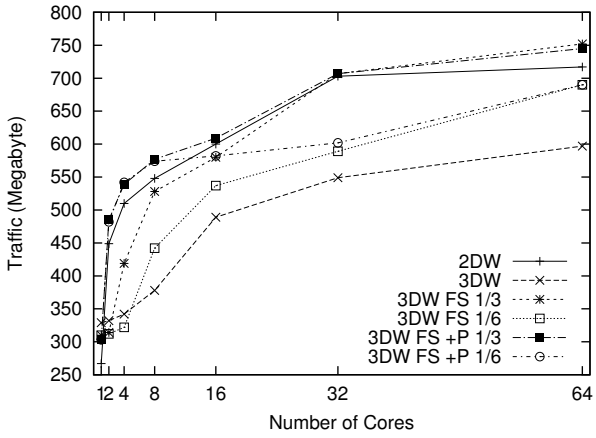
Surprisingly, the original 3D-Wave requires the least communication between L2 and L1 data caches for 8 cores or more. It transfer approximately 20% to 30% (from SD to FHD) less data than the original 2D-Wave, for 16 cores or more. This is caused by the high data locality in the original 3D-Wave technique. The 3D-Wave implementation starts processing new frames as soon as their dependencies are met. This increases the probability that the reference areas of the MB that is processed is present in the L1 data cache of another core. The probability increases because nearby area of several frames are decoded together, so the reference area is likely to be present in data caches of other cores. This reduces the data traffic because the motion compensation requires a significant portion of previous frames to be copied.

The use of FS and Priority has a negative impact on the L2 to L1 data cache traffic. The use of FS and Priority decreases the data locality, as they increase the time between processing MBs from co-located areas of consecutive frames. However, when the number of frames is sufficient to sustain a good scalability, the data traffic when using FS and Priority is still lower than the data traffic of the 2D-Wave implementation. For SD, the data traffic for FS and Priority is higher than the 2D-Wave when the available parallelism is not sufficient to leverage 32 and 64 cores. The same happens for the HD using only 3 frames in flight. For FHD, the 2D-Wave is the technique that requires the most data traffic, together with FS with 3 frames in flight. When the number of frames in flight is sufficient to leverage 32 or 64 cores, FS have 4 to 12% less data traffic than 2D-Wave. FS and Priority have 3 to 6% less data traffic than the 2D-Wave in the cases when the number of frames in flight is insufficient to utilize all the cores.

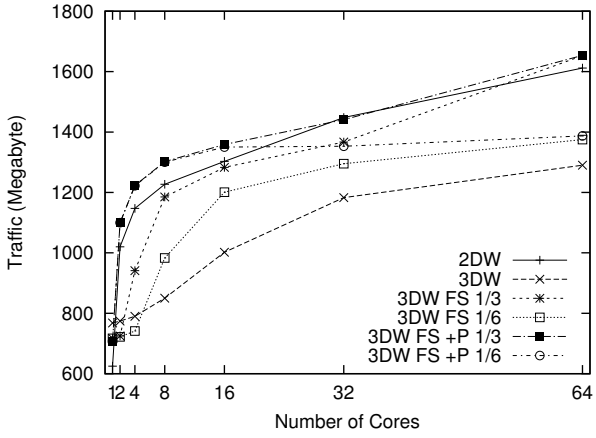
With the presented data traffic results, it is possible to calculate the L2 to L1 bandwidth requirements. The bandwidth is calculated by dividing the



(a) SD



(b) HD



(c) FHD

Figure 3.10: Frame scheduling and priority data traffic results for Rush Hour sequence.

total traffic by the time to decode the sequence in seconds. The total amount of intra chip bandwidth required for 64 cores is 21 GB/s for all resolutions of the Rush Hour sequence. The bandwidth is independent of the resolution because the number of MBs decoded per time unit per core is constant.

3.4.4 Impact of the Memory Latency

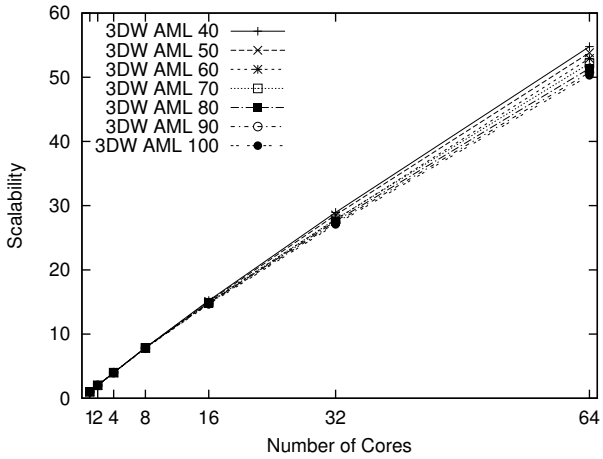
The type of interconnection used and the number of cores in the system influence the memory latency. For increasing number of cores, also the latency of an L2 to L1 data transfer increases. As mentioned in Section 3.2, the L2 cache is modeled using an Average Memory Latency (AML) that includes hits, misses, and interconnection latency. In this section, we analyze the impact of this latency on the performance by varying the AML. In the previous experiments, the AML was set to 40 cycles. In this experiment it ranges from 40 to 100 cycles in steps of 10 cycles.

Figure 3.11(a) depicts the scalability results for FHD resolution. That is, for each AML, the performance using p cores is compared to the performance using one core using the same AML. The results show that the memory latency does not significantly affect the scalability. For 64 cores, increasing the AML from 40 to 100 cycles decreases the scalability by only 10%. The scalability, however, does not equal the absolute performance. In Figure 3.11(b) the performance is depicted using the execution time on a single core with an AML of 40 cycles as baseline. The graph shows that a larger AML decreases the performance significantly. That indicates that large systems might be ineffective if the memory latency increases too much.

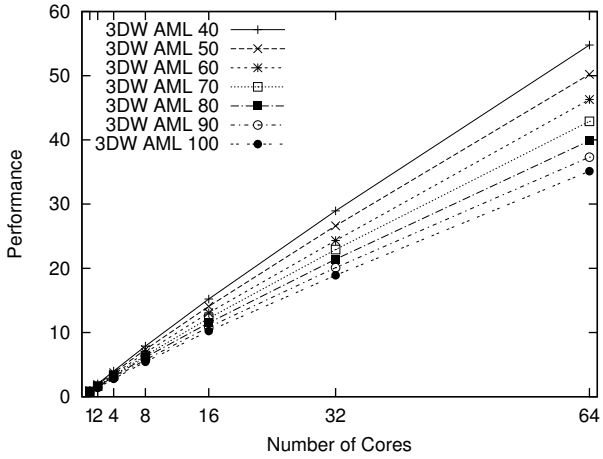
3.4.5 Impact of the L1 Cache Size

in this section, we analyze the influence of the L1 data cache size on the scalability and the amount of L2-L1 traffic. The baseline system has 4-way set-associative L1 data caches of 64KB, with LRU replacement and write allocate. By modifying the number of sets in the cache systems, different cache sizes, i.e., 16, 32, 64, 128, and 256KB, were simulated. The results for FHD resolution are depicted in Figure 3.12(a). The depicted performance is relative to the decoding time on a single core with the baseline 64KB L1 cache.

The systems with 16KB and 32KB caches have a large performance drop of approximately 45% and 30%, respectively, for any number of cores. The reason for this is depicted in Figure 3.12(b), which presents the L1-L2



(a) Scalability



(b) Performance

Figure 3.11: Scalability and performance for different Average Memory Latency (AML) values, using the 25-frame Rush Hour FHD sequence. In the scalability graph the performance is relative to the execution on a single core, but with the same AML. In the performance graph all values are relative to the execution on a single core with an AML of 40 cycles.

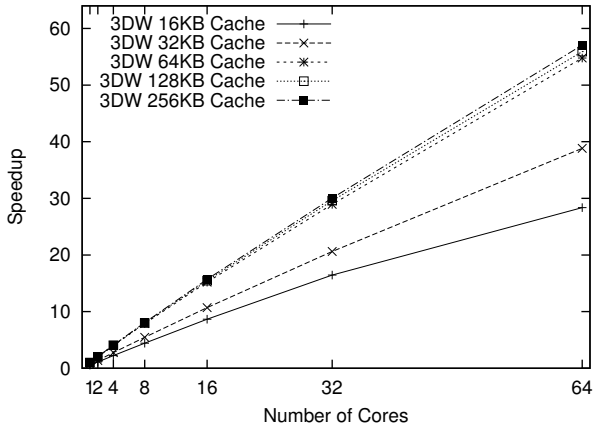
cache data traffic for FHD resolution. Compared to a system with 64KB caches, the system with 16KB caches has between 3.1 and 4.7 times more traffic while the system with 32KB caches has between 1.8 and 2.5 times more traffic. Those huge increases in data traffic are due to cache misses. For FHD resolution, one MB line occupies 45KB. Preferably, the caches should be able to store more than one MB line, as the data of each line is used in the decoding of the next line and serves as input for the motion compensation in the next frames. For FHD, the 16KB and 32KB caches suffer greatly from data thrashing. As a result, there are a lot of write backs to the L2 cache as well as reads. For smaller resolutions the effects are less prominent. For example, for SD resolution using 16KB L1 caches the data traffic increases with a factor between 1.19 and 1.66 compared to the baseline with 64KB caches. With 32KB caches, the traffic increases only by approximately 7%.

Using caches larger than 64KB provides small performance gains (less than 4%).' The reason for this is again the size of a MB line. Once the dataset fits in the cache it makes no use of the additional memory space. This is also reflected in the data traffic graph. For FHD, the system with 256KB caches has 13 to 27% less traffic than the 64KB system. For the lower resolutions, the traffic is reduced by at most 10% and the performance gain is at most 4%.

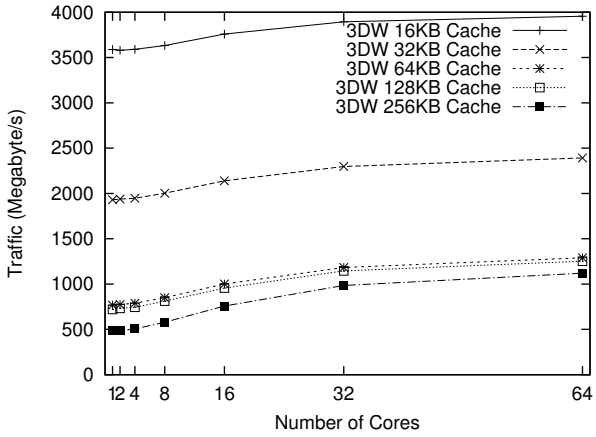
3.4.6 Impact of the Parallelization Overhead

Alvarez et al. [4] implemented the 2D-Wave approach on an architecture with 64 dual core IA-64 processors. Their results show that the scalability is severely limited by the thread synchronization and management overhead, i.e., the time it takes to submit/retrieve tasks to/from the task pool. On their platform it takes up to 9 times as long to submit/retrieve a task as it takes to decode a MB. To analyze the impact of the synchronization and management overhead on the scalability of the 3D-Wave, we increase the TP overhead by adding dummy calculations in the submit/retrieve functions of the TP.

The inserted extra overheads are 10%, 20%, 30%, 40%, 50%, and 100% of the average MB decoding time, which is 4900 cycles. Because of the Tail Submit enhancement, not every *decode_mb* task requests or submits a task to the TP. For example, on a single core, the version with 100% TP overhead is only 3% slower than the baseline version. The effects of the increased overhead are depicted in Figures 3.13(a), 3.13(b), and 3.13(c), for SD and FHD resolutions, respectively.

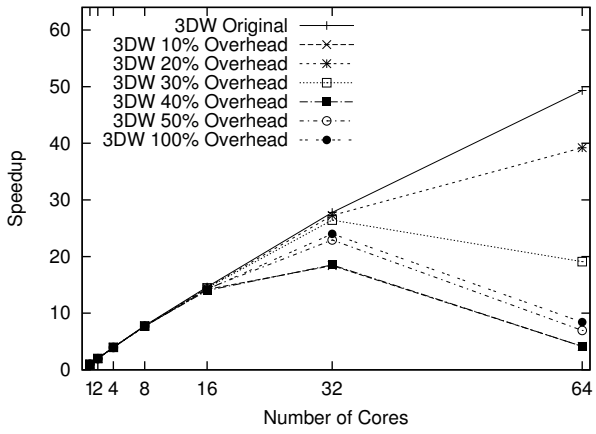


(a) Scalability

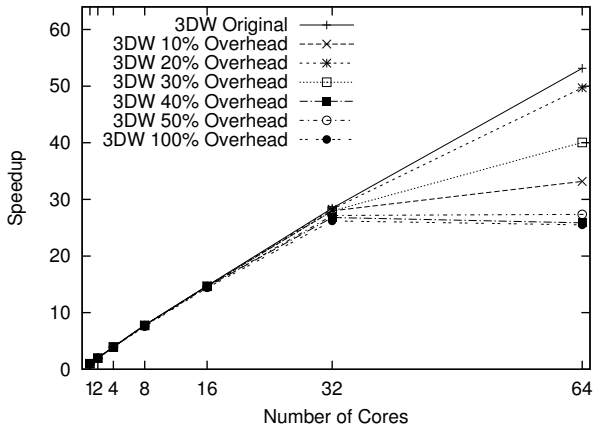


(b) Data traffic

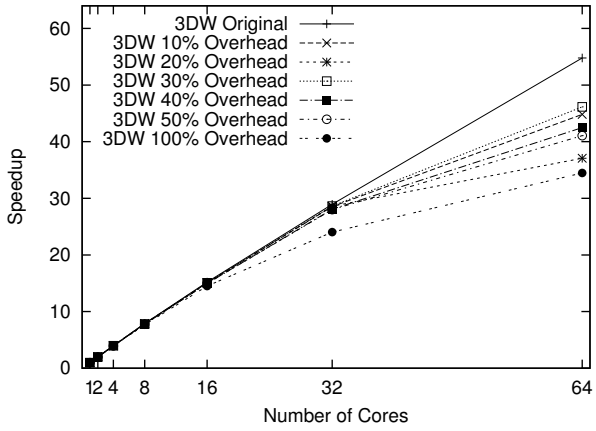
Figure 3.12: Impact of the L1 cache size on performance and L1-L2 traffic for the 25-frame Rush Hour FHD sequence.



(a) SD



(b) HD



(c) FHD

Figure 3.13: TP overhead effects on scalability for Rush Hour frames

The results for SD resolution clearly show the impact of the increased overhead on the scalability. For 32 cores the scalability is considerably reduced when the overhead is 40% or more. For 64 cores the extra overhead reduces the scalability even more. The SD resolution is severely affected by the increased overhead because the frame resolution is comparatively low and the lines are short, which increases the number of task submissions per frame. For the HD resolution, the increased overhead limits the scalability to 32 cores while for FHD it slows down the scalability, but does not limit it. As the resolution increases the number of requests to the TP per MB decreases and so does the impact of the extra overhead. These results show the drastic effects of the overhead on the scalability, even with enhancements that reduce the number of requests to the task pool. An interesting characteristic of the results is that, in some cases, a lower overhead causes a larger reduction in scalability than a higher overhead. Although we cannot fully explain the reasons of this behavior, we believe that is due to the change in the order in which tasks are submitted and consequently in their dependency chain.

3.4.7 CABAC Accelerator Requirements

Broadly speaking, H.264 decoding consists of two parts: entropy (CABAC) decoding and MB decoding. CABAC decoding of a single slice/frame is largely sequential while we have shown that MB decoding is highly parallel. We therefore assumed that CABAC decoding is performed by a specific accelerator. In this section we evaluate the performance required of such an accelerator to allow the 3D-Wave to scale to a large number of cores.

Figure 3.14 depicts the speedup as a function of the number of (MB decoding) cores for different speeds of the CABAC accelerator. The baseline accelerator, corresponding to the line labeled “no speedup”, is assumed to have the same performance as the TM3270 TriMedia processor. These results were obtained using a trace-driven, abstract-level simulator that schedules the threads given the CABAC and MB dependencies and their processing times. The traces have been obtained using the simulator described in Section 3.2 and used in the previous sections.

The results show that if CABAC decoding is not accelerated, then the speedup is limited to 7.5, no matter how many cores are employed. Quadrupling the speed of the CABAC accelerator improves the overall performance by a similar factor, achieving a speedup of almost 30 on 64 cores. When CABAC decoding is accelerated by a factor of 8, the speedup of 53.8 on 64 cores is almost the same as the results presented previously which did not con-

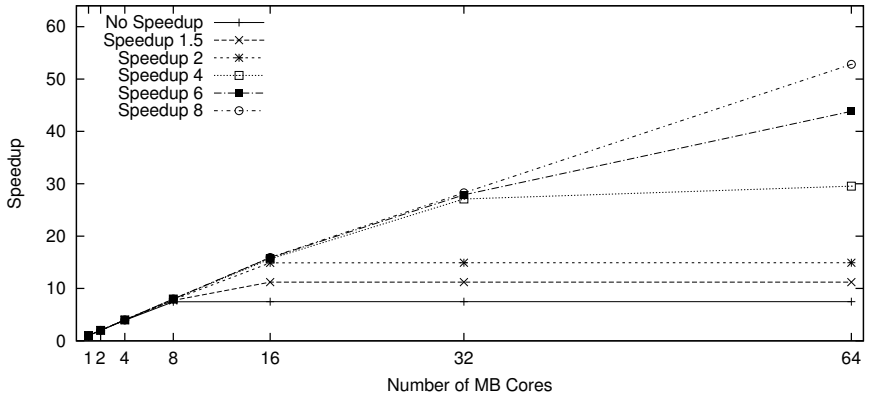


Figure 3.14: Maximum scalability per CABAC processor and accelerators

sider CABAC. There are proposals(e.g., [75]) that achieve such a speedup for CABAC decoding. This shows that the CABAC processing does not pose a strict limitation on the scalability of the 3D-Wave technique. We remark that the 3D-Wave also allows employing multiple CABAC accelerators, since different slices/frames can be CABAC decoded in parallel, as entropy-decoding dependencies do not cross slice/frame borders.

3.5 Conclusions

Future CMPs will contain dozens if not hundreds of cores. For such systems, developing parallel applications that can harness them is the key challenge. In this chapter we have contributed to this challenge by presenting a highly scalable parallel implementation of H.264 decoding. While a many-core is not necessary to achieve real-time FHD video decoding, it is likely that future video coding standards will be computationally more intensive and will be similarly block-oriented and based on motion compensation. Furthermore, decoding is part of encoding and real-time encoding is still a challenge. In addition, emerging applications such as 3D TV are based on current video coding standards.

While the idea behind the 3D-Wave was presented in the previous chapter, in this chapter we have contributed by providing an actual implementation and by providing extensive simulation results. Implementing the 3D-Wave required, for example, developing a subscription mechanism where MBs are subscribed to a so-called *Kick-off List* associated with the MBs in the reference

frame(s) they depend on. Several optimizations have been performed to reduce the overhead of this mechanism. For example, vector prediction is skipped if it has already been performed and if two reference MBs are in the same reference frame, only the one that will be decoded last is added to the list.

The simulation results show that the 3D-Wave implementation scales almost perfectly up to 64 cores. More cores could not be simulated due to limitations of the simulator. Furthermore, one of the main reasons why the speedup is slightly less than linear is that at the beginning and at the end of decoding a sequence of 25 frames, not all the cores can be used because little TLP is available. In a real sequence these periods are negligible. The presented frame scheduling and priority policies reduce the number of frames in flight and the frame latency. By applying these policies, the frame latency of the 3D-Wave is only 0.1 ms (about 1%) longer than that of the 2D-Wave.

We also measured the amount of data traffic between the shared L2 and the private L1 data caches. Obviously, increasing the number of cores increases the L2-L1 data traffic, since the cores have to communicate via the L2 cache. 64 cores generate approximately the same amount of L2-L1 traffic as 32 cores, however, and both produce roughly twice as much traffic as a single core. To our initial surprise, the original 3D-Wave generates the least amount of L2-L1 data traffic. This is because the original 3D-Wave exploits the data reuse between a MB and its reference MBs, more so than the 2D-Wave and the 3D-Wave with frame scheduling and priority.

Next we have analyzed the impact of the memory latency and of the L1 cache size. While increasing the average memory latency (AML) hardly affects the scalability (i.e., the speedup of the 3D-Wave running on p cores over the 3D-Wave running on a single core), it of course reduces the performance. Doubling the AML from 40 to 80 cycles reduces the performance on 64 cores by approximately 25%. The results for different L1 data cache sizes show that a 64KB data cache is necessary and sufficient to keep the active working set in cache. Smaller L1 data caches significantly reduce the performance, while larger L1 data caches provide little improvement. The reason is that a single line of MBs is 45KB for FHD and, therefore, caches larger than 45KB can exploit the data reuse between a MB line and the next MB line.

In addition, we have analyzed the impact of the parallelization overhead by artificially increasing the time it takes to submit/retrieve a task to/from the task pool. The 3D-Wave exploits medium-grain TLP (decoding a single MB takes roughly 5000 cycles on the TM3270), so task submission/retrieval should be efficient. Because of the tail submit optimization, however, not for every

MB a task is submitted to the task pool. The results show that even when the parallelization overhead is 50% of the MB decoding time (about 2500 cycles), the speedup on 64 cores is still higher than 41 for FHD. For SD, because it exhibits less TLP and therefore submits more tasks per MB, the effects are more dramatic.

Finally, we have analyzed the performance required of a CABAC accelerator so that CABAC decoding does not become the bottleneck that limits the scalability of the 3D-Wave. The results show that if CABAC decoding is performed by a core with the same speed as the other cores, then the speedup is limited to 7.5, no matter how many cores are employed. If CABAC decoding is accelerated by a factor of 8, however, the speedup for 64 cores is almost the same as when CABAC decoding is not considered.

4

Suitability of SIMD-Only Cores for Kernels with Divergent Branching

The last two chapters focused on the first goal of this thesis, increase the scalability of video processing for many-core processors. It was shown that there is sufficient thread-level parallelism in video processing to leverage it for the many-core processors. From this chapter on we focus on microarchitecture of the cores composing a many-core processor. As part of our second goal, this chapter evaluates the suitability of SIMD-only cores for the increasingly amount of divergent branching in video processing algorithms. The H.264 Deblocking Filter (DF) is used as a test case as it features 4 to 6 branches on the main body of the filtering functions. The Cell Synergistic Processing Element (SPE) is used to emulate the SARC Worker core for multimedia processing.

This chapter is organized as follows. Section 4.1 presents the motivation of the work in this chapter. Existing works related to SIMD processing overheads are discussed in Section 4.2. The Cell processor and its SPE are briefly described in Section 4.3. The DF is described in Section 4.4 and its implementation on the SPE is given in Section 4.5. Section 4.6 presents and analyzes the experimental results. In Section 4.7 conclusions are drawn.

4.1 Introduction

As discussed in Chapter 1, to deal with the increasing complexity of video processing coders/decoders (codecs), many processors feature Single Instruction Multiple Data (SIMD) units to accelerate multimedia processing. SIMD processing has well documented overheads, that will be briefly reviewed in Section 4.2. Cores which the instruction set is composed only by instructions that operate in SIMD fashion, called SIMD-only cores, however, introduce new challenges to processing efficiency. Two of these challenges are the efficient processing multimedia kernels with divergent branching and the efficient processing of scalar data. The first challenge is the focus of this chapter while the second is evaluated in the next chapter of this thesis.

The increasing complexity of multimedia kernels can have a negative impact on the efficiency of SIMD processing, due to the fact that presence of complex control structures in the program decreases data-level parallelism. Moreover, this can be critical on SIMD-only cores without specific units to support divergent branching, such as branch prediction and superscalar processing units.

Branch prediction and superscalar processing require complex hardware structures. Branch prediction relies on large tables to store branch histories. Superscalar processing requires complex control logic to extract ILP from the program. Because the performance gains of these structures are not scalable with the area overhead and power consumption [45], they are not featured in our Worker cores. Alternative low overhead implementations, however, are being investigated [15].

This chapter evaluates the suitability of SIMD-only cores for algorithms with highly divergent branching. As a case study, a highly adaptive filtering algorithm was chosen: the Deblocking Filter (DF) of the H.264 video compression standard. The DF is not the most time consuming kernel of H.264, but might take up to 49% of the total processing time if not vectorized using SIMD instructions along with the other kernels [6].

As mentioned in Chapter 1, we use the Cell processor SPE cores to emulate the SARC Worker cores. This allowed us to start the Worker core evaluation while the SARC simulation tools were still in development.

4.2 Related Work

Several works focused on reducing SIMD overhead, i.e., the overhead needed for bringing the data in a form amenable to SIMD processing, in scalar cores with SIMD extensions. These overheads include data alignment, packing and unpacking of subwords, and subword rearrangement. Ranganathan et al. [79] report that after vectorized with SIMD instructions, on average, 41% of their instructions for several image and video processing kernels is overhead.

Compiler techniques to reduce the overhead related to data permutations, strided accesses, and alignment constraints were presented in [80], [71], and [34], respectively. Ren et al. [80] presented a strategy to optimize data permutations needed for non-contiguous and misaligned memory references. Experiments were performed on several applications. The results showed that up to 77% of the permutation instructions were eliminated. Nuzman et al. [71] introduced an automatic compilation scheme that supports effective vectorization in the presence of interleaved data with constant strides that are powers of 2. Experimental results on a wide range of kernels showed execution time speedups of up to 3.7. Eichenberger et al. [34] presented a compilation scheme that vectorized with SIMD instructions loops in the presence of misaligned memory references. Several techniques were proposed to minimize the number of data reorganization operations. For a set of loops where 75% or more of the static memory references are misaligned, the results showed near peak performance.

Hardware techniques to reduce packing/unpacking and data rearrangement overhead were proposed in [89]. Two techniques were presented. The first technique, called extended subwords, uses four extra bits for every byte in a media register. This allows many SIMD operations to be performed without overflow and avoids packing/unpacking conversion overhead. The second technique introduces a Matrix Register File that allows flexible row-wise and column-wise access to the register file. This eliminates the costly transposition steps which are required for many multimedia kernels. Experimental results showed that these techniques have an average speedup of almost 2 compared to a conventional SIMD instruction set extension.

Another approach for matrix transposition is presented in [67]. It proposes an instruction that swaps the odd numbered elements of the first source/destination SIMD register with the even-numbered elements of the second source/destination register. It differs from regular shuffle instructions as it has two destination registers, reducing the number of steps required to trans-

pose a matrix by half. For the DF, it provides an overall performance improvement of 35%.

Alvarez et al. [8] measured the overhead of unaligned access for video processing and proposed splitting the memory bank into two to support unaligned data accesses. Results showed that unaligned access provides a speedup up to 2 for luma interpolation part of the H.264 MC, compared to the plain SIMD version.

All these works focused on SIMD overhead that is still present in SIMD-only cores. In this and in the next chapter, however, we focus on overhead specifically related to processing on SIMD-only architectures not previously studied. For example, in scalar cores with SIMD extensions scalar operations do not pose a problem, but in SIMD-only cores they may.

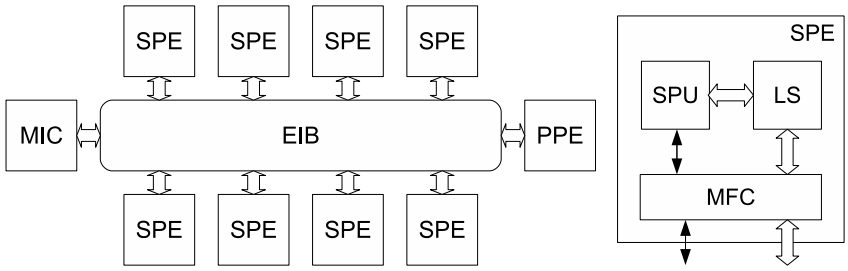
4.3 Cell Processor Architecture

This section briefly describes the The Cell Broadband Engine processor [50, 43]. The main characteristics of the Cell processor are presented with the focus on the memory system and the SIMD architecture of the SPEs.

The Cell processor is a heterogeneous multi-core processor designed for multimedia and game processing. Although originally designed for multimedia and gaming, the Cell processor has also been used as a basic block of high-performance and supercomputers. For example, it is part of the first supercomputer to run Linpack at a sustained speed in excess of 1 Pflop/s [13]. The performance and power efficiency of the Cell processor make it a suitable option to accelerate a wide range of applications. It consists of one Power Processor Element (PPE) and eight Synergistic Processing Elements connected by the Element Interconnect Bus (EIB) that consists of four 16B-wide data rings. A block diagram of the processor is depicted in Figure 4.1(a).

The PPE is a simplified version of the PowerPC processor family. It is based on IBM's 64-bit Power Architecture [1] with 128-bit SIMD media extensions. It is fully compliant with the 64-bit Power Architecture specification and can run 32-bit and 64-bit operating systems and applications. The PPE is dual-threaded and has a two-way in-order execution pipeline unit with 23 stages. The PPE has a conventional two-level cache hierarchy with split 32KB L1 instruction and data caches and a 512KB unified L2 cache.

As depicted in Figure 4.1(b), each SPE consists of a Synergistic Processing Unit (SPU), a Local Store (LS), and a Memory Flow Controller (MFC).

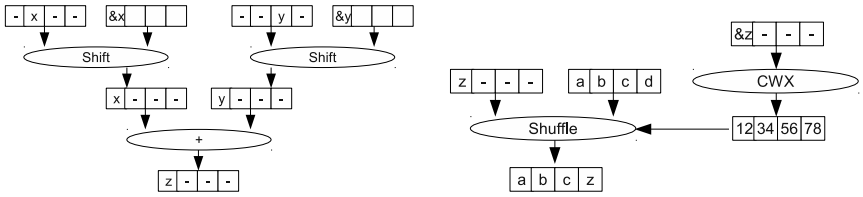


(a) Cell Broadband Engine block diagram. (b) SPE block diagram.

Figure 4.1: Cell and SPE block diagrams.

The LS is a 256KB scratchpad memory and the MFC is composed of a Direct Memory Access (DMA) engine, a memory management unit, and a bus interface. The SPUs are tailored for multimedia processing and are single-threaded, non-preemptive, two-way in-order processors. One issue slot can contain fixed- and floating-point operations and the other can contain loads/-stores, byte permutation operations, as well as branches. Branches are hinted by software and miss-predicted branches have a penalty of 18 cycles. Even unconditional branches need to be hinted in order to avoid the penalty. The register file contains 128 128-bit wide registers. All instructions are SIMD and they operate on 128-bit vectors with varying element width, i.e., $2 \times$ 64-bit, $4 \times$ 32-bit, $8 \times$ 16-bit, $16 \times$ 8-bit, or $128 \times$ 1-bit. The SPE instruction set does not contain scalar operations. Because of this, the SPE is often called a SIMD-only type of core. Furthermore, data should be 128-bit aligned.

SPEs can only access data and code stored in their 256KB LS. The LS is mapped into the main memory address space to allow LS-to-LS communication, but this memory (if cached) is not coherent in the system. To access the external memory the SPU issues a DMA request to the MFC. There are four types of DMA requests: put, get, $putlist$, and $getlist$. A put writes data from the LS to the external memory. A get copies data in the external memory to the LS. Requests can be grouped in a list with up to 1024 requests, that are issued by $putlist$ and $getlist$ requests. The DMA unit requests the data and sets a flag when the request is performed. Data are transferred in packets of at most 16KB and both the source and the target address must be 16B aligned. The DMA unit can handle up to 16 requests concurrently and data communication can be performed in parallel with computation. Double buffering can be employed to hide the data transfer latency. Double buffering is the technique of processing one block of data while fetching the next block.



(a) Addition of two scalar variables. (b) Merging and storing the scalar result.

Figure 4.2: Example of compiler managed scalar operation in the SPE.

The design decision to not support scalar and unaligned operations was taken to reduce the control complexity and to eliminate several stages from the critical memory access path [43]. To process scalar and unaligned data, the compiler inserts shuffle and shift instructions to align or allocate the data in the *preferred slot*. The preferred slot is the leftmost word in a 128-bit quadword. Figure 4.2 illustrates the steps to add two scalars, x and y, and store the result z in the rightmost slot of the quadword. The SPE compiler manages scalar operations by first moving both operands to the preferred slot of a register and then the SIMD operation corresponding to the scalar operation is performed, as depicted in Figure 4.2(a). Next, the *CWX* instruction generates the shuffle control word to store the result in its destination slot. Finally, The shuffle merges the result with the original content of the register while placing the result in its destination slot and writes back the data to the register file. These operations are depicted in Figure 4.2(b). Memory addresses for loads and stores, branch conditions, and branch addresses for register-indirect branches must be placed in the preferred slot.

For operations that involve only scalar variables, the compiler can place these variables in memory locations congruent with the preferred slot, at the price of wasting memory. If, however, the scalar is an element of a vector, then reorganization overhead is unavoidable.

For the experiment in this chapter we used a 2.4GHz Cell processor in a prototype Cell Blade [48]. For the remaining experiments in this thesis, we use the 3.2GHz Cell processor present in the Playstation 3 (PS3). It has 6 out of the 8 SPEs available, as one SPE is disabled for redundancy purposes and another one is used by the system for resources access management. Another important characteristic is that the PS3 has only 256MB of RAM. For the measurements we use the SPE has hardware counters. The SPE counter runs at a smaller frequency than the processor itself. In the PS3, it runs at 78.8 MHz, which is 40 times slower. This approach is not suitable for fine grain profiling, for

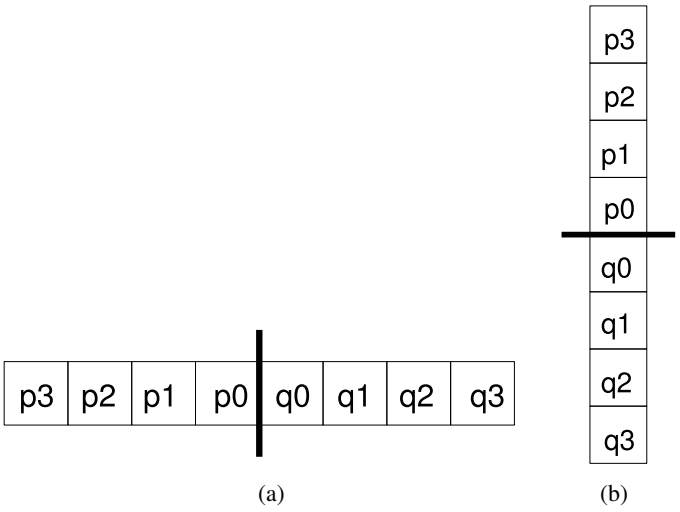


Figure 4.3: Line of pixels used for the vertical (a) and the horizontal (b) edges.

such cases we use a Cell processor simulator. Nevertheless, it is sufficiently accurate to measure the performance of functions.

4.4 Deblocking Filter

The discrete cosine transform applied in video and image compression can produce an artifact known as *blocking*; square areas in the picture. The aim of the deblocking filter (DF) is to improve the visual quality of the decoded pictures by smoothing the block edges. In H.264, this filter is mandatory. The DF is highly adaptive and has different filter strengths depending on the block types, e.g., intra- and inter-predicted types.

The filters employed in the DF are one-dimensional, i.e., they are applied to lines or columns of pixels. The bi-dimensional behavior is obtained by applying the filter to both the vertical and the horizontal edges of all 4×4 luma or chroma blocks.

The DF is applied to a line of pixels orthogonal to the block edge (see Figure 4.3). Let qi ($0 \leq i \leq 3$) denote the pixels of the current block and let pi denote the pixels of the neighboring blocks. Depending on the filter strength, the values of pixels $p2$ to $p0$ and $q0$ to $q2$ are modified, $p3$ and $q3$ always remain unaltered.

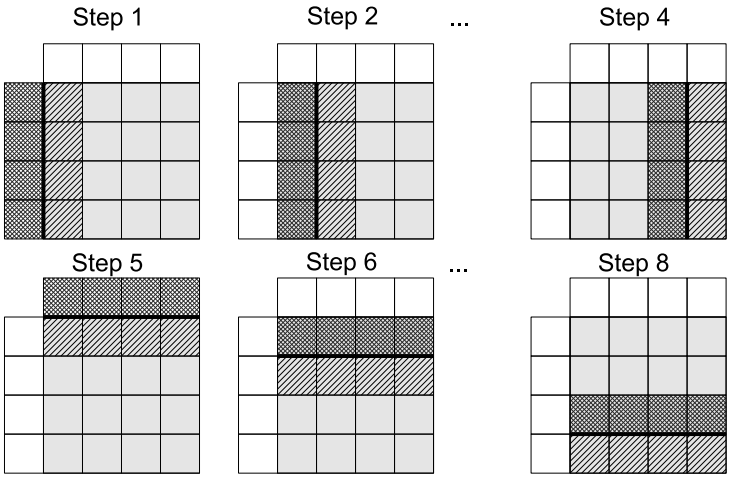


Figure 4.4: The filtering process of one macroblock.

The DF process first filters the left edges of the macroblock (MB) and then the vertical internal edges. This process is repeated for the horizontal edges. This process is illustrated in Figure 4.4, where each box represents a 4×4 luma block. The gray area represents the current MB, darkly-hatched the p blocks, and the q blocks are lightly-hatched.

The strength of the filter is determined dynamically and depends on the current quantizer, the coding of the neighboring blocks, and the gradient of the image samples across the boundary [60]. There are five Boundary Strengths (BSs) which the filter can apply, ranging from 0 (no filtering) to 4 (strongest one). Boundary strength 4 is applied between edges of two Intra Prediction MB boundary blocks. Boundary strength 3 is applied between edges of an Intra Prediction blocks. The others are used to edges between Inter prediction blocks.

Filtering is applied if the conditions $(p0 - q0) < \alpha$, $(p1 - p0) < \beta$ and $(q1 - q0) < \beta$ are met. The thresholds α and β depend on the encoder average quantization parameter over the edge. There are two filter functions. When the BS value is 4, the function depicted in Figure 4.5 is applied, where pi' is the new value for pixel pi. The function depicted in Figure 4.6 is applied for all other BS values. The presented code filters the p pixels. The equations for the q pixels are symmetrical.

```
if (p0 - q0  < (alpha << 2) + 2){
    if (p2 - p0 < beta){
        p0' = (p2 + 2*p1 + 2*p0 + 2*q0 + q1 + 4) >> 3;
        p1' = (p2 + p1 + p0 + q0 + 2) >> 2;
        p2' = (2*p3 + 3*p2 + p1 + p0 + q0 + 4) >> 3;
    }
    else
        p0' = (2*p1 + p0 + q1 + 2) >> 2;
}
else
    p0' = (2*p1 + p0 + q1 + 2) >> 2;
```

Figure 4.5: Filtering function for Intra MB boundaries.

```
clip(x, y, z){
  return x < y ? y : ( x > z ? z : x)
}
clip(x) {return clip(x, 0, 255)}

if( p2 - p0 < beta){
  p1'' = ((p2 + ((p0 + q0 + 1) >> 1)) >> 1) - p1;
  p1' = p1 + clip(p1'', -tc0, tc0);
}

delta' =  (((q0 - p0 ) << 2) + (p1 - q1) + 4) >>3;
delta  =  clip(delta',-tc, tc);
p0'    =  clip( p0 + delta );
```

Figure 4.6: Filtering function for other MB boundaries.

4.5 Implementation

In this section, the implementation of the DF on the Cell SPE is detailed. The implementation will be described in a top-down fashion starting with the main loop of the kernel and gradually goes down to the inner parts of the implementation. The focus of this analysis is the computational part of the DF.

The baseline version is a scalar implementation extracted from the FFMPEG H.264 decoder [37]. The extracted code does not include the parameter calculation of the DF. The analysis focuses on the sample filtering of the code. This scalar version is then vectorized using the SIMD intrinsics for the SPE and AltiVec intrinsics for the PPE. Because the AltiVec implementation is very similar to the SPE version, only the SPE version will be presented.

In the SPE implementations, the PPE is used only to read the parameters from the input files and to store them in main memory. After storing the parameters, the SPE threads are spawned. Thereafter, the PPE thread sends a signal to all SPEs to start the computation.

Each SPE thread processes one frame. The 3D-Wave technique was not considered because this research was performed before the 3D-Wave technique was developed. The processing of the DF per frame, however, avoids data movements between SPEs and/or between SPEs and main memory as all data dependencies are between instructions executed on the same SPE. The processing starts by reading the input pointers for the samples and parameters from the main memory.

Each frame is divided into several *MB lines* (MBs from the same row), in order to use the SPE's ability of performing computation and data communication in parallel. This partitioning is based on several factors such as the latency, maximum DMA transmission package size, number of DMA transfers, and organization of the data in the memory. The start-up latency increases with the partition size, so a large partition incurs large start-up latency. The pixel components Y, Cb, and Cr are stored in separate arrays. Partitioning into complete lines of MBs allows to load the pixel samples of one partition with three DMA transfers. Using partitions that do not cover complete lines would require a separated DMA request for each pixel line in the partition.

For every MB line there are four DMA transfers from memory to the LS. One DMA transfer is necessary for 16 lines of luma samples, two for 8 lines of each set of chroma samples, and another one for the DF parameters of the MB line. After the data is available in the LS, the processing of the MB line is performed and the results are transmitted back to the main memory.

```
DMA.get  (MB_Line[0]);

DMA.get  (MB_Line[1]);
DMA.wait (MB_Line[0]);
Filter   (MB_Line[0]);

FOR x = 2 TO frame_height_in_MB -1
{
    DMA.get  (MB_Line[x]);
    DMA.wait (MB_Line[x-1]);
    Filter   (MB_Line[x-1]);
    DMA.put  (MB_Line[x-2]);
}

Filter   (MB_Line[x-1]);
DMA.put  (MB_Line[x-2]);

DMA.put  (MB_Line[x-1]);
```

Figure 4.7: Double buffering of MB lines.

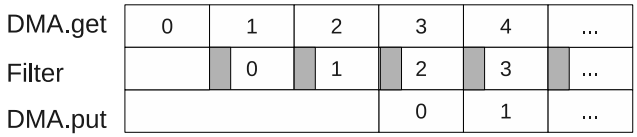


Figure 4.8: Deblocking Filter processing diagram.

The processing of the MB lines is performed as a software pipeline and uses a double buffering strategy. First, the data for the first MB line is requested, followed by the request of the data for the second MB line. After the data of the first MB line is available in the LS, it is filtered. This way the processing of MB line i is performed in parallel with the DMA transfer of of MB line $i+1$ from memory to the LS. The pseudo-code in Figure 4.7 illustrates the process. Figure 4.8 depicts the diagram of the DF process. The gray area in *Filter* indicates the waiting time for the DMA request to complete.

The processed MB line cannot be immediately transmitted back to memory. As depicted in Figure 4.4, the processing of the next MB line changes the values of the current bottom edge samples.

The filter process is performed per MB. As described in Section 4.4, there are 8 edges, four vertical and four horizontal. First the four vertical edges are filtered and then the four horizontal. The filtering process is divided into the following steps: (1) unpack the 8-bit (8b) samples of the current and left MBs to signed 16b, (2) transpose the current and left MBs so the pixels to be filtered are moved from inside the same quadword (as in Figure 4.3(a)) to different quadwords, and thus, more amenable to SIMD processing, (3) filter the vertical edges, (4) transpose the result, (5) pack back the left MB result to 8b, (6) unpack the last 4 lines of the top MB to 16b, (7) filter the horizontal edges, and finally, (8) pack back the MB result to 8b.

The computational core of the DF is the edge filtering. There are four functions required to implement the edge filtering. Luma and chroma samples require two functions each: one for Intra MB external edges blocks and another one for the other cases. These functions exhibit data-level parallelism and have been optimized with SIMD instructions of the SPE, such that the edge filtering computes 8 pixels simultaneously.

Because of the high branch penalties, it is necessary to eliminate branches whenever possible to improve performance. To perform the filtering without branches, all equations of the function have to be computed. All branches are replaced by comparisons that result in a mask. These masks are used to select the positions of the result vectors that will be written back to memory. All the filtering functions have been implemented without branches, except for the one to select between the filter processes listed above. The impact on the performance of the SIMD-only core is analyzed in the next section.

4.6 Experimental Results

In this section, the experimental results are presented. First, the input data and methodology are described. It is followed by an analysis of the results. Based on the analysis, the conclusions are drawn.

As input, the first eight frames of the Lake Wave video sequence, in the QVGA (320×240 pixels) resolution, are used. The reason why we used a QVGA video sequence is that at the time of this research was performed, only QVGA sequences were available to us. Furthermore, the goal was to compare a SIMD implementation to a scalar one and for this, the QVGA resolution is sufficient. The time is measured using the SPE hardware counters. The presented results are the average of three runs.

Figure 4.9 depicts the time in milliseconds required to filter a single frame for each version of the kernel: the scalar version running on the PPE (PPE - Scalar), the AltiVec version on the PPE (PPE - AltiVec), the scalar version on the SPE (SPE - Scalar), the SIMD version on the SPE without double buffering (SPE - SIMD no D.B.), and the SIMD version on the SPE with double buffering (SPE - SIMD D.B.). For the scalar versions, filtering a QVGA frame takes on average 2.79 and 2.72 ms on the PPE and SPE, respectively. The PPE AltiVec implementation has an average run-time of 2.15 ms, which is 30% faster than the PPE scalar version. The SPE SIMD version takes 1.14 ms per frame, while using a double buffering technique reduces the average runtime to 1.05 ms. The latter corresponds to a speedup of 2.6 compared to the SPE scalar version. For comparison, SIMD implementations of the DF for SSE2 have been presented in [57, 97] where speedups of 1.13 and 1.49 respectively, are reported.

As the AltiVec and SPE SIMD versions are almost identical, the difference in speedup obtained by vectorizing with SIMD instructions is interesting. A critical difference between the PPE AltiVec unit and the SPE is that the first one has only 32 vector registers, while the second has 128. Profiling the SPE-SIMD version using the IBM Cell Full-System Simulator [17] showed that all 128 registers are used, with an average of 120 registers alive. Because a MB is much larger than the PPE AltiVec register file, additional loads/stores are required, decreasing performance. Moreover, the SPE versions benefit from the LS, since no cache conflicts occur, and the direct access to main memory through the DMAs, as the kernel has a predefined memory access pattern.

The performance difference between the double buffering implementation with the non-double buffering is only 8%. The profiling results also show that the double buffering strategy successfully hides the communication latency. The total number of cycles the cores are stalled waiting for data from memory accounts for only 0.4% of the total runtime, on average. This shows that the DF kernel spends much more time processing than acquiring the data. As consecutive frames can be overlapped, the data communication latency influences only the start-up stage of the process.

The required SIMD overhead was also measured using the IBM Cell simulator. For profiling purposes the kernel was divided into four parts: Transposition, Pack/Unpack, Filtering, and Control. We call Control all parts of the kernel that do not fit in the first three categories. Figure 4.10 depicts the number of cycles spent in each part of the kernel. Filtering consumes 47% to 62% of the cycles required to process a frame, with an average of 58%. Approx-

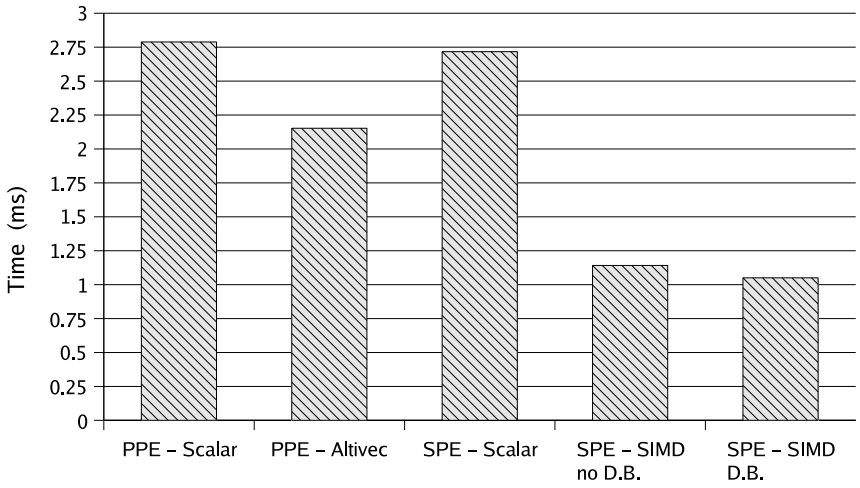


Figure 4.9: Average execution time of the deblocking filter implementations for one QVGA frame.

imately one third of the time is spent on the Transposition and Pack/Unpack parts. They consume on average 20% and 12%, respectively. The remaining 10% are used by the control structures of the kernel, e.g., data requests and function calls. The SIMD overhead for this kernel is within the previously reported range [79]. Although significant, this overhead is not the focus of this work as it has been the subject of several other works, as shown in Section 4.2.

If the known SIMD overhead is discarded, the DF vectorized with SIMD instructions has a speedup of approximately 5 compared with the PPE scalar version. This is 62.5% of the theoretical maximum (8, as 16 bits are necessary to represent the intermediate data). The 37.5% performance loss is due to two factors. First, the need of compute every possible result, one for each branch of the control statement. Second, the selection of the right result for each byte in the quadword. This performance loss is significant but acceptable if the number of branches in the original code is taken into account. To reduce this performance loss, different operators for a single SIMD word would be necessary, similar to the Imagine processor architecture [51].

The results show that SIMD-only cores can still achieve significant performance improvement over the scalar implementations, even in the presence of a significant number of branches. Moreover, the main contributor to the SPE speedup compared to the AltiVec is the large unified register file that is characteristic for a SIMD-only architecture.

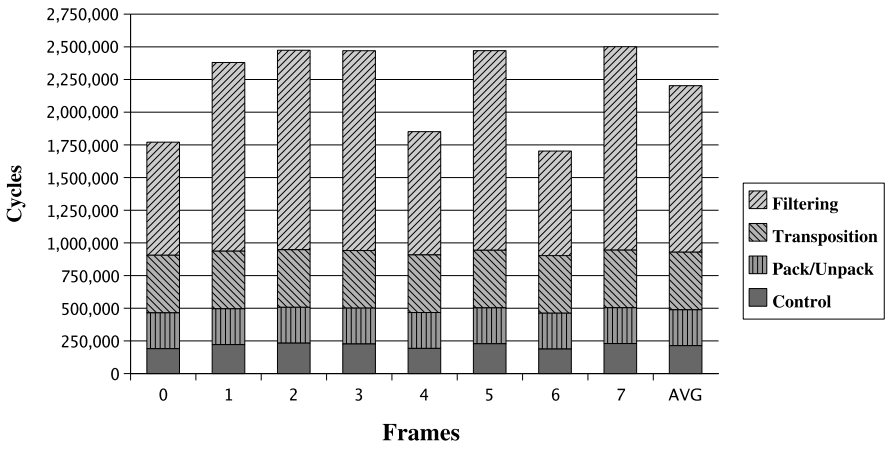


Figure 4.10: Execution breakdown of the SPE DF implementation for 8 QVGA frames.

4.7 Conclusions

This chapter presented PPE and SPE implementations of the H.264 Deblocking Filter to analyze the efficiency of video filtering on the SARC media Worker core. The Cell SPE was used to simulate the media Worker core. The DF was vectorized with SIMD instructions and ported to the SPEs of the Cell processor. Execution time was measured using hardware counters, while profiling was performed using the IBM Cell Simulator. Results show that approximately one third of the processing time of the SPE SIMD version is spent on well-known SIMD overheads such as transposition and data packing and unpacking. Despite this overhead and the high adaptivity of the kernel, the SPE version of the kernel vectorized with SIMD instructions is 2.6 times faster than the PPE and SPE scalar versions. If the known SIMD overheads are discarded, the DF vectorized with SIMD instructions reaches 62.5% of the theoretical maximum performance. These experiments show that SIMD-only cores can achieve significant performance improvement over the scalar implementations, even in the presence of many control flow statements.

5

Scalar Processing on SIMD-Only Architectures

Continuing our study on SIMD-only architectures, we focus on another one of its characteristics, while SIMD processing overheads in scalar cores with SIMD extensions are well documented in the literature [79, 89, 8], scalar processing overheads in SIMD-only cores have not yet been evaluated. In this chapter we study the latter overheads, identify their sources, and propose architectural enhancements to eliminate these overheads.

This chapter is organized as follows. Section 5.1 motivates the study and briefly describes scalar processing on SIMD-only cores. Section 5.2 presents the methodologies applied to determine the scalar processing overhead. It is followed, in Section 5.3, by brief descriptions of the evaluated kernels/applications. Experimental results are presented in Section 5.4 and options to reduce the overhead are discussed in Section 5.5. Finally, conclusions are drawn in Section 5.6.

5.1 Introduction

As argued in the previous chapter, SIMD processing is a power-efficient manner of exploiting the Data-Level Parallelism (DLP) present in many multime-

dia kernels. Unfortunately, not every parallelizable application benefits significantly from SIMD processing.

For example, sorting, information retrieval (e.g. histogram), and other kernels do not exhibit substantial amounts of DLP. These kernels, however, belong to important applications that cannot be neglected. Furthermore, even if a kernel is vectorizable using SIMD instructions, it often contains scalar operations.

As depicted in the previous chapter, to perform scalar operations and on SIMD-only cores as the SPE, the compiler inserts shuffle and shift instructions to move the data to the preferred slot. Next, the SIMD operation corresponding to the scalar operation is performed. Finally, a shuffle moves the result to the destination slot and writes back the data to the register file.

In this chapter, we evaluate SIMD-only cores for parallelizable kernels/applications that do not benefit significantly from SIMD processing. The objective of this work is to identify the amount of overhead introduced by the lack of scalar instructions and the situations that cause this overhead. In other words, the goal is to evaluate if the SARC Worker core (or SIMD-only processors in general) would profit significantly from a scalar datapath in addition to the SIMD datapath. To the best of our knowledge, we are the first to quantify the overhead caused by the lack of scalar operations in SIMD-only architectures. We study the impact of compiler-managed scalar and unaligned data for different applications. If significant overhead is incurred, it can justify modifications to the Worker core architecture. The performance degradation for scalar computations can justify the area overhead of a scalar unit or other support for scalar operations. This was the solution implemented in other many-cores such as the canceled Intel Larrabee [85] that keep scalar and vector processing datapath separated. We, however, discuss possible solutions for the scalar overhead without resorting to a complete scalar unit.

5.2 Experimental Methodologies

In this section, the methodologies used to determine the scalar processing overhead are described. Since implementing an architectural model would require too much effort for an initial evaluation, a methodology called Large-Data-Type is presented to study the overhead generated by the compiler to implement scalar operations. This methodology, however, is feasible only for relatively small kernels as it requires manual code modifications. To evaluate larger kernels and applications, a second methodology called SPE-vs-PPE is

presented. Similar to the previous chapter, the Cell SPE is used in this evaluation to represent the SARC Worker core.

5.2.1 Large-Data-Type Methodology

The Large-Data-Type (LDT) promotes 32-bit integers to 128-bit quadwords vectors to avoid shuffles/shift instructions, the main SIMD-only overheads when processing scalar data. Without these instructions, the behavior of the architecture would be equivalent to a processor with support for scalar operations. The LDT methodology highlights the differences between a SIMD-only core with and without support for scalar and unaligned operations. No influence from the DMA transfers will be considered in the performance measurements. The main microarchitectural resources focused by this methodology are the register file, its communication with the Local Store (LS), and the execution datapath. To allow this, the workload should fit in the SPE LS.

To implement the LDT methodology and simulate the behavior of a scalar unit, the kernels have been modified as follows. First, the processed data types are modified to 128-bit wide vectors. In order to avoid shuffles and shifts, the actual data is stored in the preferred slot by manually modifying the input. SPE instrinsics are used to handle comparisons, as the compiler cannot generate them directly.

The methodology is illustrated using the function depicted in Figure 5.1. This function adds the vector A to the reversed vector B and stores the result in vector C. The regular SPE implementation requires that every element of the vectors A and B are moved to the preferred slot before they are processed. This is performed by the $rotqby$ and $shufb$ instructions and causes considerable overhead. The assembly code generated by the compiler for $C[i] = A[i] + B[size - 1 - i]$; is depicted in Figure 5.3. The assembly code that implements the loop is not depicted because, as mentioned in Section 4.3 of the previous chapter, local variables are allocated in the preferred slot by the compiler and thus do not cause overhead. The resulting code of applying the LDT methodology to the function shown in Figure 5.1 is depicted in Figure 5.2. The types of the function arguments have been changed to *vector int* so that each element of the arrays is now an 128-bit wide vector. The spu_add intrinsic adds two aligned vectors. This way, the compiler generates only the core instructions without any shuffling, with the same behavior as an SPE with scalar support would have. The assembly generated after the LDT methodology has been applied is depicted in Figure 5.4. This assembly code is similar to the assembly code for the PPE, in terms of functionality. The *spu-gcc* com-

```
Scalar_Implementation(int A[], int B[], int C[],
                      int size)
{
   for (i=0; i < size; i++)
     C[i] = A[i] + B[size - 1 - i];
}
```

Figure 5.1: Example of function requiring scalar operations.

```
LDT_Implementation(vector int A[], vector int B[],
                   vector int C[], int size)
{
   for (i=0; i < size; i++)
     C[i] = spu_add(A[i],B[size - 1 - i]);
}
```

Figure 5.2: C code of the example kernel after the Large-Data-Type methodology has been applied.

piler version 4.1.1 with the *-O3* optimization flag is used for both versions of the code.

The LDT methodology, however, increases the data footprints of the kernels, as the scalars are now four times larger. This increases the amount of data transferred between the LS and the register file. This effect will be verified using the number of load/store stalls. The LDT methodology could also increase the pressure on the register file because it reduces the register file capacity from 512 scalar elements to 128 elements. Increasing the register pressure could mean that more variables need to be spilled to memory, while this would not be needed in an SPE with scalar support. To analyze this effect, we will compare the number of registers used in the LDT-emulated versions of the kernels to the number of registers needed in the original kernels.

5.2.2 SPE-vs-PPE

The purpose of the SPE-vs-PPE methodology is to evaluate the effects of compiler-managed scalar operations on large kernels and applications by com-

```
// $8 : i
// $11: address of A[i]
// $4 : address of B[size - 1 - i]
// $5 : address of C[i]

a       $2,$8,$11    //Add i to A to calc
                       distance from pref slot
lqx     $3,$8,$11    //Load A[i]
lqd     $10,0($4)    //Load B[size - 1 - i]
lqx     $7,$8,$5     //Load C[i]
cwx     $9,$8,$5     //Generate control word for
                       inserting C[i]
rotqby  $3,$3,$2     //Rotate A[i] to preferred slot
rotqby  $2,$10,$4    //Rotate B[size - 1 - i]
                       to preferred slot
a       $3,$3,$2     //Add A[i] and B[size - 1 - i]
shufb   $7,$3,$7,$9  //Shuffle result into
                      final position of C
stqx    $7,$8,$5     //Store C[i]
```

Figure 5.3: Assembly generated from the example function.

```
// $7 : i
// $8 : address of A
// $4 : address of B + size - 1 - i
// $5 : address of C
lqx     $2,$7,$8 // Load A[i]
lqd     $3,0($4) // Load B[size - 1 - i ]
a       $2,$2,$3 // Add A and B
stqx    $2,$7,$5 // Store result in C[i]
```

Figure 5.4: Assembly generated after the Large-Data-Type methodology has been applied to the example function.

paring their performance on the SPE to the their performance on a similar core with support to scalar operations. While the LDT methodology focuses on specific overheads, this methodology focuses on the overall scenario.

To compare the SPE to a processor with scalar instructions, a natural choice is the Cell PPE. The PPE runs at the same clock frequency and has several microarchitectural similarities with the SPE, such as in-order, dual-issue execution and a 23-stage pipeline that commits fixed point operations every two cycles. There are, however, fundamental architectural differences between the PPE and the SPE. The PPE is a general purpose core, while the SPE is designed specifically for multimedia and game processing. The PPE has a traditional memory hierarchy with a first and second level caches, while the SPE accesses data and code in its LS. Comparing the performance of the SPE with the PPE on large kernels reveals the scalar (in)efficiency of the processors. To concentrate the comparison on the computational part of the processor, the data set size of the kernels is limited to 32KB, corresponding to the size of the L1 data cache.

5.3 Kernels

This section describes the kernels used in this study. The selection criteria for the kernels are that they should contain scalar, hard to vectorize sections, but be parallelizable at the same time. These criteria are to match the type of applications likely to be ported to a many-core processor. There are two sets of kernels, “Small Kernels” and “Large Kernels”. Small Kernels are the kernels used to highlight a particular characteristic of the SIMD-only core that causes scalar processing overhead, using the LDT methodology. Large Kernels are used to compare the performance of the SPE with that of the PPE, using the SPE-vs-PPE methodology.

5.3.1 Small Kernels

The small kernels highlight specific characteristics of SIMD-only cores that introduce scalar computation overhead. They access unaligned data, process scattered data, or make use of indirect addressing. As described in Section 4.3, unaligned data refers to data vectors that do not start in the first slot and scalars that are not in the preferred slot. For each of these characteristics two kernels were selected. For ease of reference, pseudo-C codes are listed for each kernel.

```
saxpy(int x[], int y[], int scalar, int size){
   for (i = 0; i < size; i++)
      y[i] = scalar*x[i] + y[i];
}
```

Figure 5.5: Pseudocode for SAXPY.

```
  for (i = ORDER-1; i < SIZE+ORDER-1; i++){
    accum=0;
    for (j = 0; j < ORDER; j++)
      accum += coefficients[j] * input[i-j];
    output[i-(ORDER-1)] = accum;
  }
```

Figure 5.6: Pseudocode for the FIR filter.

5.3.1.1 Kernels that Access Unaligned Data.

Saxpy. Basic Linear Algebra Subprograms (BLAS) is a linear algebra application programming interface. It is used for scientific computations and is a part of the LINPACK benchmark. BLAS level 1 provides functionality of the form $y = \alpha \times x + y$, where x and y are vectors and α is a scalar, and it is called the SAXPY kernel. This kernel was chosen to determine the overhead of shuffle/rotate instructions needed to move the data to the preferred slot. Its pseudocode is listed in Figure 5.5. A SIMD implementation of this kernel is possible, but requires some programming effort, because the vectors may not be aligned in memory.

FIR. Finite Impulse Response (FIR) filters are implemented by a convolution of the signal with the coefficients. It requires accesses to vector elements that are not quadword aligned, both the coefficient and input values, which requires more shuffle/rotate instructions than the previous kernel. Its pseudocode is listed in Figure 5.6.

```
quickSort (int a[], int lo, int hi) {
    int i=lo, j=hi, h;
    int x=a[(lo+hi)/2];

    do{
        while (a[i]<x) i++;
        while (a[j]>x) j--;
        if (i<=j) {
            h=a[i];
            a[i]=a[j];
            a[j]=h;
            i++; j--;
        }
    } while (i<=j);

    //  recursion
    if (lo<j) quickSort(a, lo, j);
    if (i<hi) quickSort(a, i, hi);
}
```

Figure 5.7: Pseudocode for Quick Sort.

5.3.1.2 Kernels that Process Scattered Data.

QuickSort. Sorting algorithms are an important class of algorithms that are part of many applications. Sorting requires many comparisons, and aligned data accesses are hard to guarantee. QuickSort is one of the best known sorting algorithms. It works by choosing an element from the unsorted list, called the pivot, and moving the elements that are smaller to positions before the pivot and the elements larger to positions after the pivot. This procedure is recursively applied to the resulting lists until all elements of the lists are sorted. QuickSort pseudocode is listed in Figure 5.7.

Merge Sort. Conceptually, merge sort works as follows. The unsorted list is divided into two sublists of about half the size. Thereof, each sublist is sorted recursively by re-applying merge sort. Finally, the two sorted sublists are merged into one sorted list. Figure 5.8 depicts pseudocode for Merge Sort.

```
void Merge( int a[], int b[], int c[],
    int m, int n ){
  int i = 0, j = 0, k = 0;
  while (i < m && j < n){
    if (a[i] < b[j])c[k++] = a[i++];
    else           c[k++] = b[j++];
  }
  while (i < m)  c[k++] = a[i++];
  while (j < n)  c[k++] = b[j++];
}

void merge_sort( int key[], int n ){
  int *w;
    for(i = 1; i < n; i *= 2 ){
      for(j = 0; j < (n - i); j += 2 * i )
        Merge(key + j, key + j + i,
                                  w + j, i, i);
      for (j = 0; j < n; j++) key[j] = w[j];
    }
}
```

Figure 5.8: Pseudocode for Merge Sort.

```
  for (i=0; i < img_height; i++)
    for (j=0; j < img_width; j++)
      histogram[ img[i][j] ]++;
```

Figure 5.9: Pseudocode for Image Histogram.

5.3.1.3 Kernels that Require Indirect Addressing.

Image Histogram. An image histogram is a histogram which graphically represents the tonal distribution in a digital image. It plots the number of pixels for each tonal value. The horizontal axis of the graph represents the tonal variations, while the vertical axis represents the number of pixels in that particular tone. This kernel highlights the overhead for indirect address calculation. The pseudocode is depicted in Figure 5.9.

```
for(i=1; i < img_height-1; i++)
  for (j=1; j < img_width-1; j++)
  {
      GLCM[ img[i][j] ][ img[i-1][j-1] ]++;
      GLCM[ img[i][j] ][ img[i-1][j]   ]++;
      GLCM[ img[i][j] ][ img[i-1][j+1] ]++;

      GLCM[ img[i][j] ][ img[i][j-1]   ]++;
      GLCM[ img[i][j] ][ img[i][j+1]   ]++;

      GLCM[ img[i][j] ][ img[i+1][j-1] ]++;
      GLCM[ img[i][j] ][ img[i+1][j]   ]++;
      GLCM[ img[i][j] ][ img[i+1][j+1] ]++;
   }
```

Figure 5.10: Pseudocode for GLCM.

Gray-Level Co-occurrence Matrices. Gray-Level Co-occurrence Matrices (GLCM) is a tabulation of how often different combinations of pixel brightness values (gray levels) occur in an image. Second order GLCM considers the relationship between groups of two (usually neighboring) pixels in the original image. It considers the relation between two pixels at a time, called the reference and the neighbor pixel. The GLCM is used to extract statistical characteristics of the image and is used in medical imaging as well as content based image retrieval [88]. In this study, all 8 neighboring pixels are examined, as depicted in the pseudo-code in Figure 5.10. This kernel exposes the indirect address calculation overhead.

5.3.2 Large Kernels

The Large Kernels are evaluated by comparing the performance of the SPE to the performance of the PPE. The chosen kernels are the Deblocking Filter of the H.264 video decoder, which was also considered in the previous chapter, as well as the Viterbi decoder. These kernels have a more general behavior than the small kernels, mixing control and computation. They are used to analyze the impact of scalar processing overhead on SIMD-only cores to entire applications.

5.3.2.1 Deblocking Filter

As presented in detail in the previous chapter, the H.264 Deblocking Filter (DF) improves the appearance of the decoded pictures by smoothing the block edges. The DF is highly adaptive and has different filter strengths depending on the block type. The DF kernel consists of almost 400 lines of C-code and 15 functions. First, it filters the left edges of the macroblock (MB) and then it filters the vertical internal edges of its 16 4×4 blocks. This process is repeated for the horizontal edges.

The strength of the filter is determined dynamically and depends on the current quantizer, the coding of the neighboring blocks, and the gradient of the image samples across the boundary. There are five Boundary Strengths (BS) which the filter can apply, ranging from 0 (no filtering) to 4 (strongest one).

5.3.2.2 Viterbi Decoder

The Viterbi Algorithm (VA) [96] is an error-correction scheme for transmission through a noisy channel. It is used for decoding convolutional codes used in both CDMA and GSM digital cellular, dial-up modems, satellite, deep-space communications, and 802.11 wireless LANs. It is also commonly used in speech recognition, keyword spotting, computational linguistics, and bioinformatics.

A formal description of the VA is that given a sequence Z of observations of a discrete-time finite-state Markov process in memoryless noise (the interference does not depends on previous values), the VA finds the state sequence X for which the posteriori probability $P(X/Z)$ is maximal, and thus it is optimal in that sense. Therefore, the VA is a solution to the problem of Maximum A Posteriori (MAP) estimation, which tracks the state of a stochastic process with a recursive method. The MAP sequence estimation problem is formally identical to the problem of finding the shortest route through a graph.

In this study the Zero-Tail (ZT) technique [62] is used. It begins the encoding with the contents of the shift register initialized to all zeros. The Zero Tail technique works as follows. For a positive integer L, we take as the code words in our block code all sequences of length $(L + m)n$ produced by feeding the encoder with a binary sequence of length L followed by m zeros. The resultant code is an $((L+m)n, L)$ block code of rate $(1/n)(L/(L+m)) = (1/n)(1 - (m)/(L + m))$. The term $m/(L + m)$ is called the rate loss and is due to the zero tail.

5.4 Experimental Results

In this section, the experimental results are presented and analyzed. First, the results obtained using the LDT methodology are presented. It is followed by the results obtained using the SPE-vs-PPE methodology.

5.4.1 Large-Data-Type

As argued before, the LDT methodology highlights specific kernel characteristics that are known to introduce scalar processing overheads. The Small Kernels evaluated with the LDT methodology spend a significant part of their execution time on operations that require compiler-generated shuffles. Because of this, the effects of the overhead required to process scalar data can be measured. To obtain the results, the IBM Cell cycle-accurate simulator was used as it provides kernels execution details, such as the number of cycles spent in a specific area of the code, the number of single- and double-issue instruction cycles, and the number of stalls. This detailed information is necessary to evaluate all effects of the proposed methodology.

Figure 5.11 depicts the execution times of the LDT-emulated kernels normalized to the execution times of the original kernels. It also breaks down the execution time into cycles spent on actual computation, NOPs, and stalls. Stalls are further divided into load/store (L/S) stalls, stalls waiting for the shuffle unit to become available, and stalls waiting for other functional units.

Overall, the LDT-emulated versions are 19% (Merge Sort) to 57% (FIR) faster than the original versions. This performance increase comes from several sources. Without the shuffle instructions, the kernels have less operations to perform and less stalls due to structural hazards on the shuffle unit. This is represented by the decrease in computation cycles and by the elimination of shuffle stalls. The LDT methodology sometimes also affects the number of branch miss prediction stalls and other stalls, as will be discussed in more detail below.

Analyzing only the variation in computation cycles (represented by the black areas of the bars in Figure 5.11), all emulated kernels had a reduction in the number of computation cycles. For the sorting kernels, QuickSort and Merge Sort, the number of computation cycles are reduced by 21% and 27%, respectively. SAXPY present a reduction of about 40% and the remaining kernels approximately 50%. This reduction in computation cycles is mainly due to the elimination of shuffle operations. The stall breakdown shows that the

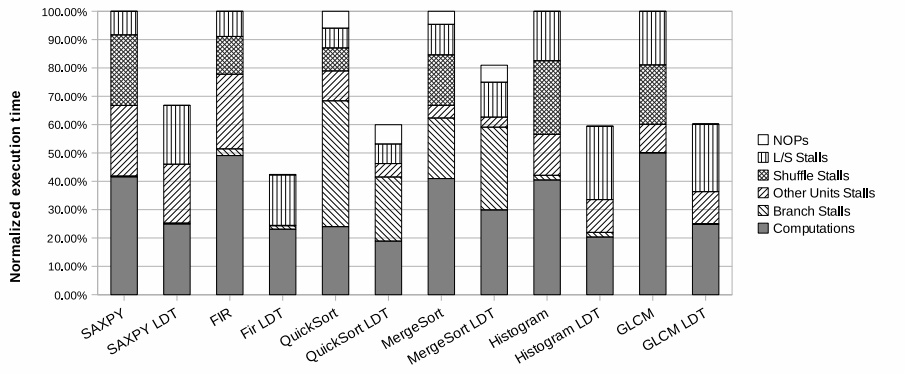


Figure 5.11: Execution times of the LDT-emulated kernels normalized to the execution times of the original kernels.

LDT methodology eliminates the shuffle stalls, as it eliminates shuffle instructions, but increases the load/store stalls, i.e., the stalls on the shuffle pipeline are transferred to the Load/Store pipeline when going from the original to the emulated version of the kernels. On average, for the original versions, the shuffle stalls account for 19% of the total number of stalls, while the Load/Store stalls account for 12%. For the emulated versions, the Load/Store stalls account for 30% of the total number of stalls. The relative increase in the number of Load/Store stalls is caused by two factors. The first is the increase in performance of the kernel. As the kernel spends less time performing arithmetic and shuffle operations, the proportion of loads and stores increases. The second effect is the increased number of loads and stores. As data are not compacted in quardwords, there are more loads and stores.

While for most kernels the percentage of branch miss prediction stall cycles does not change significantly, for the sorting kernels there is a considerable variation. For QuickSort, the number of branch miss prediction stall cycles decreases by 49% while for Merge Sort it increases by 37%. The compiler was able to increase the number of branch hints for the LDT version of QuickSort in comparison with the regular version. This explains the decrease in branch miss prediction rate for that kernel. For Merge Sort the opposite happened. The number of branch hint instructions decreased. Hint instructions have to be placed at least 8 instructions before the branch to make it useful, and more than 17 instruction to be penalty free. With the decrease of code size, due to the elimination of shuffles, the compiler has less opportunities to meet all the requirements for placing hint instructions. A vectorizable sort-

ing algorithm in tune with the characteristics of the Cell processor is presented in [41]. However, details of its performance are not presented.

Because of the small number of computations performed in each kernel, there are less than 36 live registers at any moment. For all kernels, except Merge Sort, the number of used registers decreases in the LDT version. This decrease is the result of reducing the number of intermediate steps necessary to calculate the result. Merge Sort has the same number of registers in both versions, FIR decreases the number of live registers by 14%, and the other kernels by 33%. These results show that the LDT methodology does not increase the pressure on the register file.

These results show the importance of scalar support for SIMD-only processors. The performance of every kernel increases when applying the Large-Data-Type methodology, which reveals the overhead of scalar processing on SIMD-only cores.

5.4.2 SPE-vs-PPE

In this section, the results for the SPE-vs-PPE methodology are presented and analyzed. Each kernel is executed ten times and the first run is discarded as it is used to warm-up the cache. Ten runs are performed to minimize external influences on the execution time, such as operating system context switching. The average execution time of the last 9 runs is reported.

To evaluate the performance of the kernels, the real Cell processor has been used, because the IBM Simulator does not accurately model the PPE. Execution time is measured using the hardware counters. Hardware counters have a precision of approximately one microsecond and the execution times of the kernels are considerably larger than this precision.

5.4.2.1 Deblocking Filter

The input used for testing the DF consists of the first eight frames of the Lake Wave video sequence in the QVGA (320×240 pixels) resolution. Only the 8×8 upper left MBs are filtered as a larger area would not fit in the L1 data cache. Because of this, the results differ from the ones presented in the previous chapter. Here, the presented results are the average of the runs with eight frames each. As previously mentioned, the PPE version only takes into consideration the measurements with the input data already present in L1 data cache. For the SPE version, the DMA transfer time is not included.

The scalar version running on the SPE takes on average 300 μs to filter the frame area. The scalar version running on the PPE is 12% faster; the required time to filter a frame area is on average 265 μs.

The DF has several of the typical characteristics of multimedia kernels, such as predictable data set and loops with constant number of iterations. The kernel operates on a predictable data set, which simplifies the data transfer to the Local Store. Also, the loops have static loop iteration counters, reducing loop overhead and making branch hints more effective. On the other hand, the DF is highly adaptive, featuring 4 to 6 branches in the main body of the filtering functions. Because the branch prediction in the Cell processor use simple prediction structures, as presented in Section 4.3, it has a higher miss prediction rate than high end general purpose processors. Moreover, a miss predicted branch incurs up to 18 cycles penalty. The compiler-managed overhead is masked by the miss prediction penalty, and therefore the overhead determined with the SPE-vs-PPE methodology is smaller than that found with the LDT methodology.

5.4.2.2 Viterbi Decoder

The execution time of the used Viterbi decoder is not dependent on the input data. Because of this characteristic and to avoid data transfer impact on performance, a constant input is used. The input consists of 100 blocks of 160 symbols each.

The required computation time for the SPE was 10.6 ms, while the PPE spent 6.5 ms for the same calculation. This result shows that the PPU is 39% faster than the SPE for this application. Although the Viterbi has a mix of control and computation and a large code than the Small Kernels, it is dominated by an add-compare-select structure. This structure operates on different elements of different vectors that requires them to be shifted to the preferred slot and then merged with the original data. In other words, it greatly suffers from the addressed scalar overhead. Because the add-compare-select structure is relatively small, consisting of few operations, the resulting scalar overhead is similar with the overhead found with the LDT methodology, that used relatively small kernels.

5.5 Instructions for Scalar Processing on SIMD-only Cores

To support scalar operations without sacrificing area, we propose special Scalar Load and Scalar Store instructions. These new instructions would load-/store data directly into/from the preferred slot.

For the Scalar Load an extra 4-to-1 multiplexer would be necessary to select the right word. Such an instruction was presented in [67] for accelerating table lookups. This work did not study scalar overhead but multimedia kernel acceleration in general and proposed several instructions for different bottlenecks in multimedia kernels. The instruction was applied only to table lookups in the DF kernel. It resulted in a 12% speedup for the kernel. This instruction could be used more broadly if integrated in the compiler.

For the Scalar Store a 1-to-4 demultiplexer would be necessary together with a masked write to the Local Store. The demultiplexer would not alter the store latency, since the SPE ISA includes an indexed store that adds two 32-bit words to compute the LS address. The demultiplexer is applied to the data to be stored and can operate in parallel with the address calculation. Because the Scalar Load would load the data to the preferred slot, the demultiplexer only needs to read from the preferred slot. A masked write is necessary to store only the correct word of the quadword in its final slot. For that, the write enable signal of the LS would need to be split in four individual ones, one for each word. This would require minor changes in the control logic with only the Scalar Store requiring to activate only one of the signals based on the LS address.

To fully evaluate these new instructions, the compiler would need to be modified. It is possible to write code using intrinsics to use the instructions. It would require, however, time-consuming assembly coding of kernels and applications. In spite of that, if the Scalar Load and Scalar Store instructions have the same latency as the other load and store instructions, the results should be the same as the ones acquired with the LDT methodology.

5.6 Conclusions

In this chapter, the overhead caused by scalar operations on SIMD-only architectures was analyzed. Two methodologies were presented for this analysis, the Large-Data-Type and SPU-vs-PPE methodologies. The first simulated the

behavior of a scalar unit by replacing all data types by the quadword data type, thereby eliminating the scalar processing overhead. This methodology was used to evaluate small kernels. The second methodology compared the performance of the SPE to the performance of the PPE for large kernels.

The experimental results showed that scalar support would provide considerable performance improvement for scalar, non vectorizable data kernels. Results of the LDT methodology showed that the scalar processing overhead on SIMD-only architectures ranges from 19% to 57%. For the considered large kernels, the Deblocking Filter and the Viterbi decoder, the overhead was 12% and 39%, respectively. The overhead is mainly caused by the additional shuffle instructions that need to be executed to move the scalar to the preferred slot and stalls waiting for the shuffle unit to become available.

To reduce the scalar processing overhead, a scalar unit could be added to the SPE. This, however, would require a complete re-engineering of the core including a separate register file and control logic to support many additional instructions. The resulting area and power overhead goes against the reasons given in [45] for power efficient processors. Furthermore, adding a scalar unit would be beneficial only if the area increase would be less than the performance improvement. For example, if adding a scalar unit would double the area requirements, then it would be profitable only if single-core performance more than doubles the performance, since half as many cores can be placed on a single chip. Another approach would be to add scalar instructions but to execute them on the SIMD units. This could also reduce power consumption, since the SIMD units are used to execute the scalar instructions, the unused parts can be turned off to reduce switching activity. It is reported in [43], however, that this approach was attempted during the development phase of the processor without consistent results.

To support scalar operations without sacrificing area, we proposed scalar Load and Store instructions. These new instructions would load/store data directly into/from the preferred slot and would require minimal area overhead.

6

The Multidimensional Software Cache

The next two chapters will deal with the third objective of this thesis, the improvement of the efficiency of scratchpad memories for unpredictable memory accesses. In this chapter, we propose a new type of software cache (SC) that enables the exploitation of the access behavior to reduce the traditional SC overhead. This is demonstrated by the Motion Compensation kernel from the H.264. It is also demonstrated that the proposed cache design also benefits kernels where it is not possible to exploit access behavior.

The chapter is organized as follows. Section 6.1 motivates the use of a SC for unpredictable memory accesses. Related works are presented in Section 6.2. Section 6.3 evaluates the latency and throughput of DMA operations. The MDSC implementation, properties, and its Application Programming Interface (API) are presented in Section 6.4. Section 6.5 describes the benchmarks utilized in the experiments and the MDSC optimizations for MC. Section 6.6 presents the methodology used to evaluate the proposed SC and the experimental results are presented and discussed in Section 6.7. Finally, conclusions are drawn in Section 6.8.

6.1 Introduction

Most processors use a cache to overcome the memory latency. Some processors, however, employ software-controlled high-speed internal memories or *scratchpad* memories to exploit locality, as argued in Chapter 4. Processors based on scratchpad memories are very efficient in terms of power and performance for applications with predictable memory access patterns [12]. The power efficiency is due to the simple structure of the memory compared to caches. Additionally, scratchpad memories have predictable latencies.

Many kernels, in particular multimedia kernels, have a predictable working set, which makes it possible to transfer data before the computation. Often, it is also possible to overlap computations with data transfers by means of a double buffering technique. The data in one buffer is processed while the data for the next processing stage is fetched in another buffer. In scratchpad-based systems, these data transfers usually need to be explicitly programmed using Direct Memory Access (DMA) requests. There are also many multimedia kernels, however, that process data which address is known just before it is needed. This is the case, for instance, in the Motion Compensation (MC) kernel of H.264 video decoding: Only after Motion Vector Prediction it is possible to fetch the data necessary to reconstruct the frame. Other kernels potentially have working sets that exceed the capacity of the scratchpad memory. This is the case, for example, in the Gray Level Co-occurrence Matrix (GLCM) kernel, previously described in Section 5.3.1.3 and further detailed in Section 6.5.1. It features indirect addressing and using DMA requests for each individual access is inefficient. MC is a representative kernel as its memory access pattern is similar to other important multimedia kernels, such as texture mapping. GLCM features indirect addressing that is representative of other tabulation algorithms, such as histogram calculation.

Both the MC and the GLCM kernels, however, exhibit data locality that could be exploited by a cache. In MC, the motion vectors of neighboring macroblocks (MBs) often have similar values, so their reference areas are close to each other and may even overlap. In GLCM, the difference of adjacent pixels is often small so that the kernel accesses small parts of the GLCM matrix.

In a scratchpad memory, a cache can be emulated. This is often referred to as a software cache (SC). SCs, however, incur high overhead, representing up to approximately 50% of the total application execution time [42]. Such high overheads could reduce performance compared to hand-programmed, just-in-time DMA transfers. It is therefore necessary to reduce the number of

SC lookups as much as possible. An additional feature of these kernels, as well as many other multimedia kernels, is that they access 2- or higher-dimensional data structures, and adjacent sub-rows are not stored consecutively in memory.

For such kernels that exhibit data locality which is hard to exploit with DMA transfers, we propose a Multidimensional Software Cache (MDSC). The MDSC stores 1- to 4-dimensional blocks (sub-matrices) and the cache is indexed by the matrix indices rather than a linear memory address. This approach minimizes both the DMA transfer time and the number of cache accesses. Reducing the DMA transfer time is achieved by grouping several DMA requests, thereby reducing DMA startup latency. Reducing the number of cache access is achieved by exploiting the multidimensional access behavior of the application.

6.2 Related Work

In this section, related work is discussed. First, works targeted at exploiting the multidimensional data structures used in multimedia applications are discussed. After that, works on software caches for the Cell architecture are discussed.

Cache prefeching of bi-dimensional areas to exploit the vertical locality found in multimedia applications such as video processing is proposed in several works. Cucchiara et al. [24] propose a cache prefetching strategy designed specifically to exploit the data locality found in video applications. The experimental evaluation of the technique shows that the proposed technique reduces the number of cache misses by almost half compared to the same cache configuration without prefeching. El-Mahdy et al. [35] propose an instruction for regular hardware caches to access the cache and prefetch bi-dimensional reference areas in case of a miss. It achieves speedups of almost 1.9 for MPEG2 [98] decoding and encoding applications when 7 additional cache lines are prefetched. Zatt et al. [103] show that caching the MC reference area can save up to 60% bandwidth and more than 75% of the memory cycles compared to issuing a new request for each reference area. These works show the advantages of adapting the cache to the data access patterns of the applications. In this chapter, we extend SC to higher data structure dimensions, up to 4, and exploit the multidimensionality together with known access patterns to reduce SC overhead.

The implementation of a SC for the Cell processor is an active research topic [11, 58, 18]. SCs for the Cell can be also automatically generate at com-

pile time to give the illusion of a single shared memory to the programmer [33]. Balart et al. [11] propose a SC generated by the compile that supports asynchronous transfers. The compiler uses these transfers to overlap memory transfers with computation. A speedup of 1.26 to 1.66 over synchronous transfers is reported. Chen et al. [18] propose a similar approach that supports runtime prefetching based on the access patterns. These works are complementary to our work since the MDSC can be automatically generated by the compiler.

The current version of the MDSC does not feature cache coherence. Currently, it is not needed because either the studied kernels access only read-only data or can be efficiently implemented using a map-reduce parallelization model [26] operating only on private copies of the data and thus does not require cache coherence. Cache coherence, however, is an important feature for caches in a multi-core environment. Lee et al. [58] and Seo et al. [87] propose a coherent shared memory interface for the Cell BE by using SCs. It employs a SC in the Local Store (LS) that caches memory pages, it guarantees coherence at the page level, and it uses centralized lazy release coherence.

A static analysis tool for finding the best configuration of the SC, for a given application, is proposed in [86]. The tool uses memory accesses trace files and bases its analysis on the frequency of cache accesses to each cache line and the number of accesses between two accesses to the same cache line. A similar tool for the MDSC would be desirable also as in this chapter we perform an exhaustive search to find the optimal parameters. The development of such a tool is future work.

A data movement optimization technique for software-controlled on-chip memory is presented in [40]. The authors have implemented the algorithm as an optimizing compiler and applied it to a few applications. The results reveal that the proposed technique can reduce memory stall cycles and increase performance.

6.3 Cell DMA Latency

The MDSC groups several DMA requests together in order to reduce the DMA latency. To be able to analyze this effect in this section the DMA latency is evaluated and quantified. First, we evaluate the DMA latency as a function of the DMA size, from 8 bytes to 16KB, for 1 to 6 simultaneously communicating SPEs, all SPEs available on the PS3. After that, we evaluate the latency of groups of DMA requests, or DMA lists, and compare it to the latency of individual requests.

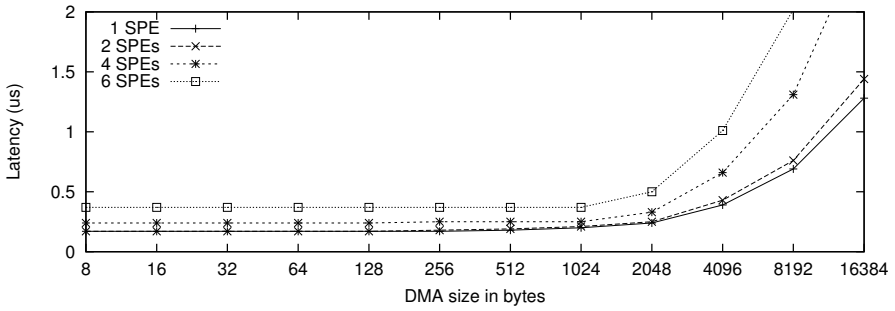


Figure 6.1: DMA latency as a function of the transfer size, for several simultaneously communicating SPEs.

Figure 6.1 depicts the DMA latency as a function of the DMA transfer size. As shown in the figure, the DMA latencies are approximately identical up to 1024 bytes for the Cell processor. After that, there is a significant increase for the 2KB and then the latencies increase linearly with the DMA transfer size. This figure also depicts the DMA latency in case of several simultaneously communicating SPUs. As the Cell processor features a dual channel memory controller, there is no difference when 1 or 2 SPUs are fetching data simultaneously. A single SPU, however, cannot simultaneously make use of both channels. Full bandwidth is achieved only when several SPUs are simultaneously accessing the external memory.

Several DMA operations can be grouped in a single DMA list operation in order to reduce the DMA startup cost. Figure 6.2 depicts the latency of DMA list operations for several numbers of requests and request sizes. We refer to the size of each individual DMA operation as the *line size*. For clarity, the results for 32- and 64-byte lines have been omitted, as they are very similar to the results for 16- and 128-byte lines. The latency for requesting the same data using sequential DMA requests is depicted for comparison.

The results show that requesting multiple lines using a single DMA list operation considerably reduces the request overhead. By using DMA list, the average request time is reduced by 50% when compared to the use of sequential DMA requests. The performance improvement ranges from 27%, when fetching 2 lines, to 69%, when fetching 64 lines.

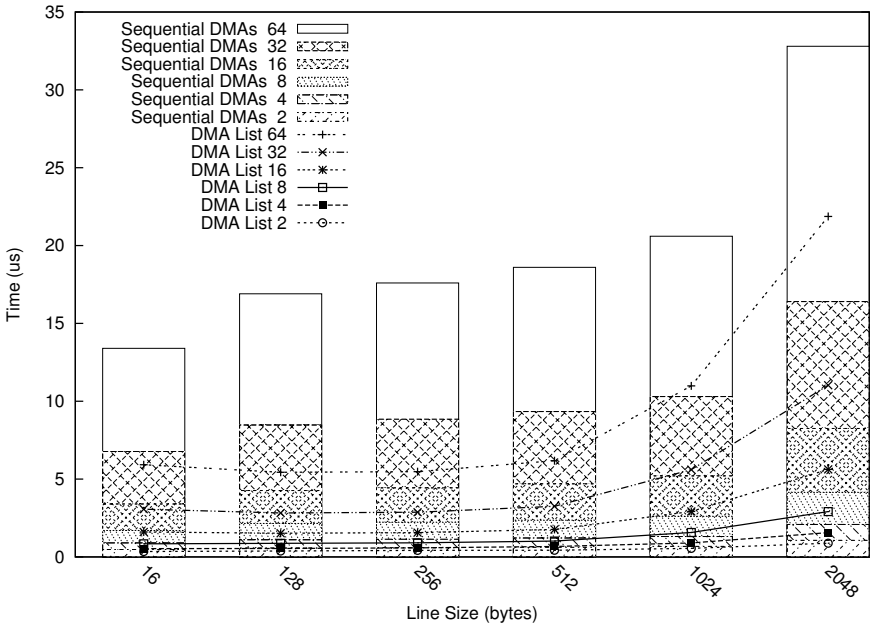


Figure 6.2: Latency of DMA list operation compared with a sequence of individual DMAs requests for the same 2D block configuration.

6.4 Multidimensional Software Cache

In this section, the Multidimensional Software Cache (MDSC) is described in detail. We start by highlighting the differences between hardware caches and SCs. After that, we present the reasons for using the data structure indices to access the cache instead of the memory address of the accessed data structure element. Next, the arguments for a multidimensional SC are given. After the motivation has been given, the proposed MDSC structure is presented.

As argued before, scratchpad memories are more efficient in terms of area and power than hardware caches [12]. They require, however, additional programming effort as explicit commands are needed to fetch data from the main memory. These commands can be automatically generated by compilers, but they are usually handled by the programmer for better performance or because of the lack of tools.

One option to increase the efficacy of scratchpad based systems is to employ a SC. SCs increase the programmability and the limited size of the LS becomes less of a concern. To access a SCs, however, incurs overhead, which

can be prohibitive. This overhead is further increased if the organization of the SC does not match the application's data access pattern. This is the case when using a generic cache for MC and for other image processing applications, such as texture mapping.

Regular hardware caches provide the abstraction of a large, fast local memory to the programmer. They capture all data and instructions used in the program. Because of its generality, the indexing by memory address becomes a natural choice. However, due to this generality and the fact that every access has to be done through the caches, it is difficult to employ application-specific techniques to reduce SC overhead.

SCs for scratchpad based processors, such as the IBM SC [101], however, capture accesses to specific data structures. Only the data structures that do not fit in the LS are likely to be accessed through the SC. In the same way as hardware caches, the IBM SC uses the memory addresses of the data to index the cache. Once again, it loses the opportunity to use the SC parameters information to exploit data locality. In this case, it becomes critical, as the access to the cache implies runtime overhead. It is possible to exploit data locality to reduce SC access overhead, due to its characteristics. These characteristics include the parameters that are known at compile time, such as data structures boundaries, and that SC accesses only few specific data structures.

To address these shortcomings of conventional SCs, the MDSC uses the indices of the accessed data structure to index the cache. So, instead of consulting the cache with a function such as $read\_SC(\&datastructure[i][j])$ we propose $read\_MDSC(\&datastructure, i, j)$, where the operation $\&$ returns a the element datastructure[i][j]. This is required to ease the calculation of data elements and cache block positions in the MDSC. Another characteristic of the MDSC is the ability to mimic in the SC the logical organization of the accessed data structure in the main memory. Replicating the memory organization in the cache set improves the performance of the cache by increasing spatial data locality. Specifically, cache blocks are 1- to 4-dimensional. Figure 6.3 illustrates 1D, 2D, and 3D cache blocks using video sequence composed of the *Lena* image. A 1D MDSC is similar to a conventional hardware cache as its stores a number of consecutive bytes (from external memory) in each block, as depicted in Figure 6.3(b). A 2D cache can be used to store rectangular image areas, as depicted in Figure 6.3(c). In this example, four vertically consecutive segments of the image lines are allocated per MDSC block. 3D MDSC blocks are a set of consecutive (in the third dimension), co-located (i.e., with the same vertical and horizontal coordinates) 2D blocks. A 3D MDSC can be used to

(a) Original image being cached.

(b) Eight 1D MDSC blocks.

(c) Two 2D MDSC blocks, four lines high.

(d) Two 3D MDSC blocks, four lines high and 2 images deep.

Figure 6.3: Examples of 1D, 2D, and 3D MDSC blocks.

store areas of a sequence of video frames, as depicted in Figure 6.3(d). Similarly, a 4D MDSC, not depicted, can be used in Multiview video processing or animated 3D representations as its blocks are sets of 3D blocks. Because of the SIMD instructions set of the SPE, 4D cache blocks are as efficient as 2D and 3D cache blocks.

The MDSC differs from a regular SC in two ways. First, it differs in the tag and index functions. In the MDSC, the tag is given by the concatenation of each index and the set is a mask operation based on the indices. The second

difference is that it employs multidimensional cache blocks instead of one-dimensional cache blocks, which allows using DMA list to read from and write to memory. A strided access to the main memory is performed by a DMA list to load the multidimensional block data.

The MDSC presents two advantages over regular SCs. First, it reduces the DMA latency by grouping several memory requests. As shown in the previous section, a single DMA list operation has a lower latency than several sequential DMAs that achieves the same effect. For conventional SCs, the access of a new image area would result in a new DMA request for each line of the new area being accessed. The second advantage is that the MDSC can be used to reduce the number of accesses to the SC. As the shape of the cache block is known, it can be used to access the data of a cache set without actually checking if the data is present in the cache. This can be done by simple pointer arithmetic. In other words, a single cache lockup is necessary for accessing an entire block. This is depicted in an example further in this section.

The associativity of the MDSC can be configured. It allows to fully or a set associative configuration and a static or dynamic implementation. A fully associative cache is possible when the number of cache blocks is small. For the fully associative and the set associative configurations, the MDSC uses a First-In-First-Out (FIFO) policy to replace the blocks when the cache or the set, respectively is fully utilized. The FIFO policy was selected due its low implementation complexity. In the static implementation, the MDSC parameters, such as block dimensions and the indices range of the data structure being cached, are constants and thus known at compile time, while the dynamic implementation allows modifying the MDSC parameters at runtime. The static implementation is more efficient than the dynamic implementation as it allows optimization of the code for several parameter options. A dynamic configuration, however, is necessary when the data to be cached can have different characteristics, such as the resolution of the video being decoded. As a result, the MDSC needs to adapt to these characteristics.

Similar to a one-dimensional SC, the MDSC performs the following steps to access an element. The flow chart of the operations is depicted in Figure 6.4.

1. *Tag Calculation*. The first operation when accessing the cache for read or write is to calculate the *tag*. The tag is formed by concatenating the up to 4 indices of the data element that is being accessed after dividing each index by the size of its respective block dimension.

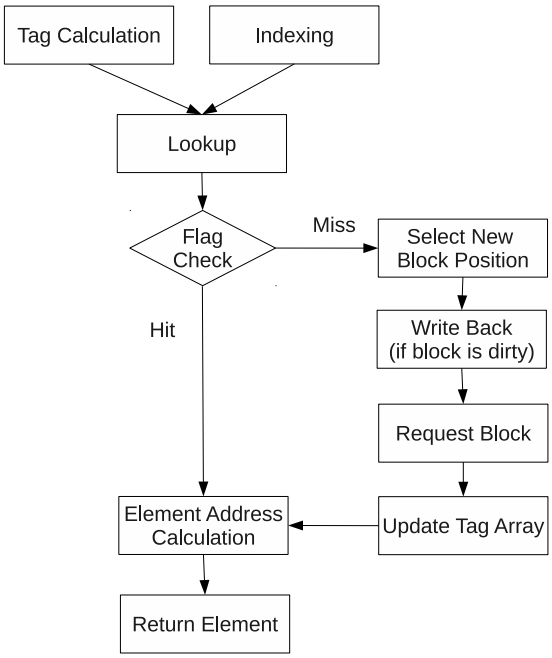


Figure 6.4: Flow chart for accessing an element from the MDSC.

2. *Indexing*. In the case of a set associative cache, a hash function is used to define in which set the block referred to by the indices is/should be stored. The difference from a regular cache hash, is that the MDSC hashes the data structure indices instead of the address of the data structure element being accessed. For a fully associative implementation, this step is not required.

3. *Lookup*. The lookup function for the MDSC is similar to the ones in regular caches. For a set associative cache, lookup checks if the tag is present in the elements of the tag array corresponding to the set produced by the indexing function. For a fully associative cache it scans the whole tag array to check if the tag is present or not. In both cases, it returns the cache block number in which the requested data is located or a flag indicating a miss.

4. *Select New Block Position*. In case of a miss, a place for the new block should be selected. For a set associative cache, the new block should be placed in one of the ways that belongs to the set computed by the indexing function. The next free block position is selected for the new block.

As described above, the MDSC uses the FIFO replacement policy to select which block should be evicted in case there is no space available.

5. *Write Back*. If the cache is not read-only and the contents of the block to be evicted was modified (marked as dirty), it should be written back to memory. The address of the block in the external memory is recovered using the base address of the data structure (passed as parameter), the tag of the block, and the size of the block dimensions. A DMA $putlist$ command is issued to write back the contents of the block.

6. *Request Block*. Similarly to the previous operation, the address of the block in the external memory is recovered using the base address of the data structure (passed as parameter), the tag of the block, and the size of the block dimensions. A DMA $getlist$ command is issued to read the block from the external memory. After that, the tag array is updated.

7. *Element Address Calculation*. The position of the requested data in the cached block is calculated using the data structure indices passed as parameters.

The MDSC API features three main functions: *read_MDSC(&datastructure, i, j)*, *write_MDSC(&datastructure, i, j, data)*, and *pointer_MDSC(&datastructure, i, j)*. The function *read_MDSC* returns the data stored at position (i, j), while *write_MDSC* writes the data at position (i, j). The function *pointer_MDSC* returns the data address (memory pointer) in the LS and marks the cache block as dirty, if it is not a read-only cache. These functions check if the 2D block, which contains *datastructure[i][j]* is present in the SC. If it is, these function return immediately. If not, these function block until the 2D block containing the *datastructure[i][j]* is fetched from the main memory to the SC and then returns to the caller. The MDSC API also includes functions for modifying the parameters of the dynamic implementation.

As the MDSC uses matrix indices to index the cache, the boundaries of the data structure need to be specified to calculate the memory address of the cache blocks. This can be done via macros in the static configuration, thus increasing the performance, or dynamically at runtime, when using the dynamic configuration. Figure 6.5 depicts how the MDSC can be configured. In this figure, the configuration for a three dimensional static MDSC for Full High Definition (1080×1920) resolution is presented.

Figure 6.6 depicts a pseudo C code for a matrix multiplication as an example of use for the MDSC. The algorithm multiplies matrices *MA* and *MB*

```
CACHE_NAME         gsc_y
CACHED_TYPE        unsigned char
CACHE_TYPE         0   // 0=read-only, 1=read-write
CACHE_STATS            // Activates statistic collection
CACHE_FULL_ASSOC   0   // 1=cache is fully associative
CACHE_LOG2NWAY     2   // Log 2 number of ways
CACHE_LOG2NSETS    5   // Log 2 number of sets
CACHE_DIM          3   // Number of block dimensions

CACHE_X_LOG2SIZE   9   // Log 2 first dimension (line size)
// When accessing the MDSC, the first index must be
// between 0 and CACHE_X_RANGE
CACHE_X_RANGE      1920
CACHE_LOG2_X_RANGE 11  // Log2(CACHE_X_RANGE)

CACHE_Y_LOG2SIZE   4   // Log 2 second dimension
                       // (number of lines)
CACHE_Y_RANGE      1088
CACHE_LOG2_Y_RANGE 11  // Log2(CACHE_Y_RANGE)

CACHE_Z_LOG2SIZE   1   // Log 2 third dimension
CACHE_Z_RANGE      256
CACHE_LOG2_Z_RANGE 8   // Log2(CACHE_Z_RANGE)

#include <mdsc-api.h>
```

Figure 6.5: The MDSC configuration interface.

and saves the result in matrix *MR*. The three matrices are square and their high and width are determined by *Matrix_size*. Matrices *MA* and *MB* do not fit in the LS and are cached by a bi-dimensional MDSC with square cache blocks of $2^{CACHE\_X\_LOG2SIZE} \times 2^{CACHE\_X\_LOG2SIZE}$, where *Matrix_size* is divisible by $2^{CACHE\_X\_LOG2SIZE}$. For brevity, and because it can be easily transfered by DMAs (not depicted), *MR* is not cached. Matrices *local_MA* and *local_MB* have the same size as the MDSC block and are used to point to the cache blocks. These matrices are used instead of pointers to avoid pointer arithmetic and improve code readability.

The matrix multiplication is performed as follows. The *for* loops in lines 21 and 22 iterate over all the elements of the matrix *MR*. The *for* loop in line 26 divides *MA* and *MB* into partitions of *local_size* size. The objective of this partitioning is, as argued before, to have only one cache access per cache block. To accomplish this, first the top leftmost element of the partitions in which elements *MA[i][k]* and *MB[k][j]* needs to be determined, as it gives the base address of the bi-dimensional MDSC block in the LS. The top leftmost element of the partition is calculated by lines 31 and 32. *k* always points to the first column or row in a partition/cache block, as stated in line 26. With *[li][k]*

and *[k][lj]* pointing to the top leftmost element in a partition, they are used to get a pointer to the base address of the bi-dimensional MDSC blocks that contain the elements to be accessed. The function *pointer_MDSC(&MA,li,k)* checks if the element *MA[li]][k]* is currently being cached, request the data if necessary, and returns the pointer to the element, that in this case is also the base address of the cache block. In lines 36 and 37, the base address of the matrices *local_MA* and *local_MB* are set to the base address of the blocks in the MDSC that contain the desired elements. Finally, in lines 40 to 46, the actual matrix multiplication can be performed for the element *MR[i,j]* without any further access to the cache while in the same partition. The modulo operations for *i* and *j* map their coordinates to the current partitions.

6.5 Studied Kernels and MDSC Enhancements

In this section, we present the kernels used for the case study. Additionally, qualitative reasons are given why the studied kernels could profit from a MDSC. First, the GLCM algorithm described once more to explain the issues related to DMA transfers when the dataset does not fit in the LS. After that, a description of MC is presented. For the MC kernel, it is possible to exploit the cache access pattern, as the MC kernel operates on rectangular areas of images. This property will used to describe several MDSC enhancements.

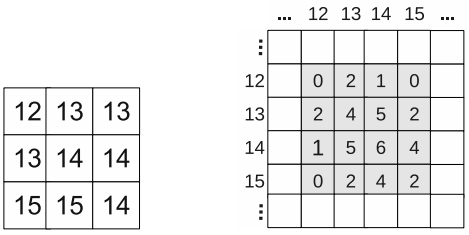
6.5.1 GLCM Kernel

As previously described in Section 5.3.1.3, the GLCM is a tabulation of how often different combinations of pixel brightness values (gray levels) occur in an image. The second order GLCM considers the relationship between groups of two (usually neighboring) pixels in the original image. It considers the relation between two pixels at a time. The pseudocode for the GLCM is depicted in Figure 5.10, page 92.

In this kernel, the source image being processed can be easily accessed through DMAs. The temporal locality of the image is very low and the spatial locality can be captured with DMAs. Also the DMA latency can be hidden using double buffering. Differently from the GLCM kernel used in the previous chapter, where the pixels of the image were quantitized so the matrix would fit in the LS, the inputs are now full range. As each pixel is 8-bits, the total GLCM matrix size is $256 \times 256 \times 4$ bytes = 256KB. This is the entire size of the LS and this is only for one color component of the image. Differently from

```
 0  /* DATA STRUCTURES IN EXTERNAL MEMORY                          */
 1
 2  int Matrix_size; //High and width of matrices
 3  int MA[Matrix_size][Matrix_size];//Matrix A in external memory
 4  int MB[Matrix_size][Matrix_size];//Matrix B in external memory
 5
 6  /*Result matrix in external memory. MR is not cached */
 7  int MR[Matrix_size][Matrix_size];
 8
 9  /* DATA STRUCTURES IN LOCAL MEMORY                            */
10  /* High and width of the local caches is the same as the MDSC*/
11  /* blocks */
12  int local_size = power(2, CACHE_X_LOG2SIZE);
13
14  /* Local arrays. Used for easy address calculation            */
15  int local_MA[local_size][local_size];
16  int local_MB[local_size][local_size];
17
18  /* PROCESSING                                                 */
19  /* MR[0][0]=MA[0][0]*MB[0][0]+                                */
20  /*          MA[0][1]*MB[1][0]+MA[0][2]*MB[2][0];              */
21  for(i=0; i < Matrix_size; i++){
22     for(j=0; j < Matrix_size; j++){
23        MR[i][j] = 0;
24
25        /* Iterates over cache blocks                          */
26        for(k=0; k < Matrix_size; k = k + local_size){
27
28           /* Calculates block borders for i and j.            */
29           /* Used to select the top leftmost position         */
30           /* of a MDSC cache block, i.e., its base address    */
31           int li = i - i%local_size;
32           int lj = j - j%local_size;
33
34           /* Changes the base address of the local matrices   */
35           /* to the base address of the MDSC block            */
36           &local_MA = pointer_MDSC(&MA,li,k);
37           &local_MB = pointer_MDSC(&MB,k,lj);
38
39           /* The actual matrix multiplication                 */
40           for(l=0; l < local_size; l++){
41              /* i and j mod local_size to access elements     */
42              /* inside the actual cache block                 */
43
44              MR[i][j] +=  local_MA[i%local_size][l] *
45                           local_MB[l][j%local_size];
46           }
47        }
48     }
49  }
```

Figure 6.6: Matrix multiplication using 2D MDSC.



(a) Subblock of a 3×3 image.

(b) Access to the GLCM matrix.

Figure 6.7: Second order GLCM of a 3×3 image.

the source image, it is not possible to determine in advance which element of the matrix will be accessed to make use of the DMAs.

Photos usually exhibit large amounts of spatial redundancy, which is exploited by image compression algorithms. The same type of redundancy can be exploited here by caches. Because the change of color is usually smooth, two-dimensional portions of the GLCM matrix are likely to be accessed close in time. In other words, the spatial redundancy of photos is translated into spatial or even temporal locality, when updating the GLCM matrix. Figure 6.7 illustrates this. Considering the top leftmost pixel, with value 12, in the image depicted in Figure 6.7(a), it access the following elements of the GLCM matrix, depicted in Figure 6.7(b): [12,13], [12,13], and [12,14].

As depicted in Figure 6.7(b), and in the pseudocode depicted in Figure 5.10, page 92, we fix the first pixel while visiting all its neighbors, i.e., for a given pixel only its GLCM matrix line will be updated while visiting its neighbor pixels. This property can be used to implement the GLCM using DMAs. For that, the entire line of the GLCM matrix corresponding to the value of the first pixel is copied to the LS, where each GLCM matrix line size is 1KB. With the GLCM matrix line in the LS, each neighbor pixel is visited and its corresponding position in the GLCM matrix line updated. After all neighbor pixels are visited, the GLCM matrix line is stored back to the main memory via DMA.

6.5.2 H.264 Motion Compensation

Motion Compensation [9] is the process of copying an area of the reference frame to reconstruct the current frame. It is depicted in Figure 6.8. The figure

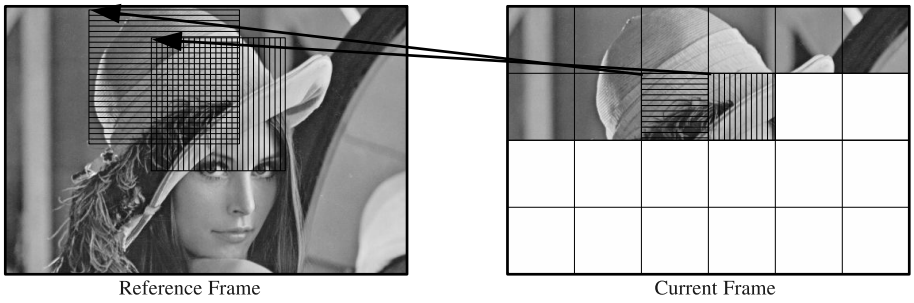


Figure 6.8: Motion Compensation of two macroblocks with respective motion vectors and reference areas.

depicts the MC for two neighbor MBs. Each MB has a Motion Vector (MV) that points to an area in the reference frame, this area is called reference area. Because of the MC applies a filtering process on the reference area to achieve quarter-sample accuracy, the reference area is larger than the MB. In H.264, a MB can be partitioned in two 16×8 or two 8×16 or even four 8×8 partitions. Each 8×8 partition can be further partitioned in up to four 4×4 partitions. Each of these partitions, also called MB partition, have its own MV. Because of MVs are usually related with its neighbors MB MVs and because of the extra data required for quarter-sample processing, reference areas of neighbor MB partitions overlaps, as depicted in Figure 6.8.

For advanced video codecs such as H.264, both the reference frame and the MVs need to be calculated. In H.264, this process is known as Motion Vector Prediction (MVP) and is part of the MC. Only after MVP it is possible to request the data necessary to reconstruct the frame. In H.264, MVs can span half of the vertical frame size and it is possible to have up to 16 frames as reference frame candidates. This makes it impossible to speculatively load all possible areas in advance.

In our vectorized SPE implementation of macroblock (MB) decoding, the MC kernel is the most time consuming, representing 62% of the total execution time. It fetches the reference area using DMA transfers and waits until the data is present in the LS. The remaining execution time is spent in DMA transfer other than the reference area (14%); deblocking filter (17%); and Inverse Discrete Cosine Transform (7%). Furthermore, the memory requests represent 75% of the execution time of the MC kernel. These numbers show how important it is to improve the performance of MC.

The unpredictability of the data accesses in MC causes two significant

problems for scratchpad memory-based processors. The first problem is that the data transfer cannot be overlapped with the computation. The process has to wait for the data to be transferred to the scratchpad memory. Because H.264 allows very fine grained area to be motion compensated, up to 4x4 pixels, the waiting time for the data can be significant. The second problem is that the data locality cannot easily be exploited. In order to do so, one has to keep track of areas present in the LS. This, however, is difficult and new data must be requested for each MB partition. Because the MVs are usually small and not randomly distributed, the same area can be copied several times. Zatt et al. [103] show that caching the reference area can save up to 60% bandwidth and more than 75% of the memory cycles.

In the next section, we investigated the data locality exhibited by the H.264 MC. After that, in Section 6.5.4, we introduce the enhancement to the MDSC for the MC kernel. These enhancements exploit the multidimensionality and the use of the indices of the data structure to index the MDSC.

6.5.3 Data Locality in MC

To evaluate the data locality, the number of bytes transferred from memory to a conventional cache is measured. H.264 sequences from HDVideoBench [7] are used as input for the experiments. As previously mentioned in Section 2.4.2, each video sequence consists of 100 frames in standard (SD), high-definition (HD), and full high-definition (FHD) resolutions at 25 frames per second. For this measurement, the motion vectors and reference indices have been extracted from the encoded sequences for each MB partition. Because of the MC quarter-pixel precision, adjacent additional areas need to be fetched from the memory. For vertical filtering, five extra pixels are required for each line, while for horizontal filtering, five extra lines are required. Details on the MC implementation can be found in [10].

A tool was developed to translate the extracted MVs to memory requests in the DineroIV [32] cache simulator input format. DineroIV was used to report the requested number of bytes for each sequence. Three simulations were performed and the results are reported in Figure 6.9. The first simulates a 1-byte cache to depict the temporal data reuse of MC. The second simulation reports the data traffic for a 16-byte cache with a 16-byte line size. These first two caches are used to calculate how much data is transfered from external memory to a scratchpad memory when using a DMA requests. The third simulation reports the data traffic for a 64KB direct mapped cache with 64-byte cache lines. The size of the original uncompressed sequence is presented as

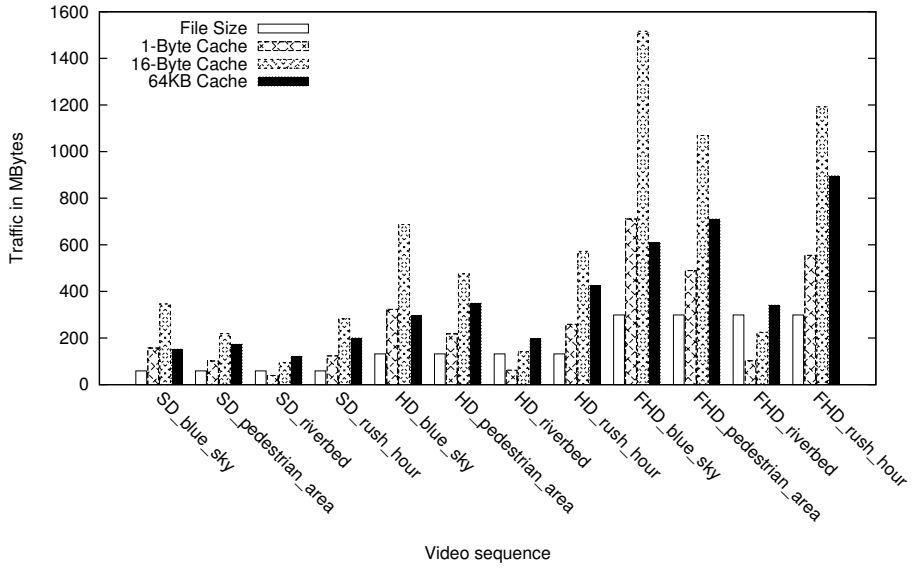


Figure 6.9: Data locality in MC.

file size. Ideally, the total traffic should have the same size as the original file size, meaning that the same data is not requested twice. The increase in data traffic, compared with the original file size, means that some of the data is transfered more than once.

The 1-byte cache assumes that it is possible to transfer individual bytes from the external memory. Because it caches Y, Cb, and Cr color components, there are no cache hits. The amount of data traffic for the 1-byte cache depicts the amount of data that is request more than once from the main memory.

The 16-byte cache consists of a single 16-byte cache line. Its purpose is to present how much data is actually transferred from the main memory, given a more realistic 16-byte memory channel. For this cache, spatial locality is exploited as, differently from the 1-byte cache, the 16-byte cache present hits as several pixels from a reference area line can be stored in a single memory word. It presents, however, a larger data traffic than the 1-byte cache as several elements of the requested memory word do not belong to the reference area.

Finally, the 64KB cache indicates how much data traffic can be saved by a regular cache. In most of the cases, it presents a lower amount data traffic than the 16-byte cache. This is expected as this cache has space to store the previous requested reference areas. For the Riverbed sequences in SD, HD, and FHD resolutions, the amount of data traffic is larger than the 16-byte

cache. This is caused by the low use of MC in the Riverbed sequence, which makes the MB featuring MC scattered over the frame with a small number of overlapping reference areas.

The results show that the sequences exhibit data locality. The MC kernel references about twice the volume of data of the original sequence. Because of memory alignment constraints, however, the actual volume of transferred data is about 3.5 times the volume of the original sequence. The 64KB cache reduces the volume of data transferred by 34% on average compared to the 16-byte cache. It reduces the data volume to 2.3 times of the original sequence. This indicates that the cache is capturing part of the data locality of the MC. It can be improved as it is an unified cache capturing the accesses to the three different color components, thus increasing conflicts.

6.5.4 The MC Enhancements

The MDSC uses the frame number and the vertical and horizontal coordinates of the MV as indices. This access method enables to exploit the access pattern as it exposes pattern specific information. Each block of the MDSC is a $x \times y$ rectangular area of a frame. The x and y values and their ranges are configurable at runtime in the dynamic implementation.

Because of reference frame areas are only likely to be reused in neighbor MBs, a fully associative configuration was selected, as it reduces the number of conflicts. A fully associative cache is possible because of the small number of blocks present in the implementation. To support different video resolutions, the MC has to use a dynamic configuration of the MDSC.

The first enhancement to the MDSC for MC is the use of a single cache lookup for the Y, Cb, and Cr caches. In H.264, the video frames are stored in YCbCr format instead of the RGB format, and each component is stored in a separated data structure. To increase the compaction, the color components (Cb and Cr) are subsampled 1:4 as they are less perceptive to the human eye. The MVs are the same for all components. Because of the subsampling, however, they need to be adjusted for the Cb and Cr components. The MDSC configuration exploits this feature and checks and requests all components at once. This reduces the number of accesses to the MDSC by a factor of 3 and overlaps the memory requests, thus reducing the memory latency.

Four additional enhancement strategies employed to reduce the number of accesses to the MDSC are described below. Each strategy builds upon the former strategy.

6.5.4.1 Extended_Line

An extended line technique was implemented, based on the technique described in [10], to reduce the number of accesses to the SC. In this technique, for each line of the reference area to be accessed through the MDSC, only the first element the line is checked in the MDSC. This is made possible by adding extra columns to the bi-dimensional cache block. These additional columns, however, are non indexable, i.e., a lookup for these columns would result in a miss (considering that they are not present in another cache block). By adding these as extra columns, it is guaranteed that if the first element of the reference area line is in the cache, all remaining elements of the reference area line area are present in the same cache block. This reduces to one the required number of cache accesses per line in the reference area.

Figure 6.10 illustrates this technique. It depicts a 2D MDSC block of 96×32 pixels, or 6×2 MBs (delimited by the dashed lines). The areas labeled *a*, *b*, and *c* denote reference area requests to the MDSC. The first element of the reference area lines are marked as black. The extra columns is depicted by the hashed area. Without the Extended_Line technique, a MDSC lookup is necessary for every first element of each quadword that belongs to the reference area line, as it can be split across two cache blocks. As indicated by the reference area *c*, with the extra columns all elements of a reference area line are enclosed in a cache block. Therefore, MDSC lookups are necessary only for the first element of the reference line.

This technique is based on the following observation. The reference area line to be accessed is at most 21 pixels long. These pixels consist of the 16 pixels of the maximum MB partition width plus 5 extra pixels for quarter-pixel filtering. These pixels can be located in up to three consecutive quadwords and requires 32 extra columns for each 2D block. Note that this implies that these additional columns can be present twice in the MDSC, as they need to be in another MDSC block to be directly accessed. As only read-only data is cached, this does not cause inconsistency problems.

6.5.4.2 Extended_XY

The Extended_XY enhancement reduces to two the required number of MDSC accesses per MB partition. This technique, which is build upon the Extended_Line technique, can be applied only when the high of the MDSC block is equal or larger than 21 lines. As the maximum MB partition plus the additional area is at most 21 lines in the vertical direction, just two accesses to the

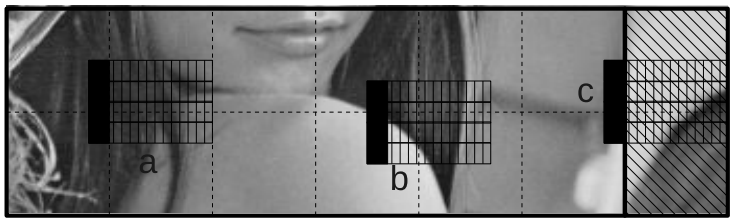


Figure 6.10: A bi-dimensional cache block with the Extended_Line enhancement.

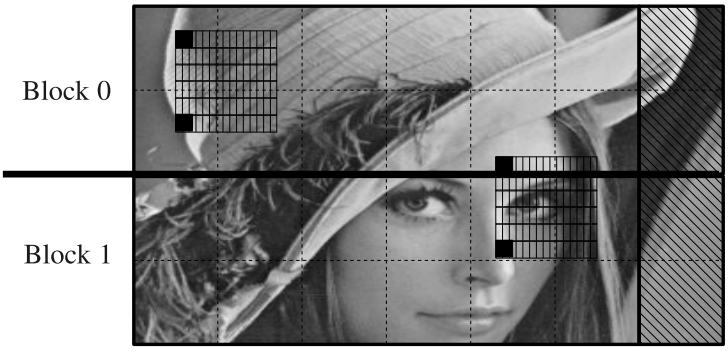


Figure 6.11: Example of use of the Extended_XY enhancement.

MDSC are sufficient to guarantee that the data are present in the cache. Only the first and last lines of the partition need to be accessed. The border between the two cached blocks is found by masking the y coordinate of the MV with the height of the block.

Figure 6.11 illustrates this concept, the same notation as described in the previous section apply. In this figure, a vertically contiguous area of the reference area is stored in MDSC blocks 0 and 1. A single reference area cannot be split in more than two cache blocks because of the Extended_Line enhancement and because the highest allowed reference area is smaller than the high of the cache block. Therefore, MDSC lookups are necessary only for the first elements of the first and last lines of the reference area, marked with black in the figure.

6.5.4.3 SIMD

The Cell SPE is a SIMD architecture. As a result, a natural step to improve the performance is to vectorized the tag search with SIMD instructions. Each tag is a 32-bit integer and the SPE supports vector operations on four 32-bit word

operands. In this optimization, four positions of the tag array are compared simultaneously with the searched tag. Once the tag is found, each of the four positions of the tag array are individually compared to find the block index.

6.5.4.4 Static

As previously stated, in the dynamic implementation the parameters for the MDSC are configurable at runtime. In this analysis, however, we use the static MDSC implementation. This means that loop boundaries are known at compile time. This allows certain loop optimizations to be performed, including the elimination of branches and loop unrolling.

6.6 Experimental Methodology

This research focuses on the performance of the cache access functions. Because of that, the kernels that access the SC will be measured. For the MC, the access to the reference area is evaluated. For the GLCM, the whole function is measured. This is because the GLCM only performs a load, an add and a store for each element. The images are loaded through explicit DMA transfers and these are not part of the GLCM calculation. The SPE hardware counters are used to measure the performance of the kernels.

The measurements were performed on a PS3, which, as described in Section 4.3, page 4.3, has only 256MB of RAM. This small amount of memory causes memory swaps with the disk. For this reason, the MC kernel was modified to access only 5 frames, which corresponds to the number of frames in the decoder frame buffer. If this would not have been done, the DMA transfer time is doubled due to memory (de)allocation routines by the OS.

The HDVideoBench [7], described in Section 2.4.2, is again used as benchmark for the experiments. All results were obtained using a single SPE. The experiments were not performed using several cores because the parallelization strategy would influence the results and the focus of this work is on single core performance. Both kernels can run in a multi-core environment without cache coherence. The GLCM kernel could have a separated GLCM matrix for each core and process a slice of the frame. After finishing the processing, the matrices would need to be added together for the final result. For MC the cache is read-only, thus it does not need cache coherence to work in a multi-core environment.

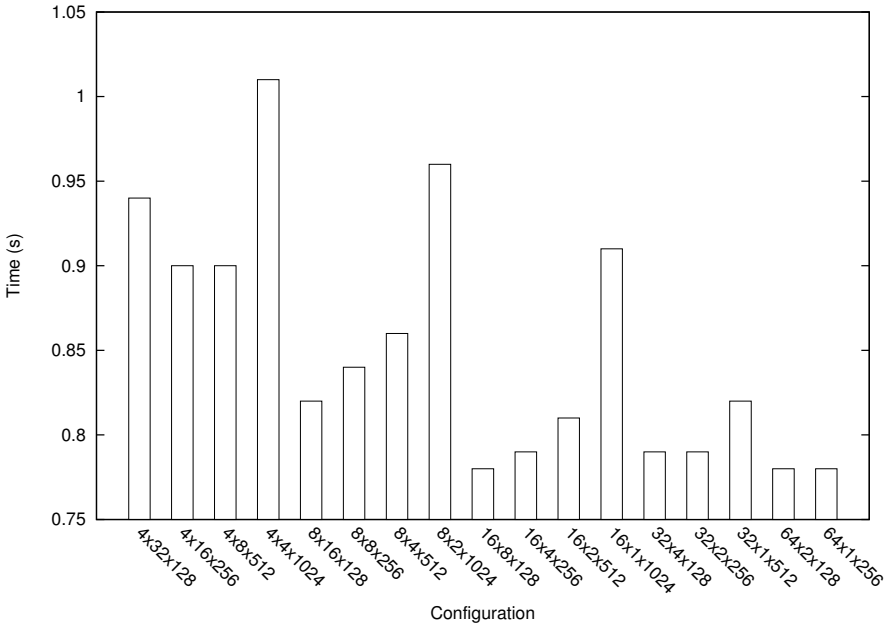


Figure 6.12: Time taken by the GLCM kernel for several MDSC configurations.

6.7 Experimental Results

6.7.1 GLCM Results

To generate input for GLCM, the first frame of each HDVideoBench sequence was transformed into an RGB image and each component was processed. Several configurations of a 4-way set associative 64KB MDSC were tested to determine the optimal configuration. The number of sets was varied from 4 to 64, the number of lines from 1 to 32, and the line size from 128 to 1024 bytes.

Figure 6.12 depicts the results of all possible 64KB configurations. In this figure each bar is labeled as $S \times L \times B$, where S, L and B are the number of sets, lines in a 2D block, and line size in bytes, respectively. Thus, the 2D block size of this MDSC configuration is 8×128. As shown in the figure, the GLCM performs better with a larger number of sets and with a smaller line size. Because of this behavior, a fully associative MDSC was not evaluated, as it requires the opposite to perform well.

The best performing MDSC configuration consists of 64 sets and uses a block size of 1×256 bytes. Surprisingly, 1-dimensional blocks yield the

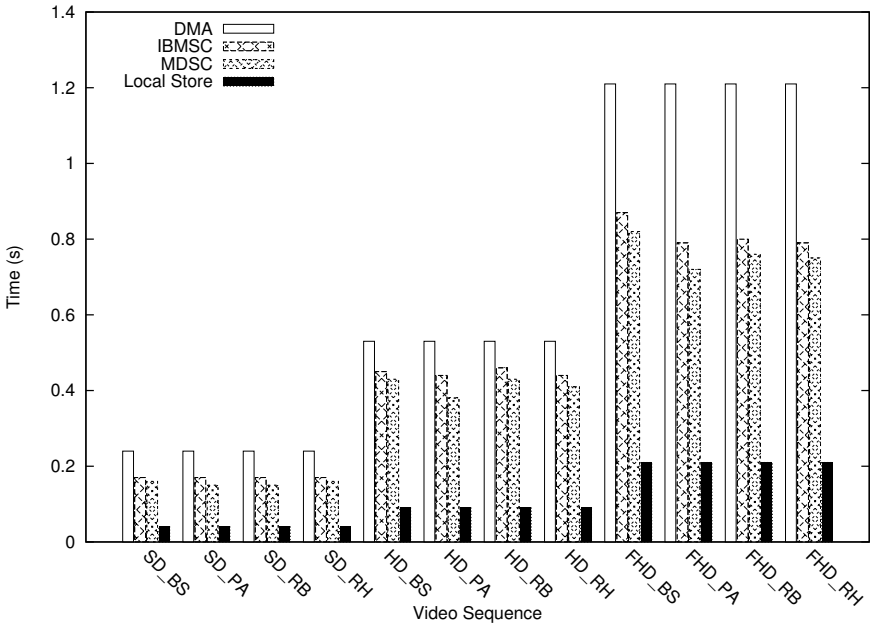


Figure 6.13: Time taken by the GLCM kernel for several video sequences when using DMA transfers (DMA), when the optimal IBM SC configuration is employed (IBM SC), when the optimal MDSC configuration is employed (MDSC), and when the GLCM matrix would fit in the Local Store.

highest performance. The reason for this is the time saved by reducing miss rate when increasing the block height is smaller than the extra time spent on the larger DMA list requests. However, the performance achieved by the configurations that use blocks of 2×128 bytes and 8×128 is less than 1% lower than the performance of the best configuration.

Figure 6.13 compares the performance of the optimal MDSC configuration to the optimal configuration of the IBM SC. Experimentally, we determined that the optimal 64KB IBM SC configuration for GLCM is 4-way set-associative, consists of 128 sets, and uses a line size of 128 bytes. For comparison purposes, the time taken by the GLCM kernel when using DMA requests, as detailed in Section 6.5.1, and the GLCM matrix fits in the LS are also depicted. To fit the GLCM in the LS, the image color resolution had to be quantitized to 6 bits, as in the experiments described in Chapter 5.

Both the IBM SC and the MDSC present performance improvement when compared with the GLCM version implemented using DMAs. The IBM SC has, on average, a speedup of 1.5 compared with the DMA version for the

FHD resolution. For the same resolution, the MDSC has an average speedup of 1.6. While for the SD resolution the speedups are similar with the FHD speedups, 1.4 and 1.5, for IBM SC and MDSC, respectively, for the HD resolution the speedups are lower, 1.2 and 1.3, for IBM SC and MDSC, respectively.

Compared to the IBM SC, the MDSC provides an 8% performance improvement on average. This performance improvement is due to the lower miss rate achieved by the MDSC. For example, for the FHD Blue Sky sequence (denoted FHD_BS in Figure 6.13), the MDSC incurs a miss rate of 2.4%, while the IBM SC incurs a miss rate of 2.6%. This 0.2% difference in miss rate translates to an 8% increase in the number of memory requests by the IBM SC compared to the MDSC. It also increases the number of times the miss handling code of the SC is executed. The miss handling code is much more time demanding than the hit handling code, as it has to choose a block to replace, calculate the block memory address based on the old tag, and issue a request for the new block.

For GLCM, the MDSC set hash function, based on indices instead of linear addresses, reduces the number of conflict misses compared to the IBM SC. The MDSC hash function more equally distributes the number of accesses over the sets. For example, for FHD Blue Sky, the average deviation of the number of accesses to each set of the MDSC is reduced by 18%, when compared with the IBM SC with the same configuration. The MDSC features the same configuration and replacement policies as the IBM SC, the only difference between them in this particular case is the index calculation. With a better distribution of the accesses, the number of replacements is lower, which reduces the miss rate.

Compared to the case when the GLCM matrix fits in the LS, which is included only for comparison, the MDSC incurs a slowdown of 3.75 on average. Considering that the GLCM kernel consists of simple processing and the fact that an access to the MDSC takes around 45 cycles, the MDSC (as well as the IBM SC) bridge the memory gap quite efficiently.

6.7.2 MC Results

In order to determine the optimal 2D block size of the MDSC for the MC kernel, first the design space was explored. The MVs from the HDVideoBench sequences were used as input. MVs and reference indices were extracted from the encoded sequences for each MB partition. Blocks of size $n \times m$ were

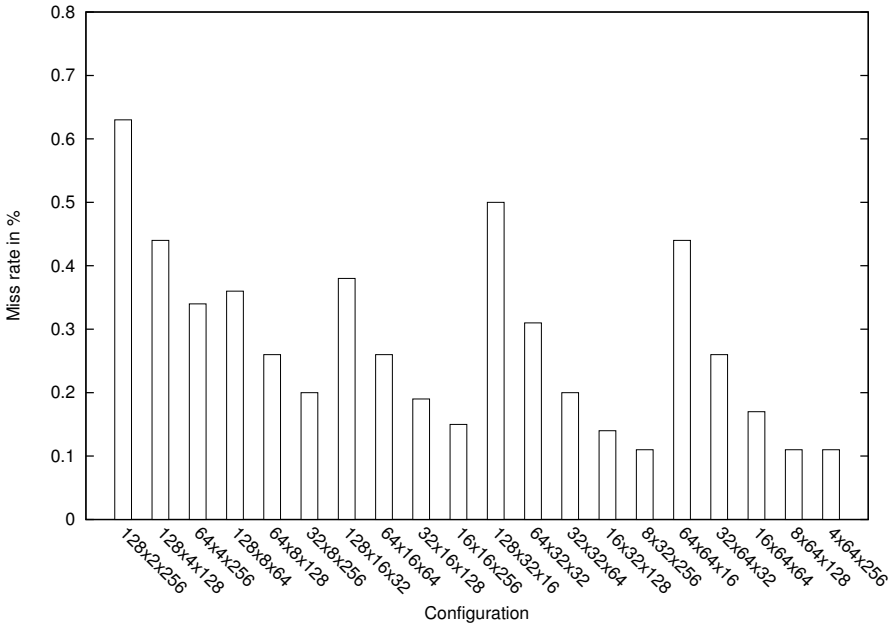


Figure 6.14: Miss rate incurred by the MC kernel for different configurations of a 96KB MDSC.

tested, for n between 1 and 64, and for m between 32 and 256. For each block size, the miss rate was calculated. The size of the MDSC was fixed at 96KB, 64KB for the Y components and 16KB each for the Cb and Cr components. As mentioned in Section 6.5.2, the MDSC for MC is fully associative. None of the enhancements discussed in Section 6.5.4 were considered in this exploration.

Figure 6.14 depicts the miss rate for each design point. It uses the same labeling style as the previous figures with the number of blocks in the cache, number of lines per block, and line size in bytes. As expected, the miss rate decreases when the 2D block size increases even though there are only few 2D blocks if they are large. The results show that having 8 64×128 blocks exhibits a miss rate of 0.11%, and that having 8 32×256 and 4 64×256 blocks exhibit the same miss rate. The 32×256 block was selected because, as depicted in Figure 6.1, fetching 256 bytes is as efficient as fetching fewer bytes. 2D blocks with 32 rows also allow to use the Extended_XY enhancement methodology.

A similar exploration was performed for the IBM SC. The IBM SC was configured to be 4-way set associative and the line size was varied from 16 to 256 bytes. As for the MDSC, a 64KB cache was used for the Y components

and two 16KB caches for both the Cb and Cr components. It should be noted that unlike the MDSC the IBM SC uses three separate caches, one for each component to avoid conflicts. The IBM SC uses the FIFO replacement policy. The best performing IBM SC configuration uses 256-byte lines and has a miss rate of 8.6% for the Y component. As the IBM SC can use only 1-dimensional blocks while the kernel processes 2-dimensional blocks, the IBM SC miss rate is much higher that the miss rate of the MDSC.

Figure 6.15 breaks down the time taken by the MC kernel when the baseline MDSC (without enhancements) is employed into the time needed to access the MDSC and the time required for the DMA transfers. For comparison purposes, the time taken by a version that does not use a software cache but fetches the reference areas from main memory using explicit, hand-programmed DMA transfers is also included and labeled *DMA*. The results include the time for frame border detection, the time to fetch the additional quarter-pixel area, the time to fetch additional Cb and Cr components, and the handling of 128-bit alignment constraints. The border detection and the alignment calculation are included in the DMA time because they are overlapped with memory transfers and account for less than 1% of the total DMA time. The baseline MDSC implementation performs an MDSC access for every 16-byte quadword. The Figure shows that the majority of the time is spent accessing the cache rather than transferring data. This is because for every MDSC access the index has to be calculated and the tag has to be compared to the tags stored in the SC. This overhead is relatively time consuming compared to the time taken by the DMA transfers. Furthermore, for all but one sequence, the version that uses hand-programmed DMAs is faster than the version that employs the baseline MDSC. The proposed enhancements exploit the SC access pattern to reduce this overhead.

Figure 6.16 compares the performance of the direct DMA version of MC, the IBM SC, the baseline MDSC, and the MDSC extended with the enhancements described in Section 6.5.4. It depicts the time in seconds to fetch the reference area from main memory to the SPE scratchpad. Our baseline for comparison is the DMA version of MC. The line labeled *Real Time* depicts the performance required for real time processing. As in the previous experiment, the DMA version includes border and alignment handling while the other versions depict the time required for MC only.

When the number of MDSC accesses is reduced with the Extended_Line technique, an average 25% improvement over the DMA version is achieved. The Extended_Line technique reduces the number of MDSC accesses by a

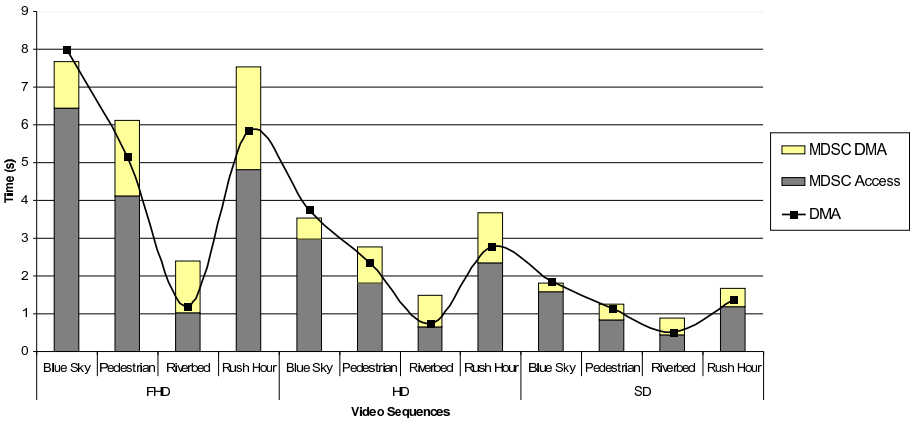


Figure 6.15: Breakdown of the time taken by the MC kernel for different input sequences when the baseline MDSC is employed and the time taken when explicit, hand-programmed DMA transfers are used.

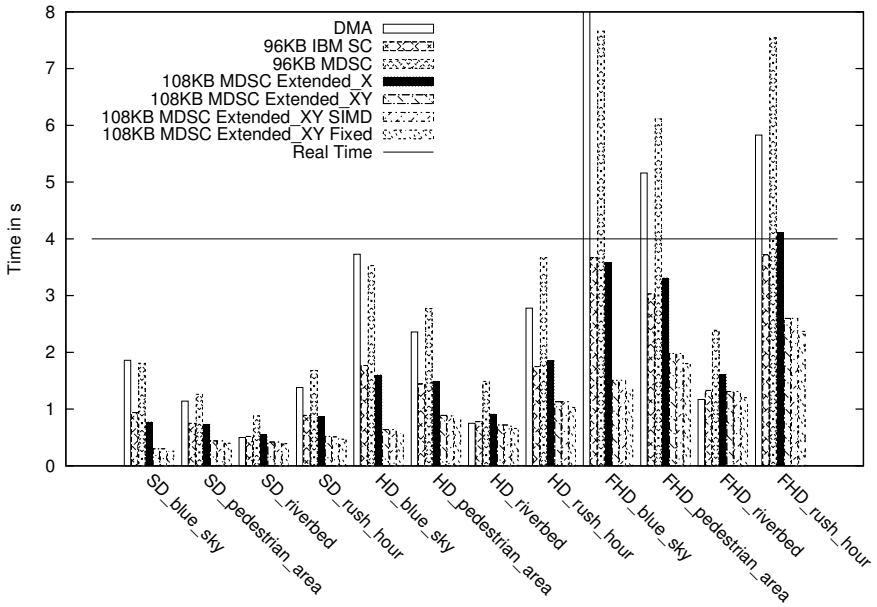


Figure 6.16: Time taken by MC for the direct DMA version, the IBM SC, the MDSC, and the various MDSC enhancements.

factor of 2, because it ensures that when the first pixel in a line is present in the MDSC, the entire line is present. By checking only the presence of the first and last line of the MB partition, as done in the Extended_XY technique, only two MDSC accesses per MB partition are needed. This substantially increases the efficiency of the MDSC implementation and achieves a 60% execution time reduction on average compared to the direct DMA version of the code.

The SIMD version of the MDSC does not provide an additional performance improvement. Its overhead cancels the benefits because of the small number of 2-dimensional blocks (eight) in the outlined MDSC configuration. Fixing the parameters of the MDSC with the static implementation so that certain compiler optimizations can be performed yields an additional 5% execution time reduction, leading to a total average performance improvement of 65%, compared to the direct DMA version of the code .

Compared to the IBM SC, the MDSC with the Extended_XY enhancement is 37% more efficient. The performance improvement increases to 43% when comparing the IBM SC to the static implementation of the MDSC. This improvement is mainly due to the reduction of the number of SC accesses. Because the MDSC uses 2-dimensional blocks and because the Extended_XY technique ensures that the entire reference area is included in at most two MDSC blocks, two MDSC accesses are sufficient to determine if the reference area is in the cache, whereas the IBM SC requires at least one access for every line in the reference area. Additionally, the MDSC exploits the relationship between the Y, Cr and Cr components to reduce the number of accesses. If an area is present in the Y cache, then it is also present in the Cb and Cr caches. Thus, only the address calculation is required to access Cb and Cr data.

Unlike for the GLCM kernel, the different indexing function does not bring improvement to the MC kernel. As for the GLCM kernel, the number of replacements and misses for MC using the IBM SC and the MDSC were compared, with the same cache block configuration. The decrease in miss rate for the MDSC compared with the miss rate of the IBM SC is practically zero.

Overall, the results show that SCs can efficiently exploit the data locality exhibited by MC. To obtain actual performance improvements, however, the number of accesses needs to be minimized. Furthermore, the MDSC allows to reduce the number of accesses more than 1-dimensional cache organizations, such as the IBM SC, thereby yielding higher overall performance.

6.8 Conclusions

In this chapter, a Multidimensional Software Cache has been proposed for systems based on scratchpad memories, such as the Cell processor. The objectives of the MDSC are to exploit the data locality that cannot easily be exploited by hand-programmed DMAs, to reduce the DMA startup overhead by employing DMA lists instead of several sequential DMAs, and to minimize the number of cache accesses by using large, multidimensional blocks. Furthermore, the cache is indexed by the indices of the base element of the block rather than the memory address, which allows to reduce the number of conflict misses. The proposed SC organization has been evaluated for the GLCM and the H.264 MC kernels, which are representative of many other multimedia kernels.

The GLCM uses indirect addressing but, because the difference between adjacent pixels is usually small, it exhibits locality between consecutive accesses to the GLCM matrix. Somewhat surprisingly, the MDSC configuration that yields the highest performance uses 1-dimensional blocks. The performance of two multidimensional configurations, however, was less than 1% lower. This indicates that, in an organization where the two memory channels could be used simultaneously by a single core, the benefits of a 2-dimensional block would be more pronounced. Compared to the heavily optimized IBM SC, the MDSC improves performance by 8%. The indexing function of the MDSC reduces the number of conflicts and accounts for the performance improvement.

For MC, first the amount of data locality that it exhibits has been analyzed. This analysis shows that MC exhibits significant amount of data locality that could be exploited by a (software) cache. Then the data access characteristics of the MC kernel has been evaluated to design an MDSC that exploits it. The proposed MDSC stores frame areas instead of blocks of consecutive memory locations. In other words, it uses 2-dimensional cache blocks instead of 1D blocks. The combination of the MDSC cache indexing mechanism and its multidimensionality has been exploited to reduce the SC overhead. Enhancements that use the mentioned combination have been proposed to reduce the number of accesses to the MDSC and its associated overhead.

For MC, the experimental results show that without tuning the MDSC to the kernel, the performance degrades compared to an implementation that uses explicit hand-programmed DMAs and does not attempt to exploit the data locality. This performance degradation is the result of the access overhead to the MDSC to check for the presence of the required data. The enhancements

proposed in order to reduce the number of accesses to the MDSC achieve an average 65% performance improvement over the hand-programmed DMA implementation. For only one sequence (Riverbed), the MDSC did not attain a performance improvement over the DMA version. The reason for this is the lack of data locality in the Riverbed sequence. The SC overhead can be reduced by using information of the kernel's access characteristics to reduce the number of cache accesses. Compared to the IBM Cell SC, the MDCS provides an improvement of 43% on average over all video sequences.

7

Hardware Support for Software Caches

In the previous chapter, we demonstrated that exploiting known application's cache access patterns significantly reduces software cache overhead. In this chapter, we follow on by proposing a hardware module to reduce software cache access overhead. More specifically, we propose an instruction to implement the cache lookup and the address calculation parts of the Multi-dimensional Software Cache (MDSC) access function.

This chapter is organized as follows. Section 7.1 motivates the use of an instruction to accelerate MDSC accesses. The proposed instruction is described in Section 7.2 and the simulation methodology is explained in Section 7.3. Results are presented and analyzed in Section 7.4 and conclusions are drawn in Section 7.5.

7.1 Introduction

As argued in Section 6.4, while traditional hardware caches are area and power consuming, the disadvantage of software caches (SCs) is that they have high processing overhead. Our aim in this chapter is to close the performance gap between software and hardware caches, while keeping the hardware overhead and power consumption at a minimum.

The previous chapter has shown that the MDSC access time dominates the execution time of the MC kernel. This is also the case for the GLCM kernel, as it will be shown in Section 7.4. Because of its impact on the execution speed, we aim to reduce the MDSC lookup overhead, and propose a dedicated instruction for that.

In this chapter, we investigate a hardware module to accelerate SCs. For this purpose, we propose a cache lookup instruction called *LookUp_SC*. The LookUp_SC instruction was specifically designed to improve the MDSC performance. The instruction is targeted at the MDSC lookup and calculates the tag and the set, searches for the tag in the tag array, and returns a hit or miss flag together with a pointer to the address of the requested data (if available). In case of a miss, it is handled by software functions.

7.2 The LookUp_SC Instruction

In this section, we describe the hardware module that implements the MDSC lookup and its functionality. The proposed hardware module is integrated in the processor pipeline and accessed through the LookUp_SC instruction. This instruction receives the access parameters and returns if the data is currently present or not together with a pointer to where the data is stored in the SPE Local Store (LS). Currently, it only supports a 4-way cache associative organization. Figure 7.1 depicts the flow chart of the MDSC modified to employ the operations performed by the LookUp_SC instruction. The proposed instruction performs the following steps:

1. Indexing. This step calculates the set based on the input indices, shifts and masks;
2. Tag Calculation. In this step the tag is calculated based on the input indices, shifts and masks;
3. Element Address Calculation. This step calculates the offset of the data inside the cache block;
4. Lookup. This step checks if the tag is present, update the address of the cache block, and returns the hit/miss flag, the position of the data, the tag, the set and the tag position.

In software, these steps take about 45 cycles on the SPE. Since the lookup needs to be performed to access each data value, even this relatively

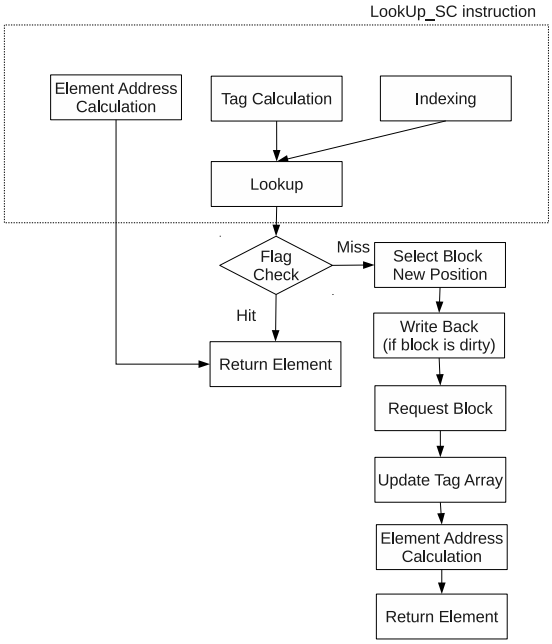


Figure 7.1: LookUp_SC instruction operations and its placement in the read_MDSC function.

small cycle count results in a significant performance penalty. The actual reading and writing of data from/to main memory are excluded from the hardware module to keep the hardware complexity and cost at a minimum level. This is feasible because the MDSC yields high hit rate compared to regular caches.

The proposed hardware accelerator differs from a regular hardware cache in two ways. First, there is no need to perform the cache access in a single or few cycles, since in a cache-based case, the cache access in on the critical path. Therefore, the high performance cache lookup implementation that is present in regular hardware caches can be replaced by a pipelined, more latency tolerant structure integrated in the existing core datapath. Second, there is no hardware handling of misses and no expensive replacement policies. These tradeoffs, while not giving the best cache configuration, aim at keeping the hardware and power consumption overheads at the minimum level, while increasing the performance compared to the complete software implementation.

The instruction has three operands: two input registers and one output register. The first register contains the at most four 32-bit indices of the multi-

dimensional cache. 32-bit integers are chosen as this is the most common type for loop indices, likely to be used to access the cache. The second input register is used to pass the cache parameters. These parameters are the base address of the SC's tag array in the LS, the size of the cache block, the number of sets, shifts and masks for each index, and the size in bytes of each dimension. A 128-bit register is capable of storing this information because of the small address space of the LS (256KB), corresponding to 18 bits addresses (aligned) the small range (usually 8 bits) of the parameters in the second registers, and the wide registers of the SPE. All these parameters only need to be packed together once, as they do not change throughout the execution.

The parameters of the LookUp_SC instruction have to be supplied for each access to allow multiple caches to be used at the same time. Alternatively, a special register could be used to store the cache parameters. This approach, however, complicates both software generation and hardware. It would increase the compiler complexity as it would have to track the accesses to the different caches to make sure each access has the right parameters set in the special register. The hardware would become more complex as two consecutive LookUp_SC instructions in the pipeline would have to access their respective parameters.

Three design choices were considered for the implementation of the tag array: using a dedicated register file, using positions in the existing register file, or using the Local Store. Each of these options is discussed in the following sections.

7.2.1 Dedicated Register File

The first option for the tag array is to use a special register file which we refer to as the Tag Array Register File (TARF). The TARF is a separate register file dedicated to store the MDSC tag array. Because of the additional resources involved, only a small number of positions would be feasible in the dedicated register file, e.g, 4 positions of 128 bits each. For this reason we have not selected this design option. Note that, as in the previous chapter, we assume that the MDSC will be used in addition to hand programmed DMA requests. If almost all data access will be to the SC, either the MDSC or a regular SC, it could justify a large TARF.

7.2.2 General Purpose Register File

It is possible to avoid the hardware overhead of the TARF using the existing register file to store the tag array. For ease of reference, this approach is called Tag Array in General Purpose (TAGP). In this design choice, the instruction accesses two registers with the required inputs of the indexing function. The calculated set serves then as the register number. This works similarly to an indirect load on the register file. The Cell SPU register file has 3 read ports, but only a few instructions use them all. This third read port can be used to access the calculated set, thus no additional register file port would be necessary. Modifications to the control logic, however, would be necessary to allow an instruction deeper in the pipeline to access the register file. Also, it requires a conflict detection mechanism for the case an instruction that accesses the 3 read ports of the register file is being processed in the same cycle as the LookUp_SC is trying to access the register file.

Compared to the TARF, this approach has the advantage of enabling a larger number of sets. This solution, however, requires the inclusion of some features in the control logic and reduces the number of registers available for other computation. Given this extra complexity, this design option was not selected.

7.2.3 Local Store

A third option is to store the tag array in the local store (TALS). This avoids the pressure on the register file and the extra area needed for an extra register file for the tag array. This option, however, has a higher latency than the previous ones as it requires to access to the LS, instead of registers. The larger latency is a good tradeoff as it offers an easier pipeline integration and the minimum hardware overhead to implement the new instruction, as presented next. Because of these advantages we chose to store the tag array in the local store.

Figure 7.2 depicts the pseudo-code for the operations performed by the LookUp_SC instruction, following the TALS design option. W, Z, Y, and X are the 32-bit indices for each of the dimensions. $Tag\_Array\_Address$ is the 16-bit address of the tag array in the LS divided by 16 to save bits. This is possible because accesses to the LS have to be quadword aligned. Each dimension of the MDSC is associated with a *shift*, a *mask*, and a *dimensionsize* parameter. Each of these parameters is 8 bits wide. *shift* is the base-2 logarithm of the size of the corresponding cache block dimension. *mask* is the base-2 logarithm of the dimension range (maximum number of bits of the index) minus the corre-

sponding *shift*. To reduce hardware complexity, the *mask* is then added to the values of the lower indices. *dimensionsize* is the base-2 logarithm size of the cache block dimension added to the lower cache block dimensions sizes, also to reduce complexity. The *address* is actually the index in the memory array that acts as the cache memory. The *n_sets* parameter carries the number of sets in the MDSC. In practice, *n_sets* is passed decremented by one, so it can be directly applied.

Despite the number of operations in the LookUp_SC instruction, it remains relatively simple due to the reduced size of its operands and the parallelism of the operations. The LookUp_SC instruction hardware is composed by the set calculation, the tag formation logic, a comparator, and the address calculation logic. The tag formation and the set calculation logic consists of a series of shifts and masks operations to select the significant bits from the indices. Similarly, the address calculation logic uses shifts, masks, and additions to find the position of the data inside the cache block. No multiplication is necessary, as the cache block dimensions are powers of 2. The *quad_result* is the resulting quadword to be returned. It consists of four 32-bit integers, corresponding to the hit flag, the index of the requested data in the data storage array, the tag, and the position of the tag in the tag array, respectively. *quad_result* is finally updated depending on the position of the *tags* quadword where the *tag* is found. The LS is accessed only once to retrieve the tag array (*tags*) vector pointed by *set*.

Most of the hardware components of the accelerator can execute in parallel. The Indexing function, the tag formation, and address calculation can be performed in parallel. After the indexing function, the logic can also start calculating *quad_result*. The indexing function is followed by the indices load and the tag formation, which is followed by the tag comparison. They need to be performed sequentially.

The SPE pipeline is divided into a front-end and a back-end part [50]. The front end pipeline is the same for all instructions and it performs the instruction fetching, instruction decoding, and accesses the register file. There are five back-end pipelines for branch, permute, load/store, fixed point, and floating point instructions. The load/store back-end pipeline would have to be modified to perform the mentioned computations.

The LookUp_SC instruction can be easily integrated in the existing MDSC code with only few modifications. The resulting code for a two-dimensional MDSC is depicted in Figure 7.3, where x and y are the indices of the structure to be accessed, ea is the address of the accessed data structure in

```
set = ((((W >> shift[w]) ^ (W >> (shift[w] + 1))) +
        ((Z >> shift[z]) ^ (Z >> (shift[z] + 1))) +
        ((Y >> shift[y]) ^ (Y >> (shift[y] + 1))) +
        ((X >> shift[x]) ^ (X >> (shift[x] + 1))))
        & (n_sets-1));

tags = Load((Tag_Array_Address + set) << 4);

tag = 1 + ((((W >> shift[w]) << mask[w]) |
            ((Z >> shift[z]) << mask[z]) |
            ((Y >> shift[y]) << mask[y]) |
             (X >> shift[x])) << 2);

address =
  (W & ~(0xFFFFFFFF << shift[w])) << dimensionsize[w] |
  (Z & ~(0xFFFFFFFF << shift[z])) << dimensionsize[z] |
  (Y & ~(0xFFFFFFFF << shift[y])) << dimensionsize[y] |
  (X & ~(0xFFFFFFFF << shift[x])) << dimensionsize[x];

quad_result = (0, ((block_size*(set<<2)) << 4) + address,
              tag, set<<2);

if (tags[0] == tag)
  return quad_result | (1,               0, 0, 0);
else if (tags[1] == tag)
  return quad_result | (1,   block_size<<4, 0, 1);
else if (tags[2] == tag)
  return quad_result | (1, 2*block_size<<4, 0, 2);
else if (tags[3] == tag)
  return quad_result | (1, 3*block_size<<4, 0, 3);
else
  return quad_result;
```

Figure 7.2: Pseudo C-code of the LookUp_SC instruction.

the external memory, $\_cache\_mem$ is the MDSC data array, and $parameters$ are the parameters of the MDSC instance. The first step is to merge all indices to a single register, as performed by the $vec\_uint4\ index = \{x, y, x, y\};$ statement. Second, the lookup function is replaced by the spu_LookUp_SC intrinsic that forms the interface to our proposed instruction. The result of the LookUp_SC instruction is then separated into regular integers by the

```
CACHED_TYPE read_MDSC( int y, int x, int ea){
  vec_uint4 index =  {x, y ,x ,y};
  quadresult = spu_LookUp_SC(index, parameters);

  int address = spu_extract(quadresult, 1);
  int tag     = spu_extract(quadresult, 2);
  int tag_pos = spu_extract(quadresult, 3);
  CACHED_TYPE ret = __cache_mem[address];

  if (unlikely (spu_extract(tagpos, 0) != 0))
    ret=__cache_mem[cache_miss(y, x, tag, tag_pos, ea)];

  return ret;
}
```

Figure 7.3: Resulting C code for the read_MDSC function integrated with the LookUp_SC instruction.

$spu\_extract$ intrinsic. The $spu\_extract$ shifts a specified word position in the quadword to the first word of the quadword, where it can be processed as a regular integer. The $unlikely$ function sets the branch hint to *not taken*. The last step is to replace the calculation of the data address with the result of the LookUp_SC instruction. The miss handling function does not need to be modified.

7.3 Experimental Methodology

We use two separate and complementary approaches to evaluate our proposed instruction. The first approach uses an actual Cell SPE core and the instruction is emulated using a load instruction. The second approach uses the CellSim Simulator [16]. In the simulator, the new instruction is added to the instruction set of the SPE.

To emulate the LookUp_SC instruction on the Cell processor, we proceed as follows. First, the kernel is profiled and broken down into the following four parts: *Processing*, consisting of the basic computational operations of the kernel without the accesses to the SC; *Access*, consisting of the accesses to the SC without considering miss handling or data transfer, i.e., it consists only of

SC hits; *Miss Handling*, consisting of the selection of the cache block to be replaced, the calculation of the memory address, and the update of the cache status; and *DMA*, corresponding to the actual transfer of data from the external memory to the LS.

The SPE hardware counter is used to profile the kernels. As there is only one hardware counter, an incremental approach is used to break down the kernel. The *Processing* time is extracted by commenting out the MDSC access. In order to isolate the *Access* time, a run of the application is performed with the accesses to the SC requesting the same data, thus with only one cold start miss. This is possible because the evaluated kernel's processing times are insensitive to the data being processed. To isolate the *Miss Handling* time, the actual data is used to access the MDSC and the DMA request commands are commented out. The *DMA* time is extracted by measuring the time it requires to run the full kernel and subtracting the time taken by the previous stages.

After the kernel has been profiled, the LookUp_SC instruction is emulated using an existing instruction. Our proposed instruction is comparable with an indexed load instruction and thus the LQX (the SPE indexed load instruction) is used. The LQX instruction performs a 32-bit addition to calculate the effective LS address. In our proposed instruction, the address calculation depends on the *set* calculation, as depicted in Figure 7.2. Because the *set* is 8 bits wide and most of the operations of the LookUp_SC can be performed in parallel, we assume that using the same latency as the LQX instruction is feasible. The LQX instruction is accessed by an SPU intrinsic. The *Access* time of the target application is then profiled again using the emulated instruction. The accelerated *Access* time is used instead of the *Access* time of the original kernel. This new kernel time is used to estimate the performance gain of the proposed instruction. This approach is valid because the LookUp_SC instruction only changes the MDSC access time with all the remaining stages intact.

We also evaluate the proposed hardware extension using the CellSim Simulator. It is not possible to use the IBM Cell Full-System simulator [17] because the source code is not available to make the necessary modifications. The CellSim Simulator was used to validate the instruction behavior and to qualitatively analyze the required hardware complexity.

CellSim is not cycle accurate as it does not distinguish between the odd and even pipelines of the SPE. To handle this limitation, the instruction fetch rate parameter can be modified. This parameter defines a constant number of instructions that the execution unit fetches per cycle. To tune this parameter,

the profiling results of the kernels are used. In our experiments, the instruction fetch rate is set so that the simulated execution time approximately matches the execution times on the actual core. The simulator reported an average runtime 7% smaller than the measurements using the real SPE cores. Because of this difference we opted to report the results acquired following our first approach, leaving the second approach only for instruction validation purposes.

In order to evaluate the LookUp_SC instruction, the MC and GLCM kernels described in Section 6.5, are used. The sequences from the HD-VideoBench, described in Section 2.4.2, are once again used as input.

Unlike in the previous chapter, however, the evaluation of the MC kernel accesses three separate MDSCs, one for each color component, each one using a static, 4-way set associative implementation. The main motivation for these changes in the MDSC for the MC kernel is that, in this chapter, we focus only on performance improvement that the our proposed instruction yields. Thus, the co-locality between color components is not exploited and the other listed enhancements, described in Section 6.5.4, to reduce the number of MDSC accesses, are not implemented. The choice of a 4-way set associative MDSC implementation is due to the fact that the LookUp_SC instruction currently does not support fully associative implementations of the MDSC. Finally, to ensure an unbiased comparison, we use a static MDSC implementation instead of a dynamic implementation because the IBM SC only supports static implementations.

7.4 Experimental Results

In this section, experimental results for the LookUp_SC instruction accelerating the MDSC are presented. Results for the GLCM and MC kernels are presented and analyzed. Three baseline versions are presented for each kernel. Results of an implementation without a SC are to state the need/impact of a SC for the given kernel. The second baseline is an implementation using a publicly available SC, the IBMSC for the Cell [101]. The third baseline uses the MDSC implementation of the kernel without the proposed instruction. We note here that while developing the LookUp_SC instruction the MDSC code for the static 4-way set associative implementation was optimized. The optimization consisted of rescheduling operations from inside the if-then-else statement that checks for cache misses to before the statement. This rescheduling increases the number of operations between the branch hint instruction and its branch target. As discussed in Section 5.4.1, the hint instruction needs to be placed

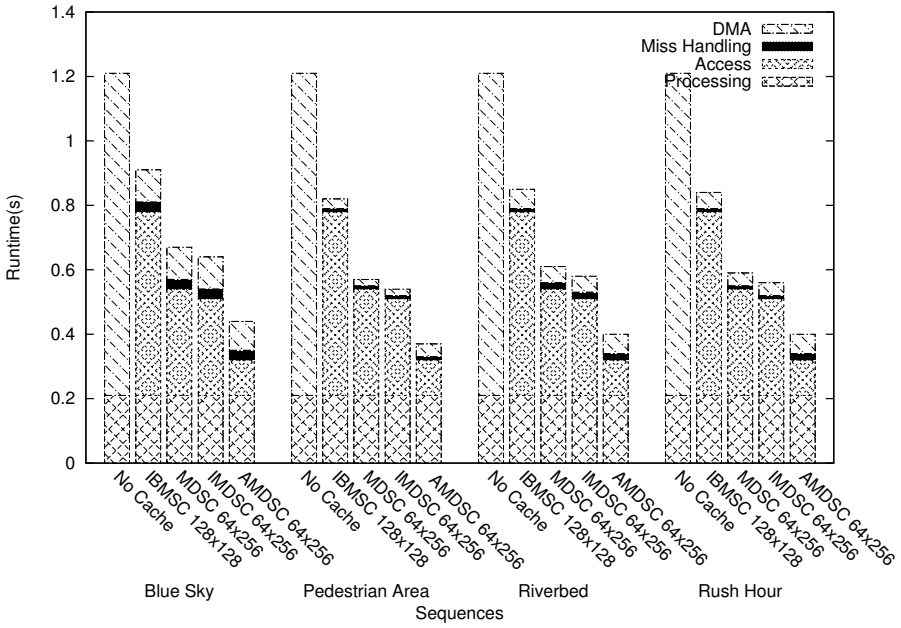


Figure 7.4: GLCM runtime of FHD sequences for No_Cache, IBMSC, MDSC, inlined MDSC, and AMDSC.

18 cycles before the branch to have a penalty free branch. Because of this optimization, the MDSC results of this chapter have a higher performance than the MDSC results presented in the previous chapter.

7.4.1 GLCM Results

In Figure 7.4, the execution time for the Full High Definition (FHD) inputs are depicted. The DMA time to fetch the picture data are not included as they can be overlapped with the computation. The *Processing* time also depicts the execution time for the GLCM if the matrix would fit in the LS. Each SC has a suffix $S \times X$ to state its configuration, where S corresponds to the number of sets and X to the number of columns in the cache block. The number of rows is not depicted as it is always one. The labeled IMDSC in Figure 7.4 refers to the MDSC with the read/write functions inlined, as it will be discussed later. Furthermore, each SC is 4-way set associative. As described in the previous chapter, we determined the optimal configuration for each SC, cache size is fixed at 64KB.

The MDSC and the IBMSC have different configuration and behaviors. Compared with the IBMSC, the MDSC has higher DMA transfer and miss handling times. This is caused by the multidimensional handling of the cache block, even if the block has only one line in this particular case. On the other hand, the MDSC has a smaller access time that compensates for the time spent in the other stages.

The SC access time dominates the execution time for the IBMSC and MDSC with 70% and 54% of the total execution time, respectively. The reason is that the GLCM kernel exhibits a very small amount of computation per access. On average, the MDSC accelerated with our proposed instruction (or AMDSC for ease of reference) speeds up the access time by a factor of 3.0, compared to the MDSC. The reduction in access time translates to a total speedup of 2.1 and 1.6 for the total kernel, when compared to IBMSC and with the regular MDSC, respectively.

In this particular kernel, the improvement is due to two factors. First, the number of instructions is reduced. The LookUp_SC instruction replaces several instructions, what leads to a performance gain. A second factor is that it also reduces the size of the cache access function. With the reduced size, the access function can be inlined in the GLCM calculation function, which further increases the speedup. Compared to the IMDSC access function, the speedups of the LookUp_SC instruction for the access time slightly decrease to 2.8. The speedup of the whole kernel is reduced from 1.6 to 1.5.

7.4.2 MC Results

In this section, the results of MC using different SCs are presented and discussed. Figure 7.5 depicts the results for the FHD sequences from the HD-VideoBench. Each MDSC has a suffix $S \times Y \times X$ to state its configuration, where S corresponds to the number of sets, Y to the number of rows in the 2-dimensional cache block, and X to the number of columns. For the IBMSC only the number of sets and number of columns (line size), in $S \times X$ format, are depicted in the suffix.

The results of the MC kernel show that the MDSC is, on average, 9% faster than the IBMSC. The IBMSC has an average hit rate of 94%, while the MDSC has an average hit rate of 98%. This explains the shorter *DMA* times for the MDSC. On the other hand, the MDSC's more complex *Miss Handling* leads to a higher Miss Handling time than incurred by the IBMSC, even with significant lower number of misses. The MDSC *Access* time is also longer

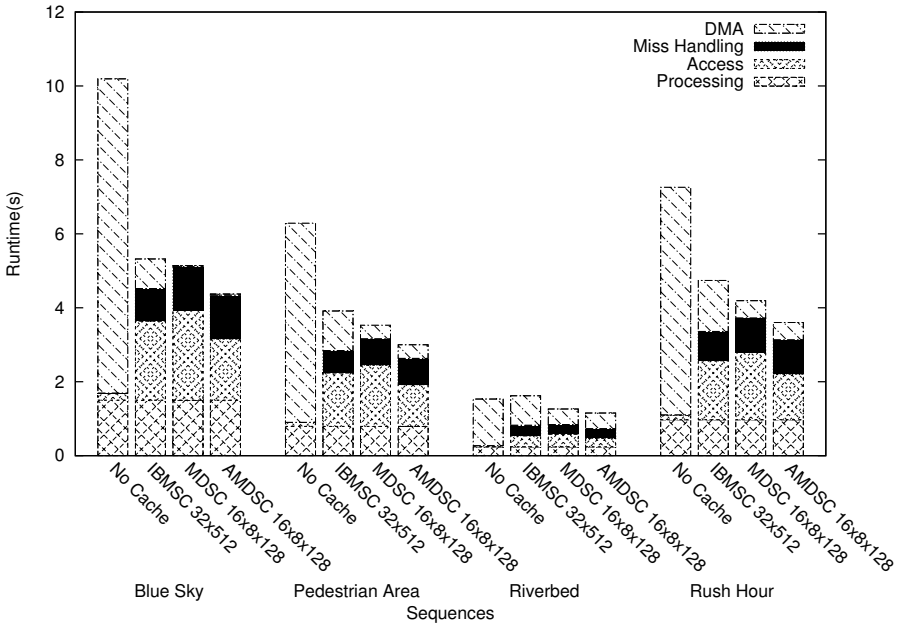


Figure 7.5: MC runtime of FHD sequences for No_Cache, IBMSC, MDSC, and AMDSC.

than the IBMSC *Access* time due to the tag calculation using a larger number of parameters.

For the MC kernel, the LookUp_SC instruction yields an average performance improvement of 47% for the AMDSC access time, compared to the MDSC access time. This relative improvement is lower than in the previous experiment. The reason for this is that the MC processing function is vectorized with SIMD instructions and operates on a temporary array. The cache access copies the reference area to this temporary array. This is performed inside a nested loop, which introduces significant overhead due to the divergent branching that implements the loop. Furthermore, the optimization of the read_MDSC function significantly reduces the number of instructions after the branch instruction of the function, which results in an inefficient branch hint for the branch that implements the loop structure. As described in Section 4.3, even unconditional branches need to be hinted in order to avoid the miss prediction penalty. On the whole MC kernel, the AMDSC performance improvements are 108%, 28%, and 16%, on average, compared to the *No Cache*, IBMSC, and MDSC implementations, respectively.

7.5 Conclusions

In this chapter, we presented a new instruction to accelerate software caches. On cores with scratchpad memories, such as the Cell SPE, software caches are an efficient way to improve the performance of applications that do not have a predictable access behavior to make efficient use of DMAs, but do exhibit locality. On the other hand, software caches incur high overhead.

The LookUp_SC instruction performs several operations and is composed of a number of hardware operations. Because several operations can be performed in parallel, the overall latency of the instruction is estimated to be the same as the latency of an indexed load instruction. As a side effect of the reduction of the number of instructions in the AMDSC access functions, it can be inlined without increasing the code size significantly. This particularly improves the performance of applications with a high ratio of number of memory accesses to computational instructions.

Our experimental results showed that the LookUp_SC instruction significantly accelerates the MDSC access function. The overall performance improvement depends on the application. For the MC kernel, the access function is accelerated by 47%, while a speedup of 3.0 is achieved on average for the GLCM access function, when compared to the MDSC access function. For the whole kernels, the AMDSC speedups are 1.16 for MC and 1.5 for GLCM, when compared with the kernels using the MDSC.

8 Conclusions

This dissertation presented several contributions to the design of multi- and many-core processors targeted mainly at video processing applications. Leveraging many-core processors for video processing is an important research challenge, as the consumer market demands higher video quality and new features. The presented contributions range from novel parallelization techniques to obtain a parallel video decoder that scales to a large number of cores, techniques to improve the scalar performance of SIMD-only cores, to techniques to exploit the data locality in applications that cannot easily be exploited in scratchpad/local store memory architectures.

This chapter is organized as follows. Section 8.1 presents a summary of the dissertation chapters and lists its main contributions. Open issues and future research directions for video processing on multi-core platforms are discussed in Section 8.2.

8.1 Summary and Contributions

This dissertation focused on many-core architectures composed of SIMD-only cores with scratchpad memories instead of caches. Furthermore, it focused on video processing applications. It covers increasing processing and power

efficiency of many-core systems in general, SIMD-only cores, and scratchpad memories. Each of these objectives was covered by two chapters. Below we summarize the dissertation and list its main contributions.

Chapter 2 presented the 3D-Wave parallelization strategy and studied its scalability to many-core processors. After discussing why the previous techniques do not efficiently scale to many-core architectures, the 3D-Wave strategy was introduced. This strategy breaks frame dependencies in a novel way by overlapping the decoding of several inter-dependent frames. A static analysis followed, showing that the technique can harvest sufficient macroblock (MB)-level parallelism to make use of many-core (64+) cores. It revealed that up to 2040 MBs can be processed in parallel for Full High Definition (FHD) sequences. Besides the available amount of MB-level parallelism, the effects of limiting resources and frame scheduling were also presented.

Taking into account the positive results of the 3D-Wave scalability study, Chapter 3 described the implementation of the 3D-Wave strategy on a simulated multi-core processor. The experimental results showed that the 3D-Wave achieves speedup of more than 50 on the 64-core processor for FHD resolution. Several additional improvements and studies have been presented. Frame priority and frame scheduling policies have been introduced to decrease the latency and memory footprint of the parallel H.264 decoder. The effects of memory latency, cache size, and synchronization latency have been studied, as well as the requirements for a CABAC accelerator.

The main contributions in Chapters 2 and 3 can be summarized as follows:

- A new strategy to parallelize H.264 decoding so that it can leverage a many-core processor.
- A static scalability study of the amount of Thread-Level Parallelism in the 3D-Wave.
- An implementation of the 3D-Wave parallelization strategy on a simulated embedded many-core system consisting of up to 64 TM3270 processors that confirms the static scalability study.
- A frame priority policy that gives priority to the oldest frame and reduces the latency to the same level as the 2D-Wave.
- A frame scheduling policy that controls the number of frames in flight.

- An Analysis of the performance requirements of the entropy decoding accelerator not to harm the 3D-Wave scalability.

Chapters 4 and 5 studied the characteristics of SIMD-only cores. Chapter 4 focused on the applicability of SIMD-only cores to the processing of video kernels with divergent branching. For this purpose, the H.264 deblocking filter was vectorized with SIMD instructions and its performance was analyzed. Although well-known SIMD overheads were still present, the overall performance gain was significant, with a speedup of 2.6 over the scalar implementation.

SIMD processing has a well known overhead but the overhead of scalar processing in SIMD-only cores has not yet been studied or documented. Chapter 5 performed this analysis. Two techniques are proposed to evaluate the overhead and to identify its source. The Large-Data-Type technique used 128-bit quadwords to represent an integer value. The PPE-versus-SPE technique compared scalar performance of the SPE with the PPE. The techniques showed a scalar processing overhead ranging form 12% to 57%. New load and store instructions have been proposed to reduce this overhead.

The main contributions of Chapters 4 and 5 were:

- A vectorization of the H.264 Deblocking Filter with SIMD instructions to study the effects of divergent branching on vectorization efficiency.
- A quantification of the overhead caused by the lack of scalar instructions support on SIMD-only architectures such as the Cell SPE.
- The identification of the sources of this overhead.
- New instructions, with minimal area overhead, that loads/stores scalars directly into/from the preferred slot.

The last part of the dissertation dealt with the memory system. More specifically, reduction of memory latency for unpredictable memory accesses, using a software cache featuring a reduced access overhead, was presented. Chapter 6 evaluated the software cache overhead and proposes a new cache organization. The Multidimensional Software Cache (MDSC) has been introduced presenting features to exploit known access behavior to reduce the number of cache lookups. Examples of MDSC enhancements that exploit know access behavior were presented for the Motion Compensation (MC) kernel. It was also demonstrated that the proposed cache design also benefits kernels

where it is not possible to exploit access behavior, such as the GLCM. Improvements ranged from 8% for GLCM to 43% for MC when compared with the IBM software cache.

After identifying the access time as one of the bottlenecks of software caches, an instruction to accelerate the cache accesses was proposed in Chapter 7. The instruction replaced the lookup function and address calculation and was composed of a small number of hardware operations. Because several steps can be performed in parallel,the latency of the instruction was stipulated to the same as the latency of an indexed load instruction. This instruction was evaluated using the MDSC and two multimedia kernels, GLCM and H.264 Motion Compensation. The results showed that the proposed instruction accelerates the software cache access time by 2.6 times. This improvement in cache access translates to a 2.1 speedup for GLCM and 1.28 for MC, when compared with the IBM software cache.

The main contributions of Chapters 6 and 7 can be summarized as follows:

- A Multidimensional Software Cache (MDSC) that caches 1- to 4-dimensional blocks of data that are logically adjacent, thereby reducing the number of cache accesses and the DMA startup overhead.
- Optimizations to the MDSC that reduce the number of cache accesses for the MC kernel.
- An instruction that accelerates the MDSC by implementing the MDSC lookup functionality.

8.2 Open Issues and Future Directions

Because of its the availability, the NXP H.264 decoder was used instead of a publicly available decoder. The heavy optimizations of the FFMPEG H.264 decoder for single-core processors make it very difficult to implement the 3D-Wave technique. However, a freely available implementation of the 3D-Wave is required to enable further research by the research community at large.

The current frame scheduling in the 3D-Wave implementation is done using a static approach. Because it does not reacts to the variability of the decoding time, it can result in a smaller than intended number of frames in flight, therefore, reducing potential MB-level parallelism. To solve this problem, the

development of an automatic frame scheduling technique that only starts to decode a new frame if some cores are idle due to insufficient TLP is desired.

The current 3D-Wave implementation focuses only on the MB decoding part. The requirements for CABAC decoding to support the 3D-Wave have been presented. However, the integration of the CABAC decoding with the MB decoding is desired to comprehensively study the issues of a complete video processing application on heterogeneous many-core processors. The entropy decoder can be parallelized in the slice/frame level and entropy decoder cores could be placed in the same local interconnect as the MB processing cores.

In this dissertation we focus on the H.264 decoder. Another interesting part of video processing is the encoder. The 3D-Wave could be implemented in the encoder to speedup this very compute intensive application. Because in the encoder the dependencies are known a priori, it would be easier to implement the 3D-Wave in the encoder than it was to implement in the decoder. It could also task-level parallelism to isolate the Motion Estimation (ME) kernel from the remaining kernels. That would be possible because ME consumes around 80% of the processing time of the video encoding process. It also would enable an efficient many-core processor consisting of small cores.

The Multiview Video Coding (MVC) [19] amendment to the H.264 standard significantly increases the number of possible frames per second to be processed, as the video now has several views for each frame. The 3D-Wave technique can be applied to this application to exploit the parallelism and provide the required computational resources from a many-core composed of power efficient cores.

With regards to SIMD processing, an option not explored in this thesis is the evaluation of longer SIMD words for multimedia kernels. A 128-bit SIMD word size was used in the Deblocking Filter kernel. It is shown that although the data size is 8 bits, a 16-bit data size has to be used to handle the intermediary results. An evaluation of multimedia kernels using a 256-bit SIMD word with the extended intermediate data size presented in [49] could be performed to asses the possible increase in performance.

The increasing presence of control flow behavior in multimedia kernels decreases performance of SIMD computation as multiple computational paths need to be executed. To reduce this performance loss, a dynamic operator across the SIMD word could be implemented, in a MIMD fashion similar to the Imagine processor [51]. A solution to adapt cost-effectively add this MIMD-like behavior to a regular SIMD datapath is required. One option to be evaluated is the use a mask and an operator selector in the third source register.

With regards to the memory hierarchy, the MDSC groups memory requests to decrease latency. However, the performance of the MDSC reflects the machine organization used for this study. The Cell processor has two memory channels, but a single SPU cannot use both channels simultaneously. A study of the MDSC in a processor with dual channel capabilities is desired as it is expected that this will increase the achieved performance improvement.

One interesting follow up of our instruction to accelerate software cache is to adapt it to conventional software caches, such the as IBM software cache. As regular caches access memory positions, the address calculation would not be accelerated by our proposed instruction. Because of this factor we expect a lower acceleration than the one achieved in this thesis. On the other hand, it would result in a smaller area overhead.

Another future work is the evaluation of the power consumption of the proposed instruction. A power consumption comparison between a hardware cache and our accelerated MDSC with scratchpad memory would give interesting quantitative results. This evaluation could be used for processor designers as another evaluation point for future power efficient processors.

Bibliography

[1] Power Architecture Version 2.02. http://www-106.ibm.com/developerworks/eserver/library/es-archguide-v2.html.

[2] International Standard of Joint Video Specification (ITU-T Rec. H. 264— ISO/IEC 14496-10 AVC), 2005.

[3] M. Alvarez, A. Ramirez, A. Azevedo, C. H. Meenderinck, B. H. H. Juurlink, and M. Valero. Scalability of Macroblock-level Parallelism for H.264 Decoding. In *Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS)*, December 2009.

[4] M. Alvarez, A. Ramirez, M. Valero, C. H. Meenderinck, A. Azevedo, and B. H. H. Juurlink. Performance Evaluation of Macroblock-level Parallelization of H.264 Decoding on a CC-NUMA Multiprocessor Architecture. In *Proceedings of the the Colombian Computing Conference*, April 2009.

[5] M. Alvarez, E. Salami, A. Ramirez, and M. Valero. A Performance Characterization of High Definition Digital Video Decoding using H.264/AVC. In *Proceedings of the IEEE International Workload Characterization Symposium*, pages 24–33, 2005.

[6] M. Alvarez, E. Salami, A. Ramirez, and M. Valero. A Performance Characterization of High Definition Digital Video Decoding Using H.264/AVC. In *Proceedings of the the IEEE International Workload Characterization Symposium*, pages 24–33, 2005.

[7] M. Alvarez, E. Salami, A. Ramirez, and M. Valero. HD-VideoBench: A Benchmark for Evaluating High Definition Digital Video Applications. In *IEEE International Symposium on Workload Characterization*, 2007.

[8] M. Alvarez, E. Salami, A. Ramirez, and M. Valero. Performance Impact of Unaligned Memory Operations in SIMD Extensions for Video

Codec Applications. In *IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 62–71, 2007.

[9] A. Azevedo, B. Zatt, L. Agostini, and S. Bampi. Motion Compensation Decoder Architecture for H.264/AVC Main Profile Targeting HDTV. In *Proceedings of the IFIP International Conference on Very Large Scale Integration*, pages 52 –57, 2006.

[10] A. Azevedo, B. Zatt, L. Agostini, and S. Bampi. MoCHA: a Bi-Predictive Motion Compensation Hardware for H.264/AVC Decoder Targeting HDTV. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1617–1620, May 2007.

[11] J. Balart, M. Gonzalez, X. Martorell, E. Ayguade, Z. Sura, T. Chen, T. Zhang, K. Obrien, and K. Obrien. A Novel Asynchronous Software Cache Implementation for the Cell-BE Processor. In *Proceedings of the International Workshop Languages and Compilers for Parallel Computing*, pages 125–140. Springer-Verlag New York Inc, 2007.

[12] Rajeshwari Banakar, Stefan Steinke, Bo sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In *Proceedings of the International Symposium on Hardware/Software Codesign (CODES)*, pages 73–78. ACM, 2002.

[13] Kevin J. Barker, Kei Davis, Adolfy Hoisie, Darren J. Kerbyson, Mike Lang, Scott Pakin, and Jose C. Sancho. Entering the Petaflop Era: the Architecture and Performance of Roadrunner. In *Proceedings of the ACM/IEEE Conference on Supercomputing (ICS)*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[14] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.

[15] M. Briejer, C. H. Meenderinck, and B. H. H. Juurlink. Extending the cell spe with energy efficient branch prediction. In *Proceedings of the Euro-Par Conference*, September 2010.

[16] F. Cabarcas, A. Rico, D. Rodenas, X. Martorell, A. Ramirez, and E. Ayguade. CellSim: A Cell Processor Simulation Infrastructure. In *HiPEAC ACACES*, pages 279–282, 2007.

[17] IBM Full-System Simulator for the Cell Broadband Engine Processor. http://www.alphaworks.ibm.com/tech/cellsystemsim.

[18] Tong Chen, Tao Zhang, Zehra Sura, and Mar Gonzales Tallada. Prefetching Irregular References for Software Cache on Cell. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 155–164, New York, NY, USA, 2008. ACM.

[19] Ying Chen, Ye-Kui Wang, Kemal Ugur, Miska M. Hannuksela, Jani Lainema, and Moncef Gabbouj. The Emerging MVC Standard for 3D Video Services. *EURASIP Journal on Applied Signal Processing*, 2009:1–13, 2009.

[20] Y.K. Chen, E.Q. Li, X. Zhou, and S. Ge. Implementation of H. 264 Encoder and Decoder on Personal Computers. *Journal of Visual Communications and Image Representation*, 17, 2006.

[21] Y.K. Chen, X. Tian, S. Ge, and M. Girkar. Towards Efficient Multi-Level Threading of H.264 Encoder on Intel Hyper-Threading Architectures. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2004.

[22] C.C. Chi, B. H. H. Juurlink, and C. H. Meenderinck. Evaluation of Parallel H.264 Decoding Strategies for the Cell Broadband Engine. In *Proceedings International Conference on Supercomputing (ICS)*, June 2010.

[23] Josh Coalson. FLAC - Free Lossless Audio Codec. http://flac.sourceforge.net/.

[24] R. Cucchiara, M. Piccardi, and A. Prati. Exploiting Cache in Multimedia. In *IEEE International Conference on Multimedia Computing and Systems*, volume 1, pages 345 –350, July 1999.

[25] L. Dagum and R. Menon. OpenMP: An Industry Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, 5(1):46 –55, 1998.

[26] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[27] L.F. Ding, W.Y. Chen, P.K. Tsung, H.K. Chiu, Y.H. Chen, P.H. Hsiao, S.Y. Chien, T.C. Chen, P.C. Lin, C.Y. Chang, et al. A 212MPixels/s 4096× 2160p Multiview Video Encoder Chip for 3d/Quad HDTV Applications. In *Proceedings of the IEEE International Solid-State Circuits Conference-Digest of Technical Papers (ISSCC)*, pages 154–155. IEEE, 2009.

[28] Dolby TrueHD Audio Coding for Future Entertainment Formats. http://www.dolby.com/uploadedFiles/Assets/US/Doc/Professional/TrueHD_Tech_Paper_Final.pdf.

[29] M. Drose, C. Clemens, and T. Sikora. Extending Single-View Scalable Video Coding to Multi-View Based on H. 264/AVC. In *Proceedings of the IEEE International Conference on Image Processing*, pages 2977–2980, 2006.

[30] AMD Phenom II Processors . http://www.amd.com/us/products/desktop/processors/phenom-ii/Pages/phenom-ii.aspx.

[31] DTS-HD Audio: Consumer White Paper for Blu-ray Disc and HD DVD Applications. http://www.dts.com/DownloadDocument.aspx?q=a7beda1e-cfe6-4ca4-b6b2-cda9554bb6a5.

[32] Jan Edler and Mark D. Hill. Dinero IV Trace-Driven Uniprocessor Cache Simulator . http://pages.cs.wisc.edu/~markhill/DineroIV/.

[33] A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo. Using Advanced Compiler Technology to Exploit the Performance of The Cell Broadband Engine Architecture. *IBM Systems Journal*, 45(1):59–84, 2006.

[34] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. Vectorization for SIMD Architectures with Alignment Constraints. *SIGPLAN Notices*, 39(6):82–93, 2004.

[35] Ahmed El-Mahdy and Ian Watson. A Two Dimensional Vector Architecture for Multimedia. In Rizos Sakellariou, John Gurd, Len Freeman,

and John Keane, editors, *Proceedings of the Euro-Par Parallel Processing*, volume 2150 of *Lecture Notes in Computer Science*, pages 687–696. Springer Berlin / Heidelberg, 2001.

[36] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the Memory Hierarchy. In *Proceedings of the ACM/IEEE Conference on Supercomputing (ICS)*, SC '06, New York, NY, USA, 2006. ACM.

[37] The FFmpeg Libavcoded. http://ffmpeg.mplayerhq.hu/.

[38] M. Flierl and B. Girod. Generalized B Pictures and the Draft H. 264/AVC Video-Compression Standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):587–597, 2003.

[39] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

[40] M. Fujita, M. Kondo, and H. Nakamura. Data Movement Optimization for Software-Controlled On-Chip Memory. In *Proceedings of the Workshop on Interaction Between Compilers and Computer Architectures (INTERACT)*, pages 120–127, Feb. 2004.

[41] Buğra Gedik, Rajesh R. Bordawekar, and Philip S. Yu. CellSort: High Performance Sorting on the Cell Processor. In *Proceedings of the International Conference on Very Large Data Bases*, pages 1286–1297. VLDB Endowment, 2007.

[42] Marc Gonzàlez, Nikola Vujic, Xavier Martorell, Eduard Ayguadé, Alexandre E. Eichenberger, Tong Chen, Zehra Sura, Tao Zhang, Kevin O'Brien, and Kathryn O'Brien. Hybrid Access-Specific Software Cache Techniques for the Cell BE Architecture. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 292–302, New York, NY, USA, 2008. ACM.

[43] M. Gschwind, H.P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic Processing in Cell's Multicore Architecture. *IEEE Micro*, 26(2):10–24, 2006.

[44] Amit Gulati and George Campbell. Efficient Mapping of the H.264 Encoding Algorithm onto Multiprocessor DSPs. *Embedded Processors for Multimedia and Communications II*, 5683(1):94–103, March 2005.

[45] HP Hofstee. Power Efficient Processor Architecture and the Cell Processor. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 258–262, Washington, DC, USA, 2005. IEEE Computer Society.

[46] Jan Hoogerbrugge and Andrei Terechko. A Multithreaded Multicore System for Embedded Media Processing. *Transactions on High-Performance Embedded Architectures and Compilers*, 4(2), 2009.

[47] M. Horowitz, A. Joch, and F. Kossentini. H.264/AVC Baseline Profile Decoder Complexity Analyis. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):704–716, 2003.

[48] Ibm bladecenter qs22. http://www-03.ibm.com/systems/bladecenter/hardware/servers/qs22/index.html.

[49] B. H. H. Juurlink, A. Shahbahrami, and S. Vassiliadis. Avoiding Data Conversions in Embedded Media Processors. In *Proceedings of the ACM Symposium on Applied Computing*, pages 901–902, March 2005.

[50] JA Kahle, MN Day, HP Hofstee, CR Johns, TR Maeurer, and D. Shippy. Introduction to the Cell Multiprocessor. *IBM Journal of Research and Development*, 49(4):589–604, 2005.

[51] B. Khailany, W.J. Dally, U.J. Kapasi, P. Mattson, J. Namkoong, J.D. Owens, B. Towles, A. Chang, and S. Rixner. Imagine: Media Processing with Streams. *Micro, IEEE*, 21(2):35–46, 2002.

[52] R. Kumar, K.I. Farkas, N.P. Jouppi, P. Ranganathan, and D.M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 81 – 92, 2003.

[53] R. Kumar, D.M. Tullsen, N.P. Jouppi, and P. Ranganathan. Heterogeneous Chip Multiprocessors. *IEEE Computer*, 38(11):32 – 38, 2005.

[54] N.A. Kurd, S. Bhamidipati, C. Mozak, J.L. Miller, T.M. Wilson, M. Nemani, and M. Chowdhury. Westmere: A Family of 32nm IA Processors. In *Proceedings of the IEEE International Solid-State Circuits*

Conference-Digest of Technical Papers (ISSCC), pages 96–97. IEEE, 2010.

[55] V. Lappalainen, A. Hallapuro, and T. D. Hamalainen. Complexity of Optimized H.26L Video Decoder Implementation. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):717–725, 2003.

[56] J. Lee, S. Moon, and W. Sung. H.264 Decoder Optimization Exploiting SIMD Instructions. In *Proceedings of the Asia-Pacific Conference on Circuits and Systems*, Dec. 2004.

[57] J. Lee, S. Moon, and W. Sung. H.264 Decoder Optimization Exploiting SIMD Instructions. In *Proceedings of the IEEE Asia-Pacific Conference on Circuits and Systems*, volume 2, 2004.

[58] Jaejin Lee, Sangmin Seo, Chihun Kim, Junghyun Kim, Posung Chun, Zehra Sura, Jungwon Kim, and SangYong Han. COMIC: A Coherent Shared Memory Interface for Cell BE. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 303–314, New York, NY, USA, 2008. ACM.

[59] P. List, A. Joch, J. Lainema, G. Bjntegaard, and M. Karczewicz. Adaptive Deblocking Filter. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):614–619, 2003.

[60] P. List, A. Joch, J. Lainema, G. Bjntegaard, and M. Karczewicz. Adaptive Deblocking Filter. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):614–619, 2003.

[61] Y. Liu and S. Oraintara. Complexity Comparison of Fast Block-Matching Motion Estimation Algorithms. In *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 2004.

[62] H. Ma and J. Wolf. On Tail Biting Convolutional Codes. *IEEE Transactions on Communications*, 34(2):104–111, 1986.

[63] HS Malvar, A. Hallapuro, M. Karczewicz, and L. Kerofsky. Low-Complexity Transform and Quantization in H. 264/AVC. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):598–603, 2003.

[64] D. Marpe, H. Schwarz, and T. Wiegand. Context-Based Adaptive Binary Arithmetic Coding in the H.264/AVC Video Compression Standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):620–636, 2003.

[65] M.D. McCool and B. D'Amora. Programming using RapidMind on the Cell BE. In *Proceedings of the ACM/IEEE Conference on Supercomputing (ICS)*, page 222. ACM, 2006.

[66] C. H. Meenderinck, A. Azevedo, B. H. H. Juurlink, M. Alvarez, and A. Ramirez. Parallel Scalability of Video Decoders. *Journal of Signal Processing Systems*, August 2008.

[67] C. H. Meenderinck and B. H. H. Juurlink. Specialization of the Cell SPE for Media Applications. In *Proceedings of the EEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, July 2009.

[68] Philipp Merkle, Karsten Müller, and Thomas Wiegand. 3D Video Coding: An Overview of Present and Upcoming Standards. *Visual Communications and Image Processing*, 7744(1):77440D, 2010.

[69] Ross Miller. Toshiba Regza GL1 3D Preview: No Frills, No Glasses, Some Issues. http://www.engadget.com/2010/10/05/toshiba-regza-gl1-3d-preview-no-frills-no-glasses-some-issues/.

[70] Nokia Europe - Nokia N8 - New Touch Screen Phone with Free Navigation and HD Video. http://europe.nokia.com/find-products/devices/nokia-n8.

[71] Dorit Nuzman, Ira Rosen, and Ayal Zaks. Auto-Vectorization of Interleaved Data for SIMD. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 132–143, New York, NY, USA, 2006. ACM.

[72] T. Oelbaum, V. Baroncini, T.K. Tan, and C. Fenimore. Subjective Quality Assessment of the Emerging AVC/H.264 Video Coding Standard. In *Proceedings of the International Broadcast Conference (IBC)*, 2004.

[73] F. Okano, M. Kanazawa, K. Mitani, K. Hamasaki, M. Sugawara, M. Seino, A. Mochimaru, and K. Doi. Ultrahigh-Definition Television System With 4000 Scanning Lines. In *Proceedings NAB Broadcast Engineering Conference*, pages 437–440, 2004.

[74] The OpenMP API specification for parallel programming. http://openmp.org/.

[75] R. R. Osorio and J. D. Bruguera. An FPGA Architecture for CABAC Decoding in Manycore Systems. In *Proceedings of the IEEE Application-Specific Systems, Architectures and Processors*, pages 293–298, July 2008.

[76] J. Ostermann, J. Bormans, P. List, D. Marpe, M. Narroschke, F. Pereira, T. Stockhammer, and T. Wedi. Video Coding with H.264/AVC: Tools, Performance, and Complexity. *IEEE Circuits and Systems Magazine*, 4(1):7–28, 2004.

[77] J. Planas, R.M. Badia, E. Ayguadé, and J. Labarta. Hierarchical task-based programming with StarSs. *International Journal of High Performance Computing Applications*, 23(3):284, 2009.

[78] A. Ramirez, F. Cabarcas, B. H. H. Juurlink, M. Alvarez Mesa, F. Sanchez, A. Azevedo, C. Meenderinck, C. Ciobanu, S. Isaza, and G. Gaydadjiev. The SARC Architecture. *IEEE Micro*, 30:16–29, 2010.

[79] Parthasarathy Ranganathan, Sarita Adve, and Norman Jouppi. Performance of Image and Video Processing with General-Purpose Processors and Media ISA Extensions. *SIGARCH Computer Architecture News*, 27(2):124–135, 1999.

[80] Gang Ren, Peng Wu, and David Padua. Optimizing Data Permutations for SIMD Devices. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 118–131, New York, NY, USA, 2006. ACM.

[81] A. Rodriguez, A. Gonzalez, and M. P. Malumbres. Hierarchical Parallelization of an H.264/AVC Video Encoder. In *Proceedings International Symposium on Parallel Computing in Electrical Engineering*, pages 363–368, 2006.

[82] M. Roitzsch. Slice-Balancing H.264 Video Encoding for Improved Scalability of Multicore Decoding. In *Proceedings of the IEEE Real-Time Systems Symposium*, 2006.

[83] SARC Project. http://www.sarc-ip.org.

[84] Ramesh Sarukkai. What's Bigger than 1080p? 4K Video Comes to YouTube. http://youtube-global.blogspot.com/2010/07/whats-bigger-than-1080p-4k-video-comes.html.

[85] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: A Many-core X86 Architecture for Visual Computing. *ACM Transactions on Graphics*, 27(3):1–15, 2008.

[86] Ganapathy Senthil, Sasikanth Gudla, and Pallav Kumar Baruah. Exploring Software Cache on the Cell BE Processor. In *Proceedings of the International Conference on High Performance Computing (HiPC)*, page 5, 2008.

[87] Sangmin Seo, Jaejin Lee, and Z. Sura. Design and Implementation of Software-Managed Caches for Multicores With Local Memory. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 55–66, Feb. 2009.

[88] A. Shahbahrami and B. H. H. Juurlink. Optimization of Content-Based Image Retrieval Functions. In *Proceedings of the IEEE International Symposium on Multimedia*, pages 607–612, December 2008.

[89] Asadollah Shahbahrami, B. H. H. Juurlink, Demid Borodin, and Stamatis Vassiliadis. Avoiding Conversion and Rearrangement Overhead in SIMD Architectures. *International Journal of Parallel Programming*, 34(3):237–260, 2006.

[90] H. Shojania, S. Sudharsanan, and Chan Wai-Yip. Performance Improvement of the H.264/AVC Deblocking Filter Using SIMD Instructions. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2006.

[91] P. Stenström. Chip-multiprocessing and Beyond. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 109–109, 2006.

[92] G.J. Sullivan, P.N. Topiwala, and A. Luthra. The H.264/AVC Advanced Video Coding Standard: Overview and Introduction to the Fidelity Range Extensions. In *Proceedings SPIE Conference on Applications of Digital Image Processing*, pages 454–474, 2004.

[93] A. Tamhankar and KR Rao. An Overview of H.264/MPEG-4 Part 10. In *Proceedings of the EURASIP Conference on Video/Image Processing and Multimedia Communications*, page 1, 2003.

[94] J. van de Waerdt, S. Vassiliadis, S. Das, S. Mirolo, C. Yen, B. Zhong, C. Basto, J. van Itegem, D. Amirtharaj, K. Kalra, et al. The TM3270 Media-Processor. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 331–342, 2005.

[95] E.B. van der Tol, E.G. Jaspers, and R.H. Gelderblom. Mapping of H.264 Decoding on a Multiprocessor Architecture. In *Proceedings of the SPIE Conference on Image and Video Communications and Processing*, 2003.

[96] A. J. Viterbi. A Personal History of the Viterbi Algorithm. *IEEE Signal Processing Magazine*, 23(4):120–142, 2006.

[97] S. Warrington, H. Shojania, and S. Sudharsanan. Performance Improvement of the H.264/AVC Deblocking Filter Using SIMD Instructions. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, page 4, 2006.

[98] T. Wiegand, H. Schwarz, A. Joch, F. Kossentini, and G.J. Sullivan. Rate-Constrained Coder Control and Comparison of Video Coding Standards. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):688 – 703, 2003.

[99] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A.Luthra. Overview of the H.264/AVC Video Coding Standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):560–576, July 2003.

[100] M. Wien. Variable Block-Size Transforms for H. 264/AVC. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):604–613, 2003.

[101] Example Library API Reference. https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/3B6ED257EE6235D900257353006E0F6A/$file/SDK_Example_Library_API_v3.0.pdf.

[102] X264. A Free H.264/AVC Encoder. http://developers.videolan.org/x264.html.

[103] B. Zatt, A. Azevedo, L. Agostini, A. Susin, and S. Bampi. Memory Hierarchy Targeting Bi-Predictive Motion Compensation for H.264/AVC Decoder. In *Proceedings of the IEEE Symposium on VLSI*, pages 445–446. IEEE Computer Society Washington, DC, USA, 2007.

[104] X. Zhou, E. Q. Li, and Y.-K. Chen. Implementation of H.264 Decoder on General-Purpose Processors with Media Instructions. In *Proceedings SPIE Conference on Image and Video Communications and Processing*, 2003.

List of Publications

International Journals

1. A. Ramirez, F. Cabarcas, B.H.H Juurlink, M. Alvarez, **A. Azevedo**, C. Meenderinck, G. Gaydajiev, C. Ciobanu, S. Isaza, F. Sanchez, *The SARC Architecture*, IEEE Micro, Volume 30, Issue 5, 2010

2. **A. Azevedo**, B.H.H. Juurlink, *A Multidimensional Software Cache for Scratchpad-Based Systems*, International Journal of Embedded and Real-Time Communication Systems (IJERTCS), Volume 1, Issue 4, 2010

3. **A. Azevedo**, B.H.H. Juurlink, C.H. Meenderinck, A. Terechko, J. Hoogerbrugge, M. Alvarez, A. Ramirez, M. Valero, *A Highly Scalable Parallel Implementation of H.264*, Transactions on High-Performance Embedded Architectures and Compilers (HiPEAC), Volume 4, Issue 2, 2009.

4. C.H. Meenderinck, **A. Azevedo**, B.H.H. Juurlink, M. Alvarez, A. Ramirez, *Parallel Scalability of Video Decoders*, Journal of Signal Processing Systems, pp. 173-194, Volume 57, Issue 2, 2009.

5. M. Alvarez, A. Ramirez, M. Valero, **A. Azevedo**, C.H. Meenderinck, B.H.H. Juurlink, *Performance Evaluation of Macroblock-level Parallelization of H.264 Decoding on a cc-NUMA Multiprocessor Architecture*, Avances en Sistemas e Informatica, Volume 6, Number 1, 2009, ISSN 1657-7663.

International Conferences

6. **A. Azevedo**, B.H.H. Juurlink, *An Instruction to Accelerate Software Caches*, Proceedings of the Conference on Architecture of Computing Systems (ARCS), 2011

7. **A. Azevedo**, B.H.H. Juurlink, *An Efficient Software Cache for H.264 Motion Compensation*, Proceedings of IEEE International Symposium on System-on-Chip (SOC), 2009.

8. **A. Azevedo**, B.H.H. Juurlink, *Scalar Processing Overhead on SIMD-Only Architectures*, Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP), 2009.

9. **A. Azevedo**, C.H. Meenderinck, B.H.H. Juurlink, A. Terechko, J. Hoogerbrugge, M. Alvarez, A. Ramirez, M. Valero, *Parallel H.264 Decoding on an Embedded Multicore Processor*, Proceedings of High-Performance Embedded Architectures and Compilers Conference (HiPEAC), 2009.

10. M. Alvarez, A. Ramirez, M. Valero, C.H. Meenderinck, **A. Azevedo**, B.H.H. Juurlink, *Performance Evaluation of Macroblock-level Parallelization of H.264 Decoding on a cc-NUMA Multiprocessor Architecture*, Proceedings of the 4th Colombian Computing Conference (4CCC), 2009.

11. M. Alvarez, A. Ramirez, **A. Azevedo**, C.H. Meenderinck, B.H.H. Juurlink, M. Valero, *Scalability of Macroblock-level Parallelism for H.264 Decoding*, Proceedings of International Conference on Parallel and Distributed Systems (ICPADS), 2009.

12. **A. Azevedo**, C.H. Meenderinck, B.H.H. Juurlink, M. Alvarez, A. Ramirez, *Analysis of Video Filtering on the Cell Processor*, Proceedings of International Symposium on Circuits and Systems (ISCAS), 2008.

13. C.H. Meenderinck, **A. Azevedo**, M. Alvarez, B.H.H. Juurlink, A. Ramirez, *Parallel Scalability of H.264*, Proceedings of Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG), 2008.

14. M. Alvarez, **A. Azevedo**, C.H. Meenderinck, B.H.H. Juurlink, A. Terechko, J. Hoogerbrugge, A. Ramirez, *Analyzing scalability limits of H.264 decoding due to TLP overhead*, HiPEAC Industrial Workshop, 2008.

15. Z. Popovic, R. Giorgi, N. Puzovic, B.H.H. Juurlink, **A. Azevedo**, *Analyzing Scalability of Deblocking Filter of H.264 via TLP exploitation*

in a new Many-Core Architecture, Proceedings of 11th EUROMICRO Conference on Digital System Design (DSD), 2008.

Local Conferences

16. R. Giorgi, Z. Popovic, N. Puzovic, **A. Azevedo**, B.H.H. Juurlink, *Exploiting Parallelism of Deblocking Filter of H.264 on DTA Architecture*, ACACES Poster Abstracts, 2008.

17. **A. Azevedo**, C.H. Meenderinck, B.H.H. Juurlink, M. Alvarez, A. Ramirez, *Analysis of Video Filtering on the Cell Processor*, Proceedings Workshop on Circuits, Systems and Signal Processing (ProRISC), 2007.

Reports

17. C.H. Meenderinck, **A. Azevedo**, B.H.H. Juurlink, M. Alvarez, A. Ramirez, *Parallel Scalability of Video Decoders*, CE technical report, 2008, CE-TR-2008-03.

Samenvatting

In dit proefschrift presenteren wij methodes en evaluaties met het doel om de efficiëntie van videocoderingsapplicaties te verbeteren voor heterogene veelkernige processoren die bestaan uit enkel-SIMD kernen met scratchpad geheugen. Onze bijdrage is drieledig: thread-level parallellisme voor veelkernige processoren, detectie van het knelpunt voor enkel-SIMD kernen, en een software cache voor kernen met scratchpad geheugen.

Eerst presenteren wij de 3D-Golf parallellisatiestrategie voor video decodering, welke schaalt naar veelkernige processoren. Deze strategie is gebaseerd op de observatie dat de afhankelijk tussen frames zich bevindt in de motion compensation kernel en dat de motion vectoren meestal een korte spanne hebben. De 3D-Golf strategie combineert macroblok-niveau parallellisme met frame- en slice-niveau parallellisme door het decoderen van frames te overlappen, terwijl de afhankelijkheden tussen macroblokken dynamisch gemanaged worden. De 3D-Golf was geïmplementeerd en geëvalueerd op een gesimuleerde meerkernige embedded processor, bestaande uit 64 kernen. Wij presenteren methodes om het geheugengebruik en wachttijd te reduceren. De effecten van geheugenwachttijd, cache-grootte, en synchronisatiewachttijd worden bestudeerd.

Het beoordelen van de geschiktheid van enkel-SIMD kernen voor de steeds complexer wordende multimedia applicaties is onze tweede bijdrage. We evalueren de geschiktheid van enkel-SIMD kernen met betrekking tot de divergerende branches in video bewerkingsalgoritmes. Het anti-blok filter van H.264 wordt gebruikt als test. Ook wordt het meerwerk, veroorzaakt door de afwezigheid van een scalar-bewerkingsunit in enkel-SIMD kernen, gemeten met twee methodes. Om scalar-ondersteuning toe te voegen aan enkel-SIMD kernen worden oplossing voorgedragen welke weinig oppervlak innemen.

Ten slotte richten we ons op de geheugen hirarchie en stellen wij een nieuwe software cache organisatie voor, welke de efficiëntie en doeltreffendheid van scratchpad geheugen verbeterd voor niet-voorspelbare en indirecte geheugen raadplegingen. De voorgestelde Multi-Dimensionale Software Cache reduceert het software cache meerwerk doordat het de programmeur in staat stelt om kennis van het raadpleeggedrag te benutten en zo het aantal raadplegingen van de software cache te reduceren, en door het groeperen van raadplegingen. Een instructie om een MDSC raadpleging te versnellen wordt gepresenteerd en geanalyseerd.

Curriculum Vitae

Arnaldo Azevedo was born on 19 of November in Natal, Rio Grande do Norte, Brazil. He received the BSc degree in computer science from the UFRN University, Natal, Brazil, in 2004 and the MSc degree in computer science from UFRGS University, Porto Alegre, Brazil, in 2006. In 2006 he started his doctoral studies at the Computer Engineering Laboratory of the Faculty of Electrical Engineering, Mathematics and Computer Science of Delft University of Technology, the Netherlands. His main research topics are multimedia processing and architectural design of multi-core processors.