Improving the Scalability of Multicore Systems

With a Focus on H.264 Video Decoding

Improving the Scalability of Multicore Systems

With a Focus on H.264 Video Decoding

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof.ir. K.Ch.A.M Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen

op vrijdag 9 juli 2010 om 15:00 uur

door

Cornelis Hermanus MEENDERINCK

elektrotechnisch ingenieur
geboren te Amsterdam, Nederland.

Dit proefschrift is goedgekeurd door de promotoren:
Prof. dr. B.H.H. Juurlink
Prof. dr. K.G.W. Goossens

Samenstelling promotiecommissie:

Rector Magnificus, voorzitter	Technische Universiteit Delft
Prof. dr. B.H.H. Juurlink, promotor	Technische Universität Berlin
Prof. dr. K.G.W. Goossens, promotor	Technische Universiteit Delft
Prof. dr. H. Corporaal	Technische Universiteit Eindhoven
Prof. dr. ir. A.J.C. van Gemund	Technische Universiteit Delft
Dr. H.P. Hofstee	IBM Systems and Technology Group
Prof. dr. K.G. Langendoen	Technische Universiteit Delft
Dr. A. Ramirez	Universitat Politecnica de Catalunya

Prof. dr. B.H.H. Juurlink was werkzaam aan de Technische Universiteit van Delft tot eind december 2009, en heeft als begeleider in belangrijke mate bijgedragen aan de totstandkoming van dit proefschrift.

ISBN: 978-90-72298-08-9

Keywords: multicore, Chip MultiProcessor (CMP), scalability, power efficiency, thread level-parallelism (TLP), H.264, video decoding, instruction set architecture, domain specific accelerator, task management

Acknowledgements: This work was supported by the European Commission in the context of the SARC integrated project #27648 (FP6).

Printed in the Netherlands

Dedicated to my wife
for loving and supporting me.

Improving the Scalability of Multicore Systems

With a Focus on H.264 Video Decoding

Abstract

In pursuit of ever increasing performance, more and more processor architectures have become multicore processors. As clock frequency was no longer increasing rapidly and ILP techniques showed diminishing results, increasing the number of cores per chip was the natural choice. The transistor budget is still increasing and thus it is expected that within ten years chips can contain hundreds of high performance cores. Scaling the number of cores, however, does not necessarily translate into an equal scaling of performance. In this thesis, we propose several techniques to improve the performance scalability of multicore systems. With those techniques we address several key challenges of the multicore area.

First, we investigate the effect of the power wall on future multicore architecture. Our model includes predictions of technology improvements, analysis of symmetric and asymmetric multicores, as well as the influence of Amdahl's Law.

Second, we investigate the parallelization of the H.264 video decoding application, thereby addressing application scalability. Existing parallelization strategies are discussed and a novel strategy is proposed. Analysis shows that using the new parallelization strategy the amount of available parallelism is in the order of thousands. Several implementations of the strategy are discussed, which show the difficulty and the possibility of actually exploiting the available parallelism.

Third, we propose an Application Specific Instruction Set (ASIP) processor for H.264 decoding, based on the Cell SPE. ASIPs are energy efficient and allow performance scaling in systems that are limited by the power budget.

Finally, we propose hardware support for task management, of which the benefits are two-fold. First, it supports the SARC programming model, which is a task-based dataflow programming model based on StarSS. By providing hardware support for the most time-consuming part of the runtime system, it improves the scalability. Second, it reduces the parallelization overhead, such as synchronization, by providing fast hardware primitives.

Acknowledgments

Many people contributed to this work in some way and I owe them a debt of gratitude. To reach this point it takes professors who teach you, colleagues who assist you, and family and friends who support you.

First of all, I thank Ben Juurlink for supervising me. You helped focusing this work, while at the same time pointing me to numerous potential ideas. You showed me what others had done in our research area and how we might either build on that or fill a gap. You suggested me to visit BSC and collaborate with them closely. Also when I wanted to visit HiPEAC meetings you supported me. Al those trips, where I met other researchers, other ideas, and other scientific work, have been of great value.

I'm also thankful to Stamatis Vassiliadis for giving me the opportunity to work on the SARC project. I profited from the synergy effect within such a large project. The shift to the SARC project also meant the end of the collaboration with Sorin Cotofana. Sorin, I thank you for the time we worked together. You were the one who introduced me into academia, you taught me how to write papers, and I still remember the conversations we had about life, religion, gastronomy, and jazz. I also thank Kees Goossens for being one of my promoters.

Over the years I have worked together with several other PhD students and some MSc students. I thank Arnaldo and Mauricio for all the work we did together on parallelizing H.264 and porting it to the Cell processor. I thank Alejandro, Felipe, Augusto, and David for all the effort they put into the simulator and helping me to use and extend it. I thank Efren for implementing the Nexus system in the simulator. I thank Carsten, Martijn, and Chi for doing their MSc project with us, by which you helped me and others. I thank Pepijn for helping me with Linux issues and being a pleasant room mate.

I thank Alex Ramirez and Xavi Martorell for warmly welcoming me in Barcelona, the collaboration, and your insight. My visits to Barcelona carried a lot of fruit. It helped me to put my research in a larger perspective; it

both broadened and deepened my knowledge, and taught me how to enjoy life in an intense way.

I thank Jan Hoogerbrugge for the valuable discussions on parallelizing H.264. I thank Stefanos Kaxiras for his input on the methodology used to estimate the power consumption of future multicores. I thank Peter Hofstee and Brian Flachs for commenting on the SPE specialization. I thank Ayal Zaks for helping me with compiler issues. I thank Yoav Etsion for the discussions and collaboration on hardware dependency resolution.

Secretaries and coffee; they always seem to go together. Whether I was in Delft or in Barcelona, the secretary's office was always my source of coffee and a chat. Lidwina and Monique, thank you for the coffee, the chats, and helping me with the administrative stuff. Lourdes, thank you for the warm welcome in BSC, the coffee, and the conversations.

I thank the European Network of Excellence on High Performance and Embedded Architecture and Compilation (HiPEAC) for financing the collaboration with BSC and some trips to the HiPEAC workshops and conferences.

I'm very grateful to my parents who raised me, who loved me, and who supported me continuously throughout the years. I thank God for giving me the capabilities of doing a PhD and being my guidance through life.

Many, many thanks to my dearest wife; the princess of my life. Thank you for loving me, for supporting me, and for taking this wonderful journey of life together with me.

C.H. Meenderinck Delft, The Netherlands, 2010

Contents

List of Figures

List of Tables

Abbreviations

ABT	Adaptive Block size Transform
ALU	Arithmetic Logic Unit
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction set Processors
AVC	Advanced Video Coding
B	Bi-directional predicted frame, slice, or macroblock
BP	Baseline Profile
CABAC	Context Adaptive Binary Arithmetic Coding
CAVLC	Context Adaptive Variable Length Coding
ccNUMA	cache coherent Non-Uniform Memory Access
CIF	Common Intermediate Formate (352×288 pixels)
CMP	Chip MultiProcessor
DCT	Discrete Cosine Transform
DF	Deblocking Filter
DLP	Data Level Parallelism
DMA	Direct Memory Access
DWT	Discrete Wavelet Transform
EC	Entropy Coding
ED	Entropy Decoding
EP	Extended Profile
esa	exhaustive search motion estimation algorithm
FHD	Full High Definition (1920×1088 pixels)
FIR	Finite Impulse Response
FMO	Flexible Macroblock Ordering
FU	Functional Unit
GOP	Group of Pictures
GPP	General Purpose Processor
GPU	Graphics Processing Unit

HD	High Definition (1280×720 pixels)
HD-DVD	High Definition Digital Video Disc
hex	hexagonal motion estimation algorithm
HiP	High Profile
I	Intra predicted frame, slice, or macroblock
IDCT	Inverse Discrete Cosine Transform
IDWT	Inverse Discrete Wavelet Transform
ILP	Instruction Level Parallelism
IPC	Instructions Per Cycles
Ipol	InterPolation
IQ	Inverse Quantization
ISA	Instruction Set Architecture
ITRS	International Technology Roadmap for Semiconductors
ITU	International Telecommunication Union
LS	Local Store
MatMul	Matrix Multiply
MB	MacroBlock
MC	Motion Compensation
ME	Motion Estimation
MFC	Memory Flow Controller
MP	Main Profile
MPEG	Moving Picture Experts Group
MT	Matrix Transposition
MV	Motion Vector
OpenMP	Open MultiProcessing
P	Predicted frame, slice, or macroblock
POSIX	Portable Operating System Interface (for uniX)
PPE	Power Processing Element
PPU	Power Processing Unit
PSNR	Peak Signal-to-Noise Ratio
QCIF	Quarter Common Intermediate Format (176×144 pixels)
RISC	Reduced Instruction Set Computer
SAT	Saturate
SD	Standard Definition (720×576 pixels)
SDK	Software Development Kit
SIMD	Single Instruction Multiple Data
SMT	Simultaneous MultiThreading
SoC	System on Chip
SPE	Synergistic Processing Element

SPMD	Single Program Multiple Data
SPU	Synergistic Processing Unit
TC	Task Controller
TLP	Thread Level Parallelism
TPU	Task Pool Unit
UVLC	Universal Variable Length Coding
VCEG	Video Coding Experts Group
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLC	Variable Length Coding
VLIW	Very Large Instruction Word

Chapter 1

Introduction

Computer architecture has reached a very interesting era where scientific research is of great importance to tackle the problems currently faced. Up to the beginning of this millennium processor performance improved mainly by techniques to exploit Instruction-Level Parallelism (ILP) and increased clock frequency. The latter caused a memory gap, as memory latency decreased at a much slower pace. This problem was relieved by using the ever growing transistor budget to build larger and larger caches.

Currently, however, the landscape has completely changed. Multicore is the new way to obtain performance improvement. From high-end servers to embedded systems, multicores have found their way to the market. For servers, chips with 6 or 8 cores are currently available but this number is increasing rapidly. At the recently held Hot Chips 2009, IBM announced its Power7 containing 8 cores with 4 threads each [54]. Moreover, 32 of those chips can be connected directly to form a single server. Sun Microsystems described the Rainbow Falls chip; 16 cores with 8 threads each [55]. AMD started shipping their 6-core Opteron this year [13] and Intel presented an 8-core Nehalem version [25].

Desktop computing is mostly based on quad cores currently. Intel sells the Core i7 [11] and AMD its Phenom X4 [1]. Higher core counts are found in special purpose architectures. The most well-known of these is the 9-core Cell processor developed by IBM, Toshiba, and Sony for the gaming industry [127]. It was groundbreaking as it is heterogeneous and uses scratchpad memory instead of coherent shared memory. The ClearSpeed CSX700 has 192 cores and is a low-power accelerator for high performance technical computing [47]. The Cisco CRS-1, also called Metro or SPP, has 188 RISC cores [14]. This

massively many-core custom network processor is used in Cisco's highest end routers. The picoChip products for wireless communication go up to 273 cores currently [128].

That brings us to the embedded market where we find, among others, the Freescale 8641D (dual core) [61], the ARM Cortex-A9 MPCore (supporting 2 to 4 cores) [3], Pluralitys Hypercore processor (capable of supporting 16 to 256 cores) [130], and Tileras Tile-Gx processor (up to 100 cores) [156]. Also among SoCs (System on Chips) multicore platforms are found, such as the Cavium Octeon (communication) [36], the Freescale QorIQ (communication) [62], and the Texas Instruments OMAP (multimedia) [155].

The list is far from complete (we have not even mentioned Graphical Processing Units (GPUs) yet), but the picture is clear: multicores are everywhere, and "they are here to stay". We have not yet explained why the shift towards multicores was made, neither why they bring along new challenges for computer architects. Those issues are addressed in the next section as we present the motivation for this work.

1.1 Motivation

The increase of processor performance is so steady that Moore's Law is applicable to it, although it was initially posed to express the doubling of the number of transistors every two years [116]. Initially the growth in performance and transistor count were directly related. The number of transistors per chip increased because their size shrunk. At the same time, smaller transistors enabled higher clock frequencies which translated into higher performance. The additional transistors also allowed techniques that exploit ILP, such as deep pipelining, branch prediction, multiple issue, out-of-order execution, etc. These two factors have for long been the main source of performance growth, but not anymore, although the transistor budget is still following Moore's Law.

The exploitation of ILP seems to have reached its limits. Some improvements are still made, but the costs are enormously high in terms of area and power consumption. In aggressive ILP cores many transistors are deployed to keep the functional units busy. The transistor budget allows this additional hardware, but the power consumption is a problem.

The power consumption of processors grew steadily over the years. Figure 1.1 depicts the power density trend of Intel processors. The chart is more than 10 years old, but it shows the trend up to that time as well as the problem.

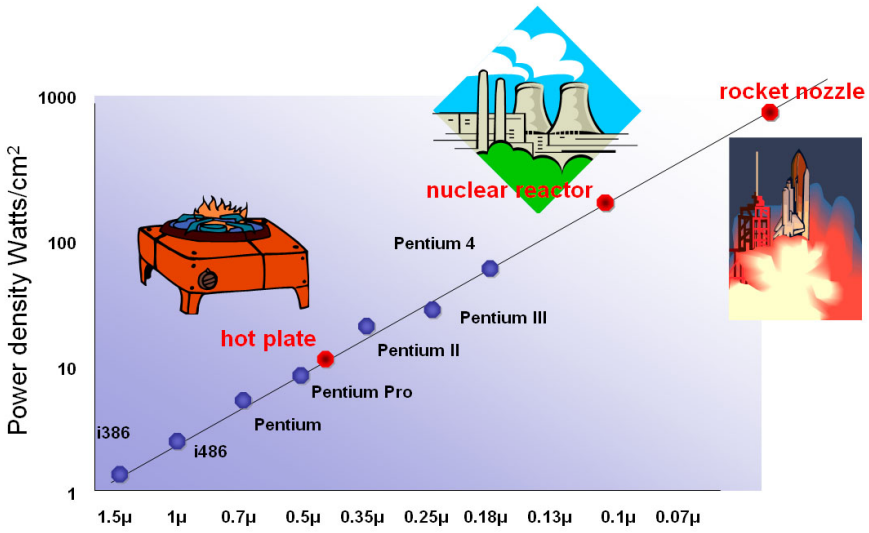


Figure 1.1: Power density trend. Source: Fred Pollack, Intel. Keynote speech Micro32, 1999.

The power consumed by the chip produces heat that should be removed to prevent the chip from burning. Packaging and cooling, however, can handle a limited amount of heat and thus the power consumption can maximally be around $200W$, assuming conventional cooling technology. That limit has been reached and thus power has become one of the dominant design constraints. The dynamic power consumption is given by $P_{dyn} = \alpha C f V^2$, where α is the transistor activity factor, C is the gate capacitance, f is the clock frequency, and V is the power supply voltage. Limits on V and C, and diminishing returns on α forces to turn to frequency to keep the power consumption within the budget. That is the main reason why frequency is hardly increasing anymore.

Without ILP improvements and clock frequency increases as sources for performance improvements, turning towards the exploitation of Thread-Level Parallelism (TLP) was the natural choice. Multithreading already found its way into processors to increase the utilization of the functional units. Putting multiple cores on a chip allows to exploit TLP to a higher degree. And the good news is: the nominal performance of a multicore is proportional to the number of cores, which in turn is proportional to the transistor budget. Therefore, due to the multicore paradigm, performance can potential keep up with Moore's Law.

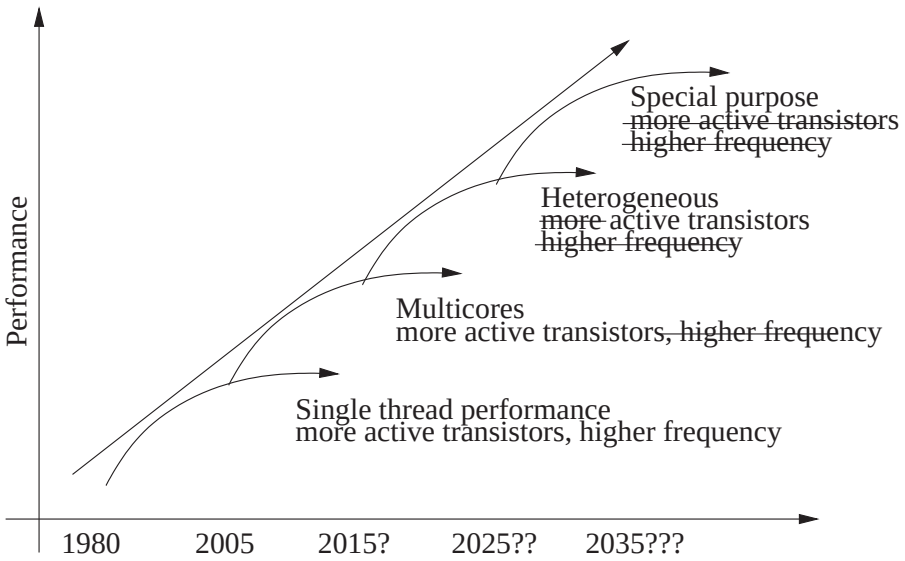


Figure 1.2: Peter Hofstee's (IBM Austin) view on how performance growth will be obtained over the years.

We write 'potentially' because performance does not automatically scale with the number of cores. Scalability is the goal, preferably into the many-core domain where hundreds or thousands of cores in a single chip efficiently and cooperatively are at work. Computer architects and even computer engineers in general are facing many challenges in the pursuit of this goal.

The first challenge is the power wall. A naive multicore architecture would have replicated cores as many as would fit on the die and use a clock frequency such that the power budget is just met. Such an approach, however, is not optimal in terms of performance. Moreover, for many-core architectures it is expected not to provide performance improvements at all.

Figure 1.2 shows Peter Hofstee's view on how performance growth will be obtained over the years. We are currently in the phase where we have more transistors but not higher frequency. As explained before, in such a situation using multiple cores instead of a single one provides performance improvements. In a next phase, there will be more transistors, but not all of them can be active concurrently (unless the frequency is reduced). In this phase a heterogeneous architecture is required to obtain performance improvements. Cores, specialized for application domains, are activated depending on the workload. Cores that are not used are shut down such that the power budget is divided

among the active cores. In the final phase, it might not be possible to put more transistors on a die. The only way to improve performance in such a situation, is to provide special purpose hardware.

Indeed, Application Specific Integrated Circuits (ASICs) show us that power efficiency can be achieved by being application specific. But in those situations where programmability, flexibility, and/or wide applicability are needed, a programmable processor is required. Thus we formulate the first challenge as follows:

Challenge 1 *To get as much as possible performance out of the power budget without significantly compromising wide applicability, programmability, and flexibility.*

Providing TLP, by putting many cores on a chip, does not automatically improve performance improvement. There are many challenges in exploiting TLP, the two most important are mentioned below.

On the algorithm or software level, we have to find TLP. Most algorithms and applications were developed for serial execution. For the near future, TLP has to be discovered within those existing algorithms and applications. For the far future, preferably new algorithms are developed, exhibiting more parallelism. Automated parallelization has been studied for a while now, but not much progress has been reported. Thus, we will have to learn how to extract parallelism there where it is not obvious.

Challenge 2 *To find parallelism in algorithms and applications, especially in applications where it is not obviously available.*

Probably the most debated multicore related topic is programmability. Multicores add another dimension to programming, making it very difficult for the average programmer to obtain good performance within reasonable time. He or she has to consider partitioning, scheduling, synchronization, maintaining dependencies, etc. For the Cell processor also Direct Memory Accesses (DMAs) have to be programmed, SPU (Synergistic Processing Unit) specific intrinsics have to be used, SIMDimization (Single Instruction Multiple Data) should be applied, and branches should be avoided.

Many of those issues require a thorough understanding of the architecture. Moreover, careful analysis of the execution is necessary to understand the behavior of the program. Only then scalability bottlenecks can be removed in order to actually benefit from the parallelization.

Therefore much research is put in this area. New programming models are proposed to ease the job of the programmer. In some cases runtime systems are added to handle scheduling and dependency resolution dynamically. Furthermore, tools are developed to assist the programmer in creating and understanding parallel programs.

Challenge 3 *To increase programmability, enabling a significant number of programmers to create efficient parallel code within reasonable time.*

A challenge that is older than multicore, but so far only briefly mentioned in this introduction, is the memory wall. The memory latency, measured in clock cycles, is growing. A processor that has to wait for data wastes resources and has lower performance. Two factors mainly cause this increased gap. First, the off-chip bandwidth grows slower than the performance of processors. Second, as feature sizes shrink the latency to reach the other side of the chip or to go off-chip, is increasing.

A cache hierarchy can be used to mitigate the effect of the power wall, but in the multicore era this poses another issue. To obtain low latency, typically the L1 cache is private to a core. Thus coherency has to be maintained among shared data, either in hardware or software. The overhead of maintaining coherency increases rapidly with the number of cores, posing a scalability problem [34]. This is exacerbated by the fact that interconnection speeds are not scaling well with technology.

Another direction is to use scratchpad memory with DMA controllers as was done in the Cell processor. It allows complete control of the data and thus performance can be optimized. Programmability, however, suffers a lot and most likely not every application domain is suitable for this type of data management. This brings about the fourth challenge:

Challenge 4 *To continuously provide each core with sufficient data to operate on.*

Having identified parallelism in the application, being able to program it, and running it on a parallel architecture does not necessarily mean that scalability has been achieved. Similar to exploiting ILP and DLP, the exploitation of TLP brings along overhead, reducing the scalability. Thus, the fifth challenge is the following:

Challenge 5 *To minimize the impact of TLP exploitation overhead on scalability and performance.*

These challenges are among the main challenges recognized by the community, although there are many more that are also important. There are challenges in the area of reliability, caused by entering the nano-scale area. Also serial performance remains an issue as shown by Hill and Marty [74]. They based their results on Amdahl's Law and its corresponding assumptions, which might turn out to be too pessimistic, though. More challenges could be mentioned here, but we will focus on how the work presented in this thesis contributes to these five important challenges.

1.2 Objectives

The research presented in this thesis was conducted in the context of the EU FP6 project SARC (Scalable computer ARChitecture). The goal of this project was to develop long-term sustainable approaches to advanced computer architecture in Europe. It focused on multicore technology for a broad range of systems, ranging from small energy critical embedded systems right up to large scale networked data servers. The project brought together research in architecture, interconnect, design space exploration, low power design, compiler construction, programming models, and runtime systems.

Our role in the project was to improve or obtain scalability in the many-core area based on architectural enhancements. Similar to Peter Hofstee's view, the SARC project assumes that heterogeneous architectures are a necessity in the many-core domain. The power budget prevents all cores being active concurrently. Thus, the transistor budget is best spent on providing cores that are specialized for an application domain. Figure 1.3 shows an example of such an architecture, as considered in the SARC project. It contains general purpose processors (P) and domain specific accelerators (A), connected in a clustered fashion. Within a cluster, the memory hierarchy is also specialized for the application domain it targets.

The clusters show similarity with the Cell processor. At the start of the project, the Cell processor was the only heterogeneous multicore processor available and thus it was chosen as the baseline architecture. Moreover, its DMA-style of accessing memory has proven to enable data optimizations for locality and thus enhance scalability.

The overall objective of this thesis, i.e., to improve scalability in the many-core area, is very general. In this section we define more concrete and specific objectives, based on the challenges mentioned in the previous section, and the context of the project. Several application domains are targeted within the

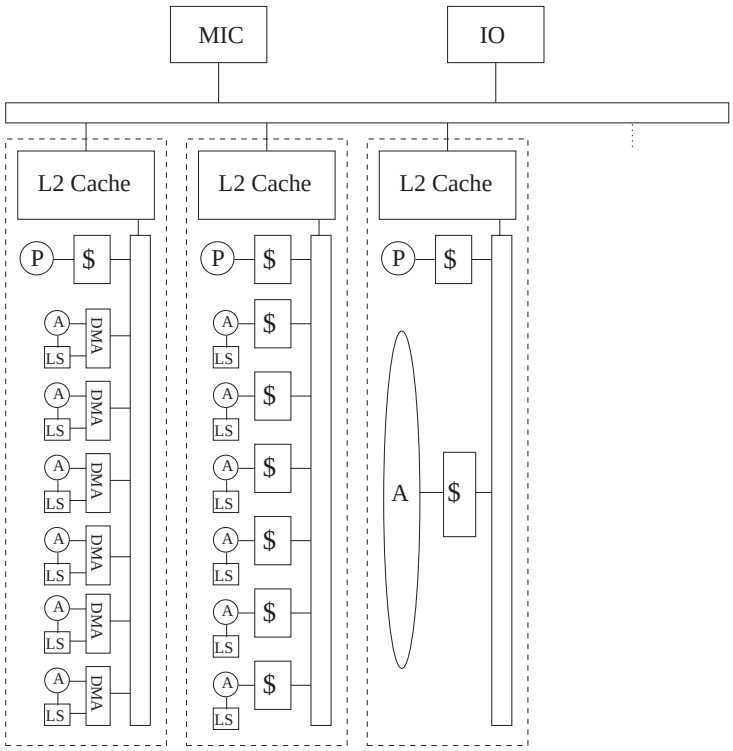


Figure 1.3: Example of the multicore architecture considered within the SARC project, in which context this research was performed.

project, of which this work focuses on the media domain. As main benchmark application, the H.264 video decoder was chosen. Media applications remain an important workload in the future and video codecs are considered to be important benchmarks for all kind of systems, ranging from low-power mobile devices to high performance systems. H.264 is the most advanced video coded currently. It is highly complex and hard to parallelize.

Objective 1 *Analyze the H.264 decoding application. Find new ways to parallelize the code in a scalable fashion. Analyze the parallelization strategies and determine bottlenecks in current parallel systems. This objective corresponds to Challenge 2.*

Objective 2 *Specialize the cores of the Cell processor for H.264 in order to obtain performance and power benefits. This objective corresponds to Challenge 1.*

Objective 3 *Design hardware mechanism to reduce the overhead of managing the parallel execution. This objective corresponds to Challenge 5.*

Objective 4 *Improve programmability by providing hardware support. This objective corresponds to Challenge 3.*

Challenge 4 is only implicitly addressed in this thesis by using scratchpad memory in combination with DMAs, and by applying double buffering.

1.3 Organization and Contributions

This work focuses on a single goal: improving/obtaining scalability into the many-core area. As scalability depends on several issues, as indicated by the challenges and objectives, this work, however, is not focused on a single issue, but rather encompasses a few. Each issue is described in a separate chapter, which are mostly self containing. Introduction to the specific issues are provided within the chapters as well as discussion of related work. An exception is Chapter 5 which extends the work described in Chapter 4.

In Chapter 2 the power wall problem is investigated in more detail. That is, Challenge 1 is investigated in more detail. Specifically, the impact on multicore architecture is investigated. In this chapter we developed a model that includes predictions of technology improvements, analysis of symmetric and asymmetric multicores, as well as the influence of Amdahl's Law.

Objective 1 is addressed in Chapter 3. The H.264 standard is reviewed, existing parallelization strategies are discussed, and a new strategy is proposed. Analysis of real video sequences reveals that the amount of parallelism available is in the order of thousands. Several parallel implementations, mainly developed by colleagues of the author, are discussed. They show the difficulty and the possibility of actually exploiting the available parallelism.

The thesis continues in Chapter 4 with the design of the SARC Media Accelerator. Based on a thorough analysis of the execution of the H.264 kernels on the Cell processor, architectural enhancements to the SPE core are proposed. Specifically, a handful of application specific instructions are added to improve performance. Kernel speedups between 1.84 and 2.37 are obtained.

One of the application specific instructions proposed in the Chapter 4 performs intra-vector operations. In Chapter 5 we investigate the applicability of such instructions to other applications. Our analysis shows that mainly two-

dimensional signal processing kernels benefit from this type of instructions. Speedups up to 2.06, with an average of 1.45 are measured.

Finally, in Chapter 6 we propose a hardware support system for task management. We analyze the scalability of manually coded task management as well as that of StarSS [129, 30]. StarSS includes a programming model, compiler, and runtime system. Although easy to program, the scalability of StarSS is limited. The proposed hardware support alleviates this problem by performing task dependency resolution in hardware.

Chapter 7 concludes this thesis. The main contributions are summarized and open issues are discussed.

Chapter 2

(When) Will Multicores hit the Power Wall?

It is commonly accepted that we have reached the *power wall*, meaning that uniprocessor performance improvements has stagnated due to power constraints. The main causes for the increased power consumption are higher clock frequencies and power inefficient techniques to exploit more Instruction-Level Parallelism (ILP), such as wide-issue superscalar execution. Hitting the power wall is also one of the reasons why industry has shifted towards multicores. Because multicores exploit explicit Thread-Level Parallelism (TLP), their cores can be simpler and do not need additional hardware to extract ILP. In other words, multicores allow exploiting parallelism in a more power-efficient way.

Figure 2.1 illustrates the power wall. Uniprocessors have basically reached the power wall. As argued above, multicores can postpone hitting the power wall but they are also expected to hit the power wall. Several questions arise such as when will multicores hit the power wall, what limitation will cause this to happen, what can computer architects do to avoid or alleviate the problem, etc. It is generally believed that power efficiency of multicores can be improved by designing asymmetric or heterogeneous multicores [74]. For example, several domain specific accelerators could be employed which are turned on and shut down according to the actual workload. But, is the power saving it provides worth the area cost? In this chapter we provide an answer to those questions.

Specifically, in this chapter we focus on technology improvements as they have been one of the main reasons for performance growth in the past. According to the ITRS roadmap [84, 85], technology improvements are expected to con-

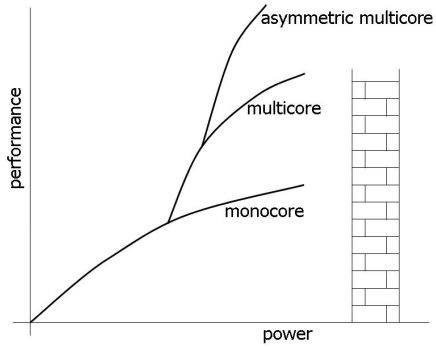


Figure 2.1: The power wall problem (taken from [114]).

tinue. The 2009 roadmap predicts a clock frequency of 14 GHz for the year 2022 and an astonishing number of available transistors. Because of power constraints, however, it might not be possible to exploit those technology improvements, e.g., it might not be possible to turn on all transistors concurrently. In this chapter we analyze the limits of performance growth due to technology improvements with respect to power constraints. We assume perfectly parallelizable applications but also performance growth for non-perfect parallelization is analyzed using both Amdahl's [24] and Gustafson's Law [70]. Based on the results of our experiments, we conclude that multicores can offer significant performance improvements in the next decade provided a number of principles are followed.

Of course our model is necessarily rudimentary. For example, it does not consider bandwidth constraints nor static power dissipation due to leakage. Nevertheless, the power wall has been predicted, multicores are expected to be a remedy, asymmetric multicores have been envisioned, but to the best of our knowledge this has never been quantified. From our results we hope to derive some principles which can be the base for future work as well as the rest of this thesis.

Complementary to this work is the extension of Amdahl's Law for energy efficiency in the many-core era by Woo and Lee [167]. For three systems (symmetric superscalar, symmetric simple, asymmetric combination) they provide equations to compute power efficiency. Their results show that energy efficiency can be improved when asymmetric multicores are used. Their methodology is based on architectural parameters, is very general, and provides rela-

tive numbers. Our work is based on technology parameters, is more specific, and provides absolute numbers. Most importantly, in our analysis the multicore is actually power constrained while in the former the power budget is not considered.

This chapter is organized as follows. The methodology used throughout this chapter is described in Section 2.1 while in Section 2.2 we present the modeled multicore architecture by scaling an existing core. The power and performance analysis of the modeled multicore is presented in Section 2.3. In Section 2.4 Amdahl's and Gustafson's Law are reviewed, which are used in Section 2.5 to analyze the effect of non-perfect application parallelization on the performance growth. Finally, Section 2.6 concludes this chapter.

2.1 Methodology

To analyze the effects of technology improvements on the performance of future multicores, and to investigate the power consumption trend, the following experiment was performed. We assume an Alpha 21264 chip (processing core and caches), scale it to future technology nodes according to the ITRS roadmap, create a hypothetical multicore consisting of the scaled cores, and derive the power numbers. Specifically, we calculate the power consumption of a multicore when all cores are active and run at the maximum possible frequency. Furthermore, we analyze the performance growth over time if the power consumption is restricted to the power budget allowed by packaging.

The Alpha 21264 [89] core was chosen as subject of this experiment for two reasons. First, the Alpha 21264 has been well documented in literature, providing the required data for the experiment. Second, it is a moderately sized core lacking the aggressive ILP techniques of current high performance cores. Thus, it is a good representative of what is generally expected to be the processing element in future many-cores. Table 2.1 provides an overview of the key parameters of the 21264 relevant for this analysis.

2.2 Scaling of the Alpha 21264

The 21264 is scaled according to data in the 2007 edition of the International Technology Roadmap for Semiconductors (ITRS) [84]. At the time this analysis was performed that was the latest version available. At time of writing this thesis though, the 2009 edition has been published. The relevant data has only

Table 2.1: Parameters of the Alpha 21264 chip.

year	1998
technology node	350 nm
supply voltage	2.2 V
die area	314 mm^2
dynamic power (400 *MHz*)	48 W
dynamic power (600 *MHz*)	70 W

slightly changed compared to the 2007 edition, resulting in moderately more optimistic results in terms of performance growth. As the general trend is not affected, in this chapter we present the original analysis using the 2007 edition of the roadmap.

The relevant parameters are given in Table 2.2. The time frame considered is 2007-2022, which is the exact time span of the roadmap. The values of the technology node and the on-chip frequency were taken from Page 79 of the Executive Summary Chapter. The on-chip frequency is based on the fundamental transistor delay, and an assumed maximum number of 12 inverter delays. The die area values were taken from the table on Page 81 of the Executive Summary. Finally, the values of the supply voltage and the gate capacitance (per micron device) were taken from the table starting at Page 11 of the Process Integration, Devices, and Structures Chapter of the roadmap.

To model the experimental multicore for future technology nodes, we scale all required parameters of the 21264 core. The values that are available in the ITRS, we use as such. The others we scale using the available parameters by taking the ratio between the original 21264 parameter values and the predictions of the roadmap. Below we describe in detail for each scaled parameter how this was done. The gate capacitance of the 21264 was not found in literature, thus we extrapolated the values reported in the roadmap and calculated a value of $1.1 \times 10^{-15} F/\mu m$ for 1998.

First, the area of one core was scaled. Let $L(t)$ be the process technology size for year t and let L_{orig} be the process technology size of the original core. The area of one 21264 core in year t will be

$$A_1(t) = A_{orig} \times \left(\frac{L(t)}{L_{orig}} \right)^2 . \tag{2.1}$$

Figure 2.2 depicts the results for the time frame considered. The area of one

Table 2.2: Technology parameters of the ITRS roadmap.

	2007	**2008**	**2009**	**2010**	**2011**	**2012**
technology (nm)	68	57	50	45	40	36
frequency (MHz)	4700	5063	5454	5875	6329	6817
die area (mm^2)	310	310	310	310	310	310
supply voltage (V)	1.1	1	1	1	1	0.9
$C_{g,total}$ ($F/\mu m$)	7.10E-16	8.40E-16	8.43E-16	8.08E-16	6.5E-16	6.29E-16
	2013	**2014**	**2015**	**2016**	**2017**	**2018**
technology (nm)	32	28	25	22	20	18
frequency (MHz)	7344	7911	8522	9180	9889	10652
die area (mm^2)	310	310	310	310	310	310
supply voltage (V)	0.9	0.9	0.8	0.8	0.7	0.7
$C_{g,total}$ ($F/\mu m$)	6.28E-16	5.59E-16	5.25E-16	5.07E-16	4.81E-16	4.58E-16
	2019	**2020**	**2021**	**2022**		
technology (nm)	16	14	13	11		
frequency (MHz)	11475	12361	13351	14343		
die area (mm^2)	310	310	310	310		
supply voltage (V)	0.7	0.65	0.65	0.65		
$C_{g,total}$ ($F/\mu m$)	4.1E-16	3.91E-16	3.62E-16	3.42E-16		

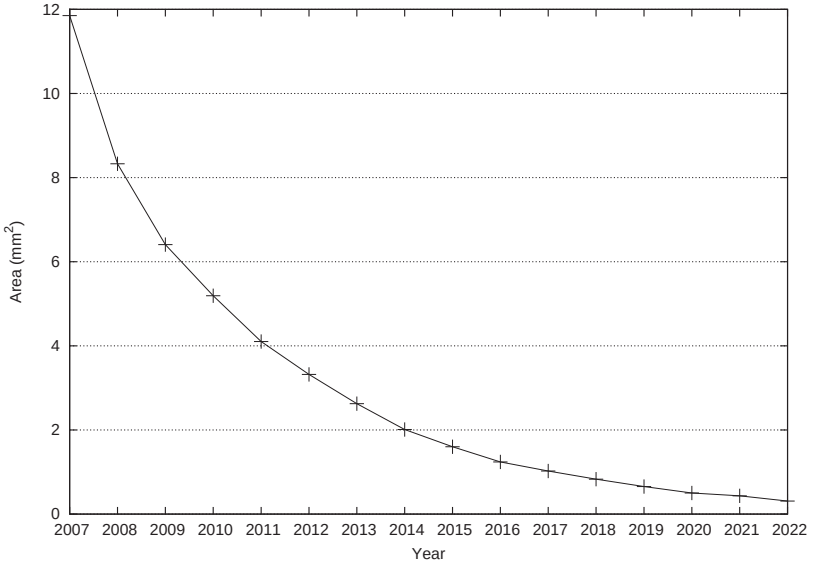


Figure 2.2: Area of one core.

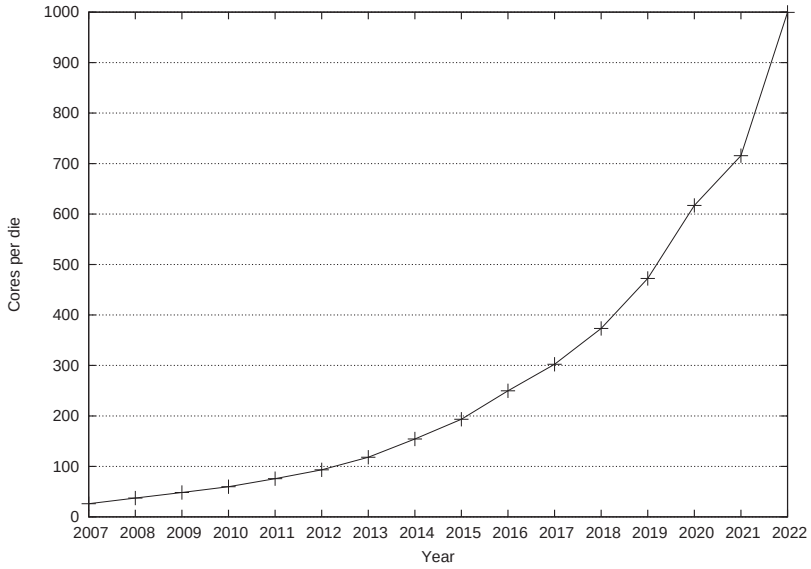


Figure 2.3: Number of cores per die.

core decreases more or less quadratically over time, and is predicted to be about one third of a square millimeter in 2022.

Second, using the scaled area of one core, the number of cores that could fit on a single die was calculated. The ITRS roadmap assumes a die area of $310mm^2$ for the entire time frame. Thus, the total number of cores per die in year t is given by

$$N(t) = \frac{310}{A_1(t)}, \tag{2.2}$$

which is depicted in Figure 2.3.

According to those results, in 2010 a 60-core chip would be possible. This seems reasonable considering the state-of-the-art multicores in 2010. The Tilera Tile-Gx100 [156] contains up to 100 cores, each having their own L1 and L2 cache and can run a full operating system autonomously. These cores are similar to the Alpha 21264 from an architectural point of view. The ClearSpeed CSX700 [47] contains 192 cores, which are simpler and targeted at exploiting data-level parallelism. Intel has created an experimental many-core called 'Single-chip Cloud Computer', which contains 48 x86 cores [82]. From 2007 on the graph shows a doubling of cores roughly every three years which is in line with the expected increase [148]. In 2022 our calculations predict a multicore with nearly one thousand cores.

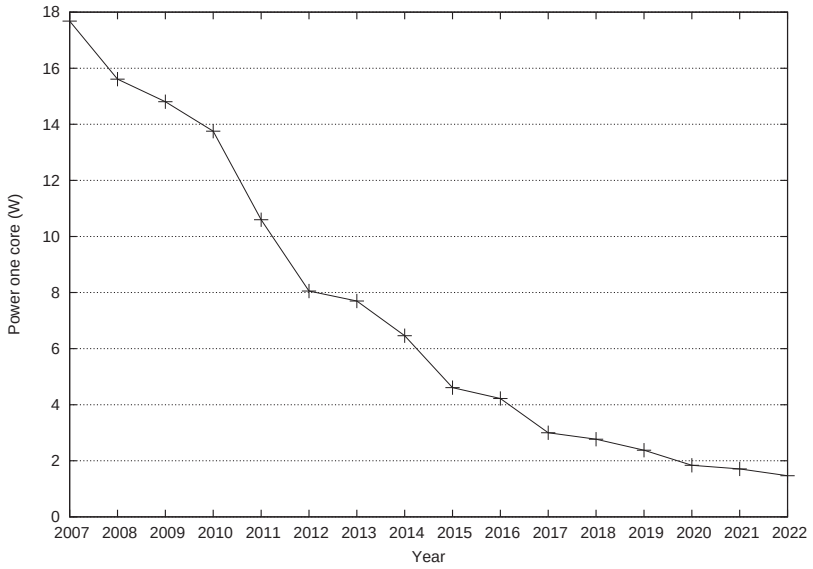


Figure 2.4: Power of one core when running at the maximum possible frequency.

Finally, the power of one core was scaled. Power consumption consists of a dynamic and a static part, of which the latter is dominated by leakage. The data required to scale the static power is not available to us and thus we restrict this power analysis to dynamic power. It is expected that leakage remains a problem and thus our estimations are conservative.

The dynamic power is given by

$$P_{dyn} = \alpha C f V^2, \tag{2.3}$$

where α is the transistor activity factor, C is the gate capacitance, f is the clock frequency, and V is the power supply voltage. The activity factor α of the 21264 processor is unknown and also depends on the application, but since this does not change with scaling, it can be assumed to be constant. The capacitance C (F) in the equation is different from capacitance $C_{g,total}$ ($F/\mu m$) in Table 2.2, but they relate to each other according to $C \propto C_{g,total} \times L$. Thus, the dynamic power at year t is calculated as:

$$P(t) = P_{orig} \times \frac{C_{g,total}(t) \times L(t)}{C_{g,total,orig} \times L_{orig}} \times \frac{f(t)}{f_{orig}} \times \left(\frac{V(t)}{V_{orig}}\right)^2. \tag{2.4}$$

Figure 2.4 depicts the power of one core over time, assuming it runs at the maximum possible frequency. As the curve shows, it roughly decreases linearly,

resulting in 1.5 W in 2022. The power density, on the other hand, increases. For example, in 2022 one core is modeled to be 0.3 mm^2 large, and thus the power density would be 5 $Watts/mm^2$. Therefore, the maximum frequency might not be obtainable.

2.3 Power and Performance Assuming Perfect Parallelization

Now that all parameters have been scaled, it is possible to calculate the power consumption of the entire multicore. It is assumed that all cores are active and thus

$$P_{total}(t) = N(t) \times P(t). \tag{2.5}$$

Figure 2.5 depicts the total power over time and shows that for the assumptions of this analysis the total power consumption in 2008 is 600 W, gradually increases, and reaches about 1.5 kW in 2022. The roadmap also predicts that the power budget allowed due to packaging constraints is 198 W. It is clear that for the entire calculated time span the power consumption of our hypothetical multicore exceeds the power budget. This is why power has become one of the main design constraints.

The figure also shows that the difference between the power budget and the power consumption of the full blown hypothetical multicore is increasing over time. That means that a large part of the technology improvement cannot be put into effect for performance growth. For example, between 2011 and 2020 the roadmap predicts that technology allows doubling the on-chip frequency. The power consumption, however, would increase with a factor 1.5. Thus, for equal power only a small frequency improvement would be possible.

Next we analyze the performance improvement that can be achieved by this multicore. Assuming that the application is perfectly parallelizable, the parameters that influence performance are frequency and the number of cores. In this case the speedup $S(t)$ in year t, relative to the original 21264 core, is given by

$$S(t) = \frac{f(t)}{f_{orig}} \times \frac{N(t)}{1}. \tag{2.6}$$

We refer to this speedup as the non-constrained speedup. It is depicted in Figure 2.6.

We are interested in the speedup achieved by multicores that meet the power budget of 198 W. As the results show the power budget is exceeded when

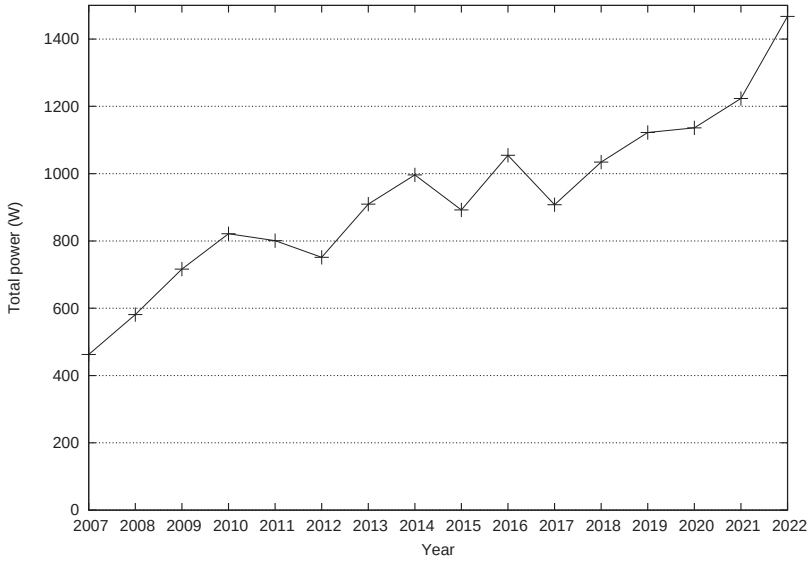


Figure 2.5: Total power consumption of the scaled multicore over the years.

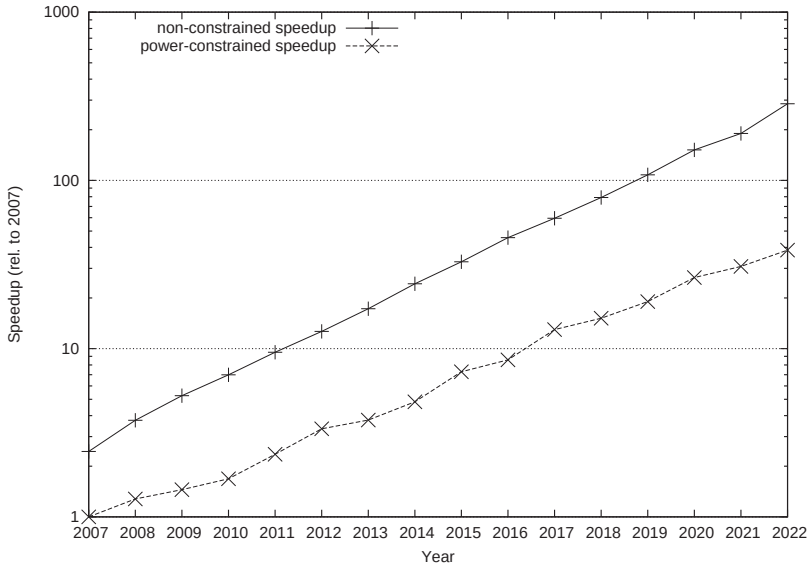


Figure 2.6: Power-constrained performance growth.

all cores are used concurrently at the maximum frequency. Thus, the non-constrained speedup is not expected to be achievable in practice. To meet the power budget, either the frequency could be scaled down or a number of cores could be shut down. Since both measures decrease the speedup linearly, the power-constrained speedup can be defined as:

$$S_{power\_constr.}(t) = \frac{f(t)}{f_{orig}} \times \frac{N(t)}{1} \times \frac{P_{budget}}{P_{total}(t)}, \tag{2.7}$$

where P_{budget} is the power budget allowed by packaging.

Figure 2.6 depicts the power-constrained performance of the case study multicore over time. To increase the readability we have normalized the result to 2007. The curve shows a performance growth of 27% per year. The non-constrained performance growth is also depicted. The latter is growing with 37% per year, and thus the gap between the two is increasing. To put these results in historical perspective we compare them to Figure 2.1 of Hennessey and Patterson [73]. From the mid-1980s to 2002 the latter graph shows an annual performance growth of 52%. Then from 2002 the annual performance growth dropped to 20%. Considering this historical perspective, the predicted annual performance growth of 27% is slightly higher, but not far off. The main conclusion from these results is that although power severely limits performance, substantial performance growth is still possible using the multicore paradigm.

2.4 Intermezzo: Amdahl vs. Gustafson

So far we assumed a perfectly parallelizable application. In practice this is not always the case as there might be code that needs to be executed serially. If this is the case we can apply either Amdahl's Law [24] or Gustafson's Law [70]. In this intermezzo we review them both briefly.

Amdahl's Law is the most well known of the two. In general terms it states that the performance improvement achieved by some feature is limited by the fraction of time the feature can be used. Originally Amdahl formulated his law to argue against parallel computers. He observed that typically 40% of the instructions were overhead which consisted of serial code that cannot be parallelized. Therefore, the speedup that can be achieved by a parallel computer

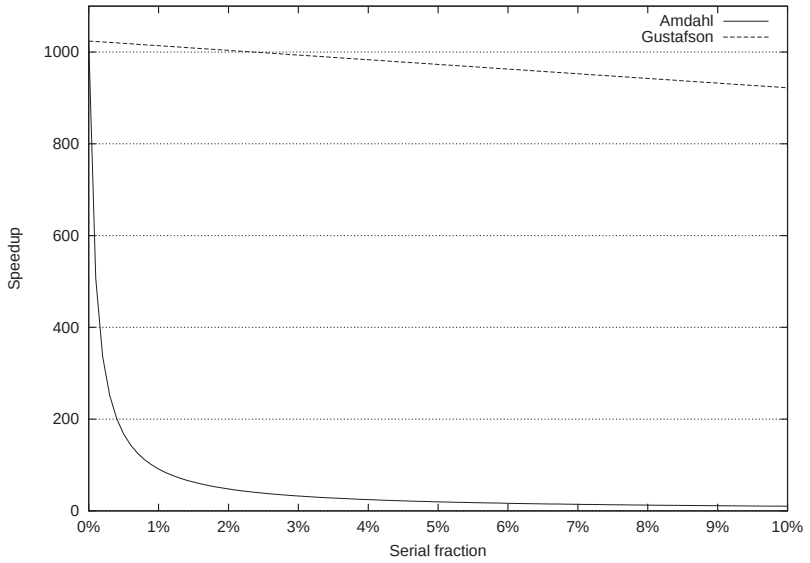


Figure 2.7: Wrong comparison of speedup under Amdahl's and Gustafson's Law for $N = 1024$.

would be very limited. In a formal way, Amdahl's Law is given by:

$$\begin{aligned} Speedup &= \frac{(s+p)}{(s+p/N)} \\ &= \frac{1}{(s+p/N)}, \end{aligned} \tag{2.8}$$

where N is the number of cores, s is the fraction of time spent on serial parts of a program, and $p = 1 - s$ is the fraction of time spent (by a single processor) on parts of the program that can be done in parallel.

Amdahl's Law assumes firstly that the serial and the parallel part both take a constant number of calculation steps independent of N, and secondly that the input size remains constant, independent of N. In other words, Amdahl does not consider scaling the problem size, while multiprocessors are generally used to solve larger problems.

Figure 2.7 depicts Amdahl's Law as a function of s for $N = 1024$. It shows that the function is very steep for s close to zero. On the other hand, Gustafson noticed that on a 1024-processor machine they achieved speedups between 1016 and 1021 for $s = 0.4 - 0.8$ percent. For $s = 0.4\%$ Amdahl's Law predicts a maximum speedup of 201 and thus Gustafson's findings seemingly

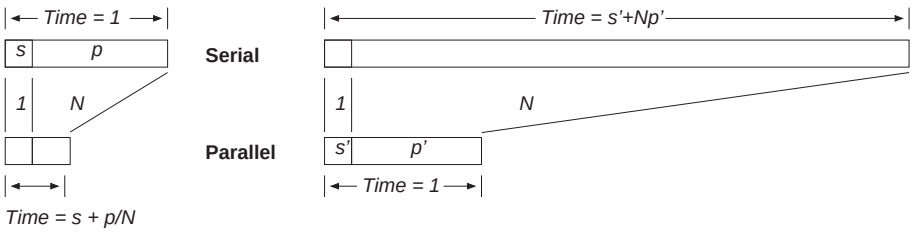


Figure 2.8: Gustafson's explanation of the difference between his speedup (right) and Amdahl's speedup (left).

conflicted with Amdahl's Law.

From these results Gustafson concluded that Amdahl's Law was inappropriate to massive parallel systems and he proposed a new law to calculate the speedup of massive parallel systems. He started reasoning from the parallel machine and defined s' and p' as the serial and parallel fractions of time spent on a parallel system*. The time on the parallel system is given by $s' + p' = 1$ and a serial processor would require time $s' + p' \times N$ to perform the same task. Gustafson explained the difference between his reasoning and that of Amdahl using Figure 2.8. The reasoning of Gustafson leads to the following equation, now known as Gustafson's Law:

$$\begin{aligned} Scaled\ speedup &= (s' + p' \times N)/(s' + p') \\ &= (s' + p' \times N) \\ &= N + (1 - N) \times s' \end{aligned} \tag{2.9}$$

Gustafson called the results of his equation the 'scaled speedup' assuming he calculated a different speedup. Figure 2.7 shows, beside the speedup under Amdahl's Law, the speedup using Gustafson's equation. The latter corresponds to the experimental findings of Gustafson.

Gustafson, however, used a different definition of the serial fraction than Amdahl did. The first measured s' in the parallel program while the latter measured s in the serial program (see Figure 2.8). Yuan Chi proved that actually the two equations are identical [139], which is not difficult to understand. Take the experiment of Gustafson with $s' = 0.4\%$ (measured on a parallel machine) and $N = 1024$, which results in speedup of 1020 using Gustafson's Law. Assume that on a parallel machine the total execution time is 1. Then the serial

*We distinguish between s and p from Amdahl and s' and p' from Gustafson. As we show later, those are not the same although apparently Gustafson did not realize that.

part takes 0.004 time units. The same task on a serial machine would take $s' + Np' = 1019.908$ time units. The fraction of the serial part on this serial run is $0.004/1019.908 = 3.92 \times 10^{-6}$. Filling this value of s into Amdahl's Law results in a speedup of 1020, exactly as Gustafson's Law predicted and the experiments showed. This also explains why Figure 2.7 provides a wrong comparison of the two laws.

That is, however, not all there is to say about the two laws. Gustafson showed that Amdahl's pessimistic prediction of the future was wrong and he also pointed out why. Amdahl assumed that the large s measured in his time would remain large in the future, i.e., he considered the fraction s to be a constant, because he assumed a fixed program and input size. Gustafson correctly observed that over time the problem size had grown and also that the serial fraction is dependent on the problem size.

Therefore, in predicting the future Gustafson assumed that the problem size scales with the number of processors. Furthermore, he assumed that the execution time of the serial part is independent on the problem size. That is, he assumed the absolute value of the serial fraction to be constant over the number of processors in contrast to Amdahl who assumed the relative value of the serial fraction to be constant over the number of processors.

The different assumptions are hidden in the two equations in the way the serial fraction is defined. Usually, when predicting the future, people simply take a fixed value s and use either of the two equations to calculate the speedup for a number of processors (see Figure 2.9). Taking a fixed value for s, however, means something different in the two laws and thus the user (unfortunately often unconsciously) agrees with the assumptions corresponding to the chosen equation. Because the assumptions are different, the equations produce different results although they are identical. It is not the different equation that produces the different results; it is the different set of assumptions that produce the different result. For example, it is possible to use Amdahl's equation with Gustafson's assumptions and find Gustafson's prediction. To do that, for every N you have to recalculate the value of s' to match Amdahl's definition of s.

Therefore, when predicting the future one has to consider the assumptions carefully. Otherwise, one is amenable to making the same mistake as Amdahl who in 1967 pessimistically predicted that there was no future for massive parallel systems. On the other hand, using Gustafson's assumptions one might obtain a too optimistic prediction. The best way is to thoroughly analyze the application domain under consideration and classify it as 'Amdahl' or 'Gustafson'. But this has to be done careful as the following example shows.

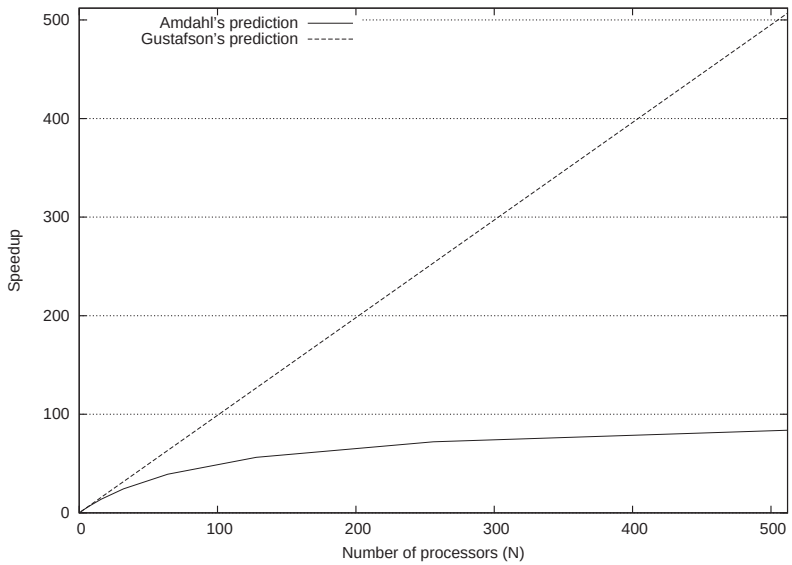


Figure 2.9: Prediction of speedup for $s = s' = 1\%$ with Amdahl's and Gustafson's assumptions.

H.264 video decoding contains an entropy decoding stage. It is parallelizable on the slice and frame level but any strategy exploiting this was generally assumed to be impractical or not scalable. Thus, often Amdahl's assumptions were used to predict a pessimistic future for parallelizing H.264. In Chapter 3, however, a new parallelization strategy is proposed where exploiting frame-level parallelism can effectively be deployed. With this parallelization strategy the 'serial' entropy decoding can be parallelized to a certain extent resulting in a much more optimistic prediction. In this case the difference is not in the assumption of a fixed problem size but that of the fixed program.

Not all application domains can be strictly classified as Amdahl or Gustafson, but something in between would be more appropriate. We will use the following equation to model that:

$$S = \frac{N}{\frac{s(N-1)}{s+\sqrt[n]{N}(1-s)} + 1}. \tag{2.10}$$

This equation is equivalent to Amdahl's and Gustafson's equations but models the assumption on the serial fraction s through the variable n. A value of $n = 1$ equals Gustafson's assumption while $n = \infty$ equals Amdahl's assumption. Figure 2.10 depicts the speedups predicted for different values of n.

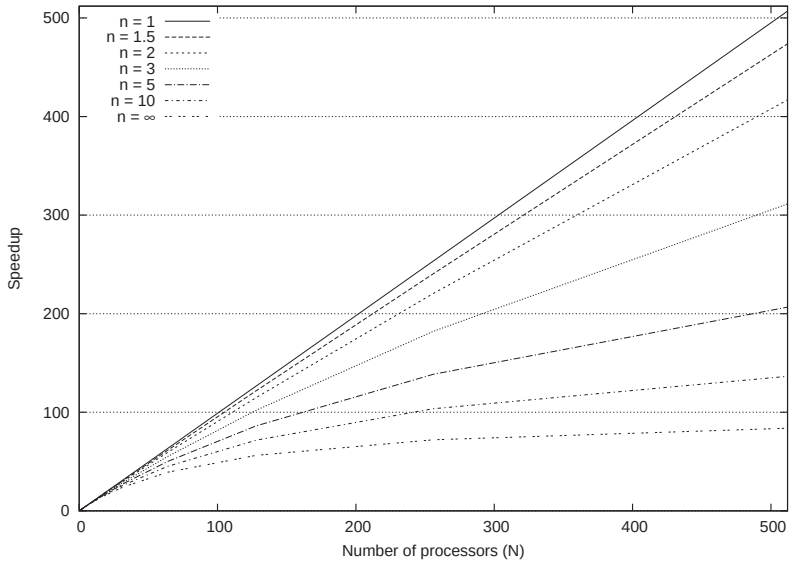


Figure 2.10: Prediction of speedup for $s = 1\%$ using Equation 2.10; $n = 1$ equals Gustafson's assumptions while $n = \infty$ equals Amdahl's assumptions.

An interesting research question is what value of n should be assigned to what application (domain). If this would be possible it would resolve the issue of which assumptions to use in predicting the future and hopefully the confusion about the two laws would diminish. Such research should go beyond a scalability analysis at the algorithm levels and include historical trends in data size and advances in algorithms.

2.5 Performance Assuming Non-Perfect Parallelization

In the previous section we showed how Amdahl's and Gustafson's assumptions and equations can be applied in predicting the future for non-perfect parallelization. The choice between the two showed to be dependent on the (assumed) characteristics of the application domain. In this section we analyze the impact of non-perfect parallelization on the power-constrained performance growth using both Amdahl's and Gustafson's assumptions. Depending on the application domain, it should be determined which one is more appropriate.

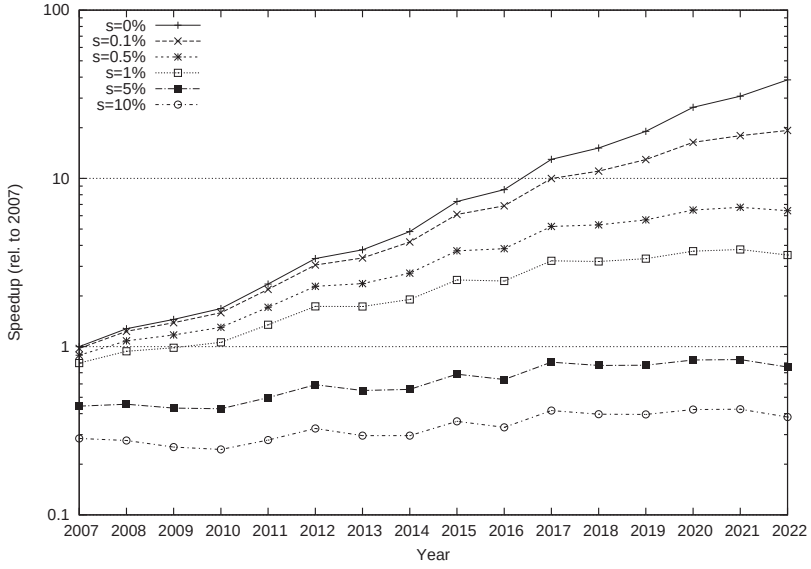


Figure 2.11: Prediction of power-constrained performance growth for several fractions of serial code s assuming a symmetric multicore and with Amdahl's assumptions.

First, we take Amdahl's assumptions to predict the power-constrained performance growth. We assume a symmetric multicore where all cores are being used during the parallel part. The clock frequency of all cores is equal and has been scaled down to meet the power budget. The power-constrained speedup that this symmetric multicore can achieve is given by

$$S_{Amdahl\_power-constr.\_sym.}(t) = \frac{1}{s + \frac{1-s}{N(t)}} \times \frac{f(t)}{f_{orig}} \times \frac{P_{budget}}{P_{total}(t)} \qquad (2.11)$$

and is depicted in Figure 2.11. We used a serial fraction s ranging from 0.1% to 10%. The figure shows that for $s = 0.1\%$ there is a slight performance drop, compared to ideal, going up to a factor of 2 for 2022. For $s = 1\%$, however, there is a performance drop of 10x for 2022 and for $s = 10\%$ there is no performance growth at all.

Indeed we see that Amdahl's prediction is pessimistic, which is an argument for asymmetric or heterogeneous multicores. If the serial part can be accelerated by deploying one faster core, more speedup through parallelism could be achieved. This serial processor could be an aggressive superscalar core, a domain specific accelerator, or a core that runs at a higher clock frequency

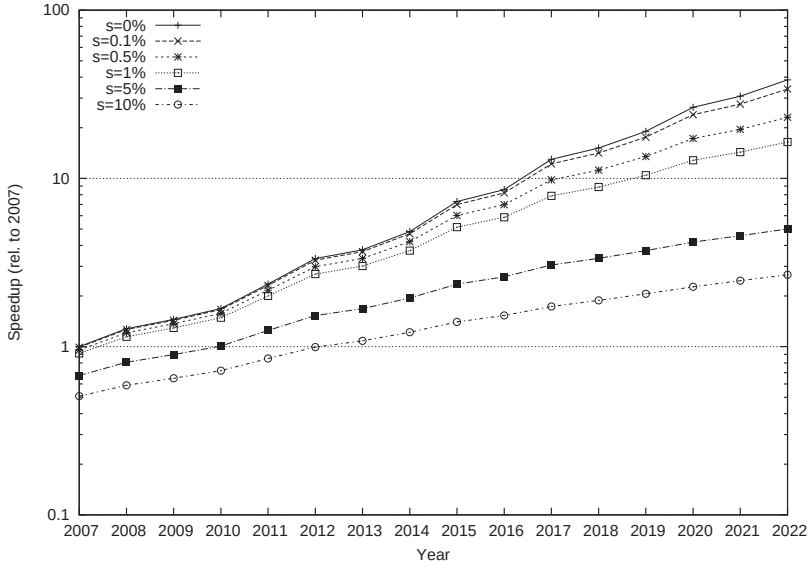


Figure 2.12: Prediction of power-constrained performance growth for several fractions of serial code s assuming an asymmetric multicore and with Amdahl's assumptions.

than the others. Using a serial processor with a higher clock frequency than the other cores, but with the same architecture, we refer to this multicore as asymmetric. When a different architecture is used for the serial processor, we refer to this multicore as heterogeneous. We analyze the effect of both type of multicores on the power constrained performance increase with Amdahl's assumptions.

In the asymmetric multicore we assume identical cores but increase the clock frequency of one core during the serial part. Note that during this time the other cores are inactive and thus the power budget is not exceeded. Limitations due to power density are not considered, though. The speedup this asymmetric multicore can achieve is given by

$$S_{Amdahl\_power-constr.\_asym.}(t) = \frac{1}{s \times \frac{f_{orig}}{f(t)} + \frac{1-s}{N(t)} \times \frac{f_{orig}}{f(t)} \times \frac{P_{total}(t)}{P_{budget}}} \tag{2.12}$$

and is depicted in Figure 2.12. Note that both this equation and Equation (2.11) become identical to Equation (2.7) if $s = 0\%$. The results show that for $s = 0.1\%$ there is only a very small performance drop compared to ideal parallelization. For $s = 1\%$ the performance drop is 2.3x in 2022 while for

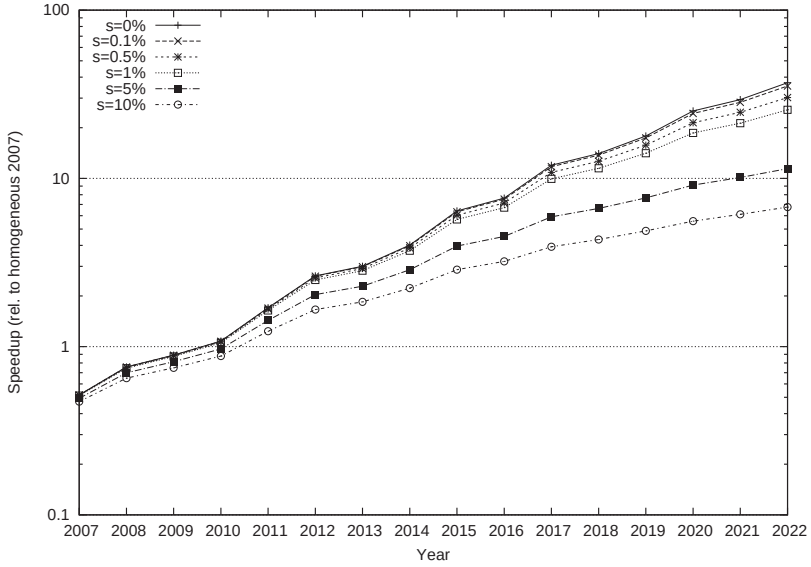


Figure 2.13: Prediction of power-constrained performance growth for several fractions of serial code s assuming a heterogeneous multicore (serial processor is 8x larger than others) and with Amdahl's assumptions.

$s = 10\%$ the performance drops 14 times compared to ideal parallelization but considerable performance growth is predicted over time. These results show that asymmetric multicores are a good choice to improve performance, if Amdahl's assumptions are correct and if the serial fraction is larger than approximately 0.5%.

To model the heterogeneous multicore we use Pollack's rule, as was done in [32, 75]. The rule states that performance increase is roughly proportional to the square root of the increase in complexity, where complexity refers to processor logic and thus equals area. A processor that is four times larger therefore is expected to have twice the performance.

To start, we model a multicore with one core that is 8 times larger than the other cores. This serial processor runs at the maximum clock frequency. The frequency of the other cores is reduced to meet the power budget of the chip. The serial part runs on the serial core, while the parallel part runs on all cores. The power constrained performance improvement for this heterogeneous multicore is depicted in Figure 2.13. Two effects are visible. First, for fully or highly parallel applications (low s), the performance is lower compared to the asymmetric multicore. This performance drop is significant for the early years

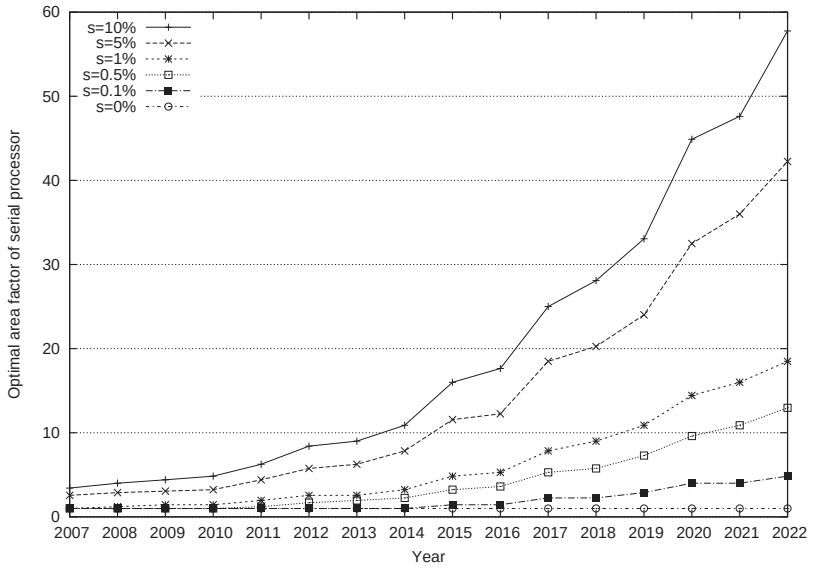


Figure 2.14: The optimal area factor of the serial processor for several fractions of serial code s.

(up to 2x) while for the later years it is negligible. Second, applications with a larger serial fraction have a better performance on the heterogeneous multicore compared to the asymmetric multicore. This performance increase is negligible for the early years, but significant (up to 2.5x) for the later years.

This example shows that a serial processor with a 8x size is not beneficial for all cases. The optimal area of the serial processor is depending on the serial fraction as well as on the technology parameters. Figure 2.14 shows the optimal area factors for several serial fractions. Figure 2.15 shows the percentage of die area such a serial processor would cost. For $s = 0\%$ the optimal area factor is always 1, which is the lowest we consider. For larger values of s, the optimal area factor increases over time. For $s = 10\%$, in 2022 the optimal area factor is 58x, which equals approximately 880 million transistors.

The power constrained performance increase for heterogeneous multicores containing the optimally sized serial core, is depicted in Figure 2.16. It differs from performance obtained with the asymmetric multicore, mainly for larger values of s. In 2022, for $s = 10\%$ the heterogeneous multicore has a 4.5 times higher performance that the asymmetric multicore. Even for smaller values of s, the performance is higher than that of the asymmetric multicore.

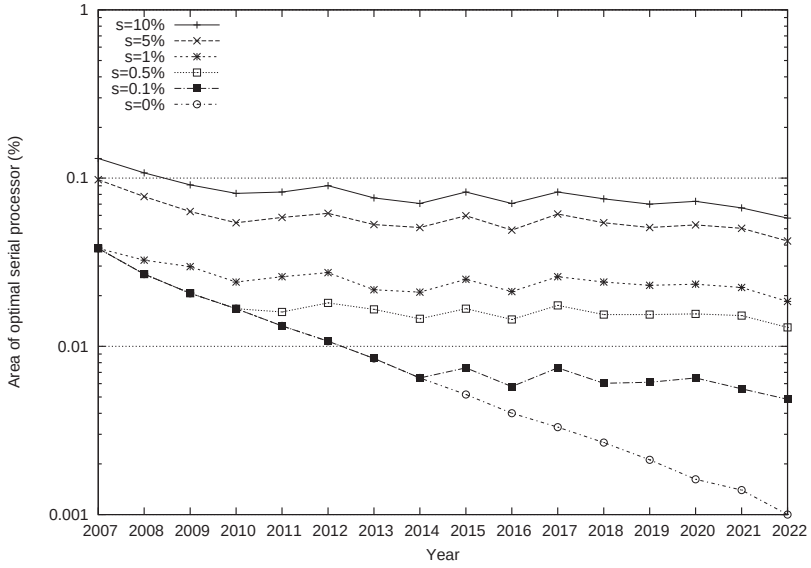


Figure 2.15: The area of the optimal serial processor (in percentage of the total die) for several fractions of serial code s.

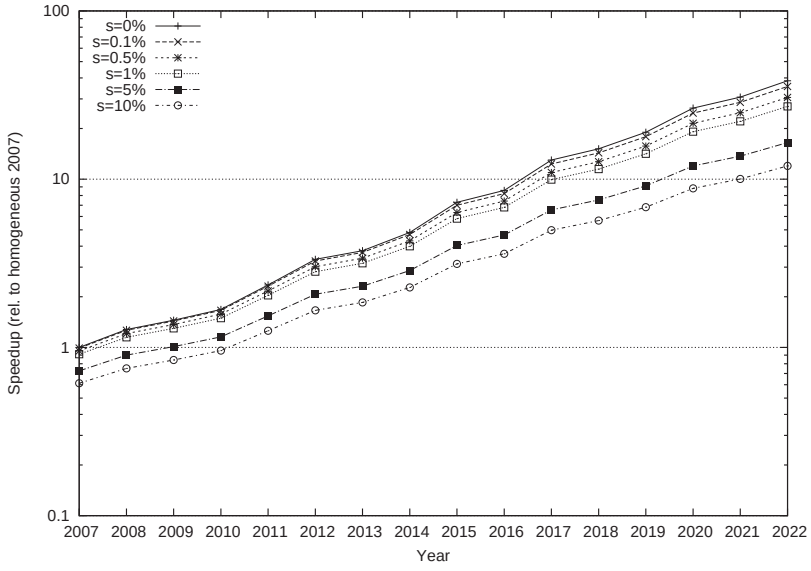


Figure 2.16: Prediction of power-constrained performance growth for several fractions of serial code s assuming a heterogeneous multicore (with optimal serial processor) and with Amdahl's assumptions.

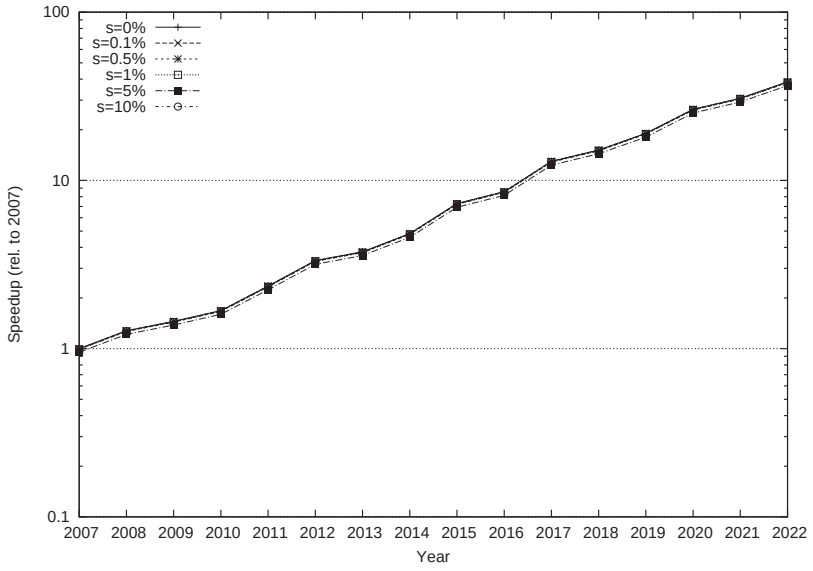


Figure 2.17: Prediction of power-constrained performance growth for several fractions of serial code s' assuming a symmetric multicore with Gustafson's assumptions.

Therefore, if Amdahl's assumptions are correct and the serial fraction is larger than 0.5%, a heterogeneous architecture is a better choice than an asymmetric architecture.

Second, we predict the power-constrained performance growth using Gustafson's assumptions. Again, we assume a symmetric multicore where all cores are being used during the parallel part. The clock frequency of all cores is equal and has been scaled down to meet the power budget. The power-constrained speedup that this symmetric multicore can achieve is given by

$$S_{Gustafson\_power-constr.\_sym.}(t) = \left(N + (1-N) \times s'\right) \times \frac{f(t)}{f_{orig}} \times \frac{P_{budget}}{P_{total}(t)} \tag{2.13}$$

and is depicted in Figure 2.17. The figure shows that for any value s' between 0% and 10% there will be no significant performance loss compared to ideal parallelization. For $s' = 10\%$ in 2022 the performance is 11% less than the ideal parallelization $s' = 0\%$ case. Thus we can conclude that homogeneous multicores are the best choice if the applications are mainly classified as 'Gustafson'. In case both application types should be covered, there is benefit

in using per-core voltage-frequency control. For 'Gustafson' type applications, a symmetric multicore is created by assigning each core an equal frequency. For 'Amdahl' type applications, an asymmetric multicore is created by assigning a higher clock frequency to the core(s) running the serial part.

2.6 Conclusions

In this chapter we analyzed the impact of the power wall on multicore design. Specifically, we investigated the limits to performance growth of multicores due to technology improvements with respect to power constraints. As a case study we modeled a multicore consisting of Alpha 21264 cores, scaled to future technology nodes according to the ITRS roadmap. Analysis showed that in 2022 such a multicore could contain nearly one thousand cores, each consuming 1.5 W when running at the maximum possible frequency of 14 GHz. The total multicore, at full blown operation, would consume 1.5 kW while the power budget predicted by the ITRS is 198 W.

From these figures it is clear that power has become a major design constraint, and will remain a major bottleneck for performance growth. It does not mean, however, that the power wall has been hit for multicores. We calculated the power-constrained performance growth and showed that technology improvements enable a doubling of performance every three years for multicores, which is in line with the number of cores per die.

There is another threat, however, for multicores which is Amdahl's Law. The speedup achieved by parallelism is limited by the size of the serial fraction s. Amdahl assumes a fixed program and input size and thus uses a constant fraction s over time. Using Amdahl's equation and assumptions we predicted the power-constrained performance growth for several fractions s. For a symmetric multicore, a serial fraction of 1% decreases the speedup by a factor of up to 10. For an asymmetric multicore, where the serial code is executed on a core that runs at a higher clock frequency, 1% of serial code reduces the achievable speedup by a factor of up to 2. This can be further reduced to 1.4 by using a heterogeneous architecture, containing one larger (and thus faster) core.

On the other hand, there is Gustafson's Law which assumes that the input size grows equally with increasing number of processors and that the serial part has a constant execution time independent of the input size. We also predicted the power-constrained performance increase using Gustafson's assumptions. The results are much more optimistic as even for $s' = 10\%$ in 2022 the performance loss is only 11% compared to ideal parallelization.

Whether Amdahl's or Gustafson's assumptions are valid for a certain application domain remains to be determined. Most likely some domains can be characterized as 'Amdahl', some as 'Gustafson' and other as something in between. This confirms that serial performance remains a challenge, as was also indicated in [74].

From the results of this chapter we conclude that in order to avoid hitting the power wall the following two principles should be followed. First, at the architectural level power efficiency has to become the main design constraint and evaluation criterion. The transistor budget is no longer the limit but the power those transistors consume. Thus performance alone should no longer be the metric but performance per watt (or a similar power efficiency metric like performance per transistor [78], $BIPS^3/W$ [95], and others [77]). Second, for application domains that follow Amdahl's assumptions asymmetric or heterogeneous designs are necessary. For those the need to speedup serial code remains. A challenge for computer architects is to combine speedup of serial code with power efficiency.

A multicore that follows these principles could for example consists of a few general purpose high speed cores (e.g., aggressive superscalar), many general purpose power-efficient cores (not wide issue, no out-of-order, no deep pipelines, etc.), and domain specific accelerators. Note that the architectural framework assumed throughout this work (see Figure 1.3) perfectly matches such a design. The accelerators provide the most power-efficient solution and also allow fast execution of serial code. For example, entropy coding in video codecs is a perfect candidate to be implemented by an accelerator. It is highly serial and benefits from high clock frequencies, but has mainly bit-level operations and thus a simple 8- or 16-bit core with a small instruction set would be best (see also Chapter 3). Furthermore, dynamic voltage/frequency scaling can be applied to optimize the performance-power balance, while hardware support for thread and task management reduces the energy of the overhead introduced by parallelism. A lot more techniques and architectural directions are possible.

Summarizing, from this study we conclude that for the next decade multicores can provide significant performance improvements without hitting the power wall, even though power severely limits performance growth. Technology improvements will provide the means. To achieve the possible performance improvements, however, power efficiency should be the main design criterion at the architectural level.

Chapter 3

Parallel Scalability of Video Decoders

With the doubling of the number of cores roughly every three years, within a decade tens to a hundred of high performance cores can fit on a chip. For power efficiency reasons those multicores might as well contain up to thousands of simple and small cores [27, 167]. A central question is whether applications scale to such large number of cores. If applications are not extensively parallelizable, cores will be left unused and performance suffers. Also, achieving power efficiency on many-cores relies on the amount of TLP available in applications.

In this chapter we investigate this issue for video decoding workloads by analyzing their parallel scalability. Multimedia applications remain important workloads in the future and video coders/decoders (codecs) are expected to be important benchmarks for all kind of systems, ranging from low power mobile devices to high performance systems. Specifically, we analyze the parallel scalability of the FFmpeg [58] H.264 decoder. We briefly review existing parallelization techniques, propose a more scalable strategy, analyze the available amount of TLP, and discuss actual implementations of the parallel H.264 decoder.

This work was performed in cooperation with two other PhD students (Arnaldo Azevedo and Mauricio Alvarez) and one MSc student (Chi Ching Chi). This chapter mainly describes the work performed by the author. For completeness though, certain work done by others is shortly presented and discussed. We will clearly indicate those sections.

This chapter is organized as follows. In Section 3.1 a brief overview of the H.264 standard is provided. Next, in Section 3.2 we describe the benchmarks used throughout this chapter. In Section 3.3 existing parallelization strategies are discussed and related work is reviewed. In Section 3.4 we propose the Dynamic 3D-Wave parallelization strategy. Using this new parallelization strategy we investigate the parallel scalability of H.264 in Section 3.5 and demonstrate that it exhibits huge amounts of parallelism. Section 3.6 describes a case study to assess the practical value of a highly parallelized decoder. Our experiences with actual implementations of the parallel H.264 decoder are discussed in Section 3.7. Section 3.8 concludes this chapter.

3.1 Overview of the H.264 Standard

Currently, the best video coding standard, in terms of compression and quality is H.264 [119]. It is used in HD-DVD and blu-ray Disc, 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 [87], H.262/MPEG-2 [86], etc. The H.264 standard [120] 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 3.1 and 3.2 depict block diagrams of the decoding and the encoding process of H.264, respectively. The main kernels are Prediction (intra prediction or motion estimation), Discrete Cosine Transform (DCT), Quantization, Deblocking filter, and Entropy Coding. These kernels process on macroblocks (MBs), which are blocks of 16×16 pixels, although the standard allows some kernels to process smaller blocks, down to 4×4. H.264 uses the YCbCr color space with mainly a 4:2:0 subsampling scheme. Each group of 2×2 pixels consists of four luma samples (Y), and two chroma samples (one Cb and one Cr). Thus, Cb and Cr are subsampled. Y is the luminance of the pixel, while Cb and Cr indicate the color as the difference from blue and red, respectively.

A movie is a sequence of images called frames, which can consist of several slices or slice groups. A slice is a partition of a frame such that all comprised MBs are in scan order (from left to right, from top to bottom). The Flexible

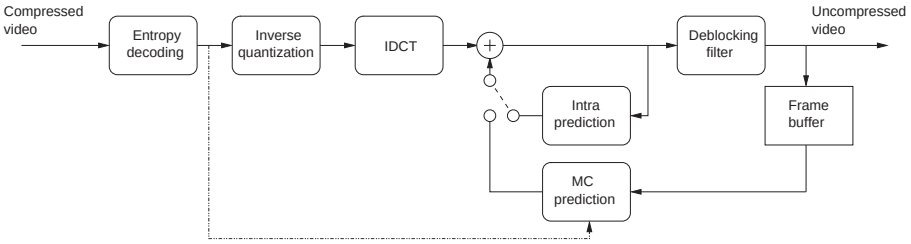


Figure 3.1: Block diagram of the decoding process.

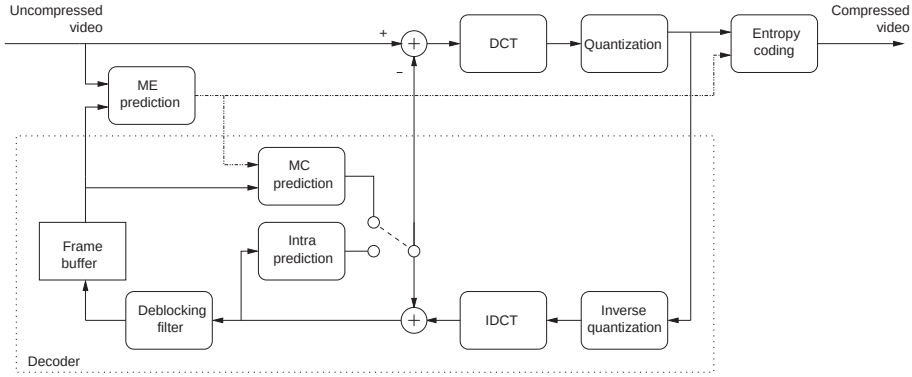


Figure 3.2: Block diagram of the encoding process.

Macroblock Ordering (FMO) feature allows slice groups to be defined, which consist of an arbitrary set of MBs. Each slice group can be partitioned into slices in the same way a frame can. Slices are self contained and can be decoded independently. Throughout this chapter the terms slice and frame are interchangeable, unless indicated differently, as we assume each frame consists of one slice.

H.264 defines three main types of slices/frames: I, P, and B-slices. An I-slice uses intra prediction and is independent of other slices. In intra prediction a MB is predicted based on adjacent blocks. A P-slice uses motion estimation and intra prediction and depends on one or more previous slices, either I, P or B. Motion estimation is used to exploit temporal correlation between slices. Finally, B-slices use bidirectional motion estimation and depend on slices from the past and future [60]. Figure 3.3 shows a typical slice order and the dependencies, assuming each frame consist of one slice only. The standard also

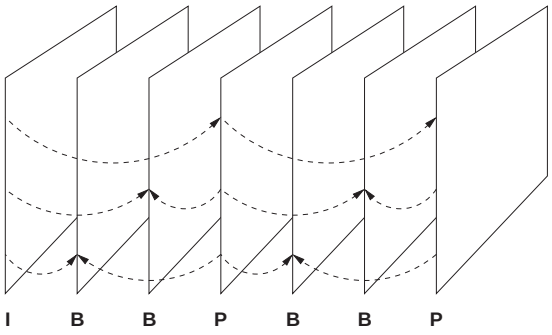


Figure 3.3: A typical frame sequence and dependencies between frames.

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 H.264 standard has many options. In the next sections we briefly mention the key features per kernel and compare them to previous standards. Table 3.1 provides an overview of the discussed features for MPEG-2, MPEG-4 ASP, and H.264. The different columns for H.264 represent profiles which are explained later. For more details the interested reader is referred to [165, 153].

Motion Estimation

Advances in motion estimation 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 motion vector(s). The motion vector is quarter sample accurate. Multiple reference frames can be used in a weighted fashion. This significantly improves the compression rate of coding occlusion areas, where an accurate prediction can only be made from a frame further in the past.

Intra Prediction

Three types of intra coding are supported, which are denoted as Intra_4x4, Intra_8x8 and Intra_16x16. 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_16x16 type is more suitable,

Table 3.1: Comparison of video coding standards and profiles.

	MPEG-2	MPEG-4 ASP	H.264 BP	H.264 MP	H.264 XP	H.264 HiP
Picture types	I, P, B	I, P, B	I, P	I, P, B	I, P, B, SI, SP	I, P, B
Flexible macroblock ordering	No	No	Yes	No	Yes	No
Motion block size	16×16	16×16, 16×8, 8×8	16×16, 16×8, 8×16, 8×8, 8×4, 4×8, 4×4	16×16, 16×8, 8×16, 8×8, 8×4, 4×8, 4×4	16×16, 16×8, 8×16, 8×8, 8×4, 4×8, 4×4	16×16, 16×8, 8×16, 8×8, 8×4, 4×8, 4×4
Multiple reference frames	No	No	Yes	Yes	Yes	Yes
Motion pel accuracy	1, 1/2	1, 1/2, 1/4	1, 1/2, 1/4	1, 1/2, 1/4	1, 1/2, 1/4	1, 1/2, 1/4
Weighted prediction	No	No	No	Yes	Yes	Yes
Transform	16×16	8×8 DCT	4×4 integer DCT	4×4 integer DCT	4×4 integer DCT	4×4, 8×8 integer DCT
In-loop deblocking filter	No	No	Yes	Yes	Yes	Yes
Entropy coding	VLC	VLC	CAVLC, UVLC	CAVLC, UVLC, CABAC	CAVLC, UVLC	CAVLC, UVLC, CABAC

for which four prediction modes are available. The high profile of H.264 also supports intra coding on 8×8 luma blocks. Chroma components are estimated for complete MBs using one specialized prediction mode.

Discrete Cosine Transform

MPEG-2 and MPEG-4 part 2 employ an 8×8 floating point transform, although often implemented using fixed point operations. Due to the decreased granularity of the motion estimation, however, there is less spatial correlation in the residual signal. Thus, standard a 4×4 (that means 2×2 for chrominance) transform is used, which is as efficient as a larger transform [109]. 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 the motion estimation [166]. Furthermore, to prevent rounding errors that occur in floating point implementations, an integer transform was chosen.

Deblocking Filter

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 [102]. 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 3.1) to improve the motion estimation.

Entropy Coding

Two classes of entropy coding are available in H.264: Variable Length Coding (VLC) and Context Adaptive Binary Arithmetic Coding (CABAC). The latter achieves up to 10% higher compression but at the cost of high computational complexity [110]. 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.

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 interoperate 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 two profiles, but was since then 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):** This is the simplest profile mainly used for video conferencing and mobile video.
- **Main Profile (MP):** This profile is intended to be used for consumer broadcast and storage applications, but is overtaken by the high profile.
- **Extended Profile (XP):** This profile is intended for streaming video and includes special capabilities to improve robustness.
- **High Profile (HiP)** This profile is 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 [149]. 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.

3.2 Benchmarks

Throughout this chapter we use the HD-VideoBench [22], which consists of movie test sequences, an encoder (X264 [168]), and a decoder (FFmpeg [58]). The benchmark contains the following test sequences:

- **rush_hour:** This sequence shows a rush-hour in the city of Munich. It has a static background and slowly moving objects on the forefront.

- **riverbed:** This sequence shows a river bed seen through waving water. It has abrupt and stochastic changes.
- **pedestrian:** This sequence is a shot of a pedestrian area in a city center. It has a static background and fast moving objects on the forefront.
- **blue_sky:** This sequence shows the top of two trees against a blue sky. The objects are static but the camera is sliding.

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 a length of 100 frames. For some experiments longer sequences were required, which we created by replicating the sequences. Unless specified otherwise, for SD and HD we used sequences of 300 frames while for FHD we used sequences of 400 frames.

The benchmark provides the test sequences in raw format. Encoding is done with the X264 encoder using the following options: 2 B-frames between I and P frames, 16 reference frames, weighted prediction, hexagonal motion estimation algorithm (hex) with maximum search range 24, one slice per frame, and adaptive block size transform. Movies encoded with this set of options represent the typical case. To test the worst case scenario we also created movies with large motion vectors by encoding the movies with the exhaustive search motion estimation algorithm (esa) and a maximum search range of 512 pixels. The first movies are marked with the suffix 'hex' while for the latter we use 'esa'.

3.3 Parallelization Strategies for H.264 *

In this section we briefly review existing parallelization strategies of H.264. An in-depth analysis of those, performed by Mauricio Alvarez, was presented in [112]. Our proposal to obtain higher scalability is described in the next section.

Parallelization strategies for H.264 can be divided in two categories: function-level and data-level decomposition. In function-level decomposition parallelism is obtained by pipelining the decoding process and assigning the different functions to different processors. In data-level decomposition parallelism is obtained by dividing the data among several processors, each running the same code.

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

Most parallelization techniques use data-decomposition as it is better in terms of load balancing and scalability. Using function-level decomposition, balancing the load is difficult because the time to execute each function is not known a priori and depends on the data being processed. Therefore, cores will often be idle waiting for a producer or consumer core, which reduces the efficiency of the system. Scalability is also difficult to achieve. The number of kernels in the H.264 decoding process is limited and therefore the scalability. Moreover, changing the number of cores used in this parallelization requires changing the mapping of kernels to cores. Function-level decomposition was applied to H.264 in a few works though [68, 135].

Data-level decomposition can be applied at different levels of the H.264 data structure (see Figure 3.4). 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 of a set of frames that can have temporal dependencies among them if motion prediction is used. Each frame consists of one or more slices, depending on the choice of the encoder. Each slice contains a set of macroblocks and there are no temporal or spatial dependencies between slices. Macroblocks 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. The next sections discus those parallelization briefly, except for the macroblock-level. The latter is discussed in more detail as the parallelization proposed in this chapter is based on macroblock-level parallelism.

3.3.1 GOP-level Parallelism

H.264 can be parallelized at the GOP level by defining a GOP size of N frames and assigning each GOP to a processor. GOP-level parallelism requires a lot of memory for storing all the frames, and therefore this technique maps well to multicomputers in which each processing node has a lot of memory resources. In multicores with a shared cache, cache pollution prevents this parallelization from being efficient. Moreover, parallelization at the GOP-level results in a very high latency which is generally only acceptable in encoding. GOP-level parallelism was applied to H.264 by [132].

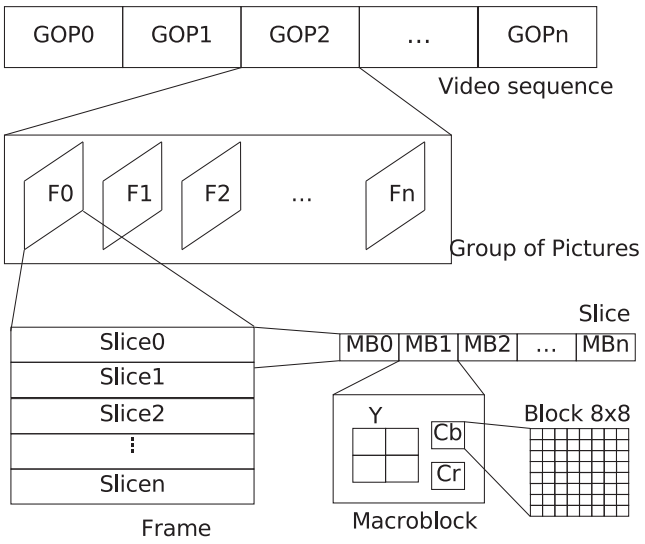


Figure 3.4: H.264 data structure.

3.3.2 Frame-level Parallelism

As shown in Figure 3.3, in a sequence of I-B-B-P frames inside a GOP, some frames are used as reference for other frames (like I and P frames) but some frames (the B frames in this case) are not. Therefore, in this case the B frames can be decoded in parallel. Frame-level parallelism has scalability problems, however, 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 is that, unlike previous video standards, in H.264 B frames can be used as reference [60]. As the encoding parameters are typically a given fact, this type of parallelism is uncertain. Frame-level parallelism has been applied to an H.264 encoder by Chen et al. [41] and has been proposed for multi-view H.264 encoding by Pang et al. [123].

3.3.3 Slice-level Parallelism

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 can be processed in parallel without dependency or ordering constraints. This allows exploitation of slice-

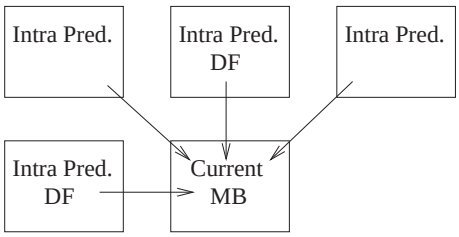


Figure 3.5: Dependencies between adjacent MBs in H.264.

level parallelism without making significant changes to the code.

The disadvantages of exploiting slice-level parallelism are the following. First, the number of slices per frame is determined by the encoder, and thus the amount of parallelism is uncertain. Second, although slices are completely independent, the deblocking filter possibly has to be applied across slice boundaries, reducing scalability. Third, there might be imbalance in the load of the cores if the decoding times of the slices are not equal. Finally, and most importantly, the bitrate increases proportional with the number of slices [113], at a rate which is unacceptable in most situations. Slice-level parallelism was applied to H.264 by [41, 133, 132]

3.3.4 Macroblock-level Parallelism

Data-level decomposition can also be applied at the MB-level. Usually MBs in a frame or slice are processed in scan order, which means starting from the top left corner of the frame and moving to the right, row after row. To exploit parallelism between MBs inside a frame 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 neighboring MBs defining a structured set of dependencies. These dependencies are shown in Figure 3.5. MBs can be processed out of scan order provided these dependencies are satisfied. Processing MBs in a diagonal wavefront manner satisfies all the dependencies and at the same time allows exploiting parallelism between MBs. We refer to this parallelization technique as 2D-Wave.

Figure 3.6 depicts an example of the 2D-Wave for a 5×5 MBs image (80×80 pixels). Each MB is labeled with a time slot that indicates the time slot at which it could be processed, assuming the MB decoding time is constant. 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

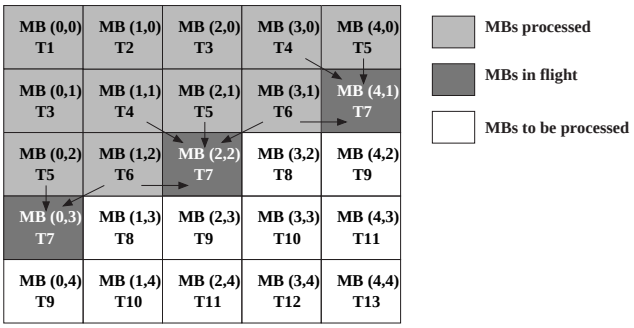


Figure 3.6: 2D-Wave approach for exploiting MB parallelism within a frame. The arrows indicate dependencies.

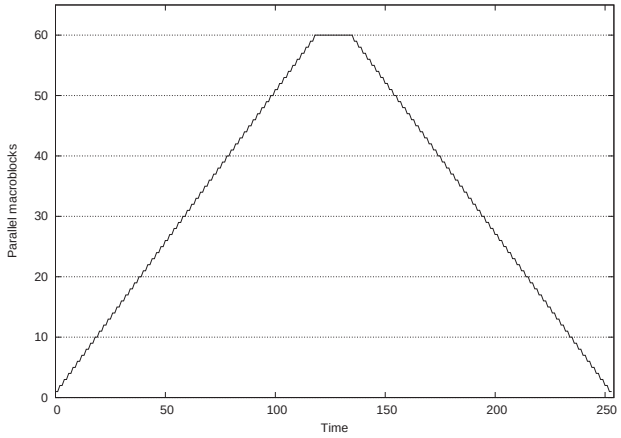


Figure 3.7: The amount of MB parallelism for a single FHD frame using the 2D-Wave approach.

in order to process each of these MBs. The maximum number of independent MBs in each frame depends on the resolution. In Table 3.2 these maximum number of independent MBs for different resolutions is stated. For a low resolution such as QCIF there are only 6 independent MBs during 4 time slots. For High Definition (1920x1088) there are 60 independent MBs during 9 slots of time. Figure 3.7 depicts the available MB parallelism over time for an FHD resolution frame, assuming that the time to decode a MB is constant.

MB-level parallelism has many advantages over other H.264 parallelization schemes. First, this scheme can have a good scalability. As shown in Table 3.2 the number of independent MBs increases with the resolution of the image.

Table 3.2: Maximum parallel MBs for several resolutions using the 2D-Wave approach. Also the number of times slots this maximum is available is stated.

Resolution		MBs	Slots
QCIF	176×144	6	4
CIF	352×288	11	8
SD	720×576	23	14
HD	1280×720	40	6
FHD	1920×1088	60	9

Second, it is possible to achieve a good load balancing if a dynamic scheduling system is used. A static scheduling approach incurs dependency stalls, as the MB decoding time varies.

This kind of MB-level parallelism, however, also has some disadvantages. The first one is that the entropy decoding cannot be parallelized using data decomposition, due to the fact that H.264 entropy coding adapts its probabilities using data from consecutive MBs. As a result 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 and in sequential order. Only after entropy decoding has been performed the parallel processing of MBs can start. This disadvantage can be mitigated by using special purpose instructions [161] or hardware accelerators [121, 39, 99] for the entropy decoding process.

The second disadvantage is that the number of independent MBs does not remain constant during the decoding of a frame, as shown in Figure 3.7. At the start of the frame there are few independent MBs, increasing though until the maximum is reached. Similarly, at the end the number of parallel MBs decreases to zero. Therefore, it is not possible to sustain a certain processing rate during the decoding of a frame. Using the parallelization strategy we propose in Section 3.4, this problem is overcome.

MB-level parallelism has been proposed in several works. Van der Tol et al. [162] have proposed H.264 MB-level parallelism for a multiprocessor system consisting of 8 TriMedia processors. No speedups were reported. Chong et al [45] have proposed another technique for exploiting MB-level parallelism in the H.264 decoder by adding a prepass stage. MBs are assigned to processors dynamically according to the estimated decoding time. By using this scheme a speedup of 3.5X has been reported on 6 processors.

Moreover, the paper suggests the combination of MB-level parallelism within

frames and among frames to increase the availability of independent MBs. The use of inter-frame MB-level parallelism is determined statically by the length of the motion vectors. Chen et al. [40] have evaluated a similar approach for the H.264 encoder on Pentium machines with SMT and multicore capabilities. In those two works the exploitation of inter-frame MB-level parallelism is limited to two consecutive frames and the identification of independent MBs is done statically by taking into account the limits of the motion vectors. Although this combination of intra-frame and inter-frame MB-level parallelism increases the amount of independent MBs, it requires that the encoder puts some limits on the length of the motion vectors in order to define statically which MBs of the next frame can start execution while decoding the current frame.

So far we only discussed MB-level parallelism within a frame. As suggested by [162], MB-level parallelism can also be exploited between frames, considering the dependencies through the motion compensation kernel are met. For H.264 *encoding* this has been implemented in X264 [168] and in the H.264 Codec that is part of the Intel Integrated Performance Primitives [83]. The first obtains a maximum speedup of approximately 16.4 using 32 threads.

Zhao et al. [170] combined inter-frame and intra-frame MB-level parallelism for H.264 encoding. This scheme is a variation of what we refer to in this thesis as static 3D-Wave. They obtained a speedup of approximately 3 using 4 cores and discarded an analysis for a high number of cores as an impractical case. The parallelization strategy that we propose in this thesis is also based on a combination of inter-frame and intra-frame MB-level parallelism. We consider H.264 decoding, which is more difficult to parallelize as encoding. Furthermore, by determining dynamically the actual dependencies between MBs, our approach is much more scalable as explained in Section 3.4.

3.3.5 Block-level Parallelism

Finally, the finest grained data-level parallelism is at the block level. Most of the computations of the H.264 kernels are performed at the block level. This applies, for example, to the interpolations that are performed in the motion compensation stage, to the IDCT, and to the deblocking filter. This level of data parallelism maps well to SIMD instructions [171, 140, 21, 96]. SIMD parallelism is orthogonal to the other levels of parallelism described above and because of that it can be mixed, for example, with MB-level parallelization to increase the performance of each thread.

3.4 Scalable MB-level Parallelism: The 3D-Wave

None of the approaches described in the previous section scales to future many-core architectures containing 100 cores or more. In the 2D-Wave there is a considerable amount of MB-level parallelism, but at the start and end of a frame there are only a few independent MBs.

In order to overcome this limitation, it is possible to exploit both intra-frame and inter-frame MB-level parallelism. Inside a frame, spatial MB-level parallelism can be exploited using the 2D-Wave scheme mentioned previously. Simultaneously, among frames temporal MB-level parallelism can be exploited.

Temporal MB-level parallelism can be exploited when considering the dependencies due to Motion Compensation (MC). MC can be regarded as copying an area, called the reference area, from the reference frame, and then to add 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 level defines, amongst others, the maximum length of the vertical component of the MV [120]. Not all encoders, however, follow the guidelines of the levels, but use the absolute limit of the standard which is 512 pixels vertical and 2048 pixels horizontal. In practice the motion estimation algorithm defines a maximum search range of dozens of pixels, because a larger search range is very computationally demanding and provides only a small benefit [104].

When the reference area has been decoded it can be used by the referencing frame. Thus it is not necessary to wait until a frame is completely decoded before decoding the next frame. The decoding process of the next frame can start after the reference areas of the reference frames are decoded. Figure 3.8 shows an example of two frames where the second depends on the first. The figure shows that MB $(2, 0)$ of frame $i + 1$ depends on MB $(2, 1)$ of frame i which has been decoded. Thus this MB can be decoded even though frame i is not completely decoded. Similarly, MB $(0, 1)$ of frame $i + 1$ depends on MB $(0, 2)$ of frame i. This dependency is resolved, as well as the intra-frame dependencies. Therefore, also this MB can be decoded.

Figure 3.8 shows a 2D-Wave decoding within each frame of the two frames. This technique can be extended to much more than two frames, adding a third dimension to the parallelization, as illustrated in Figure 3.9. Hence, we refer to this parallelization strategy as 3D-Wave.

The scheduling of MBs for decoding can be performed in a static or a dy-

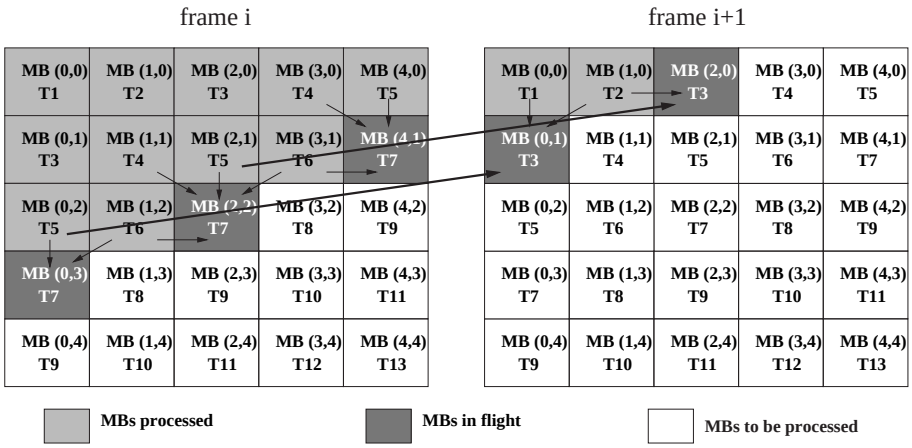


Figure 3.8: 3D-Wave strategy: intra-frame and inter-frame dependencies between MBs.

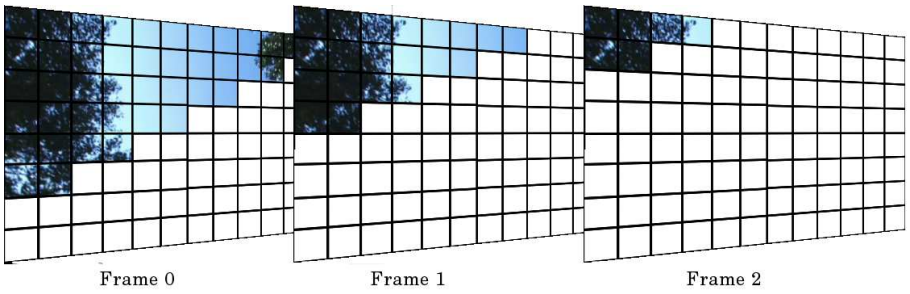


Figure 3.9: 3D-Wave strategy: frames can be decoded in parallel because inter-frame dependencies have limited spatial range.

namic way. Static 3D-Wave exploits temporal parallelism by assuming that the motion vectors have a restricted length. Based on that, it uses a fixed offset between the 2D-Waves of consecutive frames. This static approach has been implemented in previous work for the H.264 encoder [170]. It is more suitable for the encoder because it has the flexibility of selecting the size of the motion search area.

The static 3D-Wave for H.264 decoding has been investigated by Azevedo (see [113]). The amount of MB parallelism it exhibits is depending on the motion vector range (see Table 3.3). Motion vectors are typically short (< 16) and thus potentially a lot of parallelism is available. For FHD resolution even maximally 1360 MBs in parallel are available. The problem is the worst case

Table 3.3: Static 3D-Wave: maximum MB parallelism and frames in flight for several MV ranges and three resolutions.

MV range	Max # parallel MBs			Max # frames in flight		
	SD	HD	FHD	SD	HD	FHD
512	-	36	89	-	2	3
256	33	74	163	3	4	5
128	60	134	303	5	7	10
64	108	240	544	8	12	17
32	180	400	907	13	19	29
16	276	600	1360	20	28	43

scenario, which for FHD is a motion vector of length 512. As the static 3D-Wave always has to account for the worst case, the amount of parallelism that effectively can be exploited in such case is maximally 89.

Our proposal is the Dynamic 3D-Wave approach. It is not based on restricted values of the motion vectors. Instead, it uses a dynamic scheduling system in which MBs are scheduled for decoding when all the dependencies (intra-frame and inter-frame) have been satisfied. As illustrated in Figure 3.8, as soon as the search area in the reference frame has been decoded, the depending MB in a different frame can be processed. The dynamic system is best suited for the decoder taking into account that the length of motion vectors is not known a priori and that the decoding time of each MB is input-dependent. The Dynamic 3D-Wave system results in better scalability as we will show.

The dynamic scheduling can be implemented in various ways. Several issues play a role in this matter such as thread management overhead, data communication overhead, memory latency, centralized vs. distributed control, etc. In Section 3.7, while discussing our implementations of parallel H.264 decoding, we comment in more detail on the dynamic scheduling implementations.

Another issue is entropy decoding which does not exhibit TLP within a slice. Thus, entropy decoding is decoupled from the MB-decoding process using a function-level decomposition. That is, one core performs entropy decoding, which output is fed into the parallel MB-decoding process using several cores. If all cores are equal, one entropy decoding core can sustain approximately eight MB-decoding cores. Entropy decoding, however, can be parallelized at the slice level. Thus, multiple entropy decoding cores can be deployed, each processing a different slice or frame. Furthermore, entropy decoding can be accelerated by using specialized cores or ASICs. Assuming four entropy decoding cores are utilized, each twice as fast as a general purpose core, than a

total of 64 MB-decoding cores can be used efficiently. In the remainder of this chapter we focus on the parallelism in the MB decoding process, unless stated otherwise.

In the next section the parallel scalability of the Dynamic 3D-Wave is analyzed. The scalability obtained for actual implementations of the Dynamic 2D-Wave and 3D-Wave are discussed in Section 3.7.

3.5 Parallel Scalability of the Dynamic 3D-Wave

In this section we analyze the parallel scalability of the H.264 decoder by exploring the limits of the amount of TLP it exhibits. This research is similar to the ILP limit study for video applications presented in [100]. Investigating the limits to TLP, however, requires to put a boundary on the granularity of the threads. Without this boundary a single instruction can be a thread, which is clearly not feasible. Thus, using emerging techniques such as light-weight micro-threading [92], possibly even more TLP can be exploited. We show, however, that macroblock-level granularity is sufficient to sustain a many-core system.

To determine the amount of parallelism, we modified the FFmpeg H.264 decoder to analyze the dependencies in real movies, which we took from the HD-VideoBench. We analyzed the dependencies of each MB and assigned each MB a timestamp 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. As before, we assume that it takes one time unit to decode a MB.

The way inter-frame dependencies are handled needs further explanation. The pixels required for motion compensation are not only those of the sub-block the motion vector is pointing to. If the motion vector is fractional, the pixels used for MC are computed by interpolation and for this the pixels surrounding the sub-block are also required. Further, the standard defines that deblocking filtered pixels should be used for MC. The bottom and right four pixels of each

[†]The decoding order of frames is not equal to display order. The sequence I-B-B-P as in Figure 3.3 is decoded as I-P-B-B sequence.

Table 3.4: Maximum available MB parallelism and frames in flight for normal encoded movies. The maximum and the average motion vectors (in square pixels) are also stated.

	SD				HD			
	MBs	frames	max mv	avg mv	MBs	frames	max mv	avg mv
rush_hour	1199	92	380,5	1.3	2911	142	435	1.8
riverbed	1511	119	228.6	2	3738	185	496	2.2
pedestrian	1076	95	509.7	9.2	2208	131	553.5	11
blue_sky	1365	99	116	4.4	2687	135	298	5.1

	FHD			
	MBs	frames	max mv	avg mv
rush_hour	5741	211	441	2.2
riverbed	7297	245	568.9	2.6
pedestrian	4196	205	554.2	9.9
blue_sky	6041	214	498	5.6

MB are filtered only when the bottom or left MB is decoded. Thus, although a MV points to a sub-block that lies entirely in a MB A, the actual dependencies might go as far as the right and/or bottom neighbor of MB A. We took this into account by adding to each MV the pixels needed for interpolation and checking if the DF of the neighboring blocks would affect the required pixels.

To start, the maximum available MB-level parallelism is analyzed. This experiment does not consider any practical or implementation issues, but simply explores the limits of the amount of parallelism available in the application. We use the modified FFmpeg as described before and for each time slot we determined, the number of MBs that can be processed in parallel during that time slot. In addition, we keep track of the number of frames in flight. Finally, we keep track of the motion vector lengths. All experiments are performed for both normally encoded and esa encoded movies. The first uses a hexagonal (hex) motion estimation algorithm with a search range of 24 pixels, while the latter uses an exhaustive search algorithm and results in worst case motion vectors.

Table 3.4 summarizes the results for normally encoded movies and shows that a huge amount of MB-level parallelism is available. Figure 3.10 depicts the MB-level parallelism time curve for FHD (i.e., the number of MBs that can be processed in parallel in each time slot), while Figure 3.11 depicts the number of frames in flight in each time slot. For the other resolutions the time curves have a similar shape. The MB parallelism time curve shows a ramp up, a

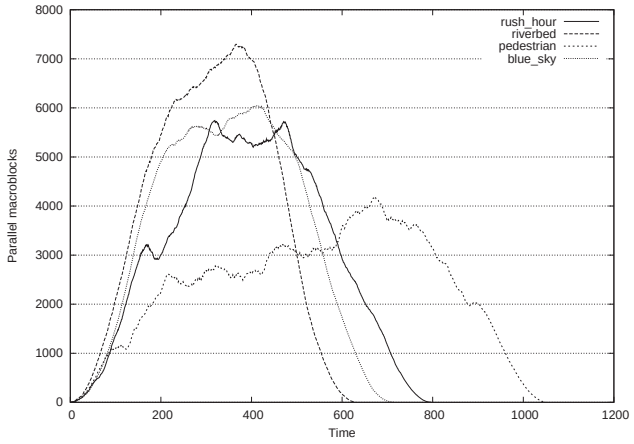


Figure 3.10: Number of parallel MBs in each time slot, using the 3D-Wave parallelization, for FHD resolution using a 400 frame sequence.

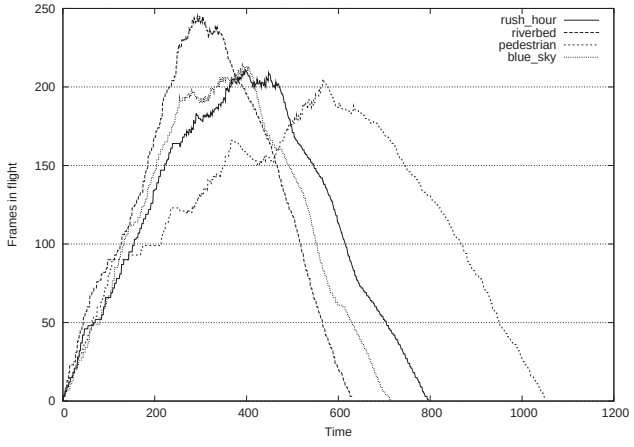


Figure 3.11: Number of frames in flight in each time slot, using the 3D-Wave parallelization, for FHD resolution using a 400 frame sequence.

Table 3.5: Maximum MB parallelism and frames in flight for esa encoded movies. Also the maximum and the average motion vectors (in square pixels) are stated.

	SD				HD			
	MBs	frames	max mv	avg mv	MBs	frames	max mv	avg mv
rush_hour	505	63	712,0	6.9	708	84	699.3	9
riverbed	150	27	704.9	62.3	251	25	716.2	63.1
pedestrian	363	50	691.5	23.5	702	70	709.9	27.7
blue_sky	138	19	682.0	10.5	257	23	711.3	13

	FHD			
	MBs	frames	max mv	avg mv
rush_hour	1277	89	706.4	9.4
riverbed	563	37	716.2	83.9
pedestrian	1298	88	708.5	23.6
blue_sky	500	24	705.7	15.7

plateau, and a ramp down. For example, for blue_sky the plateau starts at time slot 200 and last until time slot 570. Riverbed exhibits such a huge amount of MB-level parallelism that it has a small plateau. Due to the stochastic changing content of the movie, the encoder mostly uses intra-coding, resulting in very few dependencies between frames. Pedestrian exhibits the least amount of parallelism. The fast moving objects in the movie result in many large motion vectors. Especially objects moving from right to left on the screen cause large offsets between MBs in consecutive frames.

Rather surprising is the maximum MV length found in the movies (see Table 3.4). The search range of the motion estimation algorithm is limited to 24 pixels, but still lengths of more than 500 square pixels are reported. According to the developers of the X264 encoder this is caused by drifting [169]. For each MB the starting MV of the algorithm is predicted using the result of surrounding MBs. If a number of MBs in a row use the motion vector of the previous MB and add to it, the values accumulate and reach large lengths. This drifting happens only occasionally, and does not significantly affect the parallelism in the dynamic approach, but would force a static approach to use a large offset resulting in little parallelism.

To evaluate the worst case, the same experiment was performed for movies encoded using the exhaustive search algorithm (esa). Table 3.5 shows that the average MV is substantially longer than for normally encoded movies. The exhaustive search decreases the amount of parallelism significantly. The time

Table 3.6: Comparison of the maximum and average amount of MB-level parallelism for hex and esa encoded movies (FHD only).

	FHD					
	max MB			avg MB		
	hex	esa	Δ	hex	esa	Δ
rush_hour	5741	1277	-77.8%	4121.2	892.3	-78.3%
riverbed	7297	563	-92.3%	5181.0	447.7	-91.4%
pedestrian	4196	1298	-69.1%	3108.6	720.8	-76.8%
blue_sky	6041	500	-91.7%	4571.4	442.6	-90.3%

Table 3.7: Compression improvement of esa encoding relative to hex.

	SD	HD	FHD	average
rush_hour	-0.7%	0.0%	-0.3%	-0.3%
riverbed	1.4%	1.9%	3.2%	2.1%
pedestrian	0.6%	1.6%	1.2%	1.1%
blue_sky	-1.4%	-1.5%	1.4%	-0.5%
average	0.0%	0.5%	1.4%	0.6%

curves in Figures 3.12 and 3.13 have a very jagged shape with large peaks and thus the average on the plateau is much lower than the peak. In Table 3.6 the maximum and the average parallelism for normally (hex) and esa encoded movies is compared. The average is taken over all 400 frames, including ramp up and ramp down. Although the drop in parallelism is significant, the amount of parallelism is still large for this worst case scenario.

As we are comparing two encoding methods, it is important to look at some other aspects as well. In Table 3.7 we compare the compression ratios of hex and esa encoding. Only for FHD a considerable compression improvement is achieved when using esa encoding. The cost of this improvement in terms of encoding time, however, is large. A 300-frame FHD sequence takes less than 10 minutes to encode using hexagonal motion estimation and running on a modern desktop computer (Pentium D, 3.4 GHz, 2GB memory). The same sequence takes about two days to encode when using the esa option, which is almost 300 times slower. Thus we can assume that this worst case scenario is unlikely to occur in practice.

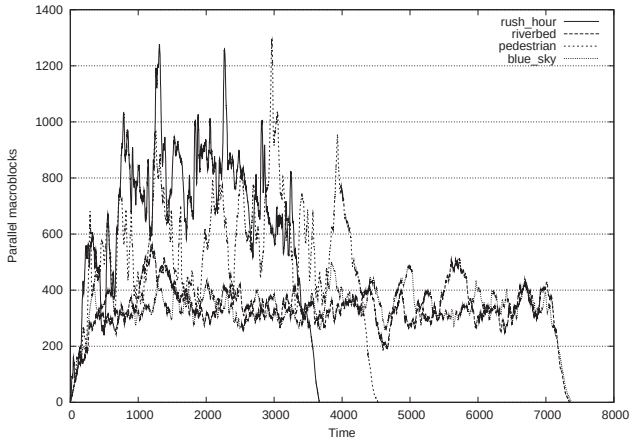


Figure 3.12: Number of parallel MBs in each time slot, using the 3D-Wave parallelization, for FHD using esa encoding.

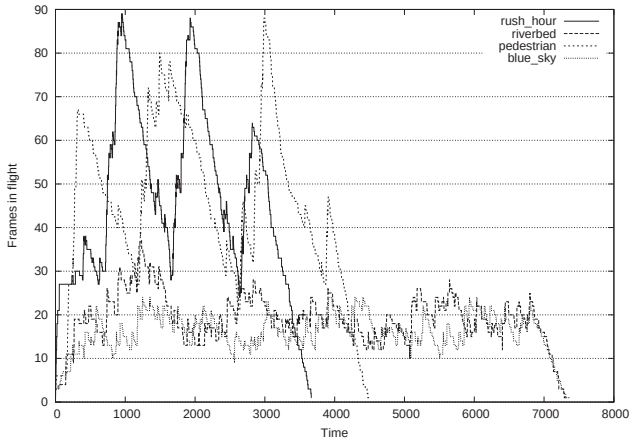


Figure 3.13: Number of frames in flight in each time slot, using the 3D-Wave parallelization, for FHD using esa encoding.

The analysis above reveals that H.264 exhibits significant amounts of MB-level parallelism. To exploit this type of parallelism on a multicore the MB decoding tasks need to be assigned to the available cores. Even in future many-cores, however, the hardware resources, e.g., the number of cores, the size and bandwidth of memory, and the NoC bandwidth, will be limited. We now investigate the impact of such resource limitations.

We model limited resources as follows. A limited number of cores is modeled by limiting the number of MBs in flight, as each MB in flight requires one core to execute on. Memory requirements are mainly proportional to the number of frames in flight, as each frame in flight should be stored in main memory. Thus limited memory is modeled by restricting the number of frames that can be in flight concurrently.

The experiment has been performed for all four movies of the benchmark, for all three resolutions, and both types of motion estimation. The effect of the limitations is similar for all movies, and thus only the results for the normally encoded blue_sky movie at FHD resolution are presented.

First, the impact of limiting the number of MBs in flight is analyzed. Figure 3.14 depicts the available MB-level parallelism for several limits on the number of MBs in flight. As can be expected, for lower limits, the height of the plateau is lower and the ramp up and ramp down are on average slightly shorter. More important is that for lower limits, the plateau becomes very flat. This translates to a high utilization rate of the available cores. Furthermore, the figure shows that the total decoding time is approximately inversely proportional to the limit on the number of MBs in flight, i.e., if the limit is decreased by a factor of 2, the decoding time increases approximately by a factor of 2 too. This is especially true for lower limits, where the core utilization is almost 100%. For higher limits, the core utilization decreases and the relation no longer holds.

Figure 3.15 depicts the number of frames in flight as a function of time when the number of MBs in flight is limited. Not surprisingly, because limiting the number of MBs in flight also limits the number of frames that can be in flight, the shape of the curves are similar to those of the available MB parallelism, although with a small periodic fluctuation.

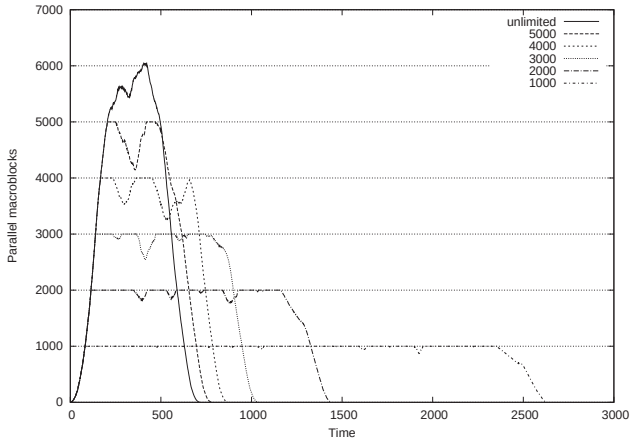


Figure 3.14: Available amount of MB-level parallelism, in each time slot, for FHD blue_sky and several limits of the number of MBs in flight.

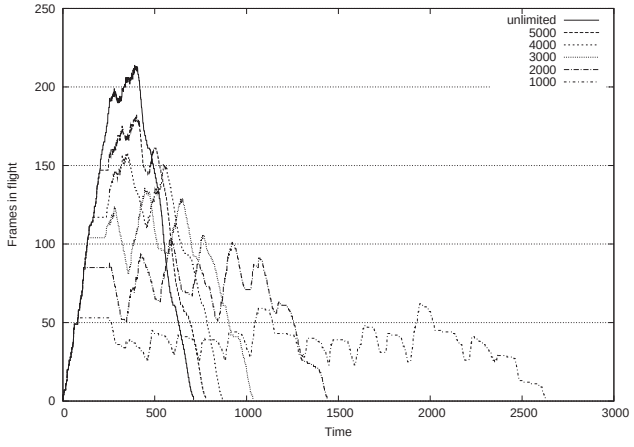


Figure 3.15: Number of frames in flight in each time slot, for FHD blue_sky and several limits of the number of MBs in flight.

Next, we analyze the impact of restricting the number of frames concurrently in flight. The MB-level parallelism curves are depicted in Figure 3.16 and show large fluctuations, possibly resulting in under-utilization of the available cores. The fluctuations are caused by the ramping effect illustrated in Figure 3.7 for one frame. It shows that at the start and end of a frame less parallelism is available. The same ramping effect occurs in this case. As a limited number of frames are in flight concurrently, and frames start decoding quickly one after another, they are approximately in the same phase. Thus, if all frames in flight are about halfway the frame, many parallel MBs are available. On the other hand, if all frames are close to the end only very little parallelism is available.

Figure 3.17 shows the actual frames in flight for several limits. The plateaus in the curves translate to full memory utilization.

From this experiment we conclude that for systems with a limited number of cores, dynamic scheduling is able to achieve near optimal performance by (almost) fully utilizing the available computational power. On the contrary, for systems where the memory size is the limitation, additional performance losses might occur because of temporal under-utilization of the available cores. In Section 3.6 we perform a case study, indicating that memory will most likely not be the limitation.

We have shown that using the Dynamic 3D-Wave strategy huge amounts of parallelism are available in H.264 decoding. Analysis of real movies has revealed that the average number of parallel MBs ranges from 3000 to 7000. This amount of MB parallelism might be larger than the number of cores available. Thus, we have evaluated the parallelism for limited resources and found that limiting the number of MBs in flight results in a equivalent larger decoding time, but a near optimal utilization of the cores.

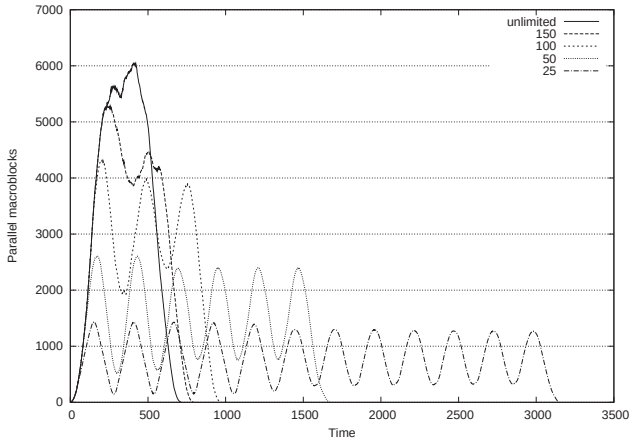


Figure 3.16: Available MB parallelism in FHD blue_sky for several limits of the number of frames in flight.

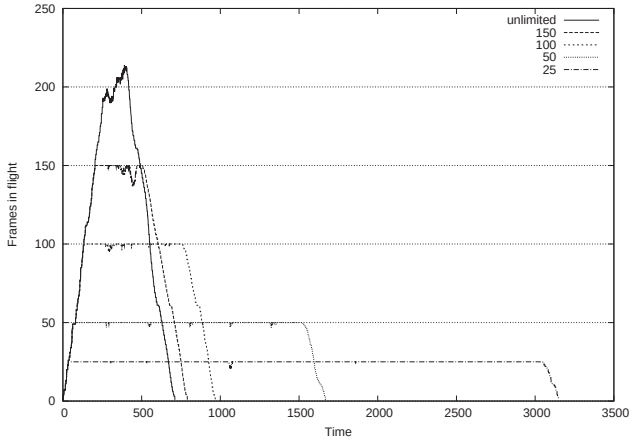


Figure 3.17: Number of actual frames in flight for FHD blue_sky, for several limits of the number of frames in flight.

3.6 Case Study: Mobile Video

So far we have mainly focused on high resolution and explored the potentially available MB-level parallelism. The 3D-Wave strategy discloses an enormous amount of parallelism, possibly more than what a high performance multicore in the near future could provide. Therefore, in this section we perform a case study to assess the practical value and possibilities of a highly parallelized H.264 decoding application.

For this case study we assume a mobile device such as the iPhone, but in the year 2015. We take a resolution of 480×320 just as the screen of the current iPhone. It is reasonable to expect that the size of mobile devices will not significantly grow, and the limited accuracy of the human eye implies that the resolution does not have to increase.

Multicores are power efficient, and thus we expect mobile devices to adopt them quickly, and therefore we project a 100-core chip in 2015. Two examples of state-of-the art embedded multicores are the Tilera Tile64 [157] and the Clearspeed CSX600 [46], containing 64 and 96 cores respectively. Both are good examples of what is already possible today and the eagerness of the embedded market to adopt the multicore paradigm. A doubling of cores every three years is expected [148]. Even if this growth would be less still the assumption of 100 cores in 2015 seems reasonable.

At time of performing this analysis (2007), the iPhone 2G was available with 8GB of memory. Using Moore's Law for memory (doubling every two years), in 2015 this mobile device would contain 128 GB. It is unclear how much of the flash drive memory is used as main memory, but for this case study we assume it is only 1%. In 2015 that would mean 1.28GB and if only half of it is available for video decoding, then still almost 1400 frames of video would fit in main memory. Thus, it seems that memory size is not going to be a limitation.

We place a limit of 1 second on the latency of the decoding process. That is a reasonable time to wait for the movie to appear on the screen, and much longer would not be acceptable. The iPhone uses a frame rate of 30 frames/second, and thus limiting the frames in flight to 30, causing a latency of 1s. This can be explained as follows. Let us assume that the clock frequency is scaled down as much as possible to save power. When the decoding of frame x is started, frame $x - 30$ has just finished and is currently displayed. But, since the framerate is 30, frame x will be displayed 1 second later. Of course this is a rough method and does not consider variations in frame decoding time, but

it is a good estimate.

Putting all assumptions together, for the future device the Dynamic 3D-Wave strategy has a limit of 100 parallel MBs and up to 30 frames can be in flight concurrently. For this case study we use the four normally encoded movies with a length of 200 frames and a resolution of 480×320.

Figure 3.18 presents the available MB-level parallelism under these assumptions as well as for unlimited resources. The figure shows that even for this small resolution, all 100 cores are utilized nearly all the time. The curves for unrestricted resources show that there is much more parallelism available than the hardware of this future mobile device has to offer. This provides opportunities for scheduling algorithms to reduce communication overhead.

Figure 3.19 depicts the number of frames in flight for this case study. The average is approximately 12 frames with peaks of up to 19. From the graphs we can also conclude that the decoding latency is approximately 0.5 second, since around 15 frames are in flight.

From this case study we draw the following conclusions:

- Even a low resolution movie exhibits sufficient parallelism to fully utilize 100 cores efficiently.
- For mobile devices, the memory size will most likely not be a limitation.
- For mobile devices with a 100-core chip, the frame latency will be short (around 0.5s).

3.7 Experiences with Parallel Implementations

Motivated by the results of the previous sections our team has implemented several parallel H.264 decoders. Azevedo [28] has implemented the Dynamic 3D-Wave on a TriMedia multicore platform during an internship at NXP. Alvarez [20] has implemented the 2D-Wave on a multiprocessor system. Finally, Chi [43, 44] has implemented the 2D-Wave on the Cell processor. The last two works are based on an unpublished initial parallelization by Paradis and Alvarez.

Although the author was not the prime developer, the experiences are very interesting and for completeness should be presented briefly in this chapter. The full experimental results and analysis can be found in [28, 20, 43, 44].

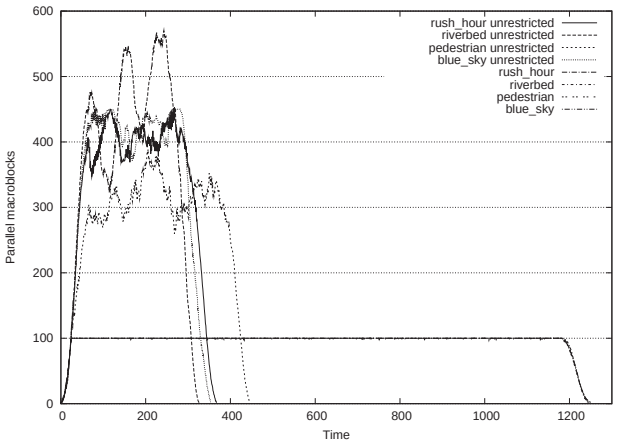


Figure 3.18: Number of parallel MBs for the mobile video case study. Also depicted is the available MB-level parallelism for the same resolution but with unrestricted resources.

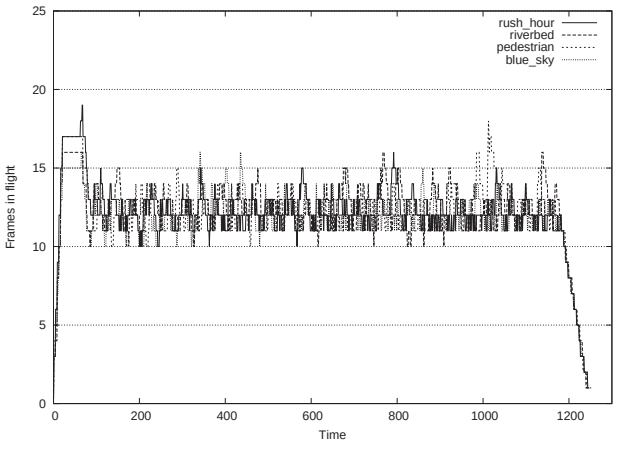


Figure 3.19: Number of frames in flight for the mobile video case study.

The scalability analysis of Section 3.5 is theoretical in nature and reveals the parallelism exhibited by the application. Exploiting this parallelism also involves the hardware architecture, the runtime system, and the operating system, whose characteristics affect the actual obtainable scalability. Therefore, these implementations are briefly reviewed in the following sections. We discuss what obstacles were encountered in achieving scalability. Also results are shown and the differences with the theoretical study are analyzed.

3.7.1 3D-Wave on a TriMedia-based Multicore

For this 3D-Wave implementation [28] the NXP H.264 decoder is used, which is highly optimized for NXPs TriMedia processor. As a starting point, Hoogerbrugge [79] already implemented the 2D-Wave, on which this implementation is based. The cycle accurate NXP proprietary simulator was used to evaluate the 3D-Wave on a TriMedia-based multicore system. It is capable of simulating up to 64 TM3270 cores [161], each with a 64KB private L1 cache, and a shared L2 cache.

The programming model follows the task pool model. The task pool library implements submission and requests of tasks to/from the task pool, synchronization mechanisms, and the task pool itself. The system contains one control processor and several worker processors. The latter request tasks from the task pool and execute it. The task pool library is very efficient as the time to request a task is only 2% of the MB decoding time.

The implementation of the 3D-Wave required quite some modifications to the code. In order to decode multiple frames concurrently some additional software buffers were required. Further, the task pool library was extended with a subscription mechanism (kick-off lists). The dependencies through the motion vectors are not known a priori. Thus, the MB decoding is split in two parts. In the first part the motion vectors are predicted. In case the reference area has been decoded, the core continues with the second part, which contains the rest of the MB decoding process. In case an unresolved dependency is met, the current MB subscribes itself to the MB it depends on. The latter starts the execution of the former MB once it has been decoded.

Due to the additional administrative overhead, the 3D-Wave is 8% slower than the 2D-Wave on a single core. The scalability of the 3D-Wave, however, is much better than that of the 2D-Wave as shown in Figure 3.20. Whereas the scalability of the 2D-Wave saturates at around 15, the 3D-Wave provides a scalability of almost 55. Note that the scalability of a case is always relative to

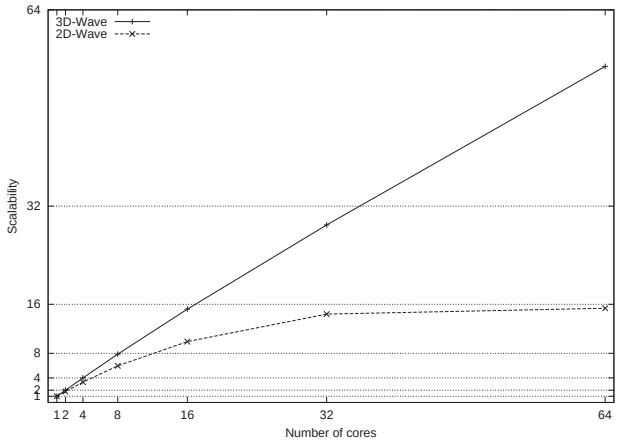


Figure 3.20: Scalability of the 3D-Wave and the 2D-Wave implementations on the TriMedia-based multicore system. A 25-frame FHD rush_hour sequence was used. Longer sequences further improve the scalability for the 3D-Wave.

the execution of that case on one core.

The 3D-Wave implementation achieves a scalability efficiency of 85%. There are two reasons why the efficiency is not higher. First, when using a large number of cores, also a large number of frames are processed simultaneous. As a consequence, cache trashing occurs and mainly the memory accesses of the motion compensation kernel suffer from cache misses. Second, at the start and the end of the sequence there is little parallelism. As the sequence used is only 25 frames long, the influence of this ramp up and ramp down is relatively large. For longer sequences the speedup would be better. This is confirmed by simulations of a 100 frame SD sequence. A similar simulation for FHD resolution could not be performed due to limited memory available in the simulator.

The scalability of the 2D-Wave is rather low compared to the available parallelism as shown in Figure 3.7. Theoretically a speedup of 32 should be possible using at least 60 cores. The theoretical analysis, however, assumes that the decoding of each MB takes equal time. In reality the decoding time heavily depends on the input data and ranges from approximately 8 to 50 μs (measured with FFmpeg on a 1.6 GHz Itanium2 processor). Alvarez [19] analyzed the effect of the varying decoding time on the scalability using the benchmark movies. On average the maximum obtainable speedup dropped from 32 to 22. It was not further investigated why the scalability of the 2D-Wave saturates at

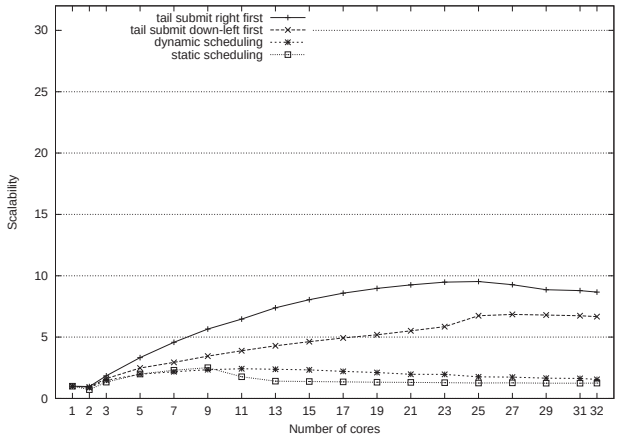


Figure 3.21: Scalability of the 2D-Wave implementation on a 128-core SGI Altix system.

around 15. A plausible explanation is that it is caused by synchronization and communication overhead.

3.7.2 2D-Wave on a Multiprocessor System

This implementation of the parallel H.264 decoder [20] was implemented and tested on an SGI Altix machine with 128 cores and a cc-NUMA (cache coherent Non-Uniform Memory Access) memory design. Blades with two dual-core Intel Itanium-II processors (1.6 GHz) and 8 GB of RAM were used. The blades are connected through a high performance interconnect (NUMAlink-4) with a 6.4 GB/s peak bandwidth.

The FFmpeg H.264 decoder was parallelized using a similar task model as described in the previous section. The worker threads are responsible for updating the dependency table. This allowed the implementation of several scheduling strategies such as tail-submit. POSIX threads were used as well as semaphores and mutexes for synchronization.

The scalability of the 2D-Wave implementation was measured with all four benchmark movies (see Section 3.2). In this case sequences of 100 frames were used. Figure 3.21 depicts the scalability obtained using up to 32 cores and for several scheduling policies. Both the static and dynamic scheduling scale very poorly. The static scheduling does not use the task pool, but uses a fixed decoding order instead, i.e., a zigzag scan order following the 2D-

Wave pattern. Due to the variable decoding time, this scheduling strategy fails to exploit the available parallelism. The dynamic scheduling implementation does use the task pool, but the overhead for submitting to and retrieving tasks is a bottleneck of the system. Using a tail-submit scheduling, the task pool is less frequently accessed and thus the scalability improves. The right-first policy (next MB is the right neighboring MB) has the best performance as it exploits more data locality than the down-left-first policy (next MB is the down-left neighboring MB).

The limited scalability of this implementation is mainly caused by synchronization. Using one core, to get a MB from the task pool takes $6.1\mu s$, while using 8 cores this time has increased to $132\mu s$. Using more cores the time increases further, but then the limited amount of available parallelism starts playing a role as well. The execution time of the get_mb function, which retrieves a MB from the task pool, is dominated by the sem_wait (semaphore wait), which in turn spends most of its time in the OS intervention. Apparently this OS intervention increases with the number of participating cores.

From this study we conclude that synchronization overhead radically affects the scalability if it is not implemented efficiently. Standard libraries such as the p-thread library do not seem to provide this efficiency.

The 3D-Wave was not implemented on this multiprocessor system. The amount of available parallelism was not the limitation, and thus there is no need to use the 3D-Wave strategy. The conclusions from this study also apply to the 3D-Wave though, which is the reason why this section was included.

3.7.3 2D-Wave on the Cell Processor

Chi [43, 44], an MSc student from Delft University of Technology, developed a very scalable implementation of the 2D-Wave on the Cell Processor. He observed that the mailbox API offered by the standard libraries has much larger latencies than the actual hardware system. Therefore, instead he used a mailbox API that accesses the problem state of the SPEs directly.

Another major improvement he made concerns the data communication. The initial implementation contained many DMAs, each transferring a part of the information required to decode the macroblock. Chi reduced the number of DMAs to a minimum by creating a new data structure containing all information in one place.

Figure 3.22 shows that this optimized Task Pool (TP) based 2D-Wave implementation achieves a scalability of 10.2 on 16 SPEs, which is approximately

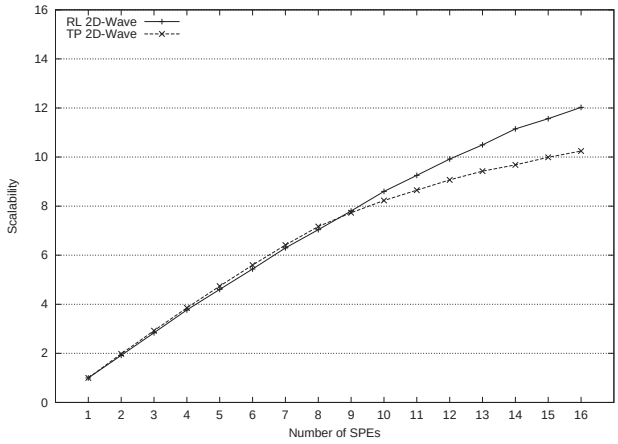


Figure 3.22: Scalability of the 2D-Wave implementations on the Cell processor. At 16 SPEs the difference in scalability is 20%. The Ring-Line (RL) based implementation, however, has twice the performance of the Task Pool (TP) based implementation.

the same as the 2D-Wave implementation on the TriMedia with equal number of cores. In this implementation, scalability is mainly limited by contention on the off-chip bus to main memory.

The TP 2D-Wave implementation does not yet fully utilize the capabilities of the Cell processor. The SPEs are connected to each other in a ring, which is exploited by the Ring-Line (RL) implementation of the 2D-Wave strategy. It uses a distributed and static scheduling instead of a dynamic task pool. Each line of macroblocks is assigned to one SPE, such that adjacent lines are processed on adjacent SPEs. Synchronization among SPEs is done by sending the output of one SPE to its neighbor. Besides removing the bottleneck of centralized control, also much more locality is exploited resulting in much less bandwidth requirements.

Figure 3.22 also shows the scalability of the ring-line implementation. It scales better than the 2D-Wave based on the task pool from 10 SPEs on, and achieves a scalability of 12.0 using 16 SPEs. The theoretical scalability limit of this implementation is lower than that of the task pool implementation due to implementation issues not discussed here. The obtained scalability is close to this theoretical limit, and therefore much higher scalability is not possible.

The actual performance of the ring-line implementation, however, is much higher than the task pool version. The latter achieves a throughput of 76 frames

per second while the former implementation achieves 149 frames per second on FHD resolution. There are two reasons for this difference. First, the lower bandwidth requirements of the ring-line 2D-Wave implementation allow it to scale to a much higher frame throughput. Second, in the ring-line implementation, the communication and computation are performed concurrently. In the task pool 2D-Wave this could be implemented, for example by applying tail-submit. It is not so easy, however, to implement and it would stress the control processor even more.

Those two implementations also show that the 2D-Wave is appropriate for small scale systems and can achieve high performance. Scalability is limited, though, to about 16 to 20 cores. For larger scale systems the 3D-Wave will have to be used. The latter is not yet implemented on the Cell processor. The 2D-Wave on Cell has shown that low latency synchronization is of vital importance to obtain scalability. Furthermore, it has shown that a smart scheduling of the data movements can significantly increase performance. Finally, we conclude from this study that a centralized control processor easily becomes a bottleneck for fine-grained thread-level parallelism, as used in our implementation of H.264 decoding.

3.8 Conclusions

In this chapter we have investigated if applications exhibit sufficient parallelism to exploit the large number of cores expected in future multicores. This is important for performance as well as for power efficiency. As a case study, we have analyzed the parallel scalability of an H.264 decoder, currently the best video coding standard.

First, we discussed possible parallelization strategies. Most promising is MB-level parallelism, which can be exploited inside a frame in a diagonal wave-front manner. As MBs in different frames are only dependent through motion vectors, intra-frame and inter-frame MB-level parallelism can be combined. Such a parallelization approach is referred to as the 3D-Wave. It has been applied to encoding before, but not to decoding as we have done. In this chapter we have observed that motion vectors typically have a limited range. Therefore, by applying dynamic scheduling a much higher scalability can be obtained. This novel parallelization strategy is called the Dynamic 3D-Wave.

Using this new parallelization strategy we have performed a limit study of the amount of MB-level parallelism available in H.264. We modified FFmpeg and analyzed real movies. The results have revealed that motion vectors are typi-

cally small on average. As a result a large amount of parallelism is available. For FHD resolution the total number of parallel MBs ranges from 4000 to 7000.

The amount of MB parallelism disclosed by the Dynamic 3D-Wave could be larger than the number of cores available. Moreover, the number of frames in flight to achieve this parallelism might require more memory than available. Thus, we have evaluated the parallelism for limited resources and found that limiting the number of MBs in flight results in a larger decoding time, but a near optimal utilization of cores. Limiting the number of frames in flight, on the other hand, results in large fluctuations of the available parallelism, likely under-utilization of the available cores and thus performance loss.

We have also performed a case study to assess the practical value and possibilities of the highly parallel H.264 decoder. In general, the results show that our strategy provides sufficient parallelism to efficiently exploit the capabilities of future many-cores, even in future mobile devices.

Finally, we have discussed a few implementations of the 2D-Wave and 3D-Wave strategy. The 3D-Wave was successfully implemented on a TriMedia-based multicore platform. With 64 cores a scalability of 55 was achieved, whereas the scalability of the 2D-Wave saturates around 15. On a single core, the 3D-Wave is 8% slower than the 2D-Wave, due to management overhead. From 4 cores on, however, the 3D-Wave outperforms the 2D-Wave. Moreover, the implementation proved that in order to scale to a large number of cores, the 3D-Wave is required. Experiences with the 2D-Wave on an SMT machine and on the Cell processor have shown that low latency synchronization is essential for fine-grained parallelism as exploited in the 3D-Wave as well as in the 2D-Wave.

Chapter 4

The SARC Media Accelerator

As has been shown in Chapter 2, the impact of the power wall is increasing over time. That means that the power budget, and neither technology nor the transistor budget, will be the main limitation to obtain higher performance. Clock frequencies will have to be throttled down and eventually it will not be possible to use all available cores concurrently. In such a situation performance increase can only be obtained by improving the power efficiency of the system. This will lead to specialization of cores (referred to as domain specific accelerators) in such way that they can run a specific domain of applications in short time using relatively less area, and thus being power efficient.

In this chapter the specialization of an existing core for H.264 [120] video decoding is presented. As discussed in Chapter 3, H.264 is the best video coding standard in terms of compression ratio and picture quality currently available and is a good representative of future media workloads. As the baseline core for specialization we take the Cell SPE. The Cell broadband engine has eight SPEs that operate as multimedia accelerators for the PowerPC (PPE) core. The Cell processor has been used in several Sony products for professional video production [144, 145]. IBM has shown the utilization of the Cell processor for video surveillance systems [103]. The SPE architecture has also been used in the SpursEngine [76] from Toshiba. It is a media oriented coprocessor, designed for video processing in consumer electronics and set-top boxes. Therefore, the SPE architecture is an excellent starting point for specialization.

As the work presented in this thesis is part of the SARC project, the specialized SPE as proposed in this chapter is referred to as the SARC Media Accelerator. It is part of the SARC architecture, which is a futuristic many-core platform

containing several domain specific accelerators, e.g., bioinformatics accelerators, vector accelerators, and media accelerators.

This chapter is organized as follows. First, related work is reviewed in Section 4.1. Section 4.2 provides a brief overview of the SPE architecture. In Section 4.3 the experimental setup is described. The architectural enhancements of and modifications to the SPE are described in Section 4.4. The resulting media accelerator is evaluated in Section 4.5. Finally, Section 4.6 concludes this chapter.

4.1 Related Work

There are many architectures that aim at speeding up media applications. First of all, there are General Purpose Processors (GPPs) with (multi)media SIMD extensions. The Intel IA-32 architecture was extended with MMX [125] and later with SSE [131]. AMD has its 3DNow! [52] technology, IBM's PowerPC was extended with Altivec/VMX [53], VIS [159] was used in Sun's SPARC architecture, the MIPS architecture was extended with MDMX [71], and MVI [71] was added to the Alpha processor. A very small media extension (16 instructions only) was the MAX-2 [97] used in HP's PA-RISC core. GPPs provide high flexibility and programmability, but are power inefficient and cannot always meet real-time constraints [113].

The second class of architectures is the multimedia processor that aims at accelerating a broad range of multimedia applications. Especially for embedded devices such a broad range of applications has to be supported. Most of the multimedia processors available in the market are actually SoCs, such as the Intel CE 2110 [10] and the Texas Instruments DaVinci Digital Media Processors [9]. Both are capable of H.264 video coding, but employ a hardware codec for it. Such platforms lack the flexibility of enabling new codecs or codec profiles.

Also academia has proposed several multimedia architectures. Among those are stream/vector processors such as Imagine [90], VIRAM [93], and ALP [134]. Prerequisites to using these architectures efficiently are certain application characteristics such as little data reuse (pixels are read once from memory and are not revisited) and high data parallelism (the same set of operations independently compute all pixels in the output image) [90]. Many media applications exhibit these characteristics. Modern video codecs such as H.264, however, are such complex that they do not have those characteristics. Motion compensation causes data to be accessed multiple times in an irregular

fashion. DLP is limited by the size of the sub-block that can be as small as 4×4 . Because macroblocks can be partitioned in sub-blocks in various ways and because each sub-block can be coded with different parameters, there is no regular loop that architectures such as the VIRAM require to exploit DLP. Moreover, due to many data dependencies parallelizing is not trivial. Therefore, we expect that stream processors are less suitable for a modern codec such as H.264.

The CSI [42], MOM [48], and MediaBreeze [152] architectures optimize SIMD extensions by reducing the overhead of data accesses using techniques as hardware address generation, hardware looping, data sectioning, alignment, reorganization, packing/unpacking, etc. The MOM architecture is matrix oriented and supports efficient memory access for small matrices as well as some matrix operations like matrix transpose. All of these architectures provide two or more dimensions of parallelism, yet recognize the need to exploit fine-grained parallelism. Some of these techniques could be effective for modern video coding. Issues like alignment, reorganization, and packing/unpacking are also addressed in the SARC Media Accelerator.

The third class of architectures is the Application-Specific Instruction set Processor (ASIP). Such architecture is optimized for one or few applications in the multimedia domain. ASIPs provide good power efficiency while maintaining flexibility and programmability and can therefore support a wide variety of standards. The SARC Media Accelerator is an ASIP based on the Cell SPE architecture. The TriMedia is a VLIW ASIP for audio and video processing. The latest version (TM3270) has been specialized for H.264 encoding and decoding [161]. Our architecture is specialized for decoding only but takes specialization a step further. Tensilica's Diamond 388VDO Video Engine has two cores and a DMA unit [15]. One core is a SIMD architecture while the other is a stream architecture. The latter is used for entropy coding and assists the SIMD core in motion prediction. Further architectural details of this processor could not be found.

An early ASIP for video coding is found in [150] and contains an instruction for motion estimation, some instructions for packing and interleaving, a specialized datapath, and IO for audio/video. Another early ASIP for video coding is found in [98] and has an interesting system architecture. A RISC core is used as control processor, a SIMD processor is used for the main decoding, and a sequence processor is used to perform entropy decoding. Moreover, the system has a local memory with a DMA unit. The cores itself though are very simple and are not specialized. In [91] an ASIP was proposed to enhance pixel

processing. It uses SIMD vectors with 8 bits for the luma component and two times 4 bits for the chroma components. This kind of SIMD can exploit only a limited amount of DLP. Furthermore, the H.264 colorspace uses mainly a 4:2:0 subsampling scheme. That means that the number of chroma samples is reduced instead of the number of bits per chroma component. Thus this architecture cannot be used for modern video codecs. Two other ASIPs are found in [126, 50]. Both processors target motion estimation only.

Not a complete ASIP, but rather an ISA specialization for video coding was presented in [31] in 1999. The authors suggested some simple instructions for the upcoming MPEG4 standard. Many of the proposed instructions, e.g., arbitrary permute, MAC, round, and average, are being used in nowadays architectures. In [138] an ISA specialization is presented for H.263. The instructions can be used for H.264 as well but are not as powerful as the ones we propose.

Finally, the last class of architectures is the ASIC. Some combinations of ASICs with GPPs for multimedia in general are mentioned above. There are also many examples of GPPs extended with an ASIC for a very specific target [118, 124, 151, 63]. ASICs are the most power-efficient solution and have found their way in many embedded systems. In systems where a large number of standards have to be supported, however, they might not be the most cost efficient solution. Furthermore, when programmability is required, for example to support future standards or profiles, ASICs are not an option.

So far we discussed architectures at a high level. There is also some work that is more specifically related to particular instructions that we propose. For clarity of exposition reasons we discuss these works after presenting the application specific instructions.

4.2 The SPE Architecture

The baseline architecture for this research is the Cell SPE (Synergistic Processing Element) [59]. At the time this research was performed (2008), the Cell processor was one of the most advanced chip multiprocessors available. It consists of one PPE and eight SPEs (Synergystic Processing Elements), as depicted in Figure 4.1. The PPE is a general-purpose PowerPC that runs the operating system and controls the SPEs. The SPEs function as accelerators; they operate autonomously but are depending on the PPE for receiving tasks to execute.

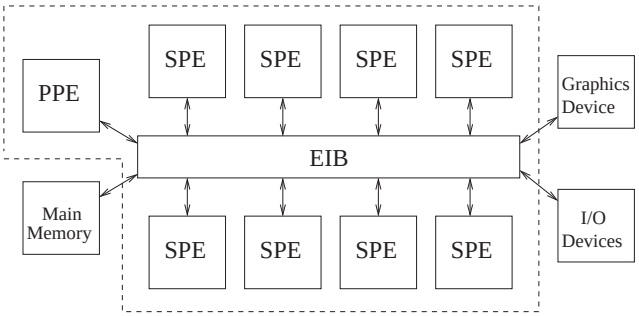


Figure 4.1: Overview of the Cell architecture.

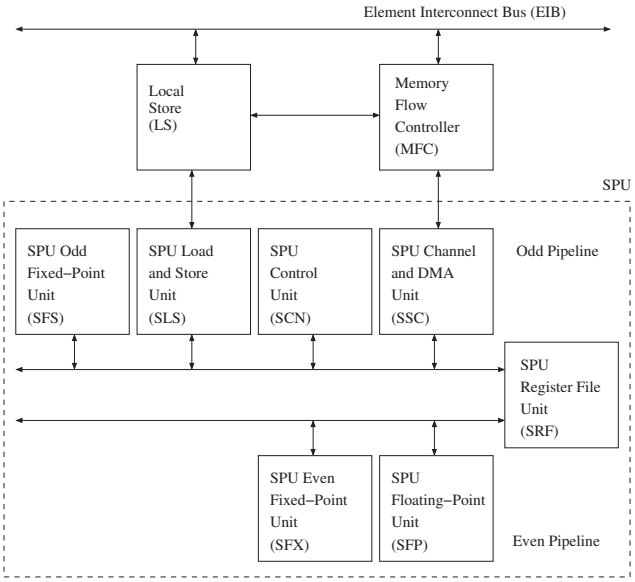


Figure 4.2: Overview of the SPE architecture (based on [80]).

The SPEs consist of a Synergistic Processing Unit (SPU), a Local Store (LS), and a Memory Flow Controller (MFC) as depicted in Figure 4.2. The SPU has direct access to the LS, but global memory can only be accessed through the MFC by DMA commands. As the MFC is autonomous, a double buffering strategy can be used to hide the latency of global memory access. While the SPU is processing a task, the MFC is loading the data needed for the next task into the LS.

The SPU has 128 registers, each of which is 128 bits wide. All data transfers between the SPU and the LS are 128-bit wide. Also the LS accesses are

quadword aligned. The ISA of the SPU is completely SIMD and the 128-bit vectors can be treated as one quadword (128-bit), two double words (64-bit), four words (32-bit), eight halfwords (16-bit), or 16 bytes.

There is no scalar datapath in the SPU, but scalar operations are performed using the SIMD registers and pipelines. For each data type, a preferred slot is defined in which the scalar value is maintained. Most of the issues of scalar computations are handled by the compiler.

The SPU has six functional units, each assigned to either the even or odd pipeline. The SPU can issue two instructions concurrently if they are located in the same doubleword and if they execute in a different pipeline. Instructions within the same pipeline retire in order.

The Cell SPE was chosen as a starting point for specialization because of its properties. Specific, the architectural features discussed next were main reasons to choose the SPE.

The fundamental idea behind the *DMA*-style of accessing global memory is the shopping list model. In this model, before processing a task all the required data is brought in the local store through DMA. For video decoding this model suits well as almost all required data is known a priori. The only exception is in the motion compensation where the motion vector can point to an arbitrary location in a previously decoded frame.

The *register file* of the SPE is rather large as it has 128 registers 128-bit each. As a result, the data processed by a kernel fits entirely in the register file in most cases. Generating code that does so, however, is not straightforward. Basically all pointers have to be replaced by arrays that can be declared `register`. Often kernels consist of multiple functions. To maintain data in the register file and have all the functions operate on these registers, function calls have to be avoided, i.e., the functions have to be replaced by macros. For large kernels such as motion compensation this is not straightforward. Moreover, the effect of this on the code size will not be insignificant and might constitute a problem.

The *Local Store* (LS) of the SPE is 256KB large and contains both data and instructions. The entire H.264 decoding application does not fit in this LS, but it does not have to. The media accelerator is meant to function in cooperation with a general-purpose core. The latter would run the main application and spawns threads to one or more accelerators. Using the 2D-Wave or 3D-Wave parallelization strategy (see Chapter 3), the accelerators are responsible for decoding single macroblocks. Thus, only the code that decodes one macroblock has to fit in the LS. The code from FFmpeg compiled for the SPE without optimizations is 256KB large. Compilation with the -O3 optimizations, however,

results in a code of 155KB, while using -Os results in 118KB. Note that also 24KB for data is required and also some additional memory for intermediate values, a stack, etc. Thus the LS size is sufficiently large but leaves little room to implement optimizations as mentioned in the previous paragraph.

The SPU is completely SIMD with a *vector width* of 128 bits and has no scalar data path. Scalar operations are performed using the SIMD path and thus a scalar path is not necessary. The width of the SIMD vectors is well-suited for H.264. Although the pixel components are encoded in 8 bits, all computation is performed using 16-bit values in order not to lose precision. That means that in one SIMD vector eight elements can be stored. Since, the kernels operate mostly on 4×4 and 8×8 sub-blocks, they would not benefit very much from longer SIMD vectors. In the deblocking filter the functions for the luma components could use a larger vector size. In the motion compensation only the functions for the Luma 16×16 , 16×8, and 8×16 mode could benefit from this. Using a longer vector size, however, would increase the overhead for all 8×8 based functions.

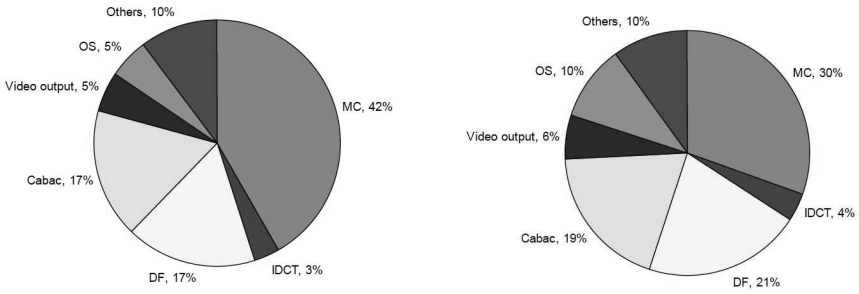
4.3 Experimental Setup

In this section we describe the experimental setup consisting of the benchmarks, the compiler, and the simulator. The target baseline architecture, the Cell SPE, was reviewed in the previous section.

4.3.1 Benchmarks

As benchmarks we used the kernels of an H.264 video decoding application. H.264. One of the best publicly available H.264 decoders is FFmpeg [58]. Colleagues of the author ported the kernels to the Cell SPE and vectorized them using the SPE SIMD instructions [29]. We used the following H.264 kernels for the analysis in this work: Inverse Discrete Cosine Transform (IDCT), Deblocking Filter (DF), and luma interpolation (Luma). The latter is one of the two elements of the motion estimation kernel. The other is the chroma interpolation kernel, which is very similar to the Luma kernel and was therefore omitted. There is also an entropy decoding kernel in H.264. We did not use this kernel in this research as it is highly bit serial and requires a different type of accelerator.

Figure 4.3 shows the execution time breakdown of the entire FFmpeg application. This analysis was performed on a PowerPC with Mac-Os-X and run-



(a) Scalar (b) Altivec

Figure 4.3: Execution time breakdown of the FFmpeg H.264 decoder for the scalar and Altivec implementation.

ning at 1.6 GHz. Both the scalar and the Altivec implementations are depicted. CABAC is one of the entropy decoding algorithms of the H.264 standard. Suppose a General Purpose Processor (GPP) performs the operating system (OS) calls, the video output, and the 'others' parts. The entropy decoding can best be performed using a special CABAC accelerator while the other kernels run on the media accelerator. Using this approach the media accelerator has the most time-consuming workload: assuming Altivec it has 55% of the workload compared to 26% for the GPP and 19% for the CABAC accelerator. Therefore, any speedup achieved in the media core, up to a factor of approximately two, directly translates to a speedup of the total application. A detailed analysis of the kernels is presented in Section 4.5.

4.3.2 Compiler

We used spu-gcc version 4.1.1. Most of the architectural enhancements we implemented were made available to the programmer by intrinsics. In some cases implementing the intrinsic would require complex modifications to the compiler. Those new instructions are only available by using inline assembly. We modified the gcc toolchain to support the new intrinsics and instructions. The procedure to modify the gcc compiler is described in [111].

The modifications assured that the compiler can handle the new instructions, but not necessarily in an optimized way. Optimizing the compiler for the new instructions is beyond the scope of this thesis.

4.3.3 Simulator

Analysis of the kernels on the Cell SPE could be done using the real processor. Analyzing architectural enhancements, however, requires an experimental environment that can be modified. Therefore we used CellSim [7]; a simulator of the Cell processor developed in the Unisim environment.

Profiling in CellSim

First, we implemented a profiling tool for CellSim as such as tool was not available. The profiling tool allows to gather many statistics (such as cycles, instructions, IPC, etc.) either from the entire application or from a specific piece of selected code. The profiler is controllable from the native code (the code running on the simulated processor). Furthermore, the profiler allows generating a trace output for detailed analysis of the execution of the kernels.

Profiling the code, however, can influence its execution, especially if the size of the profiled code is small. Each call to the profiler inserts one line of volatile inline assembly. This ensures that only code that is between the start and stop command of the profiler is measured. Because instructions are not allowed to cross this *volatile* boundary, however, there is less opportunity for the compiler to optimally schedule the instructions.

Section 4.5 presents the profiling analysis of the kernels that shows the amount of time the different execution phases take. The execution times reported in those particular analyses are off by approximately 25%. The purpose, however, is to show the relative size of the phases rather than the actual cycles spent on them. Thus, the influence of the profiling itself is not problematic.

When reporting the total execution time of the kernels, the influence of the profiling can be neglected. For this analysis only a start and stop command have to be inserted at the beginning and end of the kernel respectively. No volatile boundaries are included inside the kernel and thus all compiler optimizations are unaltered.

Configuration of CellSim

CellSim has many configuration parameters, most of which have a default value that matches the actual Cell processor. The execution latencies of all instructions classes, however, were all set by default to 6 cycles. We adjusted these to match the ones reported in [80] as described in Table 4.1. The table

Table 4.1: SPU instruction latencies.

Instruction class	Latency (cycles)	Description	CellSim pipe
SP	6	Single-precision floating-point	FP6
FI	7	Floating point integer	FP7
DP	13	Double-precision floating point	FPD
FX	2	Simple fixed point	FX2
WS	4	Word rotate and shift	FX3
BO	4	Byte operations	FXB
SH	4	Shuffle	SHUF
LS	6	Loads and stores	LSP

mentions both the name of the instruction class as used in the IBM documentation as well as the pipeline name as used in CellSim and in spu-gcc. Each instruction class maps to one of the units depicted in Figure 4.2 (see [80] for more details). The execution latency of an instruction class is the time it takes the corresponding execution unit to produce the result. More precise, it is the number of clock cycles between the time the operands enter the execution unit and the time the results is available (by forwarding) as input for another computation. Thus, the instruction latency does not include instruction fetch, decode, dependency check, issue, routing, register file access, etc. For a detailed description of the SPU pipeline, the reader is referred to [59].

The table does not include all latencies. The latency of channel instructions are modeled in a different way in CellSim and were not adjusted. The latency of branches, i.e., the branch miss penalty, was not modeled in CellSim. M. Briejer implemented this while implementing the branch hint instructions [33].

In addition, we adjusted the fetch parameters. The fetch width was set to one line of 32 instructions. The fetch buffer size was set to 78, which corresponds to 2.5 lines of 32 instructions. Finally, we adjusted the instruction fetch rate of CellSim. The real processor can fetch two instructions per cycle, but can execute them simultaneously only if one executes in the odd pipeline and the other in the even pipeline. CellSim does not distinguish between the odd and even pipeline and therefore initially fetched too many instructions per cycle on average. We determined empirically that an instruction fetch rate of 1.4 in CellSim corresponds closely with the behavior of the real processor.

Table 4.2: Comparison of execution times and instruction count in CellSim and SystemSim.

	Cycles			Instructions		
Kernel	SystemSim	CellSim	Error	SystemSim	CellSim	Error
IDCT8	917	896	-2.3%	1099	1103	0.4%
IDCT4	2571	2524	-1.9%	1853	1858	0.3%
DF	121465	122122	0.5%	101111	101480	0.4%
Luma	2534	2598	2.5%	2557	2572	0.6%

To improve simulation speed some adjustments were made to the PPE configuration parameters. The issue width was set to 1000 while the L1 cache latency was set to zero cycles. Both measures increase the IPC of the PPE, which is required to spawn the SPE threads, and thus speedup simulation time. These modifications have no effect on the simulation results, however, as the kernels run on the SPE.

Validation of CellSim

To validate the configuration of CellSim, several tests were performed. The execution times measured in CellSim were checked by comparing them to the IBM full system simulator SystemSim. We did not use the real processor as SystemSim provides more statistics and is accurate as well. These additional statistics allowed us to check the correctness of CellSim on more aspects than just execution time.

Table 4.2 shows the execution times and the dynamic instruction count of all the kernels using both SystemSim and CellSim. It shows that the difference in execution time is at most 2.5%. The number of executed instructions is slightly different. As the profiling was started and stopped at exactly the same point in the C code, we expect this difference in instruction count to be caused by the different implementations of the profiling tool.

The results presented here are measurements of entire kernel execution times. As discussed on page 81, when performing a detailed profiling analysis of the kernel phases, the profiling itself affects the execution time, resulting in a larger error.

Altogether, from these results we conclude that CellSim is sufficiently accurate to evaluate the architectural enhancements described in this chapter. Compar-

ison of other statistics generated by the two simulators also showed that CellSim is well suitable for its purpose. Off course, when evaluating architectural enhancements we always compare between results that are all obtained with CellSim.

4.4 Enhancements to the SPE Architecture

In this section we present the architectural enhancements and modifications to the SPE that create the media accelerator. The enhancements and modifications are based on the identified deficiencies in the execution of the H.264 kernels. These deficiencies are discussed in detail in Section 4.5, while discussing the results for the kernels.

To identify the deficiencies in the execution of the H.264 kernels, we performed a thorough analysis down to the assembly level. Among others, this analysis comprised the following. We compared the C code with the generated assembly code to determine which simple operations require many overhead instructions. We performed extensive profiling to determine the most time-consuming parts. Analyzing trace files allowed us to determine why certain parts were time consuming. We analyzed both C and assembly code to find opportunities for instruction collapsing.

The newly added instructions are presented in detail in this section. One of the details we show is the chosen opcode for each instruction. We do not know the structure in the opcodes and thus we chose available ones. We show them, to prove that the added instructions fit (or not) in the ISA.

The notation used in this thesis to describe the instruction set architecture follows the notation used in the IBM documentation with a few extensions. First, temporary values w are 64-bit wide. Second, temporary values p are 16-bit wide. Third, temporary values d are 8-bit wide. An extensive description of the notation is provided in [81], for example.

In the code depicted in this thesis we use the following notation for vector types. A `vector signed short` type is denoted as `vsint16_t` while a `vector unsigned char` type is denoted as `vuint8_t`. In general the integer types are denoted as `v[s/u]int[8/16/32]_t`.

This section is organized as follows. First, we present four categories of new instructions that we added to the ISA. Finally, we propose a modification to the architecture that allows unaligned memory accesses to the local store.

4.4.1 Accelerating Scalar Operations

Since the SPU has a SIMD-only architecture, scalar operations are performed by placing the scalar in the so-called preferred slot and using the SIMD FUs. For operations that involve only scalar variables this works well; the compiler places the scalar variable in a memory location congruent with the preferred slot even if this wastes memory. But if the operand is a vector element a lot of rearrangement overhead is required to move the desired element to the preferred slot. Rather than providing a full scalar datapath, which would increase the silicon area significantly, we propose adding a few scalar instructions.

Lds Instructions

The lds (LoaD Scalar) instructions load a scalar from a quadword unaligned address inside the local store and store it in the preferred slot of a register. The original SPU architecture allows only quadword aligned accesses with size of 128 bits. To load a scalar from a non quadword-aligned location, a lot of additional instructions are required.

Listing 4.1 shows an example of an array access. The left part of the figure shows the C code including the declaration of the used array and index variable. The middle of the figure shows the assembly code generated by the compiler for the original SPU architecture. The first instruction (`shli`) performs a shift left on the index variable `i`, stored in `$5` by 2 bits. That is, the index is multiplied by 4 and the result is the distance of the desired element to the start of the array in bytes. The second instruction (`ila`) loads the start address of the array into a register. The third instruction (`lqx`) loads the quadword (128 bits) that contains the desired element. It adds the index in bytes to the start address of the array, truncates it to a multiple of 16, and accesses the quadword at that address. At this point the desired element of the array is in register `$4` but not necessarily in the preferred slot. The fourth instruction (`a`) adds the index in bytes to the start address of the array. The four least significant bits of this result is used by the last instruction (`rotqby`) to rotate the loaded value such that the desired element is in the preferred slot.

Using the `ldsw` (LoaD Scalar Word) instruction the same array access can be done in two instructions, as shown on the right side of Listing 4.1. The first instruction (`ila`) stores the base address of the array in a register. Next, the `ldsw` instruction takes as input the index (`$5`), the start address of the array (`$6`), and the destination register (`$2`). It loads the corresponding quadword from the local store, then selects the correct word and puts it in the preferred

Listing 4.1: Example of how the lds instructions speed up code. Left is an array access in plain C code, in the middle is the original assembly code, while the right depicts the assembly code using the `ldsw` instruction.

```
1  int array[8];          shli   $2,$5,2      ila    $6,array
2  int i;                 ila    $6,array     ldsw   $2,$6,$5
3  int a=array[i];        lqx    $4,$2,$6
4                         a      $2,$2,$6
5                         rotqby $2,$4,$2
```

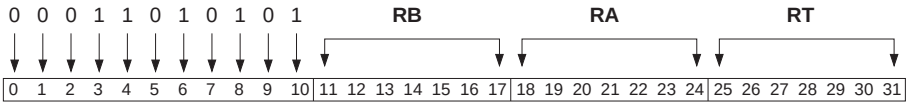
slot, and finally stores the result in the destination register.

The addressing mode is base plus scaled index, i.e., the effective address of `ldsw $2,$6,$5` is `$6 + 4x$5`. The ldsw instruction loads a word (four bytes) and thus its index is multiplied by four. For the other lds instructions the index is scaled in a similar way. The scalars that are loaded should be aligned according to their size. For example, words should be word aligned. The base address provided to the instruction does not have to be properly aligned, as the effective address is truncated according to the data size (see also instruction details below). Note that the base plus scaled index addressing mode is supported in several ISAs (e.g., x86), but not found to be very useful because the compiler can often eliminate the index variables. In this case, however, the datapath is completely SIMD and the index is used to select a specific element of the loaded quadword.

Loads and stores are handled by the SLS (SPU Load and Store) execution unit. The SLS has direct access to the local store and to the register file unit. Besides loads and stores, the SLS also handles DMA requests to the LS and load branch-target-buffer instructions. In order to implement the `ldsw` instruction a few modifications to the SLS unit are required. First, computing the target address is slightly different than for a normal load. Instead of adding the two operands, the index should be multiplied by four before adding it to the start address of the array. This multiplication can be done by a simple shift operation. Second, for each outstanding load (probably maintained in a buffer) two more select bits are required to store the SIMD slot in which the element will be located. These two bits are bits 2 and 3 of the target address. For normal loads, these two bits should be set to zero. Finally, when the quadword is received from the local store, the correct word should be shifted to the preferred slot. This can be done by inserting a 4-to-1 mux in the data path which is controlled by the select bits.

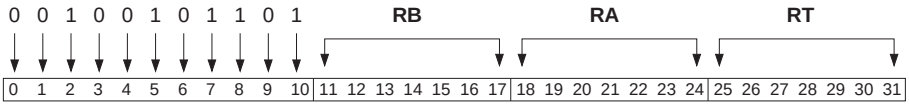
The lds instructions fit the RR format. The instruction details are depicted below. Although for the benchmarks used in this work only the `ldsw` instruction is required, for completeness we defined the lds instructions for halfwords and bytes as well. These element sizes are used frequently in media applications and thus it is reasonable to provide them. Doublewords are not considered in this work as neither long integers nor single precision floating point are used in the media applications targeted. Furthermore, the SPU ISA does not contain any long integer operations.

ldsw rt,ra,rb

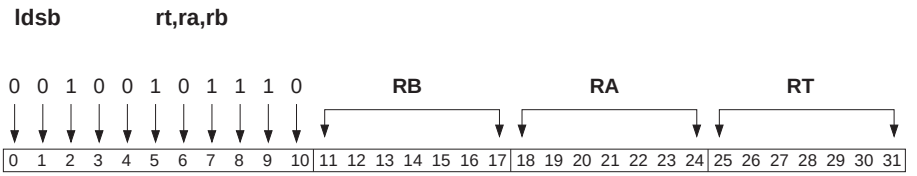


$$
\begin{array}{ll}
\text{LSA} & \leftarrow (RA^{0:3} + RB^{0:3} << 2) \text{ \& LSLR \& 0xFFFFFFF0} \\
b & \leftarrow (RA^{0:3} + RB^{0:3} << 2) \text{ \& 0x0000000C} \\
Q & \leftarrow \text{LocStor(LSA,16)} \\
RT^{0:3} & \leftarrow Q^{b::4}
\end{array}
$$

ldshw rt,ra,rb



$$
\begin{array}{ll}
\text{LSA} & \leftarrow (RA^{0:3} + RB^{0:3} << 1) \text{ \& LSLR \& 0xFFFFFFF0} \\
b & \leftarrow (RA^{0:3} + RB^{0:3} << 1) \text{ \& 0x0000000E} \\
Q & \leftarrow \text{LocStor(LSA,16)} \\
RT^{0:1} & \leftarrow Q^{b::2}
\end{array}
$$



$$
\begin{aligned}
\text{LSA} &\leftarrow (RA^{0:3} + RB^{0:3}) \text{ \& LSLR \& 0xFFFFFFF0} \\
b &\leftarrow (RA^{0:3} + RB^{0:3}) \text{ \& 0x0000000F} \\
Q &\leftarrow \text{LocStor(LSA,16)} \\
RT^0 &\leftarrow Q^b
\end{aligned}
$$

The lds instructions were made available to the programmer through intrinsics of the form `d = spu_ldsx(array,index)` where $x \in b, hw, w$, `array` is a pointer to a word, and `index` is a word. The compiler generates both the `ldsx` and the `ila` instruction required to load the desired element of the array.

The lds instructions were added to the simulator and the latency was set to seven cycles for the following reasons. A normal load or store operation takes six cycles, and we added one cycle to account for the extra mux in the datapath.

As2ve Instructions

As2ve (Add Scalar to Vector element) instructions add a scalar to a specific element of a vector. As the SPU architecture is completely SIMD, scalar operations often require a considerable amount of time. Listing 4.2 shows an example from the IDCT8 kernel where the constant value 32 has to be added to the first element of a vector of halfwords.

The middle of the figure depicts how the normal SPU ISA handles this code. The pointer `*block` is stored in `$4`. First, it is calculated (`ai`) by how much bytes the vector must be rotated to put the desired element in the preferred slot. This shift count is stored in `$2`. Second, the vector is loaded (`lqd`) into register `$8`. Third, a mask is created (`chd`) and stored in `$7` which is needed by the shuffle latter on. Fourth, the vector is rotated (`rotqby`) based on the shift count. Fifth, the value 32 is added (`ahi`) to the desired element that now is in the preferred slot. Finally, the result of the addition is shuffled (`shufb`) back into its original position of the vector. Note that this compiler generated code is not optimal. The shift count is known a priori as pointer `*block` is quadword aligned. Apparently the compiler is not aware of that.

Listing 4.2: Example of how the as2ve instructions speed up code. Left are two lines of code from the IDCT8 kernel, in the middle is the normal assembly code, while the right depicts the assembly code using as2ve instructions.

```
1 vsint16_t *block;   ai      $2,$4,14      lqd       $8,0($4)
2 block[0] += 32;     lqd     $8,0($4)      as2vehwi  $8,32,0
3                     chd     $7,0($4)
4                     rotqby  $2,$8,$2
5                     ahi     $2,$2,32
6                     shufb   $6,$2,$8,$7
```

```
as2veb    rt,ra,immb    \\rt[immb] += ra      \ for vector
as2vebi   rt,imma,immb  \\rt[immb] += imma    /  of bytes.
as2vehw   rt,ra,immb    \\rt[immb] += ra      \ for vector
as2vehwi  rt,imma,immb  \\rt[immb] += imma    /  of halfwords.
as2vew    rt,ra,immb    \\rt[immb] += ra      \ for vector
as2vewi   rt,imma,immb  \\rt[immb] += imma    /  of words.
```

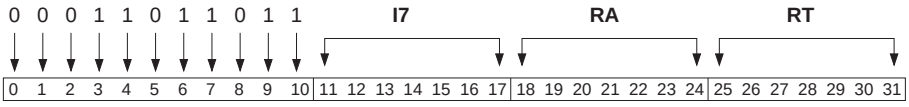
Figure 4.4: Overview of the as2ve instructions.

Using the as2ve instructions the same addition of the scalar to the vector can be done in one instruction (or two, if the load is included as in the example). In the example of Listing 4.2 we used the `as2vehwi` instruction that adds a halfword immediate to a vector of halfwords. Figure 4.4 shows all the as2ve instructions. Only the instructions for bytes and halfwords are used in the benchmarks, but we provide the word versions as well as media application sometimes use this data type. For each data type there are two instructions. The first assumes the scalar is hold in the preferred slot of a register. The second assumes the scalar is put as immediate value in the instruction.

One might consider the lds instruction, augmented with an sts (store scalar) to perform this operation. This might, however, incur a lot of overhead. Suppose the as2ve operation is in the middle of a series of instructions that operate on variables in the register file. In that case, first the vector should be stored in memory, second using an lds instruction the vector element is loaded into the register file, third the addition can be done, fourth an sts instruction has to be used to store the vector element back into the vector, and finally the entire vector has to be loaded again for further operation. This method requires four overhead instructions that are dependent and have a large latency.

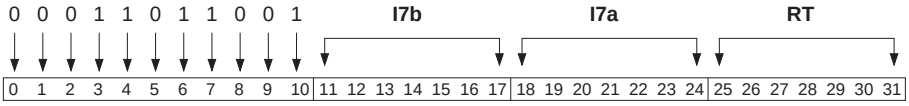
The as2ve instructions fit the RI7 format except the `as2vebi`, `as2vehwi`, and `as2vewi` instructions. The latter have two immediate values and one register value, instead of one immediate value and two register values. The instruction details are depicted below.

as2veb rt,ra,imm



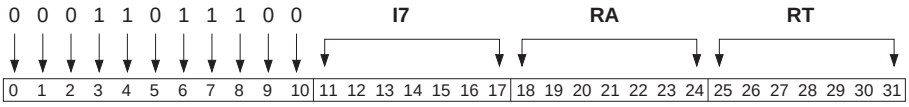
$b \leftarrow RA^{3}$
$c \leftarrow$ RepLeftBit(I7,8) & 0x0F
$d \leftarrow RT^{c}$
$RT^{c} \leftarrow d + b$

as2vebi rt,imma,immb



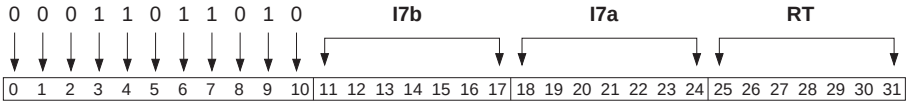
$b \leftarrow$ RepLeftBit(I7a,8)
$c \leftarrow$ RepLeftBit(I7b,8) & 0x0F
$d \leftarrow RT^{c}$
$RT^{c} \leftarrow d + b$

as2vehw rt,ra,imm



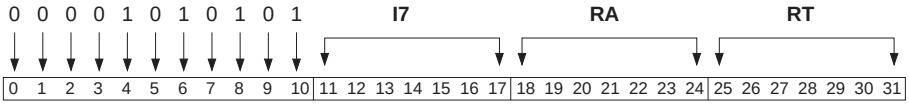
$s \leftarrow RA^{2:3}$
$b \leftarrow$ RepLeftBit(I7,8) & 0x07
$r \leftarrow RT^{2b::2}$
$RT^{2b::2} \leftarrow r + s$

as2vehwi rt,imma,immb



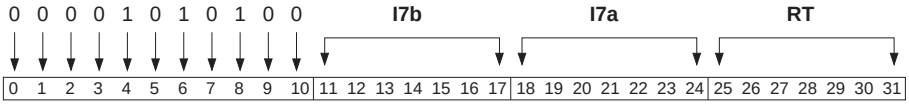
s $\leftarrow$ RepLeftBit(I7a,16)
b $\leftarrow$ RepLeftBit(I7b,8) & 0x07
r $\leftarrow RT^{2b::2}$
$RT^{2b::2}$ $\leftarrow r + s$

as2vew rt,ra,imm



t $\leftarrow RA^{0:3}$
b $\leftarrow$ RepLeftBit(I7,8) & 0x03
u $\leftarrow RT^{4b::4}$
$RT^{4b::4}$ $\leftarrow t + u$

as2vewi rt,imma,immb



t $\leftarrow$ RepLeftBit(I7a,32)
b $\leftarrow$ RepLeftBit(I7b,8) & 0x03
u $\leftarrow RT^{4b::4}$
$RT^{4b::4}$ $\leftarrow t + u$

The as2ve instructions were made available to the programmer through intrinsics of the form `a = spu_as2vebi(a,c,x)` where `a` is the vector operand. The latter is both input and output, which is a case the compiler was not build for. Therefore, not in all cases the intrinsics worked properly and we had to use inline assembly as follows:

```
asm ("as2vebi %0,c,x"      \
        : "=r" (a)         \
        : "0" (a)          \
    );
```

The instructions were added to the simulator and the latency was set to three cycles. A possible implementation uses the FX2 (fixed point) pipeline and replicates the scalar to all SIMD slots of input a. The vector goes to input b of the ALU. Some additional control logic selects the proper SIMD slot to do addition while the others transfer input b. The latency of the FX2 pipeline is two cycles. Adding another cycle, for the increases complexity, results in three cycles.

4.4.2 Accelerating Saturation and Packing

The elementary data type to store pixel information is a byte. Computation, however, is done using halfwords (16 bits) in order not to lose precision. This introduces some overhead known as pack/unpack. In this section we propose three instruction classes that reduce the cost of this overhead. They also accelerate saturation, which is often performed in conjunction with a pack operation.

Clip Instructions

Clip instructions saturate the elements of a vector between two values. A clip instruction can be used to emulate saturating arithmetic but also saturation between arbitrary boundaries. For example, in de deblocking filter kernel the index of a table lookup has to be clipped between 0 and 51. In the normal SPU architecture this operation requires four instructions (see Listing 4.3). First, two masks are created by comparing the input vector with the lower and upper boundaries. Second, using the masks the result is selected from the input and the boundary vectors.

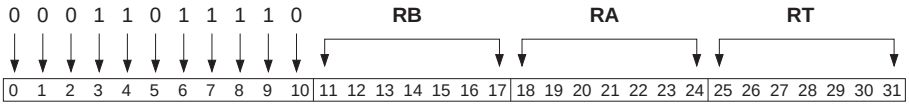
The clip operation is mainly used in integer and short computations. The instruction could be implemented for bytes as well, but we do not expect any

Listing 4.3: C code of a clip function using the normal SPU ISA. The clip instructions perform this operation in one step.

```
1 vsint16_t clip(vsint16_t a, vsint16_t min, vsint16_t max) {
2     vuint16_t min_mask, max_mask;
3     min_mask = spu_cmpgt(min, a);
4     max_mask = spu_cmpgt(a, max);
5     vsint16_t temp = spu_sel(a, min, min_mask);
6     return spu_sel(temp, max, max_mask);
7 }
```

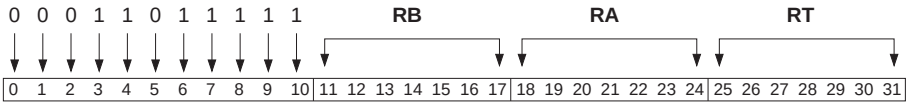
application to use it. The clip instructions fit the RR format if the source and destination register (of the vector to clip) are the same. The instruction details are depicted below.

cliphw rt,ra,rb



for $j = 0$ to 7
    if $RT^{2j::2} > RB^{2j::2}$ then $RT^{2j::2} \leftarrow RB^{2j::2}$
    if $RT^{2j::2} < RA^{2j::2}$ then $RT^{2j::2} \leftarrow RA^{2j::2}$
end

clipw rt,ra,rb



for $j = 0$ to 3
    if $RT^{2j::4} > RB^{2j::4}$ then $RT^{2j::4} \leftarrow RB^{2j::4}$
    if $RT^{2j::4} < RA^{2j::4}$ then $RT^{2j::4} \leftarrow RA^{2j::4}$
end

The clip instructions were made available to the programmer through intrinsics of the form `a = spu_cliphw(a, min, max)` where the operands all have the same data type. We created intrinsics for both vectors and scalars. Note that the same instruction can be used for both.

Using these intrinsics to perform a saturation to 8 or 16-bit values, requires the programmer to define two variables containing the minimum and maximum value. To ease programming intrinsics of the form `a = spu_satub(a)` (saturate to unsigned byte value) can be created. This requires some small modifications to the machine description in the back-end of the compiler, such that it creates the two vectors containing the minimum and maximum values.

In certain cases the clip intrinsics suffered the same problem the as2ve intrinsics, namely that the compiler did not understand that the source and destination registers are the same. The solution is to use inline assembly as follows:

```
asm ("clipw %0,%1,%2"                        \
      : "=r" (a), "=r" (amin), "=r" (amax) \
      : "0" (a), "1" (amin), "2" (amax)    \
    );
```

The clip instructions were added to the simulator and the latency was set to four cycles. A normal compare operation takes two cycles. The two compares can be done in parallel but that requires that three quadwords are read from the register file into the functional unit. Another approach would be to serialize the operation. First two operands are loaded, while these are compared and the result is selected the third operand is loaded, and finally the latter is compared to the result of the first compare. We added one cycle to account for loading three quadwords and also one cycle to account for the extra mux in the datapath that selects the correct value.

Sat_pack Instructions

The sat_pack instructions takes two vectors of a larger data type, and saturates and packs them into one vector of a smaller data type. This operation is useful, for example, whenever data has been processed in signed 16-bit values but has to be stored to memory as unsigned 8-bit values. Listing 4.4 shows such a piece of code from the Luma kernel that can entirely be replaced by one sat_pack instruction. The kernel operates on 16×16 matrices (entire macroblocks). The vectors `va` and `vb` together contain one row of the output matrix.

The sat_pack instructions fit the RR instruction format. For each type of operands sat_pack instructions could be implemented, but for H.264 decod-

Listing 4.4: Example code from the Luma kernel that can be replaced by one sat_pack instruction. The code saturates the vectors `va` and `vb` between 0 and 255, and packs all 16 elements into one vector of unsigned bytes.

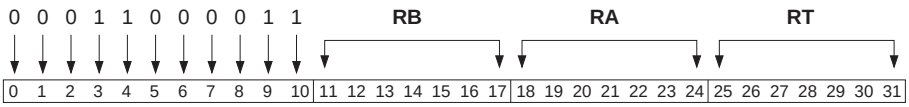
```
1  vsint16_t va, vb, vt, sat;
2  vuint8_t vt_u8;
3  vuint8_t mask = {0x01,0x03,0x05,0x07,0x09,0x0B,0x0D,0x0F,\
4                   0x11,0x13,0x15,0x17,0x19,0x1B,0x1D,0x1F}
5  vsint16_t vzero = spu_splats(0);
6  vsint16_t vmax = spu_splats(255);
7  sat = spu_cmpgt(va,vzero);
8  va = spu_and(va,sat);
9  sat = spu_cmpgt(va,vmax);
10 va = spu_sel(va,vmax,sat);
11 sat = spu_cmpgt(vb,vzero);
12 vb = spu_and(vb,sat);
13 sat = spu_cmpgt(vb,vmax);
14 vb = spu_sel(vb,vmax,sat);
15 vt_u8 = spu_shuffle(va,vb,mask);
```

ing only packing from 16-bit signed to 8-bit unsigned is required. For audio processing packing from 32-bit signed to 16-bit signed can be useful. We implemented two different ways of packing. The `sp2x` (x=ub,hw) instruction packs the elements of the first and second operand to the first and second doubleword respectively. The `spil2x` instruction interleaves the elements of the two operands, i.e., the elements of the first operand go to the odd elements and those of the second operand to the even elements. The latter instruction is useful when the unpack was performed by a multiply instruction. The instruction details are depicted below.

The sat_pack instructions were made available to the programmer through intrinsics and were added to the simulator. The latency was set to two cycles as the complexity of the instruction is similar to that of fixed point operations.

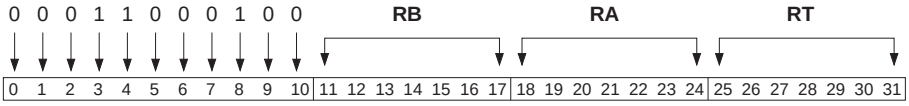
A combination of pack and saturation is also found in the pack.sss and pack.uss instructions of the IA64 architecture [16]. Also MMX has instructions that combine pack and saturate. In the TriMedia there are a few instructions that include a clip operation.

sp2ub rt,ra,rb



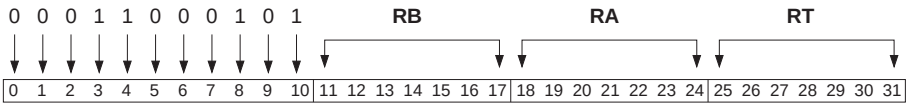
for $j = 0$ to 7
    r $\leftarrow RA^{2j::2}$
    if $r > 255$ then $r \leftarrow 255$
    if $r < 0$ then $r \leftarrow 0$
    $RT^{j} \leftarrow r_{0:7}$
    r $\leftarrow RB^{2j::2}$
    if $r > 255$ then $r \leftarrow 255$
    if $r < 0$ then $r \leftarrow 0$
    $RT^{j+8} \leftarrow r_{0:7}$
end

spil2ub rt,ra,rb



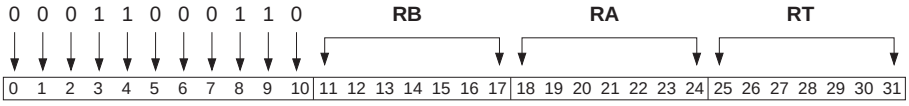
for $j = 0$ to 7
    r $\leftarrow RA^{2j::2}$
    if $r > 255$ then $r \leftarrow 255$
    if $r < 0$ then $r \leftarrow 0$
    $RT^{2*j+1} \leftarrow r_{0:7}$
    r $\leftarrow RB^{2j::2}$
    if $r > 255$ then $r \leftarrow 255$
    if $r < 0$ then $r \leftarrow 0$
    $RT^{2*j} \leftarrow r_{0:7}$
end

sp2hw rt,ra,rb



for $j = 0$ to 3
    t $\leftarrow RA^{4j::4}$
    if $t > 32767$ then $t \leftarrow 32767$
    if $t < -32768$ then $t \leftarrow -32768$
    $RT^{j::2} \leftarrow t_{0:15}$
    t $\leftarrow RB^{4j::4}$
    if $t > 32767$ then $t \leftarrow 32767$
    if $t < -32768$ then $t \leftarrow -32768$
    $RT^{j+8::2} \leftarrow t_{0:15}$
end

spil2hw rt,ra,rb



for $j = 0$ to 3
    t $\leftarrow RA^{4j::4}$
    if $t > 32767$ then $t \leftarrow 32767$
    if $t < -32768$ then $t \leftarrow -32768$
    $RT^{4*j+2::2} \leftarrow t_{0:15}$
    t $\leftarrow RB^{4j::4}$
    if $t > 32767$ then $t \leftarrow 32767$
    if $t < -32768$ then $t \leftarrow -32768$
    $RT^{4*j::2} \leftarrow t_{0:15}$
end

Listing 4.5: Example code from the IDCT8 kernel that can be replaced by one asp instruction. The code adds the vectors `va` and `vb`, saturates them between 0 and 255, and packs the elements to 8-bit values.

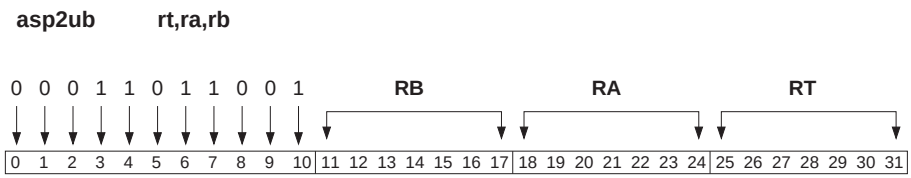
```
1  vsint16_t va, vb, vt, sat;
2  vuint8_t vt_u8;
3  const vuint8_t packu16 =
4                  {0x01,0x03,0x05,0x07,0x09,0x0B,0x0D,0x0F,\
5                   0x11,0x13,0x15,0x17,0x19,0x1B,0x1D,0x1F}
6  vsint16_t vzero = spu_splats(0);
7  vsint16_t vmax = spu_splats(255);
8  vt = spu_add(va,vb);
9  sat = spu_cmpgt(vt,vzero);
10 vt = spu_and(vt,sat);
11 sat = spu_cmpgt(vt,vmax);
12 vt = spu_sel(vt,vmax,sat);
13 vt_u8 = spu_shuffle(vt,vzero,packu16);
```

Asp Instructions

Although the IDCT kernels require some packing operations, it cannot utilize the sat_pack instructions due to the way the data is stored in memory. Especially for this kernel, the asp (Add, Saturate, and Pack) instructions were developed which combine an addition, a saturation, and a pack.

Listing 4.5 shows a piece of code from the IDCT kernel that entirely can be replaced by one asp instruction. The code adds the result of the IDCT to the predicted picture. First, it adds two vectors of signed halfwords. Second, the elements of the vector are saturated between 0 and 255. Finally, the elements are packed to bytes and stored in the first half of the vector. Using the normal SPU ISA many instructions and auxiliary variables are needed. The asp instructions are tailored for this kind of operation and can perform the entire operation at once. Note, that if only a saturate and a pack is required, one of the operands can be assigned a zero vector.

The asp instructions fit the RR format. For each type of operands an asp instruction could be implemented, but we found it only necessary to implement one: `asp2ub` (add, saturate, and pack to unsigned bytes). It takes as input two vectors of signed halfwords and stores the output in a vector of unsigned bytes. The eight elements are stored in the first eight slots while the last eight slots are assigned value zero. The instruction details are depicted below.



for $j = 0$ to 7
  r $\leftarrow RA^{2j::2} + RB^{2j::2}$
  if $r > 255$ then $r \leftarrow 255$
  if $r < 0$ then $r \leftarrow 0$
  $RT^{j} \leftarrow r_{0:7}$
end

The asp instruction was made available to the programmer through an intrinsic. The instruction was added to the simulator and the latency was set to four cycles. A normal addition costs two cycles and we added two more cycles for the saturation and the data rearrangement.

4.4.3 Accelerating Matrix Transposition

Matrix Transposition (MT) is at the heart of many media algorithms and can take up a large part of the execution time. Typically, multimedia data is organized in blocks (matrices) of 16×16, 8×8, and sometimes smaller. This allows utilization of SIMD instructions to exploit DLP. Computations, however, have often to be performed row-wise and column-wise. Thus matrix transposes are required to arrange the data for SIMD processing.

In this section we show how fast matrix transposition at low cost can be achieved using swapoe (swap odd-even) instructions. But first we review existing methodologies for matrix transposition using SIMD.

Conventional SIMD Matrix Transposition

The conventional way of performing a MT with SIMD instructions is to perform a series of permutations. We review an 8×8 transpose in AltiVec as an example. Altivec has 128-bit SIMD registers and thus we assume a matrix of halfwords, stored in eight registers (`a0` through `a7`). The code in Listing 4.6 performs the MT and stores the result in registers (`b0` through `b7`).

Listing 4.6: An 8×8 matrix transpose using Altivec instructions.

```
1  TRANSPOSE_8_ALTIVEC(a0,a1,a2,a3,a4,a5,a6,a7,\
2                      b0,b1,b2,b3,b4,b5,b6,b7){
3      b0 = vec_mergeh( a0, a4 );
4      b1 = vec_mergel( a0, a4 );
5      b2 = vec_mergeh( a1, a5 );
6      b3 = vec_mergel( a1, a5 );
7      b4 = vec_mergeh( a2, a6 );
8      b5 = vec_mergel( a2, a6 );
9      b6 = vec_mergeh( a3, a7 );
10     b7 = vec_mergel( a3, a7 );
11     a0 = vec_mergeh( b0, b4 );
12     a1 = vec_mergel( b0, b4 );
13     a2 = vec_mergeh( b1, b5 );
14     a3 = vec_mergel( b1, b5 );
15     a4 = vec_mergeh( b2, b6 );
16     a5 = vec_mergel( b2, b6 );
17     a6 = vec_mergeh( b3, b7 );
18     a7 = vec_mergel( b3, b7 );
19     b0 = vec_mergeh( a0, a4 );
20     b1 = vec_mergel( a0, a4 );
21     b2 = vec_mergeh( a1, a5 );
22     b3 = vec_mergel( a1, a5 );
23     b4 = vec_mergeh( a2, a6 );
24     b5 = vec_mergel( a2, a6 );
25     b6 = vec_mergeh( a3, a7 );
26     b7 = vec_mergel( a3, a7 );
27 }
```

The instruction `vec_mergeh(ra, rb)` functions as follows. Assume that the elements of each vector are numbered beginning with 0. The even-numbered elements of the result are taken, in order, from the elements in the most significant 8 bytes of `ra`. The odd-numbered elements of the result are taken, in order, from the elements in the most significant 8 bytes of `rb`. The instruction `vec_mergel(ra, rb)` functions similar but takes the elements from the least significant 8 bytes.

The AltiVec MT requires 24 instructions for an 8×8 matrix. In general this approach requires $\log n$ steps of n instructions. Thus in total $n \log n$ instructions are required. For $n = 16$ 64 AltiVec or SPU instructions are required.

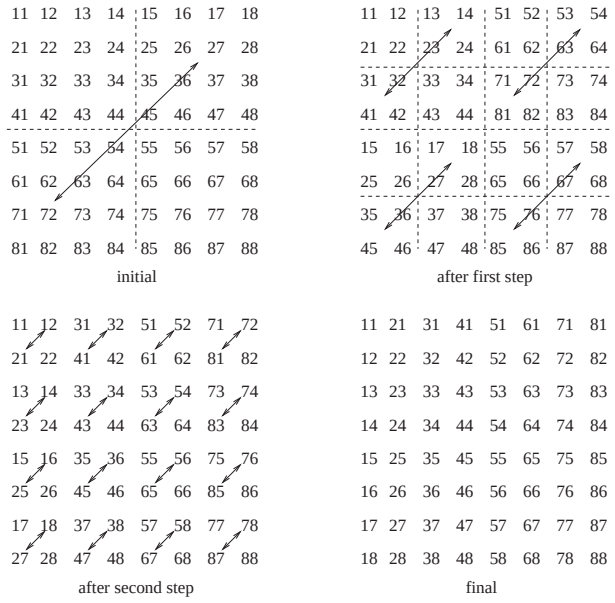


Figure 4.5: Eklundh's matrix transpose algorithm.

Matrix Transposition using Swapoe Instructions

In this section we present the swapoe instructions and explain how they speed up matrix transposition. The swapoe instructions are similar to the mix instructions introduced by Lee [97] and used in HP's MAX-2 multimedia extension to the PA-RISC architecture. Although the mix instructions found their way into the IA-64 architecture because of the collaboration between HP and Intel, no modern multimedia extension incorporates it. We rediscover the swapoe instructions and show how it is valuable for contemporary media applications.

The swapoe instructions are inspired by Eklundh's recursive matrix transposition algorithm [56], which is illustrated in 4.5 for an 8×8 matrix. The new instructions have two source registers and two destination registers. In this chapter we assume that the source and the destination registers are the same, but if allowed by the instruction format, they can also be different. Using the same registers for source and destination has the advantage that the transpose will be performed in-situ. The complete set of swapoe instructions is depicted in Figure 4.6.

```
swapoedw  ra,rb          \\Swap doublewords
swapoew   ra,rb          \\Swap words
swapoehw  ra,rb          \\Swap halfwords
swapoeb   ra,rb          \\Swap bytes
```

Figure 4.6: Overview of the swapoe instructions.

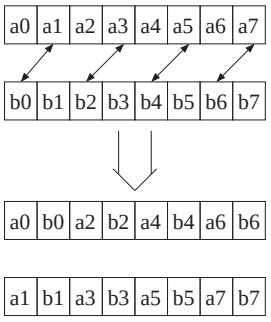


Figure 4.7: Graphical representation of the `swapoehw` instruction. The odd elements of the `a` register are swapped with the even elements of the `b` register.

The instructions swap the odd elements of register `ra` with the even elements of `rb`. Figure 4.7 shows the `swapoehw` instruction as example. On top the two registers before the instruction is executed are depicted. The bottom of the figure shows the contents of the register after the `swapoehw`. The other instructions operate similar but on different element sizes.

The swapoe instructions directly implement the steps of Eklundh's matrix transpose algorithm. The 8×8 MT function using swapoe is shown in Listing 4.7. This implementation requires 12 instructions. Note that this method performs the MT in-situ, i.e., no additional registers are required for temporal storage. Depending on the size of the register file and the kernel the MT is used in, this might save a lot of loads and stores and further increase performance.

In general for an $n \times n$ matrix each step can be accomplished using $n/2$ instructions. Since there are $\log n$ stages, the total number of instructions is $(n \log n)/2$, assuming the matrix is already loaded in the register file. Thus, when $n = 16$ 32 instructions are required. Indeed our approach also requires half the number of instructions compared to the AltiVec implementation.

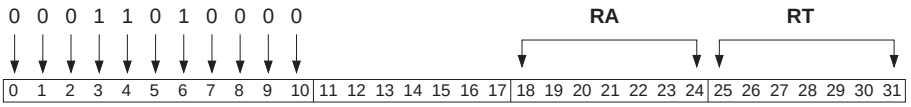
Listing 4.7: An 8×8 matrix transpose using the swapoe instructions.

```
1  TRANSPOSE_8_SWAPOE(a0,a1,a2,a3,a4,a5,a6,a7) {
2      swapoedw(a0, a4);
3      swapoedw(a1, a5);
4      swapoedw(a2, a6);
5      swapoedw(a3, a7);
6      swapoew(a0, a2);
7      swapoew(a1, a3);
8      swapoew(a4, a6);
9      swapoew(a5, a7);
10     swapoehw(a0, a1);
11     swapoehw(a2, a3);
12     swapoehw(a4, a5);
13     swapoehw(a6, a7);
14 }
```

Implementation Details

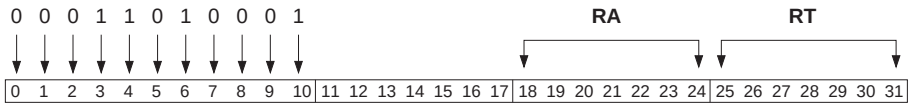
The swapoe instructions have two register operands and thus fit the RR instruction format of the SPU [81], where the RB slot is not used. Below the details of all swapoe instructions are provided.

swapoedw rt,ra



$$
\begin{aligned}
w &\leftarrow RT^{8:15} \\
RT^{8:15} &\leftarrow RA^{0:7} \\
RA^{0:7} &\leftarrow w
\end{aligned}
$$

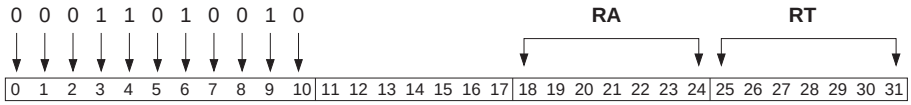
swapoew rt,ra



$t \leftarrow RT^{4:7}$
$RT^{4:7} \leftarrow RA^{0:3}$
$RA^{0:3} \leftarrow t$

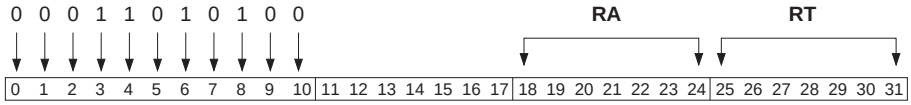
$t \leftarrow RT^{12:15}$
$RT^{12:15} \leftarrow RA^{8:11}$
$RA^{8:11} \leftarrow t$

swapoehw rt,ra



for $j = 0$ to 15 by 4
    $r \leftarrow RT^{j+2::2}$
    $RT^{j+2::2} \leftarrow RA^{j::2}$
    $RA^{j::2} \leftarrow r$
end

swapoeb rt,ra



for $j = 0$ to 15 by 2
    $b \leftarrow RT^{2+1}$
    $RT^{j+1} \leftarrow RA^{2}$
    $RA^{j} \leftarrow b$
end

All swapoe instructions were made accessible from C code through intrinsics. Using the intrinsic, however, can cause problems due to the following. The SPU compiler assumes that for an instruction `mnemonic rt, ra, [rb/rc/im]` only register `rt` is modified. As a consequence, the compiler assumes the variable located in register `ra` is also available in memory, and might decide to re-use the register for another variable. To avoid this problem inline assembly was used, which allows the programmer to specify the mode of each register:

```
asm volatile ("swapoedw %0,%1"    \
        : "=r" (a0), "=r" (a4)    \
        : "0" (a0), "1" (a4)      \
        );
```

Note that we used the keyword `volatile` here. It was not required to maintain correct order of execution. We used it to enforce the compiler to maintain the hand optimized order of instructions. Simulations confirmed that indeed the hand optimized order of these swapoe instruction resulted in faster execution compared to compiler scheduled swapoe instructions.

The swapoe instructions were added to the simulator and to the compiler. We assume a latency of three clock cycles for the predefined permutations of the swapoe instructions. The instructions, however, write to two registers instead of one. Writing to two registers can be serialized if only one write-port is available. The SPU register file probably has two write-ports as it has an odd and even pipeline. Thus, the swapoe instructions could as well use both these write ports. This might, however, cause a stall in the other pipeline. We account for the second write, no matter how it is implemented, by adding one extra cycle. So in total the latency of the swapoe instructions was set to four clock cycles.

Related Work

A linear time matrix transposition method was proposed in [72], which utilizes a special vector register file that allows diagonal access. The in-situ matrix transpose (IPMT) algorithm allows a matrix, stored in register file, to be transposed by performing a number of rotations on the diagonal registers. The number of instructions required by this approach, assuming the buffered implementation, is $2n - 2$. For the common cases $n = 16$ and $n = 8$ this approach requires 30 and 14 instructions, respectively which both are approximately twice as fast as the AltiVec matrix transpose. Our implementation using the swapoe instruction takes 32 and 12 instructions for $n = 16$ and $n = 8$, respec-

Listing 4.8: Example of how the sfxsh instructions speed up code of the IDCT kernel. Left are two lines of C code, in the middle is the normal assembly code, while the right depicts the assembly code using sfxsh instructions.

```
1 b1 = a7>>2 + a1;  rotmahi $11,$9,-2   shaddhw $11,$11,$9,2
2 b3 = a3 + a5>>2;  ah      $11,$11,$4  shaddhw $8,$8,$15,2
3                   rotmahi $8,$15,-2
4                   ah      $8,$8,$7
```

tively. Thus our approach is as fast as IPMT but is much simpler as we use a conventional register file.

As mentioned in Section 4.1, the MOM architecture provides a matrix transpose instruction. According to [48] this transpose takes eight cycles for an 8×8 matrix. As the architecture is only described at the ISA level, it is unclear how this is implemented and what the impact on the performance is.

4.4.4 Accelerating Arithmetic Operations

In this section we propose several instruction classes that enhance the execution of regular arithmetic expressions in video decoding.

Sfxsh Instructions

Sfxsh (Simple FiXed point & SHift) instructions combine simple fixed point operations with a shift operation. Listing 4.8 shows an example from the IDCT8 kernel where one operand is shifted and added to the second. Using a normal ISA, these two lines of code would require four instructions.

This kind of code is found frequently in the kernels we investigated. More general, they can be characterized as follows. There are two register operands, a simple arithmetic operation (such as addition and subtraction), and one shift right by a small constant amount of bits. Our analysis showed that the instructions depicted in Figure 4.8 are beneficial. Our benchmarks only used the instructions that operate on vectors of halfwords, but we provide the same operations for vectors of words as well. To prevent overflow, computation is generally not performed using bytes but halfwords. Thus, the sfxsh instructions for bytes do not seem to be useful and are omitted.

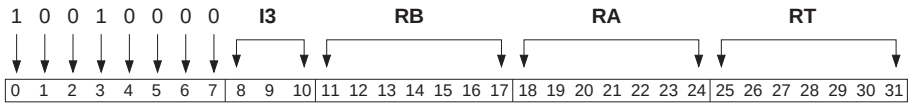
The sfxsh instructions could fit the RRR format of the SPU ISA if the immediate value would be stored in a register. This approach, however, would

```
shaddhw  rt,ra,rb,imm  \\rt = ra + rb>>imm     \
shsubhw  rt,ra,rb,imm  \\rt = ra - rb>>imm     | for vector
addshhw  rt,ra,rb,imm  \\rt = (ra + rb)>>imm   | of halfwords
subshhw  rt,ra,rb,imm  \\rt = (ra - rb)>>imm   /
shaddw   rt,ra,rb,imm  \\rt = ra + rb>>imm     \
shsubw   rt,ra,rb,imm  \\rt = ra - rb>>imm     | for vector
addshw   rt,ra,rb,imm  \\rt = (ra + rb)>>imm   | of words
subshw   rt,ra,rb,imm  \\rt = (ra - rb)>>imm   /
```

Figure 4.8: Overview of the sfxsh instructions.

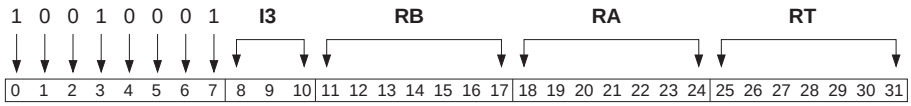
cause an unnecessary load to a register. Furthermore, there are not enough opcodes of the RRR format available to implement all instructions. It is the case, though, that the shift operation is only performed by a few bits, i.e., the shift count is between 0 and 7. Thus, the immediate value can be represented by 3 bits only. Using this observation we defined the RRI3 instruction format that uses 8 bits for the opcode, 3 bits for the immediate value, and 7 bits for each of the three register operands. The instruction details, depicted below, show the RRI3 format graphically.

shaddhw rt,ra,rb,imm



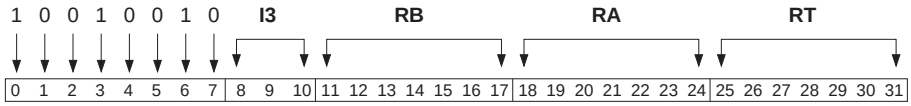
s $\leftarrow$ RepLeftBit(I3,16)
for $j = 0$ to 15 by 2
    p $\leftarrow RT^{j::2}$
    for $b = 0$ to 15
        if $(b - s) > 0$ then $r_b \leftarrow p_{b-s}$
        else $r_b \leftarrow 0$
    end
    $RB^{j::2} \leftarrow RA^{j::2} + r$
end

shsubhw rt,ra,rb,imm



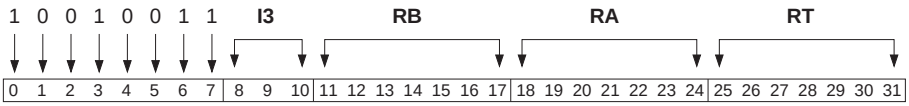
s ← RepLeftBit(I3,16)
for $j = 0$ to 15 by 2
    p ← $RT^{j::2}$
    for $b = 0$ to 15
        if $(b - s) > 0$ then $r_b \leftarrow p_{b-s}$
        else $r_b \leftarrow 0$
    end
    $RB^{j::2} \leftarrow RA^{j::2} - r$
end

addshhw rt,ra,rb,imm



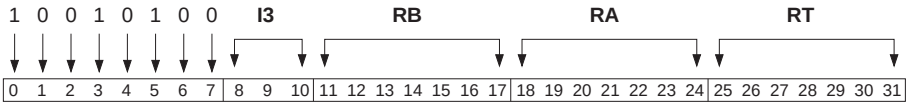
s ← RepLeftBit(I3,16)
for $j = 0$ to 15 by 2
    p ← $RA^{j::2} + RT^{j::2}$
    for $b = 0$ to 15
        if $(b - s) > 0$ then $r_b \leftarrow p_{b-s}$
        else $r_b \leftarrow 0$
    end
    $RB^{j::2} \leftarrow r$
end

subshhw rt,ra,rb,imm



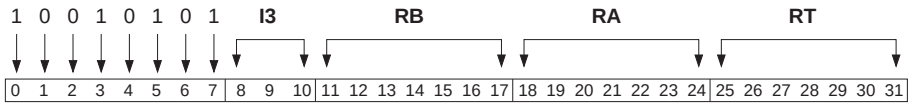
s ← RepLeftBit(I3,16)
for $j = 0$ to 15 by 2
    p ← $RA^{j::2} - RT^{j::2}$
    for $b = 0$ to 15
        if $(b - s) > 0$ then $r_b \leftarrow p_{b-s}$
        else $r_b \leftarrow 0$
    end
    $RB^{j::2} \leftarrow r$
end

shaddw rt,ra,rb,imm



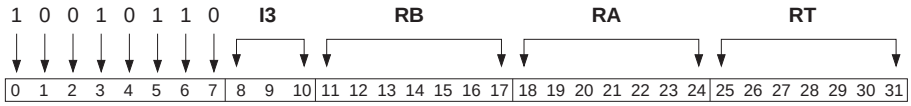
s ← RepLeftBit(I3,16)
for $j = 0$ to 15 by 4
    t ← $RT^{j::4}$
    for $b = 0$ to 31
        if $(b - s) > 0$ then $u_b \leftarrow t_{b-s}$
        else $u_b \leftarrow 0$
    end
    $RB^{j::4} \leftarrow RA^{j::4} + u$
end

shsubw rt,ra,rb,imm



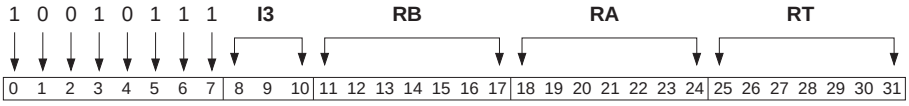
s ← RepLeftBit(I3,16)
for $j = 0$ to 15 by 4
    t ← $RT^{j::4}$
    for $b = 0$ to 31
        if $(b - s) > 0$ then $u_b \leftarrow t_{b-s}$
        else $u_b \leftarrow 0$
    end
    $RB^{j::4} \leftarrow RA^{j::4} - u$
end

addshw rt,ra,rb,imm



s ← RepLeftBit(I3,16)
for $j = 0$ to 15 by 4
    t ← $RA^{j::4} + RT^{j::4}$
    for $b = 0$ to 31
        if $(b - s) > 0$ then $u_b \leftarrow t_{b-s}$
        else $u_b \leftarrow 0$
    end
    $RB^{j::4} \leftarrow u$
end

subshw rt,ra,rb,imm



s ← RepLeftBit(I3,16)
for $j = 0$ to 15 by 4
    t ← $RA^{j::4} - RT^{j::4}$
    for $b = 0$ to 31
        if $(b - s) > 0$ then $u_b \leftarrow t_{b-s}$
        else $u_b \leftarrow 0$
    end
    $RB^{j::4} \leftarrow u$
end

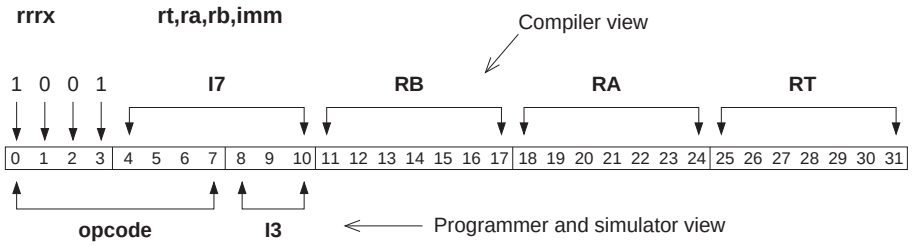


Figure 4.9: The `rrrx` instruction of format RRR was used inside the compiler to handle the sfxsh instructions with format RRI3. At the programmers side an offset was added to the immediate value to set bits 4 to 7.

Listing 4.9: Defines translate the sfxsh intrinsics to the rrrx intrinsic and add the offset to the immediate value.

```
1 #define spu_shaddhw(a,b,c)     spu_rrrx(a,b,c);
2 #define spu_shsubhw(a,b,c)     spu_rrrx(a,b,c+8);
3 #define spu_addshhw(a,b,c)     spu_rrrx(a,b,c+16);
4 #define spu_subshhw(a,b,c)     spu_rrrx(a,b,c+24);
5 #define spu_shaddw(a,b,c)      spu_rrrx(a,b,c+32);
6 #define spu_shsubw(a,b,c)      spu_rrrx(a,b,c+40);
7 #define spu_addshw(a,b,c)      spu_rrrx(a,b,c+48);
8 #define spu_subshw(a,b,c)      spu_rrrx(a,b,c+56);
```

The sfxsh instructions were made available to the programmer through intrinsics. We did not modify the compiler to handle the new instruction format. Instead we used a little trick at the programmer's side, such that the compiler could handle the sfxsh instructions as if they were of the RRR format. Inside the compiler we created the instruction `rrrx rt,ra,rb,imm` using the RRR format except that we replaced `rc` with an immediate (see Figure 4.9). The opcode is 4 bits and has value `0x480` (`1001`). This instruction is available through the intrinsic `d = spu_rrrx(a,b,imm)`. To create the instruction word corresponding to the sfxsh instructions we created the defines of Listing 4.9 that add an offset to the immediate value. The offset will change bits 4 to 7 of the instruction word such that the right RRI3 opcode is created.

The simulator was modified to recognize the new RRI3 instruction format and the new instructions. The latency of the sfxsh instructions was set to five cycles. Normal shift and rotate instructions take four cycles. We added one cycle to this for the extra addition or subtraction.

Listing 4.10: Example code from the Luma kernel that can be replaced by one mpytr instruction. In the code, first the odd elements of `input` are multiplied by constant 5, followed by the even elements. A shuffle packs the two vectors of words into one vector of halfwords.

```
1 vsint16_t v5 = spu_splats(5);
2 vuint8_t mask = {0x02,0x03,0x12,0x13,0x06,0x07,0x16,0x17,\
3                  0x0A,0x0B,0x1A,0x1B,0x0E,0x0F,0x1E,0x1F};
4 vsint32_t A = spu_mulo(input, 5);
5 vsint32_t B = spu_mule(input, v5);
6 vsint16_t result = spu_shuffle(B, A, mask);
```

Mpytr Instructions

The Mpytr (multiply and truncate) instructions perform a multiply or multiply-add on vectors of halfwords and stores the results as halfwords as well by truncating the most significant bits. The normal multiply instructions in the SPU ISA take halfword operands as input and produce word values. The reason is of course not to lose precision, but the side effect is that only four SIMD slots are used while the input vector consists of eight elements.

In video decoding the primary input and output data type is unsigned byte. In order not to lose precision, computations are mostly performed in signed halfword values. In the Luma and Chroma interpolation filters several multiplications are performed. Using the normal ISA the intermediate results are words (32 bits) while we know from the algorithm that the maximum possible value can be represented by 15 bits. Using the mpytr instructions, the conversion to 32-bit values is avoided.

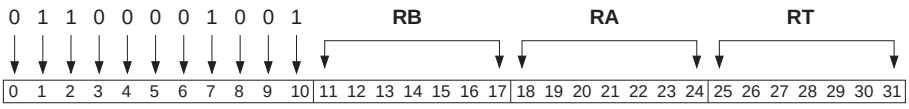
Listing 4.10 shows a piece of code from the Luma kernel that multiplies the vector `input`, containing eight signed halfwords, with a constant 5. First the odd elements are multiplied (`spu_mulo`) followed by the even elements (`spu_mule`). Using a shuffle the two vectors of words are packed into one vector of halfwords. Furthermore, there is some overhead involved to create a vector of constants and to load the mask. This overhead though is performed once per function call and thus shared among a few multiplications.

The code of Listing 4.10 can be replaced by one `mpytrhwi` instruction, which multiplies a vector of signed halfwords by an unsigned immediate value. The other instructions in this class are `mpytrhw` (multiply two vectors of signed halfwords) and `maddtrhw` (multiply-add two vectors of signed halfwords). We do not expect mpytr instructions for unsigned halfwords to be useful as

most media computations are done using signed values. Mpytr instructions for bytes do not seem useful as unpacking to 16-bit is generally necessary and thus normal byte multiplication instructions can be used (see next section). Mpytr instructions for words might be useful in certain cases. As we are not aware of any, we did not implement those instructions. Further investigation of applications, for example audio processing, would be useful in this respect.

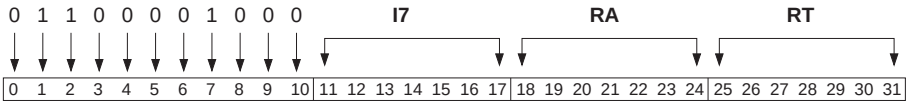
The `mpytrhw` instruction has the RR format and the `mpytrhwi` has the RI7 format. Thus the latter has seven bits for the immediate value which is enough for video decoding. The RI10 format could be used as well in case more bits are necessary for the immediate value. The `maddtrhw` instruction has the RRR format. The instruction details are depicted below.

mpytrhw rt,ra,rb

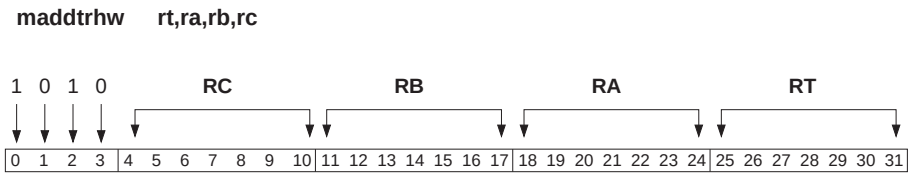


for $j = 0$ to 7
  t $\leftarrow RA^{2j::2} * RB^{2j::2}$
  $RT^{2j::2}$ $\leftarrow t_{0:15}$
end

mpytrhwi rt,ra,imm



for $j = 0$ to 7
  t $\leftarrow RA^{2j::2} * I7$
  $RT^{2j::2}$ $\leftarrow t_{0:15}$
end



for $j = 0$ to 7
$\quad t \leftarrow RA^{2j::2} * RB^{2j::2} + RC^{2j::2}$
$\quad RT^{2j::2} \leftarrow t_{0:15}$
end

The mpytr instructions were made available to the programmer through intrinsics and were added to the simulator. All regular multiply instructions are executed in the FP7 pipeline which has a latency of seven cycles. The mpytr instructions are no more complex than the other multiply instructions and thus the latency of those was set to seven cycles.

Mpy_byte Instructions

The mpy_byte instructions perform a multiply on vectors of unsigned byte elements and produce vectors of unsigned halfwords. These instructions are not novel at all, but the SPU ISA did not contain any byte multiplication instructions although they are useful for many media applications.

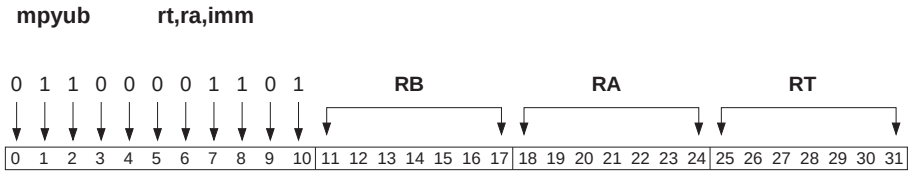
In video decoding typically the byte input data is first unpacked to halfwords, after which the arithmetic computations are performed. In the previous section we showed how the mpytr instructions can prevent the overhead of going to words when performing a multiplication. Another way to resolve this issue is to do multiplications on the byte values directly and get the unpack to halfwords for free.

As byte multiplication instructions are not part of the normal SPU ISA we implemented them. The `mpyubi` and `mpyhhubi` instructions multiply a vector of unsigned bytes with an immediate value and store the results as unsigned halfwords. The first instruction operates on the odd elements of the input vector while the latter operates on the even elements. These instructions fit the RI7 format. Although not used in the benchmarks, we also added the `mpyub` and `mpyhhub` instructions to the ISA. These instructions are the same as the previous ones, but take two registers operands as input.

The `maddub` and `msubub` perform a multiply-add and multiply-sub operation on vectors of unsigned bytes and store the result as a vector of unsigned halfwords. These instructions operate on the odd elements of the input vectors. These two instructions fit the RRR format. Only a few opcodes of the RRR format are available and in the normal SPU ISA only two were left unused. We already used these opcodes, however, for the `rrrx` and `maddtrhw` instructions. To be able to use the `maddub` and `msubub` anyway, we removed two floating point instructions from the ISA. For a media accelerator these floating point instructions are not needed and thus this choice is justified. If a more general purpose core is targeted, the `msubub` is most likely not worth removing another instruction for. Simulation results showed that this instruction provides very limited benefit compared to others.

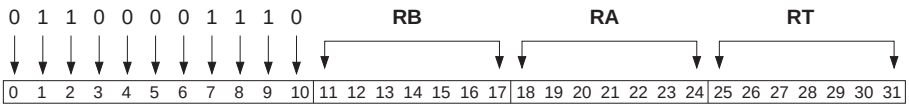
We also implemented the `mpyhhaub` which multiply-adds the even elements of vectors of bytes. Similar to the existing `mpyhha` instruction it has the RR format and thus writes the result to the register of the first operand. We expected this instruction to be a good addition to the `maddub` instruction. Simulation results, however, showed no speedup using this instruction compared to using a shuffle and a normal multiply-add. Therefore we did not include it in the accelerator architecture. For the same reason we omitted the `mpyhhsub` (multiply high high and sub unsigned bytes) instruction.

We did not implement any multiplication instructions for signed bytes. We are not aware of any media application using this data type. The details of the new byte multiplication instructions that we implemented are depicted below.



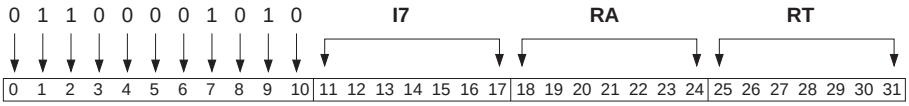
for $j = 0$ to 7
$\quad RT^{2j::2} \leftarrow RA^{2j+1} \mid * \mid RB^{2j+1}$
end

mpyhhub rt,ra,imm



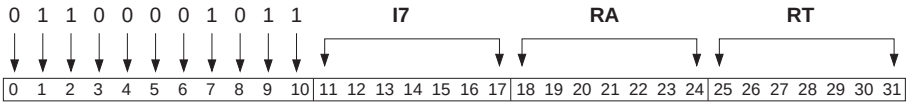
for $j = 0$ to 7
$\quad RT^{2j::2} \leftarrow RA^{2j} \mid * \mid RB^{2j}$
end

mpyubi rt,ra,imm



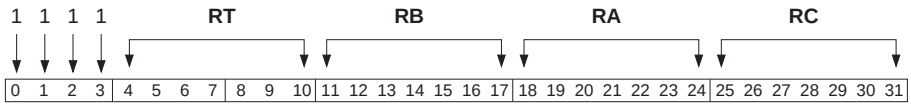
for $j = 0$ to 7
$\quad RT^{2j::2} \leftarrow RA^{2j+1} \mid * \mid I7$
end

mpyhhubi rt,ra,imm



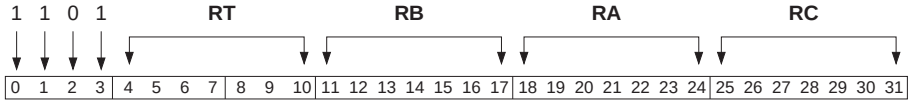
for $j = 0$ to 7
$\quad RT^{2j::2} \leftarrow RA^{2j} \mid * \mid I7$
end

maddub rt,ra,rb,rc



for $j = 0$ to 7
$\quad RT^{2j::2} \leftarrow RA^{2j+1} \mid * \mid RB^{2j+1} + RC^{2j+1}$
end

msubub rt,ra,rb,rc



for $j = 0$ to 7
$\quad RT^{2j::2} \leftarrow RA^{2j+1} \mid * \mid RB^{2j+1} - RC^{2j+1}$
end

The mpy_byte instructions were made available to the programmer through intrinsics and were added to the simulator. Just as the mpytr instruction, the mpy_byte instructions are no more complex than the other multiply instructions and thus the latency of those was set to seven cycles.

IDCT8 Instruction

The Inverse Discrete Cosine Transform (IDCT) is an operation often found in still picture and video processing. The IDCT is typically performed on 8×8 or 4×4 pixel blocks. Performing the IDCT comprises a one-dimensional IDCT row-wise followed by a one-dimensional IDCT on each column. FFmpeg uses the LLM IDCT algorithm [105]. The computational structure of the one-dimensional IDCT8 is depicted in Figure 4.10. If elements 0 through 7 are a row of the matrix, then this diagram represents a row-wise one-dimensional IDCT8. The diagram also shows that the IDCT consists of simple additions, subtractions, and some shifts.

The typical way to SIMDimize this one-dimensional IDCT is to perform the operations on vectors of elements. Thus for the row-wise 1D IDCT, a vector

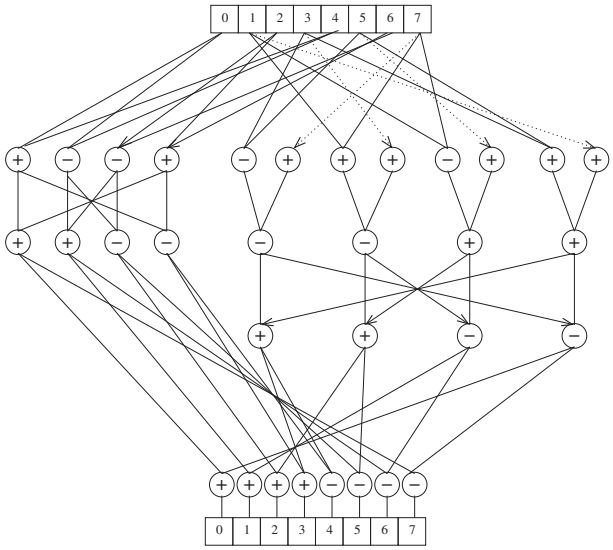


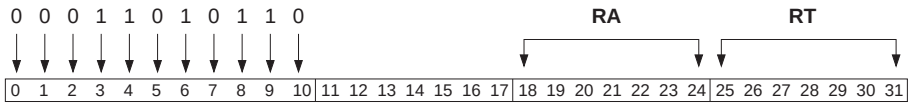
Figure 4.10: Computational structure of the one-dimensional IDCT8. Arrows indicate a shift right by one bit. A dotted line means that the two input operands are the same.

would be a column of the matrix, while for the column-wise 1D IDCT a vector would be a row of the matrix. Assuming the sub-block is stored in memory row major (row elements in consecutive addresses), the whole IDCT is performed in four steps: matrix transpose (MT); row-wise 1D IDCT; MT; column-wise 1D IDCT.

We propose the `IDCT8` instruction that performs the row-wise 1D IDCT8. This intra-vector instruction takes one row of the matrix as input, performs the arithmetic operations as depicted in Figure 4.10, and stores the result in a register. Thus, for an 8×8 matrix the total row-wise 1D IDCT8 takes eight instructions. Besides the reduction in the number of executed instructions, the two MTs are also avoided. This is because using the `IDCT8` instruction the whole IDCT can be performed in two steps: row-wise 1D IDCT using the `IDCT8` instruction, followed by a conventional column-wise 1D IDCT.

The `IDCT8` instruction fits the RR format where the RB slot is not used. Register RA is the input register while RT is the output register. Usually for the IDCT8 these two would be the same but the instruction allows them to be different. The `IDCT8` instruction assumes its operand to be a vector of signed halfwords. The instruction details are depicted below.

idct8 **rt,ra**



$$
\begin{array}{llllll}
r0 & \leftarrow RA^{0:1} & +RA^{8:9} & & & \\
r2 & \leftarrow RA^{0:1} & -RA^{8:9} & & & \\
r4 & \leftarrow RA^{2:3}/2 & -RA^{12:13} & & & \\
r6 & \leftarrow RA^{2:3} & +RA^{12:13}/2 & & & \\
 & & & & & \\
s0 & \leftarrow r0 + r6 & & & & \\
s2 & \leftarrow r2 + r4 & & & & \\
s4 & \leftarrow r2 - r4 & & & & \\
s6 & \leftarrow r0 - r6 & & & & \\
 & & & & & \\
r1 & \leftarrow RA^{10:11} & -RA^{6:7} & -(RA^{14:15} & +RA^{14:15}/2) & \\
r3 & \leftarrow RA^{2:3} & +RA^{14:15} & -(RA^{6:7} & +RA^{6:7}/2) & \\
r5 & \leftarrow RA^{14:15} & -RA^{2:3} & +(RA^{10:11} & +RA^{10:11}/2) & \\
r7 & \leftarrow RA^{6:7} & +RA^{10:11} & +(RA^{2:3} & +RA^{2:3}/2) & \\
 & & & & & \\
s1 & \leftarrow r7/4 + r1 & & & & \\
s3 & \leftarrow r3 + r5/4 & & & & \\
s5 & \leftarrow r3/4 - r5 & & & & \\
s7 & \leftarrow r7 - r1/4 & & & & \\
 & & & & & \\
RT^{0:1} & \leftarrow s0 + s7 & & & & \\
RT^{2:3} & \leftarrow s2 + s5 & & & & \\
RT^{4:5} & \leftarrow s4 + s3 & & & & \\
RT^{6:7} & \leftarrow s6 + s1 & & & & \\
RT^{8:9} & \leftarrow s6 - s1 & & & & \\
RT^{10:11} & \leftarrow s4 - s3 & & & & \\
RT^{12:13} & \leftarrow s2 - s5 & & & & \\
RT^{14:15} & \leftarrow s0 - s7 & & & &
\end{array}
$$

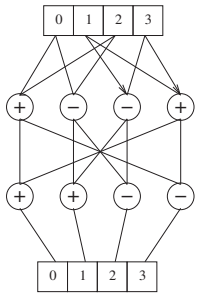


Figure 4.11: Computational structure of the one-dimensional IDCT4. An arrow indicates a shift right by one bit.

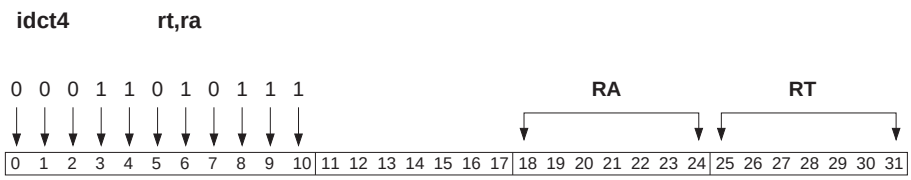
The `IDCT8` instruction was made available to the programmer through an intrinsic. The instruction was added to the simulator and the latency was set to eight cycles for the following reason. Fixed point operations take two cycles in the SPU architecture. The `IDCT8` instruction performs four steps of simple fixed point operations and thus the latency of the new instruction is at most four times two making a total of eight cycles.

IDCT4 Instruction

The IDCT4 is similar to the IDCT8 but operates on a 4×4 pixel block. The computational structure of the one-dimensional IDCT4 is simpler than that of the IDCT8 as shown in Figure 4.11.

For the IDCT4 kernel a similar approach could be used as for the IDCT8. As a 4×4 matrix of halfwords, however, completely fits in two registers of 128-bits, the entire two-dimensional IDCT4 could be performed using one instruction with two operands. Such an instruction would read the matrix from its two operand registers, perform the arithmetic operations, and store the result back in the two registers. Storing the result could be done in one cycle if the register file has two write ports, or otherwise two cycles are required.

The `IDCT4` instructions fits the RR format where the RB slot is not used. Register RA is the input and output register that contains the first two rows of the matrix while RT holds the last two rows. The IDCT4 instruction assumes its operands to be vectors of signed halfwords. The instruction details are depicted below.



for $i = 0$ to 8 by 8

$$
\begin{array}{lll}
r0 & \leftarrow RA^{i::2} & +RA^{(i+4)::2} \\
r1 & \leftarrow RA^{i::2} & -RA^{(i+4)::2} \\
r2 & \leftarrow (RA^{(i+2)::2} >> 1) & -RA^{(i+6)::2} \\
r3 & \leftarrow RA^{(i+2)::2} & +(RA^{(i+6)::2} >> 1)
\end{array}
$$

$$
\begin{array}{lll}
Q^{i::2} & \leftarrow r0 & +r3 \\
Q^{(i+2)::2} & \leftarrow r1 & +r2 \\
Q^{(i+4)::2} & \leftarrow r1 & -r2 \\
Q^{(i+6)::2} & \leftarrow r0 & -r3
\end{array}
$$

$$
\begin{array}{lll}
r0 & \leftarrow RT^{i::2} & +RT^{(i+4)::2} \\
r1 & \leftarrow RT^{i::2} & -RT^{(i+4)::2} \\
r2 & \leftarrow (RT^{(i+2)::2} >> 1) & -RT^{(i+6)::2} \\
r3 & \leftarrow RT^{(i+2)::2} & +(RT^{(i+6)::2} >> 1)
\end{array}
$$

$$
\begin{array}{lll}
R^{i::2} & \leftarrow r0 & +r3 \\
R^{(i+2)::2} & \leftarrow r1 & +r2 \\
R^{(i+4)::2} & \leftarrow r1 & -r2 \\
R^{(i+6)::2} & \leftarrow r0 & -r3
\end{array}
$$

end

for $i = 0$ to 3

$$
\begin{array}{ll}
r0 & \leftarrow Q^{2i::2} + R^{2i::2} \\
r1 & \leftarrow Q^{2i::2} - R^{2i::2} \\
r2 & \leftarrow (Q^{2(i+4)::2} >> 1) - R^{2(i+4)::2} \\
r3 & \leftarrow Q^{2(i+4)::2} + (R^{2(i+4)::2} >> 1)
\end{array}
$$

$$
\begin{array}{ll}
RA^{2i::2} & \leftarrow ((r0 + r3) >> 6) \\
RA^{2(i+4)::2} & \leftarrow ((r1 + r2) >> 6) \\
RT^{2i::2} & \leftarrow ((r1 - r2) >> 6) \\
RA^{2(i+4)::2} & \leftarrow ((r0 - r3) >> 6)
\end{array}
$$

end

As the instruction details show, the result is shifted right by six bits before writing to the registers. This operation is not depicted in Figure 4.11 but is always performed after the two one-dimensional IDCTs have been computed. Therefore we added this operation to the `IDCT4` instruction.

The `IDCT4` instruction was made available to the programmer through inline assembly. Using inline assembly it is possible to specify that both operands of the instruction are read and written. The following code shows how to do this:

```
asm volatile ("idct4 %0,%1"         \
        : "=r" (v0), "=r" (v1)      \
        : "=0" (v0), "=1" (v1)      \
        );
```

The instruction was added to the simulator and the latency was set to ten cycles for the following reason. Fixed point operations take two cycles in the SPU architecture. The `IDCT4` instruction performs four steps of simple fixed point operations making a total of eight. We also count one cycle for the final shift (although this could be hardwired) and one cycle for writing the second result to the register file.

Related Work

The sfxsh instructions collapse two operations in one instructions. The operation pattern of the sfxsh instructions is frequently found in the H.264 kernels and is therefore useful to provide in hardware. In [26] it is described how such recurring operation patterns can automatically be determined.

The (inverse) discrete cosine transform has been used in many applications that compress image data. Several algorithms exist to compute the (I)DCT and many hardware implementations have been proposed. Some of the more recent proposals can be found in [69] and [66], both of which provide a good overview of other implementations in the past. Simai et al. [141] proposed an implementation for a programmable media processor. They extended the TriMedia processor with a reconfigurable execution unit and demonstrated how to implement the IDCT on it.

The (I)DCT algorithm used in the H.264 standard is a simplified version that is much less complex than previous algorithms. For example, it uses only addition, subtraction, and shifts where other algorithm required many multiplications. This algorithm was proposed in [109, 108] for the 4×4 transform while the algorithm for the 8×8 transform was proposed in [67]. To the best of our knowledge, the only hardware implementation of this algorithm was

Listing 4.11: Example code from the Luma kernel that loads one quadword from an unaligned memory address (`src`).

```
1 int perm_src = (unsigned int) src & 15;
2 vuint8_t srcA = *(vuint8_t *)(src);
3 vuint8_t srcB = *(vuint8_t *)(src+16);
4 vuint8_t src = spu_or(spu_slqwbyte(srcA, perm_src),
5                       spu_rlmaskqwbyte(srcB, perm_src-16));
```

proposed in [49]. It performs the 8×8 2D IDCT in a sequential manner, i.e., it consumes and produces one sample per cycle. Their hardware unit was designed to be used in a streaming environment and therefore is not very suitable for a programmable media accelerator. Another difference with our approach is that we exploit DLP to reduce the latency of the IDCT operation.

Similar to our work, in [137] the authors proposed some techniques to speedup the Discrete Wavelet Transform (DWT) on SIMD architectures. DWT has a higher compression ratio but is computationally more complex than the DCT. The authors proposed to add a MAC instruction to the ISA, use the extended subwords technique, and use a special matrix register file.

4.4.5 Accelerating Unaligned Memory Accesses

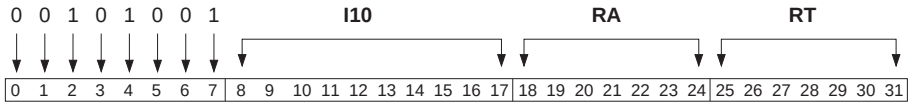
In the SPE architecture, accesses to the local store are quadword aligned. The four least significant bits of a memory address used in a load or store instruction are truncated. Thus, byte aligned memory addresses can be used but the LS store always returns an aligned quadword.

If a quadword has to be loaded or stored from an unaligned address this has to be done in several steps. Listing 4.11 shows an example of an unaligned load taken from the Luma kernel. First, two aligned quadwords are loaded, each containing a part of the targeted quadword. Next, the two quadwords are shifted left and right respectively to put the elements in the correct slot. Finally, the two words are combined using an OR instruction. The procedure for an unaligned store is even more complicated and involves two loads and two stores.

To overcome this limitation of the SPE architecture we implemented unaligned loads and stores. To assure backward compatibility of code the existing load and store operations were not altered. Instead, we added new instructions that perform the unaligned loads and stores. The details of these instructions are

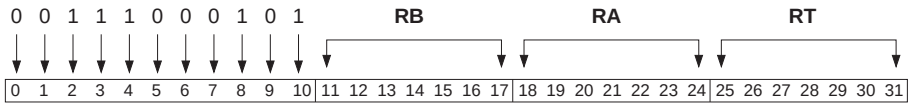
depicted below. We did not implement the unaligned version of the a-form Load and Store (L/S) instructions as we did not find those necessary.

lqdu rt,symbol(ra)



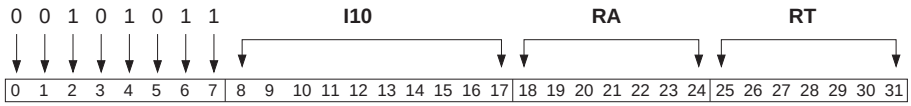
$LSA \leftarrow$ (RepLeftBit(I10 || 0b0000,32) + $RA^{0:3}$ & LSLR
$RT \leftarrow$ LocStor($LSA, 16$)

lqxu rt,ra,rb

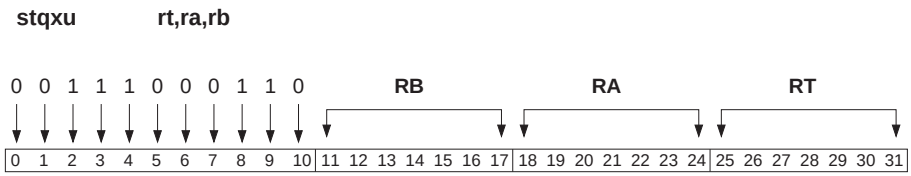


$LSA \leftarrow (RA^{0:3} + RB^{0:3})$ & LSLR
$RT \leftarrow$ LocStor($LSA, 16$)

stqdu rt,symbol(ra)



$LSA \leftarrow$ (RepLeftBit(I10 || 0b0000,32) + $RA^{0:3}$ & LSLR
LocStor($LSA, 16$) $\leftarrow RT$



LSA	$\leftarrow (RA^{0:3} + RB^{0:3})$ & LSLR
LocStor$(LSA, 16)$	$\leftarrow RT$

To allow byte aligned accesses, some modifications to the local store are required. A quadword might be aligned across two memory lines. Thus additional hardware is required to load or store the misaligned quadword from memory. To account for this we added one cycle to the latency of the local store.

The unaligned L/S instructions were made available to the programmer through inline assembly only. Modifying the compiler to automatically use the unaligned L/S instructions is a complex task beyond the scope of this thesis. We tried to use intrinsics but we could not make them work properly within reasonable time.

Analysis of the assembly code revealed that the compiler was not able to schedule the unaligned L/S instructions as optimal as it does for normal L/S instructions. Therefore, we optimized the scheduling of the unaligned L/S instructions manually. We did not hand optimize the scheduling of other instructions to keep the comparison fair.

Unaligned memory access has been applied before. In vector processing, of which SIMD is a sub class, there is often a mismatch between the data order used in memory and the data order required for computation. Therefore, most vector and DSP processors provide complex memory access that can be divided in three groups: unaligned access, strided access, and irregular access. In SIMD processors, due to the short vectors, mainly unaligned access is an issue. Multimedia SIMD extensions such as MMX, and SSE support unaligned accesses in hardware at the cost of some additional latency. In other SIMD architectures, such as Altivec/VMX, VIS, and 3DNow! alignment has to be performed in software, which is very costly in terms of latency.

Alignment issues on different multimedia SIMD architectures has been investigated in [136]. The paper describes several software approaches to minimize the overhead costs. The benefits of supporting unaligned access in hardware

is studied in detail in [23] for the Altivec/VMX architecture. Although the research is general, the authors give an example of how unaligned access could be implemented. Similar work is presented in [38, 37], but these focus more on the actual memory access architecture and compiler optimizations. The latest TriMedia processor, which was specialized for H.264 supports unaligned memory access.

4.5 Performance Evaluation

In the previous section we have presented several ISA enhancements to accelerate media applications, especially H.264 decoding. This section evaluates the performance improvement obtained with the proposed instruction on the kernels of our benchmark suite. Each kernel is analyzed by means of profiling, the identified deficiencies are described, and it is shown how the proposed architectural enhancements resolve the deficiencies.

4.5.1 Results for the IDCT8 Kernel

The first benchmark considered is the IDCT kernel, which is the smallest of all. It takes as input a macroblock of 16×16 16-bit elements, divides it in blocks of 8×8 or 4×4 . On each sub-block it performs a one-dimensional transformation first row-wise then column-wise, thus a two-dimensional transformation in total. The size of the sub-blocks depends on the mode of the macroblock. We will refer to these modes as IDCT8 and IDCT4, respectively and treat them as different kernels. This section describes the IDCT8 kernel while the next section describes the IDCT4 kernel.

The IDCT8 kernel was analyzed by splitting the kernel in parts and measuring the execution times taken by each part. The kernel consists of four 8×8 IDCTs, which are identical. In this analysis of the kernel we measured one 8×8 IDCT only. Table 4.3 presents the results. We remark again that this low-level profiling influences the compiler optimizations and thus the measured execution times. For the purpose of this analysis, however, it suffices to know the relative execution times.

We briefly describe the parts of the IDCT8 function. Although we profiled only one 8×8 IDCT, we included the overhead caused by the loop that starts the four 8×8 IDCTs needed to process the entire macroblock. This overhead includes calculating the pointers to the source and destination address of the data and the stride. The initialization of variables mainly consists of creating or loading

Table 4.3: Profiling of the baseline IDCT8 kernel. Both the original and the optimized version are depicted. No architectural enhancements are included.

	Original			Optimized		
	cycles		%	cycles		%
Loop overhead	33		6%	36		11%
Initialize variables	11		2%	11		3%
Load data & offset	18		4%	30		9%
Control		62	12%		77	24%
First transpose	28		5%	27		8%
First 1D IDCT	38		7%	38		12%
Second transpose	33		6%	32		10%
Second 1D IDCT	37		7%	37		11%
Shift right 6	12		2%	12		4%
IDCT		148	28%		146	45%
Calculate dst shift	8		2%	14		4%
Addition and store	300		58%	89		27%
Addition		308	60%		103	31%
Total		518	100%		326	100%

constants in the register file. Loading the data includes address computation but also adding an offset to the first element of the 8×8 matrix for rounding purposes. The real computation of the IDCT involves two transposes, two sequences of one-dimensional IDCTs, and finally a divide by 64 (or a shift right by 6 bits).

Another part of the kernel adds the result of the IDCT to the result of the prediction stage. In the original code this part consumes most of the execution time. Analysis of the trace files revealed that it has a very low IPC because the compiler is not able to schedule that part of the code in an efficient way. The *addition and store* part consists basically of one macro for each matrix row. The macro loads one row of the matrix containing the result of the prediction stage, aligns the data, unpacks it, adds the result of the IDCT to it, packs the result, and stores it in memory. This macro expands into twelve instructions that are completely serial and have many dependencies. The compiler schedules these blocks of twelve instructions sequentially resulting in an IPC of only 0.36 for this part. In other words, instructions are mainly stalled, waiting for the previous instruction to finish.

We optimized this part of the code by interleaving the instructions corresponding to the different blocks manually. This is possible as processing the different rows of the matrix is completely independent. Furthermore, we removed one unnecessary operation and combined a shift and a shuffle in one shuffle. These optimizations speed up this part of the kernel by a factor of three, as depicted in the right side of Table 4.3.

We chose to use the optimized version of the kernel as our baseline for evaluating architectural enhancements. If the non-optimized kernel would have been taken, it would be to easy to achieve speedup by architectural enhancements resulting in biased results.

Besides profiling the kernel, we also analyzed the assembly code of the kernel and the trace files generated by the simulator. By doing so we identified a number of deficiencies in the execution of the IDCT8 kernel. Below we present the identified deficiencies and show how the proposed architectural enhancements of Section 4.4 resolve these issues.

- Matrix transposes: In order to utilize the SIMD capabilities, each matrix has to be transposed twice. This takes a lot of time, about as much as the actual IDCT computation does. Matrix transposes can be accelerated using the swapoe instructions. Using the IDCT8 instruction, however, the matrix transposes are completely avoided.

- Scalar operations: Due to the lack of a scalar data path, a scalar computation that involves a vector element requires a lot of instructions. Scalars are stored in the preferred slot of a vector. If one of the operands is a vector element, it has to be shifted to the preferred slot, the computation has to be performed, and using a shuffle the result has to be stored in the original vector. The as2ve instructions allow to add a scalar to one vector element in one instruction.

- Lack of saturating arithmetic: The SPUs do not have saturating arithmetic and thus this has to be performed in software. This involves unpacking, adding, comparing, selecting, and packing. The asp instruction contains saturating arithmetic and can be used to avoid software saturation.

- Pack/unpack & alignment: Packing and unpacking has to be performed explicitly and is done using shuffle instructions. Moreover, most packing and unpacking require alignment which has to be performed explicitly

as well. Consider an unpack from eight bytes elements to halfwords elements. If the programmer cannot be sure whether the eight bytes are in the lower or upper part of the vector, he or she has to check the address, compute the shift and rotate the data. After that the unpack can be done. In the IDCT kernel this is the case. A part of this problem (mainly packing) is solved by the asp instruction. Alignment issues remain a deficiency.

Besides the enhancements mentioned above, also the sfxsh instructions are beneficial to the IDCT8 kernel. Table 4.4 shows the effect of the enhancements on the execution time and the instruction count of the IDCT8 kernel. First, it shows the effect of each of the enhancement separately. Finally, also the combination of all enhancements was analyzed. This set of enhancements excluded the swapoe instructions as they become obsolete when using the IDCT8 instruction, which eliminates the matrix transposes. For this analysis we run the IDCT kernel on one complete macroblock. We remark again that this kernel-level profiling does not influence the compiler optimizations and thus these results are accurate.

The speedup achieved by each enhancement separately is between 0.99 and 1.39. Not surprisingly, the IDCT8 instruction achieves the largest speedup, mostly due to avoiding matrix transposes. The swapoe instructions do not provide speedup in this kernel. The amount of ILP in the kernel is low and thus the critical path in the dependency graph determines the latency of the kernel. Although using swapoe instructions reduces the number of executed instructions, the critical path of the matrix transpose does not change. Therefore, the critical path of the entire kernel does not change and neither does the latency.

Table 4.4: Speedup and reduction in instruction count of the IDCT8 kernel using the proposed architectural enhancements. The effect of each separate enhancement is listed as well as the aggregate effect.

Enhancement	Cycles			Instructions		
	Baseline	Enhanced	Speedup	Baseline	Enhanced	Reduction
Swapoe	896	908	0.99	1103	988	1.12
IDCT8	896	645	1.39	1103	756	1.46
Sfxsh	896	840	1.07	1103	984	1.12
Asp	896	781	1.15	1103	928	1.19
As2ve	896	865	1.04	1103	1072	1.03
All (but swapoe)	896	488	1.84	1103	488	2.26

The speedup achieved when combining all architectural enhancements is 1.84. Note that also a reduction in the number of instructions of 2.26 is achieved. The latter is important for power efficiency, which has become a design constraint in contemporary processor design (see also Chapter 2). The achieved speedup is directly related to the reduction in instruction count, but does not necessarily has the same magnitude. For most kernels the speedup is lower than the instruction count reduction. This is expected since multiple shorter-latency instructions have been replaced by a single instruction with a longer latency. In other words, the latency of the arithmetic operations stays the same. For example, the latency of the IDCT8 instructions is assumed to be 8 cycles, because there are 4 simple fixed point operations on the critical path, each taking two cycles. The application-specific instructions we have introduced mainly reduce the overhead and eliminate stalls due to data dependencies.

4.5.2 Results for the IDCT4 kernel

Besides the IDCT8 kernel we also investigated the IDCT4 kernel. The IDCT4 kernel splits a MB into 16 sub-blocks of 4×4 pixels and performs the transform on these sub-blocks. We analyzed this kernel by profiling the execution time spent on the different parts. The IDCT4 kernel has the same execution steps as the IDCT8 and the explanation of those can be found in the previous section. As in the IDCT8 we measured the execution of one 4×4 IDCT only. Table 4.5 presents the results.

We optimized the code of the IDCT4 kernel in the same way we optimized the IDCT8 kernel. In the *addition and store* part we interleaved intrinsics in order to achieve a higher ILP. Furthermore, we removed one unnecessary operation and combined a shift and a shuffle in one shuffle. Again, these optimizations speed up this part of the kernel by a factor three as depicted in the right side of Table 4.5. For the same reasons as mentioned in the previous section we chose the optimized version as our baseline for evaluating architectural enhancements.

We identified the deficiencies in the execution of the IDCT4 kernel by analyzing the profiling results and a trace output. The identified deficiencies are similar as for the IDCT8 kernel, but there are some differences. Below we explain the deficiencies and show how the proposed architectural enhancements of Section 4.4 resolve these issues.

- Matrix transpositions: Because of the small matrix size the transpositions are less costly but more of them are required so they still take up

Table 4.5: Profiling of the baseline IDCT4 kernel. Both the original and the optimized version are depicted. No architectural enhancements are included.

	Original		Optimized	
	cycles	%	cycles	%
Loop overhead	33	10%	33	15%
Initialize variables	11	3%	11	5%
Load data & offset	29	9%	30	13%
Control	73	23%	74	33%
First transpose	22	7%	22	10%
First 1D IDCT	15	5%	16	7%
Second transpose	25	8%	24	11%
Second 1D IDCT	15	5%	16	7%
Shift right 6	11	3%	12	5%
Idct	88	27%	90	40%
Calculate dst shift	7	2%	9	4%
Addition and store	156	48%	53	24%
Addition	163	50%	62	27%
Total	324	100%	226	100%

about 20% of the execution time. Swapoe instructions are of no value for a 4×4 matrix as it fits entirely in two registers, and thus shuffle operations are sufficient. The IDCT4 instruction, on the other hand, completely removes all matrix transpositions.

- Unused SIMD slots: All SIMD computations are performed row or column-wise. As the matrix is 4×4 and the elements are 16-bit only half of the SIMD slots are used. The IDCT4 instruction does utilize all SIMD slots by exploiting the fact that an entire 4×4 matrix can be stored in two registers. This increases the efficiency of the computational part of the kernel.

- Scalar operations: Same as for IDCT8 kernel; see page 129 for a full description. The as2ve instructions were used to mitigate the effect of this deficiency.

- Lack of saturating arithmetic: Same as for IDCT8 kernel; see page 129 for a full description. The asp instruction was used to mitigate the effect of this deficiency.

- Pack, unpack & alignment: Same as for IDCT8 kernel; see page 130 for a full description. The asp instruction was used to resolve the pack/unpack deficiency. Alignment issues remain a deficiency.

Besides the enhancements mentioned above, we also used the unaligned L/S instructions. By doing this we avoided some branches that caused many stalls.

Table 4.6 shows what the effect of the enhancements is on the execution time and the instruction count of the IDCT4 kernel. First it shows what the effect is of each enhancement separately. Finally, also the combination of all enhancements was analyzed. For this analysis we run the IDCT4 kernel on one complete macroblock.

The speedup achieved by the enhancements is between 1.05 and 1.23. A synergistic effect happens when combining all enhancements such that the total speedup is larger than the multiplication of the separate speedups. The source of this synergistic effect seems to be the compiler optimization process. The speedup achieved when combining all architectural enhancements is 2.37 and the number of instructions is reduced by a factor 2.67.

4.5.3 Results for the Deblocking Filter Kernel

The Deblocking Filter (DF) kernel is applied to remove an artifact introduced by the IDCT. The IDCT operates on 4×4 or 8×8 sub-blocks and as a result square areas might become visible in the decoded picture, which is called the 'blocking' artifact. The DF kernel smoothes these block edges and thus improves the appearance of the decoded picture.

The process of deblocking a macroblock is depicted in Figure 4.12. Each

Table 4.6: Speedup and reduction in instruction count of the IDCT4 kernel using the proposed architectural enhancements. The effect of each separate enhancement is listed as well as the aggregate effect.

Enhancement	Cycles			Instructions		
	Baseline	Enhanced	Speedup	Baseline	Enhanced	Reduction
IDCT4	2524	2059	1.23	1858	1304	1.42
Asp	2524	2277	1.11	1858	1512	1.23
As2ve	2524	2407	1.05	1858	1756	1.06
Unaligned L/S	2524	2086	1.21	1858	1720	1.08
All	2524	1063	2.37	1858	696	2.67

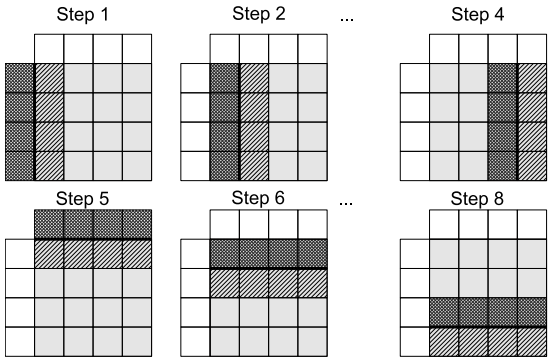


Figure 4.12: The process of deblocking one macroblock. Each square is a 4×4 sub-block. Also the left and top neighboring sub-blocks are involved in the process.

square represents a 4×4 sub-block. Note that also the left and top neighboring sub-blocks are depicted as they are involved in the filtering. The filter is applied on all edges, starting with the vertical ones from left to right followed by the horizontal ones from top to bottom. The filter is applied to each line of pixels perpendicular to the edge and with a range of four pixels to each side of the edge. The strength of the filter is determined dynamically based on certain properties of the macroblock, but more importantly on the pixel gradient across the edge. This adaptiveness decreases the available DLP, but still an efficient SIMD implementation for the Cell SPU is possible [29].

We analyzed the code of the DF kernel by profiling. Because the code of the kernel is not as small as the IDCT kernels we profiled the code on several levels, of which we present here the most important results.

Table 4.7 shows the profiling results of the top level, i.e., the function 'filter_mb_spu'. This function performs the filtering of one macroblock. Packing and unpacking is necessary because the data is stored in memory as bytes but processing is performed using halfwords. The matrix transposes are needed to rearrange the data such that it is amenable to SIMD processing. Analyzing the assembly code revealed that the loaded data was too large to be maintained in the register file during the entire execution. The compiler therefore generated a lot of loads and stores before and after each processing stage.

The total DF kernel consists of several functions. To find the most time-consuming parts we analyzed the number of function calls as well as the total amount of time spent in each function (including the functions that it calls).

Table 4.7: Top-level profiling of the DF kernel for one macroblock. Both the original and the optimized version are depicted. No architectural enhancements are included.

	Original		Optimized	
	cycles	%	cycles	%
Initialize variables	22	0%	22	0%
Load & unpack data	761	10%	761	10%
First transpose	451	6%	451	6%
Filter horizontal	2741	34%	2539	34%
Second transpose	448	6%	448	6%
Filter vertical	2748	34%	2546	34%
Pack and store data	806	10%	806	11%
Total	7977	100%	7573	100%

Table 4.8: Profiling of the functions of the DF kernel when filtering 4×4 macroblocks. The number of function calls and the time spent in the function is mentioned. Functions of level x only call functions of level $x + 1$.

	Calls	Time	Level
filter_mb_spu	16	100%	1
filter_mb_edgeh	240	53%	2
filter_mb_edgecv	112	18%	2
h264_v_loop_filter_luma_c	192	19%	3
h264_loop_filter_chroma	64	5%	3
h264_loop_filter_chroma_intra	48	1%	3
clip	704	5%	3
clip_altivec	640	4%	4
clip_uint8_altivec	512	3%	4

The results are depicted in Table 4.8 which also states the level of each function. Functions of level x only call functions of level $x + 1$. For this analysis we filtered an area of 4×4 macroblocks of the top-left corner of a frame.

A large amount of time is spent in the 'filter_mb_edgeh' function which mainly calls 'h264_v_loop_filter_luma_c'. The latter, however, is only 19% of the total execution time while the first is 53%. Further profiling revealed that in the first a lot of time is taken by table lookups which are time-consuming on the SPU because of alignment issues.

Table 4.8 also shows that 12% is spent in clip functions. The 'clip' and 'clip_altivec' functions are used to set the index of table lookups within the range of the table (mainly between 0 and 51). The 'clip_uint8_altivec' function is used to saturate a halfword between 0 and 255. This is necessary because the SPU does not support saturating arithmetic.

While analyzing the kernel we also looked for non-optimal code. In this kernel we did not find much to optimize. We did change some code that generated constants. The code was written in normal C and the compiler generated non-optimal assembly code. We rewrote the code using SPU intrinsics. The right side of Table 4.7 shows the profiling results of the optimized kernel. As with the previous kernels, we chose the optimized version as the baseline for evaluating architectural enhancements.

We identified a number of deficiencies in the execution of the DF kernel. Below they are presented and we show how the proposed architectural enhancements, proposed in Section 4.4, resolve these issues.

- Matrix transpositions: In order to utilize the SIMD capabilities each data matrix has to be transposed twice. This rearrangement overhead takes up about 12% of the total execution time of the DF kernel. As argued before, matrix transposition can be accelerated using the swapoe instructions.

- Data size: The total data size of the baseline kernel exceeds the size of the register file and therefore a lot of intermediate loads and stores are required. The main cause of the large data size is that the matrix transposes require twice as many registers: one set of registers for the original and one for the transposed. The swapoe instructions perform a matrix transpose in place and thus require half as many of registers. Simulation showed that using the swapoe instructions the entire data set fits in the register file and thus many loads and stores are prevented.

- Scalar operations: As with the previous kernels, scalar operations turn out to be very costly. In the DF kernel this manifests in the table lookups. Listing 4.1 of Section 4.4.1 shows that a table lookup costs five instructions because of alignment issues. Using the lds instructions a table lookup takes only two instructions.

- Lack of saturating function/arithmetic: The output of the DF kernel consists of unsigned bytes. Thus, the result of the filtering has to be saturated between 0 and 255. Many SIMD architectures offer saturating

arithmetic which performs saturation automatically after each arithmetic operation. In the DF kernel, however, halfwords are needed for the intermediate results anyway, and thus one saturate at the end suffices. Implementing many saturating instructions is costly in terms of opcodes. The clip instruction is in that sense a good solution as it provides a fast saturation with only one instruction per data type. Moreover, the clip instruction can also be used for other purposes such as limiting the index of a table lookup to the size of the table.

- Pack & unpack: In the SPE architecture, packing and unpacking has to be performed explicitly and is done using shuffle instructions. The asp or sat_pack instructions are of no help in the DF kernel as the saturate and the pack are not coupled together. Automatic (un)pack at load and store would be beneficial. As others have investigated this before [143, 42], we do not include this technique in our architectural enhancements.

- SIMDimization: Because of the adaptiveness of the filter, limited DLP is available. Five different filter strengths are applied depending on the smoothness of the edge. That means that across the SIMD slots different filters might be required. The SIMDimization strategy applied is to compute all five filters and to select the right result for each SIMD slot. Off course this requires extra computation. None of the proposed architectural enhancements resolves this issue.

Table 4.9 presents the speedup and instruction count reduction of the DF kernel using the proposed architectural enhancements. For this analysis we also applied the DF to the top left 4×4 macroblocks of a frame from a real video sequence. In contrast to the IDCT, for this kernel, the swapoe instructions have a large impact on the execution time, mainly due to avoiding loads and stores. To achieve this some modifications to the code were required such that the compiler could ascertain all data accesses at compiler time. These modifications included loop unrolling, substituting pointers for array indices, and inlining of functions.

Both the lds and clip instructions provide a moderate speedup separately but significant when all enhancements are applied concurrently. When combining all three enhancements, again a synergistic effect happens as the total speedup is larger than the multiplication of the separate speedups. The reduction in the number of instructions is proportional to the speedups. Combining all enhancements the number of instructions is reduced with a factor 1.75.

Table 4.9: Speedup and reduction in instruction count of the DF kernel using the proposed architectural enhancements. The effect of each separate enhancement is listed as well as the aggregate effect.

Enhancement	Cycles			Instructions		
	Baseline	Enhanced	Speedup	Baseline	Enhanced	Reduction
Swapoe	122135	90518	1.35	101480	74504	1.36
Lds	122135	108995	1.12	101480	89692	1.13
Clip	122135	114755	1.06	101480	96312	1.05
All	122135	66334	1.84	101480	58088	1.75

For the DF kernel the speedup is larger than the reduction in instructions for the following reason. When using the swapoe instructions it is possible to keep the entire macroblock in the register file. To achieve this some modifications to the code were required, such as the replacement of functions by macros. This reduces the number of branches significantly and with that the number of branch miss stall cycles. Specifically, in the baseline kernel there were 15,341 branch miss stall cycles, while for the enhanced kernel this was only 4,529.

4.5.4 Results for the Luma/Chroma Interpolation Kernel

The Luma and Chroma interpolation kernels together perform the motion compensation. The kernels are applied to blocks of 16×16, 16×8, 8×16, 8×8, 8×4, 4×8, or 4×4 pixels. Note that there is only one chroma sample per 2×2 pixels. Otherwise the two kernels are identical and thus we focus on the Luma kernel only.

The accuracy of the motion compensation is a quarter of the distance between luma samples. If the motion vector points to an integer position, the predicted block consists of the samples the motion vector points to in the reference frame. If the motion vector points to a half-sample position, the predicted block is obtained by interpolating the surrounding pixels. This is done by applying a one-dimensional 6-tap FIR filter, which is given by:

$$p[0]' = (p[-2] - 5p[-1] + 20p[0] + 20p[1] - 5p[2] + p[3]) + 16) >> 5. \quad (4.1)$$

This filter is applied horizontally and/or vertically. Prediction values at quarter-sample positions are generated by averaging samples at integer- and half-sample positions. For more information on the motion compensation of H.264 the reader is referred to [165].

Table 4.10: Profiling results of the baseline Luma kernel. Both the original and the optimized version are depicted. No architectural enhancements are included.

	Original		Optimized	
	cycles	%	cycles	%
Initialization of variables	44	1%	27	1%
Loop overhead	752	16%	896	21%
Loading, aligning, and unpacking	1630	36%	1546	37%
Computing interpolation filter	1024	23%	976	23%
Saturating and pack	512	11%	416	10%
Storing result	592	13%	368	9%
Total	4554	100%	4229	100%

The code of the Luma kernel contains many functions as there are many possible combinations of block sizes and quarter sample positions. For the analysis in this work we chose an input block size of 16×16 and a quarter sample prediction position that requires both horizontal and vertical interpolation. This filter function is one of the most compute intensive ones. The name of this filter function in the FFmpeg code is `put_h264_qpel16_mc11_spu`. The input for this kernel is one macroblock and thus the filter function is run once.

We analyzed the code of the Luma kernel by profiling it using CellSim. The filter function calls three subfunctions: `put_h264_qpel_v_lowpass_spu` performs vertical interpolation and consumes 33% of the total time, `put_h264_qpel_h_lowpass_spu` (52%) performs horizontal interpolation, and `put_pixels_16_l2_spu` (16%) averages the result of the first two functions and stores it at the destination address. We divided the functions into the six parts shown in Table 4.10 and measured how many cycles were spent in each part combined for all three functions. Only 22.5% of the cycles is spent on actual computation of the interpolation. Most of the time is taken by loading/storing and the corresponding rearrangement of data to make it amenable to SIMD processing.

While analyzing the kernel we also looked for code optimization possibilities. We performed three optimizations. The first optimization is in the horizontal and vertical interpolation functions. The original code used only the `spu_madd` intrinsic, which multiplies the odd elements only, and used shuffles to multiply even elements. We used the `spu_mule` and `spu_mulo` in

places where they are beneficial and thereby avoided some of the shuffles. We also developed a version that employed the `spu_mhhadd` instruction, which multiply-adds the even elements. It turned out not to provide any speedup.

The second code optimization was performed in the `put_pixels_16_l2_spu` function. The function computes the average of two 16×16 matrices of which it assumes that they are stored in memory at a non quadword aligned address. Therefore this function contains a lot of alignment overhead. In only a few cases, however, the input matrices are indeed stored non quadword aligned. In most cases the alignment overhead can be avoided. For the filter function that we chose as subject of this analysis this is always the case. Thus we created another version of this function called `put_pixels_16_l2_spu_aligned` and used that one in our filter function.

The third optimization was performed on all three functions mentioned. All these functions contain a loop that iterates over all the rows of the block being processed. The height of the block is a parameter of the function. Therefore, the loop count cannot be determined at compile time and thus the compiler does not perform loop unrolling optimizations. The loop bodies, however, are rather small and some contain very little ILP. Thus without loop unrolling a lot of dependency stalls occur. The height of the block can take only two values, however. Thus, we created two versions of each function, one for each block height. We applied this enhancement to the three functions only when beneficial. For example, for the baseline kernel we applied this optimization to the smallest subfunction only.

The two rightmost columns of Table 4.10 show the profiling results of the optimized kernel. The third code optimization, however, is not reflected in these results. The ILP increase it obtained was destroyed by the insertion of the volatile profiler commands. In total the optimized version is about 33% faster than the original one. As with the previous kernels, we chose the optimized version as the baseline for evaluating architectural enhancements.

We identified a number of deficiencies in the execution of the Luma kernel. Below we present them and show how the proposed architectural enhancements of Section 4.4 can resolve these issues.

- Unaligned loads: The Luma kernel loads the sub-block where the motion vector is pointing to. In most cases by far this motion vector points to a non-quadword aligned memory address. To load such a misaligned matrix into the register file, a lot of overhead is required. This main deficiency of the Luma kernel was resolved by using the unaligned load instructions.

- Multiplication overhead: The input and output of the Luma kernel is a matrix of bytes while computations are performed on halfwords in order not to lose precision. When a multiplication is performed, the instructions produce 32-bit results although the actual value never exceeds 15 bits. Thus a multiplication of one vector is performed as a multiplication of the odd elements, a multiplication of the even elements, and a shuffle to merge the odd and even elements back into one vector. This overhead can be avoided by either using the mpytr instructions or the mpy_byte instructions.

  The mpytr instructions multiply vectors of halfwords and produce vectors of halfwords. The mpy_byte instructions multiply vectors of bytes and produce vectors of halfwords. The mpy_byte instructions, therefore, allow performing the multiplication on the input data directly, which is in bytes. In general that will not always be possible, but for the Luma kernel it is. The additional benefit of the mpy_byte instructions is that they unpack the input data automatically, making the explicit unpacks superfluous.

  In the horizontal interpolation function the mpy_byte instructions provided highest speedup while in the vertical interpolation function the mpytr instructions did. Due to the way these functions are SIMDimized, the horizontal interpolation function contains many more unpack operations which are performed automatically by the mpy_byte instructions. The vertical interpolation function requires few unpack operations and does not benefit that much from the auto unpack of the mpy_byte instructions. Moreover, using the mpy_byte instructions more instructions are required for the vertical interpolation.

- Lack of saturation function/arithmetic: Just as the other kernels the result of the Luma kernel has to be saturated between 0 and 255 and packed to bytes. The clip instruction can be used for this purpose but the sat_pack instruction provides more benefit to the overall performance.

- Pack & unpack: Packing and unpacking has to be performed explicitly and is implemented using shuffle instructions. Automatic (un)pack during load and store would be beneficial, but is not investigated in this work, as mentioned before. The sat_pack instruction provides speedup as it combines a saturate and a pack.

Besides the architectural enhancements mentioned above, also the sfxsh instructions are beneficial to the Luma kernel, although very little. Table 4.11

Table 4.11: Speedup and instruction count reduction of the Luma kernel using the proposed architectural enhancements. The effect of each separate enhancement is listed as well as the aggregate effect.

Enhancement	Cycles			Instructions		
	Baseline	Enhanced	Speedup	Baseline	Enhanced	Reduction
Mpytr	2461	2238	1.10	2473	2163	1.14
Mpy_byte	2461	2175	1.13	2473	2056	1.20
Clip	2461	2333	1.05	2473	2311	1.07
Sat_pack	2461	2065	1.19	2473	2212	1.12
Sfxsh	2461	2375	1.04	2473	2409	1.03
Unaligned L/S	2461	1915	1.29	2473	1936	1.28
All (but Clip)	2461	1068	2.30	2473	1059	2.34

presents the speedup and instruction count reduction of the Luma kernel using the proposed architectural enhancements. For this analysis we run the Luma kernel on one entire macroblock.

The mpytr, mpy_byte, and sat_pack instructions provide a moderate speedup. The clip and sfxsh instructions are used only a few times in the Luma kernel and therefore also provide a small speedup. The unaligned L/S instructions provide the highest speedup as loading, storing, and alignment consumes a large part of the total execution time of the baseline kernel. Combining all enhancements, the clip instruction is not used as the saturation it performs is done by the sat_pack instructions. The total speedup achieved is 2.30, while the instruction count is reduced by a factor of 2.34.

4.5.5 Summary and Discussion

To summarize, Figure 4.13 and Table 4.12 provide an overview of all architectural enhancements and modifications and the effect on the execution time and instruction count of all the kernels. For each kernel a significant speedup was obtained using a handful of specialized instructions only.

The performance improvements come at the cost of additional area. Most instructions require a few additions to the data path of existing pipelines and the corresponding control logic. For example, the as2ve (Add Scalar to Vector Element) instructions require the scalar to be replicated among the SIMD slots, while the control logic selects which slot adds the scalar input to the vector input. The sat_pack (Saturate & Pack) and asp (Add, Saturate, & Pack)

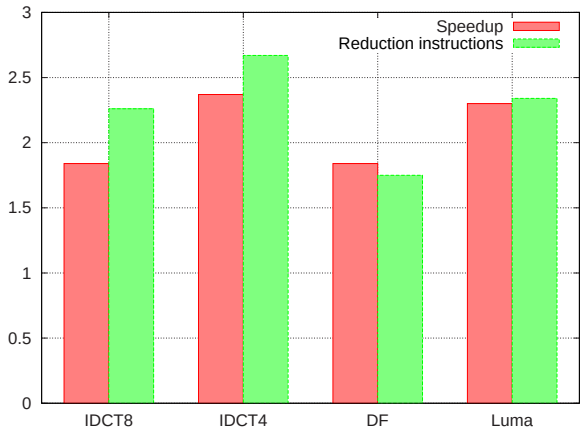


Figure 4.13: Speedup and instruction count reduction achieved on the H.264 kernels.

instructions require the pipeline to be extended with logic for saturation and pack.

The swapoe (Swap Odd-Even) instructions have two output operands. These could be written in the register file serially, if one port is available. To write the results to the register file in one cycle, two ports are needed. Either the port of the other pipeline is used, or an additional port is implemented. In any case, the forwarding network has to be extended, requiring significant area.

The IDCT8 and IDCT4 instructions most likely implemented in a special intra-vector unit. In the next chapter this is described in more detail.

Finally, two instructions require some modifications to the local store. First, the unaligned L/S instructions require the local store to shift the data into the right position. Furthermore, the required data might be stored across two memory lines. This required more logic to handle. The lds (LoaD Scalar) instruction requires similar shift operations. Thus, the logic required for these two instructions possibly can be combined.

Table 4.12: Overview of the effect of all architectural enhancements and modifications on the execution time and instruction count of the kernels.

	Speedup				Reduction instructions			
	IDCT8	IDCT4	DF	Luma	IDCT8	IDCT4	DF	Luma
As2ve	1.04	1.05			1.03	1.06		
Asp	1.15	1.11			1.19	1.23		
Clip			1.06	1.05			1.05	1.07
IDCT4		1.23				1.42		
IDCT8	1.39				1.46			
Lds			1.12				1.13	
Mpytr				1.10				1.14
Mpy_byte				1.13				1.20
Sat_pack				1.19				1.12
Sfxsh	1.07			1.04	1.12			1.03
Swapoe	0.99		1.35		1.12		1.36	
Unaligned L/S		1.21		1.29		1.08		1.28
Total	1.84	2.37	1.84	2.30	2.26	2.67	1.75	2.34

4.6 Conclusions

In this chapter we described the specialization of the Cell SPE for H.264 decoding. The SPE architecture is an excellent starting point for this specialization as it has a large register file, a local store, and a DMA-style of accessing global memory. This is very suitable for H.264 decoding because most data accesses it performs are know in advance. We enhanced the architecture with twelve instruction classes, most of which have not been reported before, and which can be divided in five groups. First, we added instructions to enable efficient scalar-vector operations, because the SPE has a SIMD only architecture. Second, we added several instructions that perform saturation and packing as needed by the H.264 kernels. Third, we added the `swapoe` instructions to perform fast matrix transposition, which is needed by many media kernels. Fourth, we added specialized arithmetic instructions that collapse frequently used simple arithmetic operations. Finally, we added support for non-quadword aligned access to the local store, which is especially needed for motion compensation. The speedup achieved on the H.264 kernels is between 1.84x and 2.37x, while the dynamic instruction count is reduced by a factor between 1.75x and 2.67x. The largest performance improvements are achieved by the `swapoe` instructions because it allowed more efficient usage of the register file, by the `IDCT` instructions because they completely avoid matrix transposes, and by the unaligned load/store instructions because of the large overhead reduction.

The IDCT instructions proposed in this chapter perform intra-vector operations. In general SIMD instructions perform inter-vector operations, i.e., they perform an operation on elements that are in the same slot, but in different vectors. The next chapter investigates if such intra-vector SIMD instructions are beneficial for other kernels and other application domains as well.

Chapter 5

Intra-Vector Instructions

In the previous chapter we proposed the SARC Media accelerator to achieve performance scalability by specialization. Among the newly proposed instructions were two intra-vector instructions. Those instructions operate on data elements within a vector, in contrast to normal SIMD operations that perform operations among elements that are in the same SIMD slot of different vectors. Those intra-vector instructions were designed because we noticed that two-dimensional kernels often cannot fully exploit the DLP offered by the SIMD unit. For vertical processing, the data is aligned in a fashion that fits SIMD processing. Horizontal processing, though, requires significant data rearrangement before SIMD processing can be applied. This overhead was reduced by using intra-vector instructions that break with the strict separation of SIMD lanes.

Intra-vector instructions can be found in several commercial processors. Most general purpose SIMD extensions, such as SSE and AltiVec, have some limited form of intra-vector instructions. They range from non-arithmetic operations such as permute, pack, unpack, and merge, to arithmetic operations such as dot product, sum across, and extract minimum. Intra-vector instructions are more common in the DSP area. NXP's EVP [160, 115] has intra-vector processing capabilities that allows, for example, arbitrary reordering of data within a vector as needed for FFT butterflies, pilot channel removal, and other computations common in communication signal processing. The Sandblaster architecture [117] has several intra-vector reduction operations, such as the multiply and sum operations. Furthermore, there are intra-vector instructions for Viterbi and Turbo decoding. More general intra-vector permutations are provided in the CEVA-XC [8], Lucent's 16210 [163], and SODA [101] architectures.

Most of the works above target the communication domain. The only intra-vector instructions that can be applied to other application domains are general purpose and contain a limited form of intra-vector operations. In this chapter we analyze the applicability of intra-vector instructions to kernels in the media domain. Furthermore, we apply intra-vector instructions to solve the data rearrangement problem in two-dimensional kernels, which, to the best of our knowledge, has not been done before.

This chapter is organized as follows. The experimental setup is described in Section 5.1. Next, in Section 5.2, the application and evaluation of intra-vector instructions in several kernels is presented. Section 5.3 discusses the results while Section 5.4 concludes this chapter.

5.1 Experimental Setup

As in the previous chapter, the Cell SPE core was used as baseline architecture and simulation was used to evaluate the performance of the enhanced kernels. For more details on the SPE architecture, the reader is referred to Section 4.2. The simulator and compiler used throughout this chapter are also equivalent to those used in the previous chapter. We describe them briefly here, but focus on issues related to this work. More details on the simulator and the compiler can be found in Section 4.3.

5.1.1 Simulator

CellSim [7] was used as the simulation platform. Although it models the architecture rather accurately, it does not distinguish between the odd and even pipeline. Instead, the simulator provides a parameter (IPC) that controls the instruction fetch rate. By default, this parameter is set to two, which is the maximum fetch rate. In most applications the fetch rate is lower, and therefore this parameter should be adjusted accordingly.

To determine the correct parameter value, we compared the execution times measured in CellSim with those obtained with the IBM full system simulator SystemSim. We did not use the real processor as SystemSim provides more statistics and is accurate as well. Those additional statistics allowed us to check the correctness of CellSim on more aspects than just execution time. We observed that an instruction fetch rate of 1.4 in CellSim corresponds best with the behavior of the real processor. For one highly optimized kernel, a value of 1.7 was used though. Table 5.1 shows the execution times of all the kernels using

Table 5.1: Validation of CellSim.

	IPC	Cycles		
Kernel	(fetch rate)	SystemSim	CellSim	Error
IDCT8	1.4	917	896	-2.3%
IDCT4	1.4	2571	2524	-1.9%
DF	1.4	121465	122122	0.5%
IPol	1.4	2534	2598	2.5%
MatMul	1.7	84180	85894	2.0%
DWT	1.4	962640	994541	3.2%

both SystemSim and CellSim. It shows that the difference in execution time is at most 3.2%. Comparison of other statistics generated by the two simulators also showed that CellSim is well suitable for its purpose.

5.1.2 Compiler

The compiler used throughout this work is spu-gcc version 4.1.1. The implemented intra-vector instructions were made available to the programmer through intrinsics or inline assembly in case intrinsics would require complex modifications. The modifications assured that the compiler can handle the new instructions, but not necessarily in an optimized way. Optimizing the compiler for the new instructions is beyond the scope of this project.

5.1.3 Benchmark

We used six different kernels to demonstrate the effectiveness of intra-vector instructions. The kernels we used were either develop in-house before or were taken from CellBench [5], which is a collection of publicly available Cell kernels. We examined all kernels available, searching for opportunities to apply intra-vector instructions. That is, we looked at inefficiencies of the SIMDimization and inefficiencies in the implementation of the algorithm. Devising intra-vector instructions for a kernel requires a thorough understanding of the algorithm the kernel implements. Six kernels seemed amenable to the intra-vector paradigm, but we expect experts to be able to implement intra-vector instructions in many more kernels. The kernels used throughout this work are IDCT8, IDCT4, IPol (InterPolation), and DF (Deblocking Filter), originating from the FFmpeg H.264 video decoder [58]. Furthermore, we

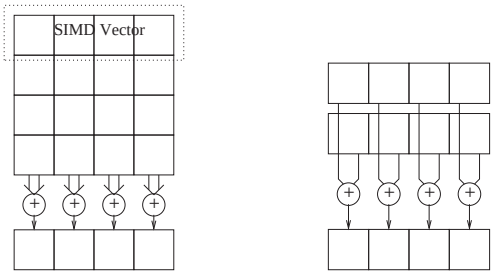


Figure 5.1: Example of vertical processing of a matrix. Adding the elements of each column (a) can efficiently be performed with traditional SIMD instructions, such as the `vector-add` instruction (b).

used the matrix multiply (MatMul) kernel from the Cell SDK. The last kernel is DWT (Discrete Wavelet Transform), taken from JasPer JPEG-2000 for Cell [6].

5.2 The Intra-Vector SIMD Instructions

The principle of intra-vector instructions is well explained by the simple example of Figures 5.1 and 5.2. In both figures the elements of a matrix are processed. The matrix is stored in row-major order and, in this case but not necessarily, one row fits in one SIMD vector. Figure 5.1(a) shows a vertical processing, i.e., the elements of each column are added. Such a computation can efficiently be performed by traditional SIMD instructions, as the `vector-add` depicted in Figure 5.1(b).

In Figure 5.2(a), the matrix is processed in horizontal mode, i.e., the elements of each row are added. To perform this computation using traditional SIMD instructions, the data has to be rearranged, for example by a matrix transposition. To avoid this overhead, intra-vector instructions can be used, such as the AltiVec `sum-across` instruction depicted in Figure 5.2(b).

Intra-vector instructions are not limited to matrix oriented kernels. The example shows however, that kernels that operate on matrices and perform both vertical and horizontal processing are most likely to be amenable to the intra-vector paradigm. Also, the benefit of intra-vector instructions is not only avoiding rearrangement overhead as we will show in this section. Finally, we remark that the intra-vector instructions presented in this work are purposely

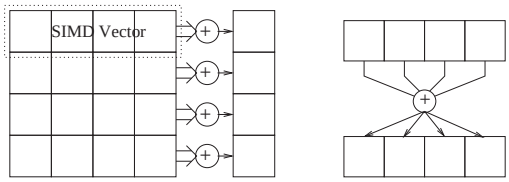


Figure 5.2: Example of horizontal processing of a matrix. Adding the elements of each row (a) cannot efficiently be performed with traditional SIMD instructions. Intra-vector instructions, such as the AltiVec `sum-across` instruction (b) do allow efficient computation.

application specific and thus more complex than the `sum-across` instruction of the AltiVec ISA.

For six kernels intra-vector instructions were developed and evaluated. The next sections describe the intra-vector instructions for the kernels in detail and evaluates performance. As the intra-vector IDCT8 and IDCT4 instructions were presented in detail in the previous chapter, for those only the results are discussed briefly. Figure 5.3 shows the speedup and instruction count reduction of all the kernels. Figure 5.4 shows a breakdown of the execution time of the kernels. The following sections refer to those graphs.

5.2.1 IDCT8 Kernel

The Inverse Discrete Cosine Transform (IDCT) is an operation often found in picture and movie processing. The IDCT is typically performed on 8×8 or 4×4 pixel blocks. Performing the IDCT comprises a one-dimensional IDCT row-wise followed by a one-dimensional IDCT column-wise. FFmpeg uses the LLM IDCT algorithm [105], which consists of simple additions, subtractions, and some shifts.

Using the intra-vector `IDCT8` instruction, presented in detail in Section 4.4.4, the row-wise IDCT can be performed without applying a matrix transposition. The column-wise IDCT is done with the normal SIMD instructions.

Figure 5.3 shows that a speedup of 1.39 was achieved by using the `IDCT8` instruction. Avoiding the matrix transpose was the main contributor to the speedup followed by instruction collapsing. The number of executed instructions was reduced by 1.46X.

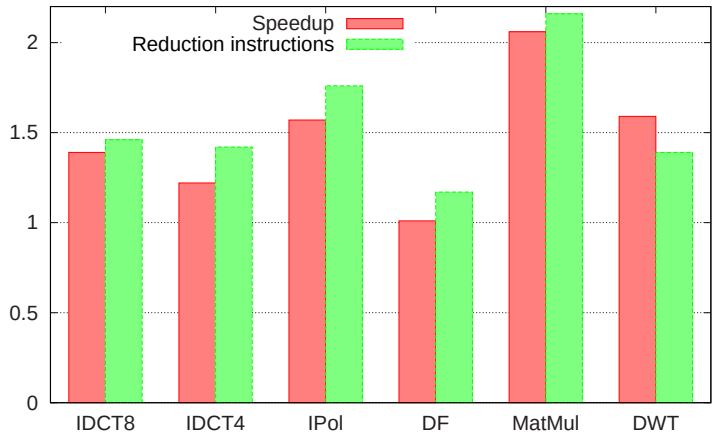


Figure 5.3: Speedup and instruction count reduction achieved using intra-vector instructions.

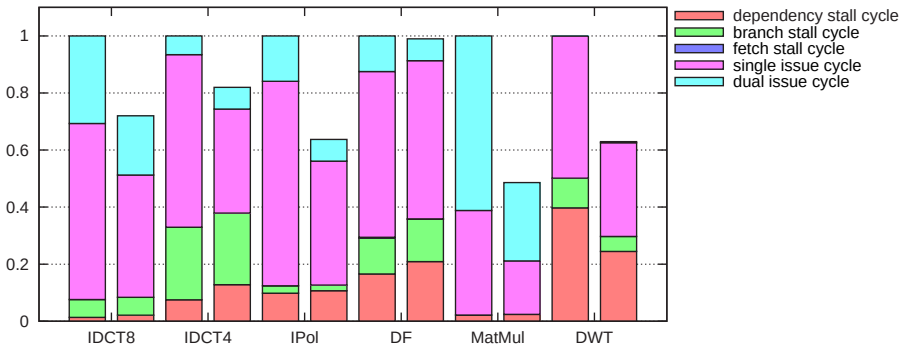


Figure 5.4: Breakdown of the execution cycles normalized to the baseline kernel. For each kernel, the left is the baseline while the right is the enhanced version using intra-vector instructions.

5.2.2 IDCT4 Kernel

The IDCT4 kernel is similar to the IDCT8 kernel but operates on a 4×4 pixel block. The computational structure of the one-dimensional IDCT4 is even simpler than that of the IDCT8. Moreover, as a 4×4 matrix of halfwords completely fits in two registers of 128-bits, the entire two-dimensional IDCT4 could be performed using one instruction with two operands. This intra-vector `IDCT4` instruction reads the matrix from its two operand registers, performs the arithmetic operations, and stores the result back in the two registers. Storing the result could be done in one cycle if the the register file has two write ports, or otherwise two cycles are required. The `IDCT4` instruction is described in detail in Section 4.4.4.

Figure 5.3 shows that a speedup of 1.22 was achieved. Again, avoiding the matrix transposition was the main contributor to the speedup followed by instruction collapsing. The number of executed instructions was reduced by 1.42X. Compared to the `IDCT8` instruction, a significant lower speedup is obtained, although the instruction count reduction has only decreased a bit. This is because the speedup of the IDCT4 kernel was limited by dependencies. The inner loop of the kernel is small and by replacing multiple instructions with the `idct4` intra-vector instruction, the few instructions left were highly depending. Figure 5.4 shows that the dependency stall cycles nearly doubled.

5.2.3 Interpolation Kernel (IPol)

Motion compensation in H.264 has quarter pixel accuracy. In case a motion vector points to an integer position, the corresponding pixel is the predicted value. Otherwise, the prediction is obtained by interpolation of the neighboring pixels. Figure 5.5 illustrates this interpolation process. Pixels at half sample position are obtained by applying a 6-tap FIR filter as follows:

$$b = (20(G + H) - 5(F + I) + E + J + 16) >> 5 \tag{5.1}$$

$$h = (20(G + M) - 5(C + R) + A + T + 16) >> 5 \tag{5.2}$$

The result of the filter is clipped between 0 and 255. Pixels at quarter sample position are obtained by averaging the nearest half and full sample pixels. Thus for every non integer motion vector, at least one 6 tap FIR filter has to be applied.

This filter is one of the most time-consuming parts of the motion compensation kernel. Both the horizontal and the vertical filter can be SIMDimized to exploit

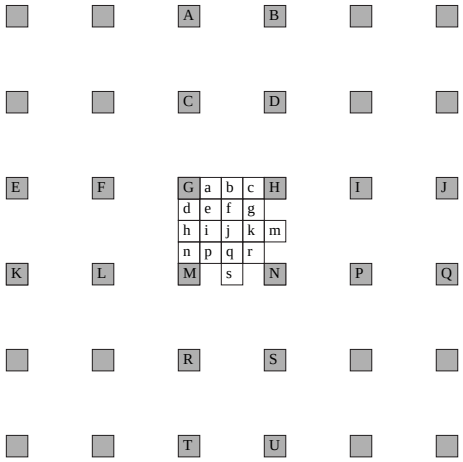


Figure 5.5: Pixels at fractional sample position (lower case letters) are determined by applying an interpolation filter on the full sample pixels (upper case). Picture taken from [165].

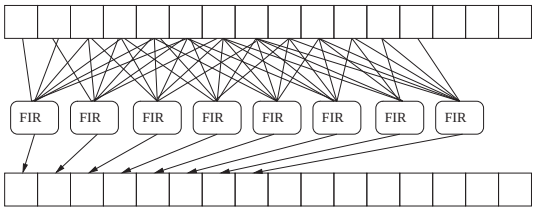


Figure 5.6: The `ipol` instruction computes eight FIR filters in parallel.

DLP, however, the first incurs considerable more overhead. When applying the horizontal filter, the six input elements are located consecutive in memory and thus and intra-vector instruction could compute the entire filter at once.

Therefore, we propose the `ipol` instruction. It computes eight 6 tap FIR filters as indicated in Figure 5.6. Both the input and output operands are bytes. The computational structure of each FIR filter is depicted in Figure 5.7. The multiplications are exchanged for shifts and additions to decrease latency. The result of the filter is also saturated between 0 and 255 in order to be able to store the results as bytes. Note, that although the input and output operands of the instruction are bytes, internally halfwords have to be used in order not to lose precision.

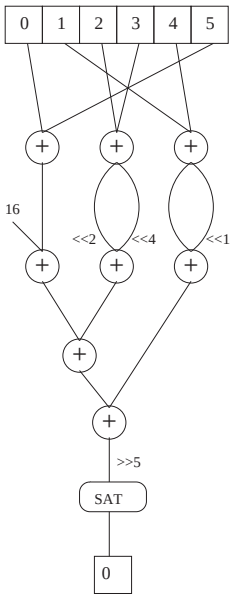
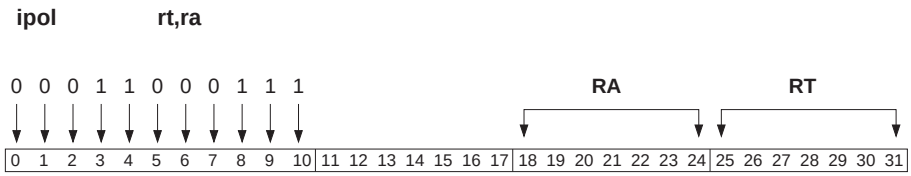


Figure 5.7: The computational structure of the FIR filter.

The `ipol` instruction fits the RR format where the RB slot is not used. Register RA is the input register while RT is the output register. The instruction details are depicted below.



for $j = 0$ to 5
$\quad r \leftarrow \quad 20 * (RA^{(j+2)::2} + RA^{(j+3)::2}) - 5 * (RA^{(j+1)::2} + RA^{(j+4)::2})$
$\qquad\quad +RA^{j::2} + RA^{(j+5)::2} + 16$
$\quad r \leftarrow \quad r >> 5$
$\quad$ if $r > 255$ then $\quad r \leftarrow 255$
$\quad$ if $r < 0$ then $\quad r \leftarrow 0$
$\quad RT^{j::2} \leftarrow r$
end

Table 5.2: Speedup and instruction count reduction obtained with the `ipol` instruction in the IPol kernel.

	Baseline	Enhanced	Improvement	
Cycles	2390	1523	867	1.57
Instructions	2473	1402	1071	1.76

The `ipol` instruction was made available to the programmer through an intrinsic. The instruction was added to the simulator and the latency was set to ten cycles for the following reason. Fixed point operations take two cycles in the SPU architecture. The `ipol` instruction performs four steps of simple fixed point operations. We added another two cycles for the final shift and saturation, making a total of ten.

We applied the `ipol` instruction to the IPol kernel. This kernel consists of many functions, each for a different combination of block size and quarter sample position. The `ipol` instruction can be used in any of those functions where a horizontal filter is applied, irrespective of the block size. We evaluated the `ipol` instruction with the function that computes the predicted value for a block of 16×16 pixels with a motion vector at quarter sample position for both the x and y coordinates. This function computes both a horizontal and vertical filter and is a good representative.

The speedup and instruction count reduction for the IPol kernel is shown in Table 5.2. A speedup of 1.57 is achieved, on the total two-dimensional kernel. This speedup is mainly achieved by reducing the number of instructions by 1.76 times. Each `ipol` instruction replaces 67 instructions from the baseline kernel, including the following:

- Creation of auxiliary variables, such as vector of constants and masks for shuffling.
- Aligning 6 input vectors (requiring some branches), instead of one.
- Unpack from bytes to halfwords and pack vice versa.
- Computation: 8 multiply-adds (each requiring a pack from words to halfwords), 8 additions, 2 subtractions, 2 shifts, and 8 instructions for saturation.

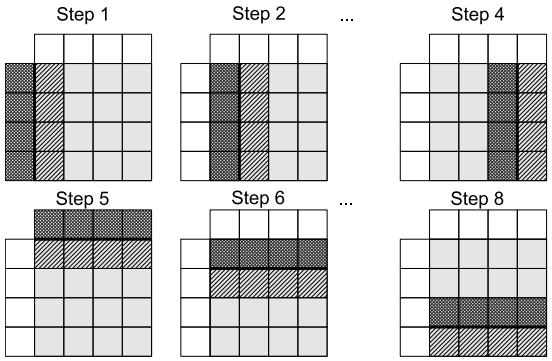


Figure 5.8: The process of deblocking one macroblock. Each square is a 4×4 sub-block. Also the left and top neighboring sub-blocks are involved in the process.

Note that the `ipol` instruction is only applied in the horizontal mode, and therefore replacing 67 instructions with one, results in an instruction count reduction of 1.76 and not higher.

The speedup of the IPol kernel is lower than the instruction count reduction, which is the case for all but one kernel. The reason is the following. Multiple instructions are replaced by one. This decreases the code size but also the average distance between dependent instructions. Thus, there is less opportunity to schedule two instructions in the same cycle, or dependency stalls might even occur. This is reflected in Figure 5.4 which shows a breakdown of the execution cycles. It shows that the enhanced IPol kernel has much less dual issue cycles than the baseline.

5.2.4 Deblocking Filter Kernel (DF)

The deblocking filter kernel is applied to remove an artifact introduced by the IDCT. The latter operates on 4×4 or 8×8 sub-blocks and as a result square areas might become visible in the decoded picture, which is called the 'blocking' artifact. The DF kernel smoothes these block edges and thus improves the appearance of the decoded picture.

The process of deblocking a macroblock is depicted in Figure 5.8. Each square represents a 4×4 sub-block. Also the left and top neighboring sub-blocks are depicted as they are involved in the filtering. The filter is applied on all edges, starting with the vertical ones from left to right followed by the horizontal ones

Listing 5.1: Serial code of the df_y1 filter function of the DF kernel.

```
1  if(abs(p0 - q0) < alpha &&
2     abs(p1 - p0) < beta &&
3     abs(q1 - q0) < beta) {
4
5      if(abs(p0 - q0) < ((alpha >> 2) + 2)){
6          if(abs(p2 - p0) < beta) {
7              p0 = (p2 + 2*p1 + 2*p0 + 2*q0 + q1 + 4) >> 3;
8              p1 = (p2 + p1 + p0 + q0 + 2) >> 2;
9              p2 = (2*p3 + 3*p2 + p1 + p0 + q0 + 4) >> 3;
10         } else {
11             p0 = (2*p1 + p0 + q1 + 2) >> 2;
12         }
13         if(abs(q2 - q0 ) < beta) {
14             q0 = (p1 + 2*p0 + 2*q0 + 2*q1 + q2 + 4) >> 3;
15             q1 = (p0 + q0 + q1 + q2 + 2) >> 2;
16             q2 = (2*q3 + 3*q2 + q1 + q0 + p0 + 4) >> 3;
17         } else {
18             q0 = (2*q1 + q0 + p1 + 2) >> 2;
19         }
20     } else{
21         p0 = (2*p1 + p0 + q1 + 2) >> 2;
22         q0 = (2*q1 + q0 + p1 + 2) >> 2;
23     }
24 }
```

from top to bottom. The filter is applied to each line of pixels perpendicular to the edge and with a range of four pixels to each side of the edge. The strength of the filter is determined dynamically based on certain properties of the macroblock, but more importantly on the pixel gradient across the edge.

Listing 5.1 depicts the serial code of one of the two filter functions (referred to as df_y1) for the luma components. The DF kernel also contains two filter functions for the chroma components. The function of which the code is depicted is the most complex one. Loading and storing of data is not included. The variables `alpha` and `beta` are inputs of the filter function. The pixels are denoted as `{p3,p2,p1,p0,q0,q1,q2,q3}`, where `px` is a pixel on the left (top) side of the edge, while `qx` is a pixel on the right (bottom) side of the edge, assuming vertical (horizontal) filtering.

An efficient SIMD implementation for the Cell SPU was created by Azevedo [29]. The SIMDimized version only performs filtering in horizontal mode, thus the macroblock had to be transposed in order to perform vertical filtering. The transposition overhead is approximately 12% of the execution time of the kernel. The procedure using the SIMD implementation is as follows: transpose all sub blocks involved, perform horizontal filtering, transpose all sub blocks back, perform horizontal filtering again.

Using intra-vector instructions for the vertical filtering, this transposition overhead can be avoided. Furthermore, instruction collapsing can be applied, reducing the time spent in computation. The procedure for the DF kernel using intra-vector instructions is the following: perform vertical filtering using intra-vector instructions, perform horizontal filtering using the SIMDimized filter functions.

We implemented several intra-vector instructions for the DF kernel. As said before, the kernel contains two filter functions for the luma components and two for the chroma components. All functions are similar although these for the luma components are a bit more complex. We started implementing intra-vector instructions for the luma filter functions and, as we show later, found out that those do not provide any speedup. Therefore, we did not implement intra-vector instructions for the chroma functions. We do report on those for the luma functions as it provides insight in conditions for effective usage of intra-vector instructions.

The two luma filter functions are referred to as df_y1 and df_y2. The corresponding ffmpeg function names are ‘filter_mb_edgeh’ and ‘h264_loop_filter_luma_c’, respectively. The functions differ to much to utilize the same intra-vector instructions. Separate instructions were implemented instead, with the beginning of its mnemonic indicating for which function it can be used.

The computation structure of the df_y1 function is depicted in Figure 5.9. This entire set of operations could be implemented in one instruction, but at the cost of area and latency. Moreover, the latency might be larger than the depth of the current SPU pipeline. In such case, the number of forwarding stages will have to be increased, again at the cost of area. Instead of using one instruction for the entire filter function, the set of operations could be split in parts and implemented in different instructions. We decided to explore both options, referring to the first as ‘complex’ and the latter as ‘simple’.

Using the complex intra-vector instructions, the code of the filter functions becomes rather simple and short, although some overhead is introduced to

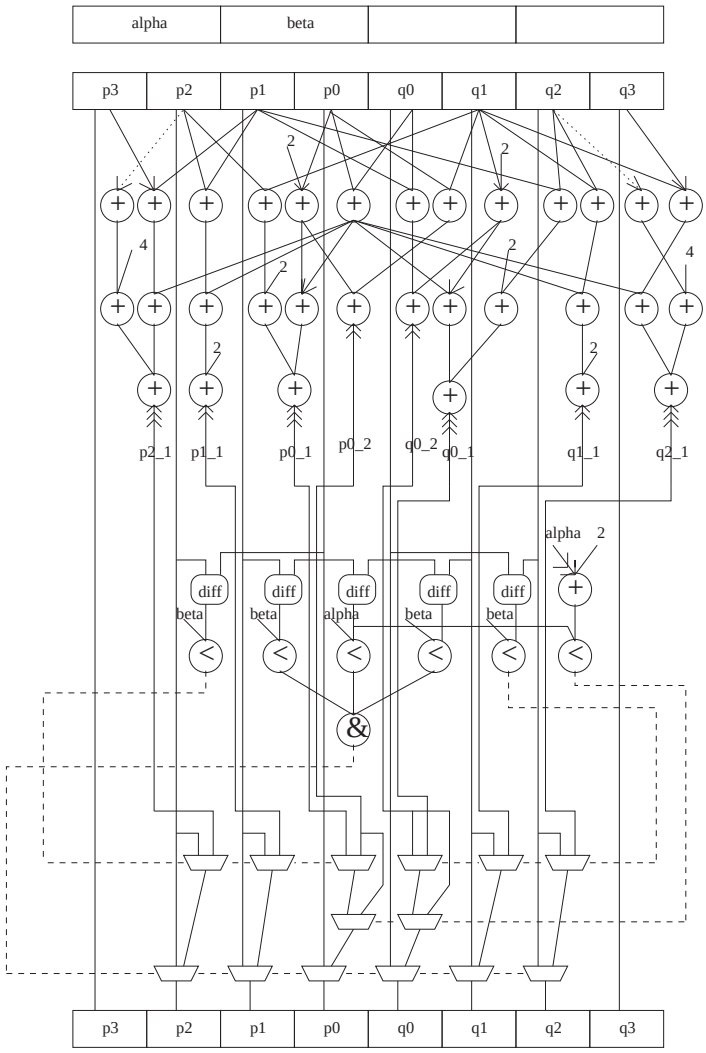
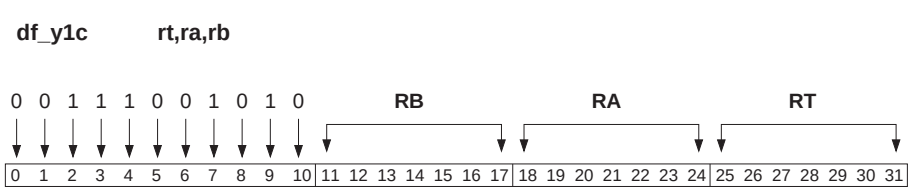


Figure 5.9: Computational structure of the df_y1 filter function. The `df_y1c` instruction implements this entire computation. An arrow done means a shift right by one bit while arrows up represent shifts left. A dotted line means the two input operands are the same.

rearrange the input data amenable to the new instructions. As a clarification, the code the of the df_y1 function is presented in Listing 5.2. Pixels are stored, in this case, as vectors of halfwords. As each vector contains eight pixels, data is stored in memory as 8×8 matrices and the functions filter eight rows or

columns. In half of the cases, the pixels needed are stored within one 8×8 matrix (pointed to by `*pix1`). In the other cases, however, the pixels are stored across two matrices (pointed to by `*pix1` and `*pix2`) and shuffles are required to combine the input pixels for each row into one vector and split the output pixels to store them in the original position. As the evaluation will show, this overhead is large.

The `df_y1c` and `df_y2c` instructions both fit the RR instructions format. Operand RA contains the pixels, while RB contains the parameters `alpha` and `beta`. The details of the `df_y1c` instruction are depicted below.



$t \leftarrow RB^{0::4}$
$u \leftarrow RB^{5::4}$
if(abs$(RA^{6::2} - RA^{8::2}) < t$ &&
abs$(RA^{4::2} - RA^{6::2}) < u$ &&
abs$(RA^{10::2} - RA^{8::2}) < u$) then
if(abs$(RA^{6::2} - RA^{8::2}) < ((t >> 2) + 2)$) then
if(abs$(RA^{2::2} - RA^{6::2}) < u$) then

$$RT^{6::2} \leftarrow (RA^{2::2} + 2 * RA^{4::2} + 2 * RA^{6::2} + 2 * RA^{8::2} + RA^{10::2} + 4) >> 3$$

$$RT^{4::2} \leftarrow (RA^{2::2} + RA^{4::2} + RA^{6::2} + RA^{8::2} + 2) >> 2$$

$$RT^{2::2} \leftarrow (2 * RA^{0::2} + 3 * RA^{2::2} + RA^{4::2} + RA^{6::2} + RA^{8::2} + 4) >> 3$$

else $RT^{6::2} \leftarrow (2 * RA^{4::2} + RA^{6::2} + RA^{10::2} + 2) >> 2$
if(abs$(RA^{12::2} - RA^{8::2}) < u$) then

$$RT^{8::2} \leftarrow (RA^{4::2} + 2 * RA^{6::2} + 2 * RA^{8::2} + 2 * RA^{10::2} + RA^{12::2} + 4) >> 3$$

$$RT^{10::2} \leftarrow (RA^{6::2} + RA^{8::2} + RA^{10::2} + RA^{12::2} + 2) >> 2$$

$$RT^{12::2} \leftarrow (2 * RA^{14::2} + 3 * RA^{12::2} + RA^{10::2} + RA^{8::2} + RA^{6::2} + 4) >> 3$$

else $RT^{8::2} \leftarrow (2 * RA^{10::2} + RA^{8::2} + RA^{4::2} + 2) >> 2$
else
$RT^{6::2} \leftarrow (2 * RA^{4::2} + RA^{6::2} + RA^{10::2} + 2) >> 2;$
$RT^{8::2} \leftarrow (2 * RA^{10::2} + RA^{8::2} + RA^{4::2} + 2) >> 2;$

Listing 5.2: Code of the df_y1 filter functions of the DF kernel using the complex DF intra-vector instruction.

```
1  const vuint8_t combine     =
2                  {0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x0f,
3                   0x10,0x11,0x12,0x13,0x14,0x15,0x16,0x17};
4  const vuint8_t split_low   =
5                  {0x10,0x11,0x12,0x13,0x14,0x15,0x16,0x17,
6                   0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07};
7  const vuint8_t split_high =
8                  {0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x0f,
9                   0x18,0x19,0x1a,0x1b,0x1c,0x1d,0x1e,0x1f};
10 vsint32_t params = (vsint32_t) alpha;
11 params = spu_insert(beta, params, 1);
12
13 for(i = 0; i < 8; i++ ) {
14     vuint16_t pixels;
15     if(do_combine){
16         //load & rearrange input data
17         pixels = spu_shuffle(*(pix1+i), *(pix2+i), combine);
18
19         //compute filter
20         vuint16_t result = spu_df_y1c(pixels, params);
21
22         //rearrange and store output data
23         *(pix1+i) = spu_shuffle(result1, *(pix1+i),
24                                 split_low);
25         *(pix2+i) = spu_shuffle(result1, *(pix2+i),
26                                 split_high);
27     } else {
28         //load data
29         pixels = *(pix1+i);
30
31         //compute filter
32         vuint16_t result = spu_df_y1c(pixels, params);
33
34         //store data
35         *(pix1+i) = (vsint16_t) result1;
36     }
37 }
```

Table 5.3: Speedup and instruction count reduction obtained with the complex DF instructions in the DF kernel.

	Baseline	Enhanced	Improvement	
Cycles	122135	120876	1259	1.01
Instructions	101480	86376	15104	1.17

The new instructions were made available to the programmer through intrinsics (see also Listing 5.2). The instructions were added to the simulator and the latency of both was set to 14 cycles. There are five stages of fixed point operations, each accounted for by two cycles. The 3-bit AND operation and the three stages of muxes are accounted for by another two cycles. Finally, we added two more cycles for the complex routing in the first two stages (see Figure 5.9). The complexity of the `df_y2c` instruction is similar and thus has the same latency.

The simple DF intra-vector instructions implement only part of the computational structure of Figure 5.9. The `df_y1_p1` instructions computes the values `{p2_1,p1_1,p0_1,q0_1,q1_1,q2_1}`, while `df_y1_p2` computes the values `{p0_2,q0_2}`. To select the proper values for each pixel, two masks are created by instructions `df_y1_m1` and `df_y1_m2`. Listing 5.3 shows how these new functions work together to compute the df_y1 filter function. For the df_y2 function a similar approach was used.

All the simple DF intra-vector instructions fit the RR instruction format. They were added to the simulator and made available to the programmer by intrinsics. Considering the latency, the instructions were divided in two groups. Those that create a mask were assigned a latency of four cycles, just as any other instruction in the SPU ISA that creates a mask. The other instructions were assigned a latency of seven cycles. They mostly have three stages of fixed point operations, each accounted for by two cycles. Further, we added one cycles for the complex routing of the input values.

Table 5.3 and 5.4 present the speedup and reduction of the instruction count using the complex and simple DF instructions, respectively. In both cases the enhanced kernel performs worse than the baseline kernel. To understand this, we summarize all the effects of using intra-vector instructions.

First, two transpositions of the pixel data are avoided, having a positive impact on the performance. On the other hand, new data rearrangement has been introduced as in half of the cases the input data is not aligned within one vector.

Listing 5.3: Code of the df_y1 filter functions of the DF kernel using the simple DF intra-vector instructions.

```
1  // initialization omitted due to space constraints
2
3  for(i = 0; i < 8; i++ ) {
4      vuint16_t pixels;
5      if(do_combine){
6          //load & rearrange input data
7          pixels = spu_shuffle(*(pix1+i), *(pix2+i), combine);
8
9          //compute the filter
10         vuint8_t mask1, mask2;
11         mask1 = spu_df_y1_m1(pixels, params);
12         mask2 = spu_df_y1_m2(pixels, params);
13         vuint16_t result1 = spu_df_y1_p1(pixels);
14         vuint16_t result2 = spu_df_y1_p2(pixels);
15         result1 = spu_shuffle(result1, result2, mask2);
16         result1 = spu_shuffle(result1, pixels, mask1);
17
18         //rearrange & store output data
19         *(pix1+i) = spu_shuffle(result1, *(pix1+i),
20                                 split_low);
21         *(pix2+i) = spu_shuffle(result1, *(pix2+i),
22                                 split_high);
23     } else {
24         //load data
25         pixels = *(pix1+i);
26
27         //compute the filter
28         vuint8_t mask1, mask2;
29         mask1 = spu_df_y1_m1(pixels, params);
30         mask2 = spu_df_y1_m2(pixels, params);
31         vuint16_t result1 = spu_df_y1_p1(pixels);
32         vuint16_t result2 = spu_df_y1_p2(pixels);
33         result1 = spu_shuffle(result1, result2, mask2);
34         result1 = spu_shuffle(result1, pixels, mask1);
35
36         //store data
37         *(pix1+i) = (vsint16_t) result1;
38     }
39 }
```

Table 5.4: Speedup and instruction count reduction obtained with the simple DF instructions in the DF kernel.

	Baseline	Enhanced	Improvement	
Cycles	122135	120596	1539	1.01
Instructions	101480	89704	11776	1.13

The latter rearrangement requires some shuffles but also additional loads and stores. For example, the code of Listing 5.2 performs 33 loads and 16 stores. That is an overhead of 4 loads and two stores per intra-vector instruction.

Second, the resulting code has many dependencies within few instructions. For example, the code of Listing 5.2 has the following instruction order: loads (2), shuffle, df_y1c, loads (2), shuffles (2), stores (2). Only the instructions that are grouped within this list are independent. Consecutive groups are dependent and cause pipeline stalls. Even when applying loop unrolling this function takes more than three times the number of cycles compared to the SIMD version. The latter takes 77 cycles and 97 instructions while the enhanced takes 275 cycles and 107 instructions. In total 131 cycles are dependency stalls, while most other cycles are single issue. From these numbers we conclude that dependencies are the main cause of the loss of performance.

Third, the DLP exploited by the DF intra-vector instructions is smaller than in the SIMD version. Some of the intra-vector instructions compute only two outputs, while the SIMD version always computes eight outputs. This loss of DLP is compensated by the instruction collapsing, especially in the complex intra-vector instructions.

Finally, applying intra-vector instructions on the DF kernel, introduced some branches, which are costly in the SPE architecture. For example, the code of Listing 5.2 incurs 30 branch stall cycles in half the cases.

We conclude the following from the experiments with this kernel. First, intra-vector instructions are likely not to be efficient if data is not naturally aligned within vectors. Second, performance gains obtained by intra-vector instructions are likely to be proportional to the number of its output elements.

5.2.5 Matrix Multiply Kernel (MatMul)

Dense matrix-matrix multiplication is used in many scientific applications. As the matrices are often large (M=1024 or larger) and the algorithm has a time

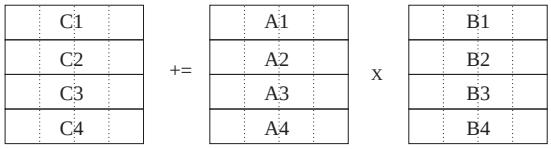


Figure 5.10: Data layout of the baseline 4×4 matrix multiply. The vectors of matrix A have to be broken down in order to compute the matrix multiply.

complexity of $O(M^3)$, fast execution of the matrix multiply (in short Matmul) kernel is of vital importance.

An efficient implementation using the multicore and SIMD capabilities of the Cell processor was provided with SDK 3.0. The matrix multiplication is broken down into multiplications of 64×64 matrices, which fit in the SPE local store size. Within the SPE this multiplication is broken down into smaller parts in two steps, such that in the end matrices of 4×4 are multiplied. The SIMD width of the SPU is four slots of single precision floats, and thus one row of the 4×4 matrix is contained within one SIMD vector as depicted in Figure 5.10. A 4×4 matrix multiply is part of a larger matrix multiplication, and thus the result has to be added to the intermediate value in the output matrix, i.e., $C+ = A \times B$.

In order to utilize the SIMD capabilities, the elements of matrix A have to be extracted and splatted. For example, the first row of matrix C is computed as:

$$C1+ = a1\_1 * B1 + a1\_2 * B2 + a1\_3 * B3 + a1\_4 * B4, \tag{5.3}$$

where $a1_x$ is the xth element of vector A1. Listing 5.4 shows the SPU code that performs this computation. The masks `splatx` are used multiple times and do not constitute a large overhead. Extracting and splatting the elements of matrix A does constitute a significant overhead, although in the optimized implementation of the 64×64 matrix multiplication the result is reused four times, decreasing the total overhead.

Using the intra-vector paradigm, the matrix multiplication could be broken down into blocks of 2×2. A 2×2 block contains four floats which fit in one SIMD vector. Thus a 2×2 matrix multiply can be done in one instruction. We propose the `fmatmul rt,ra,rb` (Floating MATrix MULtiply) and `fmma rt,ra,rb,rc` (Floating Matrix Multiply Add) instructions that multiply the 2×2 matrices contained in `ra` and `rb` and stores the result in register `rt`. The `fmma` instructions also adds the matrix contained in `rc` to the result.

Listing 5.4: Example code that computes the first output row of a 4×4 matrix multiply.

```
1  vector uchar splat0 = {0,1,2,3, 16,17,18,19,
2                         0,1,2,3, 16,17,18,19};
3  vector uchar splat1, splat2, splat3;
4
5  splat1 = spu_or(splat0, 4);
6  splat2 = spu_or(splat0, 8);
7  splat3 = spu_or(splat0, 12);
8
9  a1_1 = spu_shuffle(A1, A1, splat0);
10 a1_2 = spu_shuffle(A1, A1, splat1);
11 a1_3 = spu_shuffle(A1, A1, splat2);
12 a1_4 = spu_shuffle(A1, A1, splat3);
13
14 C1 = spu_madd(a1_1, B1, C1);
15 C1 = spu_madd(a1_2, B2, C1);
16 C1 = spu_madd(a1_3, B3, C1);
17 C1 = spu_madd(a1_4, B4, C1);
```

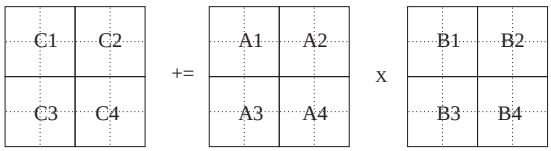


Figure 5.11: Data layout of the enhanced 4×4 matrix multiply using 2×2 sub blocks. Each 2×2 block is located within one SIMD vector.

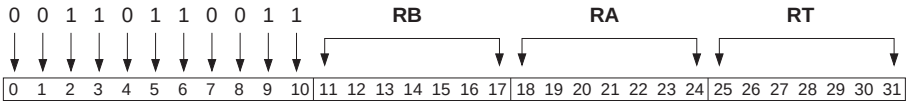
Assuming the data of a 4×4 matrix is rearranged such that the four 2×2 blocks are located within SIMD vectors (as indicated in Figure 5.11), one block of the output can be computed with two instructions only, as shown in Listng 5.5. Usually the input data is not arranged in such a way, and therefore additional conversion overhead is required. As we show later, the overhead does not counterbalance the benefits.

The `fmatmul` instruction fits the RR instruction format. The instruction details are depicted below. The `ffma` instruction fits the RRR instruction format. Only a few opcodes for this instruction format are available in the SPU. Optionally, the RR format could be used if register `rt` is used for both input and output. The instruction details as we implemented it, are depicted below.

Listing 5.5: The code of Listing 5.4 can be replaced by the following when using the intra-vector `fmma` instruction.

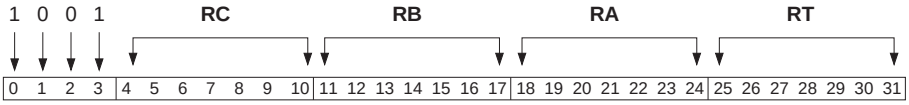
```
1 C1 = spu_fmma(A1, B1, C1);
2 C1 = spu_fmma(A2, B3, C1);
```

fmatmul rt,ra,rb



$$
\begin{array}{lll}
RT^{0::4} & \leftarrow RA^{0::4} * RB^{0::4} & +RA^{4::4} * RB^{8::4} \\
RT^{4::4} & \leftarrow RA^{0::4} * RB^{4::4} & +RA^{4::4} * RB^{12::4} \\
RT^{8::4} & \leftarrow RA^{8::4} * RB^{0::4} & +RA^{12::4} * RB^{8::4} \\
RT^{12::4} & \leftarrow RA^{8::4} * RB^{4::4} & +RA^{12::4} * RB^{12::4}
\end{array}
$$

fmma rt,ra,rb,rc



$$
\begin{array}{llll}
RT^{0::4} & \leftarrow RA^{0::4} * RB^{0::4} & +RA^{4::4} * RB^{8::4} & +RC^{0::4} \\
RT^{4::4} & \leftarrow RA^{0::4} * RB^{4::4} & +RA^{4::4} * RB^{12::4} & +RC^{4::4} \\
RT^{8::4} & \leftarrow RA^{8::4} * RB^{0::4} & +RA^{12::4} * RB^{8::4} & +RC^{8::4} \\
RT^{12::4} & \leftarrow RA^{8::4} * RB^{4::4} & +RA^{12::4} * RB^{12::4} & +RC^{12::4}
\end{array}
$$

The `fmatmul` and `fmma` instruction were made available to the programmer trough the intrinsics `d = spu_fmma(a,b,c);` and `d = spu_fmatmul(a,b);`. The instructions were added to the simulator and the compiler. The latency of the `fmatmul` instruction was set to six cycles. A normal floating point multiply-add takes six cycles. The `fmatmul` performs twice the number of multiplies, but those can be done in parallel. The `fmatmul` instruction performs an addition, just like the normal multiply add. Thus the latencies of those two are the same. The final addition of the `fmma` instruction, though, has three inputs instead of two and we account for this by adding two cycles to the latency. This is rather pessimistic, but on the safe side.

Table 5.5: Speedup and instruction count reduction obtained with the `fmatmul` instruction in the Matmul kernel.

	Baseline	Enhanced	Improvement	
Cycles	85894	69094	16800	1.24
Instructions	136585	111048	25537	1.23

Table 5.6: Speedup and instruction count reduction obtained with the `fmma` instruction in the Matmul kernel.

	Baseline	Enhanced	Improvement	
Cycles	85894	41735	44159	2.06
Instructions	136585	63239	73346	2.16

We applied the intra-vector instructions to the Matmul kernel taken from the Cell SDK. The kernel contains several versions of the SPU code that multiplies the 64×64 blocks, each with a higher degree of optimization. We used the optimized SIMD version, which is written in C code. There is also an even more optimized version written in assembly. We will indicate the expected performance of our enhanced kernel when written in assembly.

We did not change the general strategy of the 64×64 multiply function, but only changed the code at the subblock-level. We measured the time of one 64×64 matrix multiply add only. The Matmul kernel uses double buffering and is computation bound, and thus this measure is representative for the entire kernel. In the measurement results presented here we do not include the overhead cost for rearranging the data. This rearrangement has time a complexity of $O(M^2)$ and is fully data parallel. The matrix multiplication itself has a time complexity of $O(M^3)$ and thus the overhead cost decreases exponentially for increasing matrix sizes. For $M = 64$ the overhead is 19.1%, but for $M = 1024$ (which is a more realistic matrix size) the overhead is only 1.2%.

Table 5.5 shows the results obtained with the `fmatmul` instruction. A speedup of 1.24 is achieved and a similar reduction in the number of executed instructions. The speedup is significant but limited. Every `fmatmul` instruction is followed by an addition, mitigating much of the benefits. This is confirmed by the results of the `fmma` instruction, depicted in Table 5.6. A speedup of 2.06 is achieved which is much larger than the former because many add instructions are collapsed into the `fmma` instruction.

The speedup is also much larger than that of the previous kernels because the intra-vector instructions affected the entire kernel, while for the previous kernels it affected the horizontal mode only. Figure 5.4 shows that both the baseline and the enhanced code are very optimal as there are only very few dependency stalls and a high rate of dual issue cycles.

As mentioned above, the Matmul kernel can be further optimized when written in assembly. The SDK assembly version takes 65665 cycles and executes 105801 instructions. The improvements in performance, compared to the version we used, is due to optimizations in register mapping and instruction scheduling. The first resulted in a decreased number of loads and stores while the latter improved the dual issue rate. Our enhanced kernel, written in C code, outperforms the assembly code of the SDK. Furthermore, the same optimization techniques used in the SDK assembly version can be applied to our enhanced kernel. Our enhanced kernel will benefit less from the register mapping as it is already more optimal in terms of loads/stores. Instruction scheduling, however, will increase performance more than was the case in the SDK code. The SDK assembly code has an IPC of 1.61 because there are many more even pipeline instructions than there are odd pipeline instructions. Our enhanced kernel has a more balanced number of even and odd pipeline instructions and could achieve an IPC of maximally 1.81. Thus only scheduling could decrease the execution time of the enhanced kernel to 34902 cycles. From this we conclude that an assembly optimized version of our enhanced kernel will outperform the SDK assembly version by a factor near to 2x.

This kernel shows that with some understanding of the algorithm, the implementation can be modified to use intra-vector instructions. By rearranging the data, overhead can be avoided and operations can be collapsed into one instruction.

5.2.6 Discrete Wavelet Transform Kernel (DWT)

The Discrete Wavelet Transform (DWT) allows to generate a compact representation of a signal, taking advantage of data correlation in space and frequency [107, 51]. It has found its use in different fields, ranging from signal processing, image compression, data analysis, and partial differential equations, to denoising. There exist several implementations of the DWT, such as convolution based and lifting scheme methods. The last allows the most efficient implementation of the 2D DWT and is used in the JPEG-2000 standard.

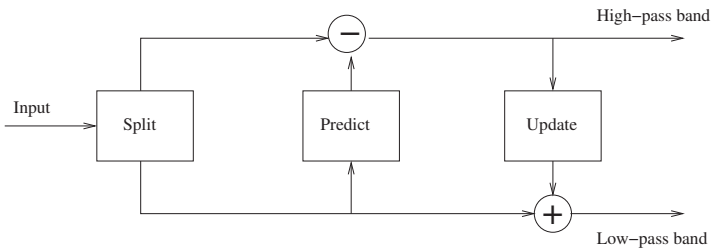


Figure 5.12: The three phases of the forward DWT lifting scheme. The inverse DWT applies the inverse operations on reverse order.

A Cell implementation of JPEG-2000 is available from the CellBuzz project [6, 88]. The inverse DWT kernel from the lossless decoder formed the baseline kernel for this experiment. The forward DWT is very similar to the inverse DWT and would provide similar results as obtained with the latter. Also the DWT in the lossy decoder is similar to the one of the lossless decoder. The baseline inverse DWT kernel is SIMDized.

The DWT kernel applies five levels of decomposition. In each decomposition step the image is processed horizontally and vertically, constituting a 2D transform. In each of the 1D transforms a vector of pixels is split into two parts as indicated in Figure 5.12 for the forward DWT. After splitting, in two lifting steps the high pass and low pass bands are generated. The inverse DWT performs the inverse operations in the reverse order.

Figure 5.13 shows schematically the operations performed by the inverse DWT. The example is simplified and does not show the processing of the samples at the beginning and end of the vector. Those are handled in a slightly different manner. The input vector consists of two parts; the low pass band on the left and the high pass band on the right. First, one lifting step is applied on the low pass band followed by a lifting step on the high pass band. Finally, the two are combined (inverse split) and the original samples are obtained.

In Figure 5.14 the same process is depicted in a different way. It shows that four adjacent output samples are depending on three samples from the low pass band and four samples from the high pass band. Each sample is a 32-bit integer and thus four samples fit in one SIMD vector of the SPE. Consequently, the computational structure of the figure can be computed by one intra-vector instruction, the verb!idwt! instruction.

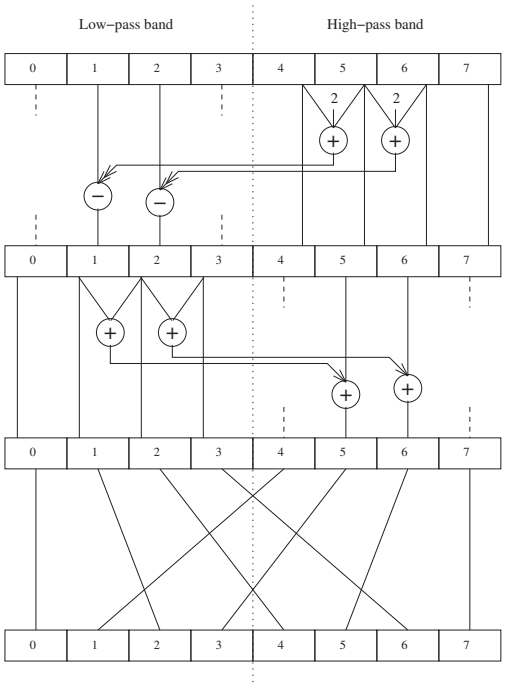


Figure 5.13: Schematic overview of the computational structure of the inverse DWT on an example vector of eight samples. An arrow indicates a shift right by one bit (two bits for a double arrow). The edges of the two sub bands are processed slightly different and not shown in this diagram.

The `idwt` instruction is efficient in the horizontal inverse DWT as both the input and output vector have elements in consecutive memory locations. To apply the `idwt` instruction in the vertical DWT a lot of rearrangement overhead is required. Moreover, conventional SIMDimization is efficient in the vertical DWT, and therefore there is no need to apply the intra-vector instruction to the vertical DWT.

As mentioned above, processing the first and last samples of a vector are done slightly different. Therefore, the `idwt` instruction can only be applied to the body of the vector. The head, consisting of four output samples, is performed using scalar operations as well as the tail, consisting of at least one sample. The size of the vectors is usually large. Assuming an image of 512×512 pixels, the vector sizes for the DWT are 32, 64, 128, 256, and 512.

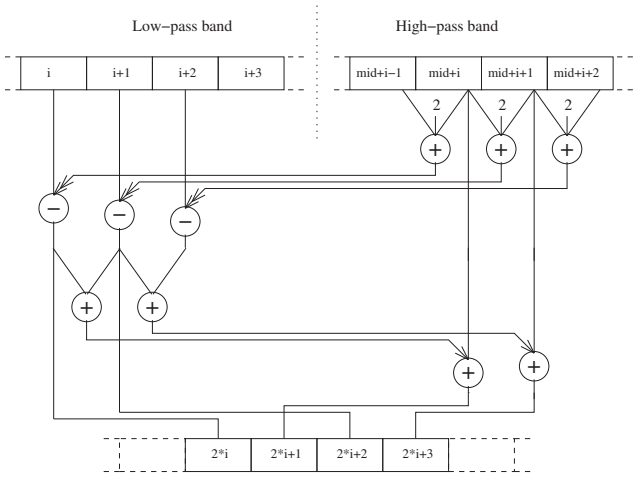


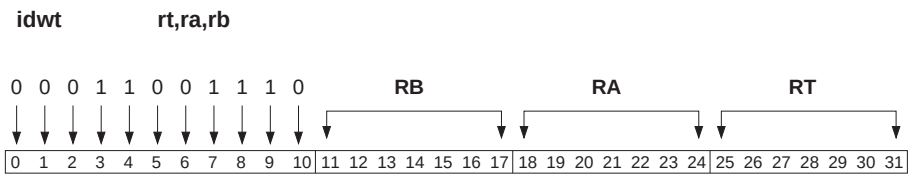
Figure 5.14: A different representation of the computational structure of the inverse DWT. This time a larger vector is assumed and only partially displayed. All stages are combined in one step. The computations depicted can be performed with the `idwt` intra-vector instruction.

The `idwt` instruction fits the RR instruction format. The instruction details are depicted below. The corresponding intrinsic `d = spu_idwt(a,b)` was added to the compiler. The latency of the instruction in the simulator was set to eight cycles. The critical path contains four simple fixed point operations, each taking two cycles and adding up to eight. We assume the latency of the shifts can be neglected.

We evaluated the `idwt` intra-vector instruction using the Cell implementation of JPEG-2000. The DWT kernel in this application is a cooperation between the PPU and the SPUs. The first divides the work while the SPUs DMA in the data and perform the actual DWT. We decoded a picture of 512×512 pixels and measured the actual time spent on the DWT in the SPUs. The results are depicted in Table 5.7 and show that a significant speedup of 1.59 is obtained. Interestingly, the speedup is larger than the instruction count reduction. Figure 5.4 shows that the baseline kernel suffered severely from dependency and branch stalls. The new method of processing the array, enabled by intra-vector instructions, proves to be more efficient as both types of stall cycles are substantially reduced. The DWT can also be applied to a one-dimensional data array. In that case the speedup achieved will even be larger. We obtained a speedup of 2.53 on an array of 512 pixels.

Table 5.7: Speedup and instruction count reduction obtained with the `idwt` instruction in the inverse DWT kernel.

	Baseline	Enhanced	Improvement	
Cycles	10948039	6893905	4054134	1.59
Instructions	5117073	3685579	1431494	1.39



$$
\begin{aligned}
t0 &\leftarrow RA^{0::4} - ((RB^{0::4} + RB^{4::4} + 2) >> 2) \\
t2 &\leftarrow RA^{4::4} - ((RB^{4::4} + RB^{8::4} + 2) >> 2) \\
t &\leftarrow RA^{8::4} - ((RB^{8::4} + RB^{12::4} + 2) >> 2) \\
t1 &\leftarrow RB^{4::4} + ((t0 + t2) >> 1) \\
t3 &\leftarrow RB^{8::4} + ((t2 + t) >> 1) \\
\\
RT^{0::4} &\leftarrow t0 \\
RT^{4::4} &\leftarrow t1 \\
RT^{8::4} &\leftarrow t2 \\
RT^{12::4} &\leftarrow t3
\end{aligned}
$$

Whereas the inner loop of all other kernels operate on small sub-blocks, in the DWT kernel entire rows and columns are processed. Therefore, the speedup of the kernel is depending on the input size. So far, an input image of 512×512 pixels was assumed. Figure 5.15 shows that for smaller input sizes, the speedup and instruction count reductions are actually larger. For smaller input sizes, the startup overhead (processing the edges, initializing variables, etc.) dominates. This overhead is larger in the baseline kernel than is the case in the enhanced kernel. Therefore, the speedup is larger for small input sizes and flattens out for larger sizes.

The DWT kernel is another example of how large speedups can be achieved by intra-vector instructions. It shows that they can enable a more efficient processing of the data. The DWT kernel also shows that intra-vector instruction can be applied to one-dimensional kernels.

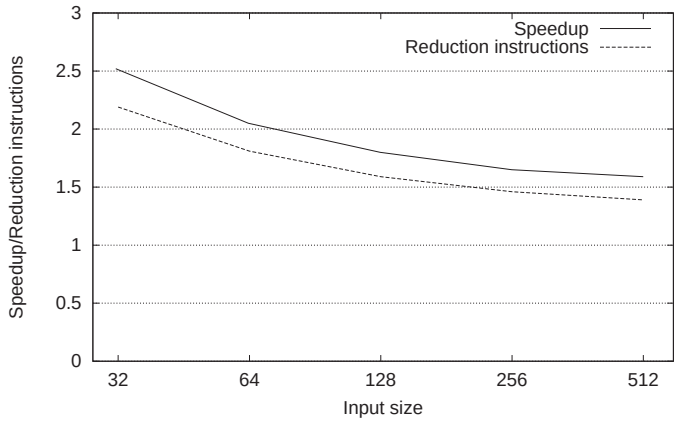


Figure 5.15: Speedup and instruction count reduction obtained for the DWT kernel with different input sizes.

5.3 Discussion

The results obtained are good, but some questions arise about implementability and cost. In this section we discuss those issues.

The first question is what the influence of the new instructions is on the latency of the existing instructions. Adding the intra-vector instructions to the existing functional units increases their complexity and latency. It seems best, though, to put all intra-vector instructions in one new functional unit, except those for the MatMul kernel, as they perform floating point operations in contrast to the others. This functional unit can consist of a series of ALUs and interconnect such that each intra-vector instruction can be mapped to it. A similar structure of ALUs and interconnect has been used in [158, 172] for collapsing automatically generated instruction set extensions. Most likely, by adding such a unit the latency of the total pipeline is increased, meaning that the write-back is delayed. Due to the existence of the forwarding network, however, the other instructions are not affected. The forwarding network becomes larger though, and thus there is some additional area cost here. Future work is to investigate a smart implementation of such intra-vector unit. Another approach would be to map the intra-vector instructions to extensible processors such as the ARC700 [2], Tensilica's XTensa [154], or Altera's NioSII [18]. It has yet to be investigated if this would be possible, though. The instructions for the MatMul

kernel probably are best added to the existing single precision unit. Similarly this would increase the latency of the pipeline.

A second issue is the area cost of the intra-vector instructions. Without an actual hardware implementation, some back of the envelope calculations are the best we can do. The intra-vector unit needs five levels of simple fixed point operations (no multiplication) and interconnect in between. The fixed point unit of the SPE performs these operations and from the floor plan we determined that the area of the fixed point unit is 3% of the total SPE. This fixed point unit can perform much more diverse operations as needed by the intra-vector unit, and also can operate on 32-bit values, while the intra-vector unit needs to operate on 8 and 16-bit values only. On the other hand, the intra-vector unit requires more ALUs per level and additional interconnect. We estimate that the area taken per level is 1.5 times that of the fixed point unit. Therefore, adding the intra-vector unit to the SPE requires approximately an additional $5 \times 3 \times 1.5 = 22.5\%$ of area. Addition of the MatMul instructions to the single precision pipeline is expected to double its size, requiring 10% of additional area.

Another question might be whether the latencies of the instructions are correct. Discussion with experts showed us that our methodology of determining the latencies is on the safe side. We also analyzed the speedups for larger latencies, however. Figure 5.16 shows that the average speedup only drops from 1.45 to 1.31 in the unlikely case when the latencies would be twice as large.

The speedups achieved seems to be worth the area cost. The average speedup is obtained on kernels only, though. The speedup on complete applications will be lower, although in the signal processing domain generally the kernels dominate the execution time. Application specific processors targeted at signal processing therefore seem to be most suitable for specialized intra-vector instructions as we presented them.

5.4 Conclusions

In this chapter we have shown that intra-vector SIMD instructions are an effective technique for the specialization of cores, especially in the signal processing domain. For six kernels intra-vector instructions have been developed and simulations have shown that a speedup of up to 2.06 was achieved, with an average of 1.45.

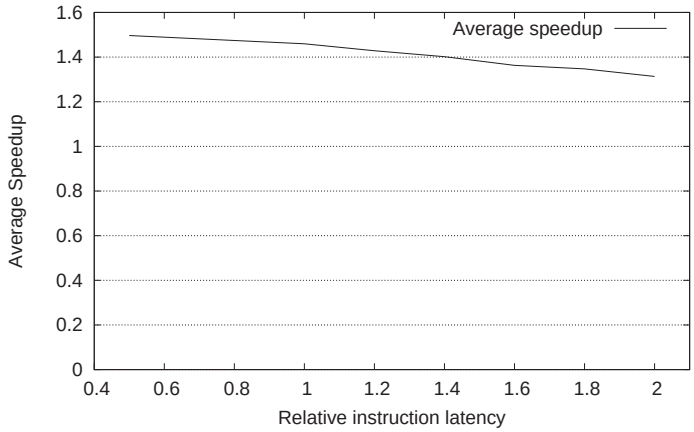


Figure 5.16: The average speedup as a function of the relative instruction latencies.

We have shown how intra-vector instructions reduce the data rearrangement overhead in two-dimensional kernels. The efficiency depends on several factors. The ability to collapse a large number of instructions increases the efficiency. On the other hand, efficiency is limited if a lot of rearrangement overhead is required for intra-vector processing, if many dependency stalls occur because the distance between dependent instructions has decreased, or if there is a low amount of DLP in intra-vector instructions.

The evaluation was performed with the Cell SPE as baseline architecture. The results on different architectures will be slightly different. The general conclusion of this chapter, however, remains.

Chapter 6

Hardware Task Management Support for the StarSS Programming Model: the Nexus System

Successful deployment of many-core chips depends on the availability of parallel applications. Creating efficient parallel applications, though, is currently cumbersome and complicated as, besides parallelization, it requires task scheduling, synchronization, programming or optimizing data transfers, etc. Our experiences with the implementation of parallel H.264 decoding for several platforms, as described in Chapter 3, confirm this statement.

Currently there are several efforts to solve the programmability issues of multicores. Among these is the task-based StarSS programming model [129]. It requires the programmer only to indicate functions, referred to as tasks, that are amenable to parallel execution and to specify the input and output operands. All the other issues, such as determining task dependencies, scheduling, programming and issuing data transfers, etc., are automatically handled by StarSS.

The task management performed by the StarSS runtime system is rather laborious. This overhead affects the scalability and performance, and potentially limits the applicability of StarSS applications. In this chapter we study the benefits of accelerating task management with hardware support. We show that for fine-grained tasks, the runtime overhead greatly reduces performance and severely limits scalability. Through extensive analysis of the current sys-

tem's bottlenecks, we specify the requirements of a hardware support system.

As an example of such a system, we present Nexus: a hardware task management support system compliant with the StarSS programming model. We present the design of the hardware, as well as the software interface. The Nexus system is evaluated by means of simulation, which shows that the requirements are met. By using the Nexus system, StarSS can be efficiently used with much smaller task sizes.

Some other works have proposed hardware support for task management. Most are limited, however, to scheduling of independent tasks leaving it up to the programmer to deliver tasks at the appropriate time. Carbon [94] provides hardware task queuing enabling low latency task retrieval. In [79] a TriMedia-based multicore system is proposed containing a centralized task scheduling unit based on Carbon. For a multicore media SoC, Ou et al. [122] propose a hardware interface to manage tasks on the DSP, mainly to avoid OS intervention. Architectural support for the Cilk programming model has been proposed in [106], mainly focusing on cache coherency.

A few works include hardware task dependency resolution. In [142] a look ahead task management unit has been proposed reducing the task retrieval latency. It uses task dependencies in a reversed way to predict due available tasks. Because of the programmable nature, its latency is rather large. The latter issue has been resolved in [17] at the cost of generality: the proposed hardware is specific for H.264 decoding. For System-on-Chips (SoCs), an Application Specific Instruction set Processor (ASIP) has been proposed to speedup software task management [35]. Independently, Etsion et al. [57] have also developed hardware support for the StarSS programming model. They observe a similarity between input and output dependencies of tasks and instructions and have proposed task management hardware that works similar to an out-of-order instruction scheduler.

A different approach called Decoupled Thread Architecture (DTA) has been proposed by Giorgi et al. [64] and is based on thread level data-flow. Threads can be as small as functions and thus their approach is applicable to fine-grained parallelization. Data is sent from a producer to a consumer thread directly by distributed hardware units. This data communication also functions as synchronization. In [65] an implementation of DTA for the Cell processor has been described. Also based on the data-flow principle is the approach that has been proposed by Stavrou et al. [146]. In contrast to DTA, this approach is coarse grained, especially the scheduling among cores. A programming model for these two data-flow approaches has been presented in [147].

Listing 6.1: Basics of the StarSS programming model.

```
1  int *A[N][N];
2
3  #pragma css task input(base[16][16])
4                   output(this[16][16])
5  void foo(int* base, int* this){
6    ...
7  }
8
9  void main(){
10   int i,j;
11   ...
12   #pragma css start
13   foo(A[0][0], A[i][j]);
14   ...
15   #pragma css finish
16   ...
17 }
```

This chapter is organized as follows. The StarSS programming model and the target processor platform are briefly reviewed in Section 6.1. The benchmarks used throughout this chapter are described in Section 6.2. In Section 6.3 the StarSS runtime system is analyzed. Section 6.4 compares the performance of the current StarSS system to that of a manually parallelized implementation and illustrates the need for hardware task management support. The proposed Nexus system is described in Section 6.5, while its performance is evaluated in Section 6.6. Future work is described in Section 6.7 and Section 6.8 concludes the chapter.

6.1 Background

The StarSS programming model is based on simple annotations of serial code, by adding pragmas. The main pragmas are illustrated in Listing 6.1. The pragmas `css start` and `css finish` indicate the initialization and finalization of the StarSS runtime system. Tasks are functions that are annotated with the `css task` pragma, including a specification of the input and output operands. In addition to the pragmas illustrated in this example, StarSS provides several synchronization pragmas.

The current implementation of StarSS uses one core with two threads to control the application, while the others (worker cores) execute the parallel tasks. A source-to-source compiler processes the annotated serial code and generates the codes for the control and worker cores. The first thread of the control core runs the main program code that adds tasks and the corresponding part of the runtime system. The second thread handles the communication with the worker cores and performs the scheduling of tasks.

Tasks are added dynamically to the runtime system, which builds the task dependency graph based on the addresses of input and output operands. Tasks, whose dependencies are met, are scheduled for execution on the worker cores. Furthermore, the runtime system manages data transfers between main memory and local scratchpad memories, if applicable. It tries to minimize the execution time by applying a few optimizations. First, it creates groups of tasks, referred to as bundles within StarSS. Using bundles reduces the per-task overhead for scheduling. In addition, the runtime system optimizes for data locality by assigning chains within the task graph to bundles. Within such bundles, data produced by one task is used by a next, and thus locality is exploited.

Throughout this chapter, we use the StarSS instantiation for the Cell Broadband Engine, which is called CellSS. We use CellSS version 2.2 [4], which is the latest release at time of writing. It has an improved runtime system with significantly lower overhead than prior releases.

CellSS has several configuration parameters that can be used to tune the behavior of the runtime system. Three of these significantly affect the execution of the benchmarks we use: (scheduler.min_tasks, scheduler.initial_tasks, scheduler.max_strand_size), of which the default values are $(16, 128, 8)$. All three relate to assigning ready tasks to the worker cores, which is referred to as scheduling in CellSS. Scheduler.min_tasks defines the minimum number of ready tasks that should be available to start the scheduling process. Scheduler.initial_tasks is a similar threshold, but only used for the first time scheduling is applied. Scheduler.max_strand_size defines the maximum bundle size.

Measuring the execution time of the benchmarks using CellSS includes only the parallel section. The startup and finish phase of CellSS are relatively large compared to the small benchmark program. Therefore, these phases are excluded from the measurements. In practice this means that the timing is started directly after the `#pragma css start` and stopped just before the `#pragma css finish`, but after a `#pragma css barrier` (see Listing 6.2 in the next section). Putting the measurement points outside these pragmas would include the startup and finish phase.

Our experimental platform is the Cell processor, which was briefly described in Section 4.2. It contains one PowerPC Processing Element (PPE) and eight Synergistic Processing Elements (SPEs). The CellSS runtime system runs on two threads of the PPE. The SPEs wait to receive a signal from the helper thread indicating what task to perform. Once they are ready, they signal back and wait for another task. The SPEs have scratchpad memory only and use DMA (Direct Memory Access) commands to load data from main memory into their local store. When using bundles, CellSS applies double buffering, which hides the memory latency by performing data transfers concurrently with task execution. The measurements are performed on a Cell blade containing two Cell processors, running at $3.2GHz$. Thus, in total 16 SPEs are available.

6.2 Benchmarks

To analyze the scalability of StarSS under different circumstances, we used three synthetic benchmarks. We used synthetic benchmarks rather than real applications because it allows controlling the application parameters such as the dependency pattern, the task execution time (and, hence, the computation to communication ratio), etc. The three benchmarks are referred to as ND (No Dependencies), SD (Simple Dependencies), and CD (Complex Dependencies). The dependency pattern of the CD benchmark is similar to the dependency pattern in H.264 decoding (see Figure 3.6).

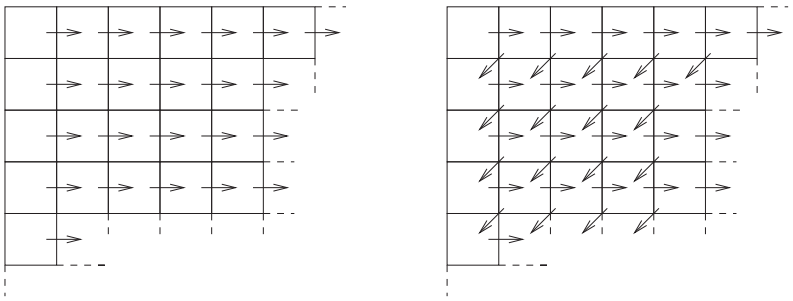
All three benchmarks process a matrix of 1024×1024 integers in blocks of 16×16 (see Listing 6.2). Note that A is a matrix of pointers to the 16×16 blocks and thus has size 64×64 and not 1024×1024. Each block operation is one task, adding up to a total of 4096 tasks. The function `spend_some_time()` allows us to modify the task execution time. The task execution time is varied from approximately $2\mu s$ to over $2ms$.

In the ND benchmark all tasks are independent of each other and thus the application is fully parallel. In the SD benchmark, each task depends on its left neighbor if it exists, as depicted in Figure 6.1(a). Thus, all 64 rows are independent and can be processed in parallel. Finally, in the CD benchmark, each task depends on its left and top-right neighbor if it exists, as depicted in Figure 6.1(b). Listing 6.2 shows the simplified code of the CD benchmark, including the annotations.

Due to the task dependencies, the parallelism in the CD benchmark suffers from ramping, similarly as depicted in Figure 3.7. That is, during the execu-

Listing 6.2: The simplified code of the CD benchmark including StarSS annotations.

```
1  int *A[64][64];
2
3  typedef struct params{
4    int y;
5    int x;
6  } params_t;
7
8  #pragma css task input(prms, left[16][16],
9                         topright[16][16])
10                  inout(this[16][16])
11 void task(params_t* prms, int* left, int* topright,
12           int* this){
13   ...
14   //compute
15   ..
16   spend_some_time();
17   return;
18 }
19
20 void main(){
21   int i,j;
22   params_t my_params;
23
24   init_matrix(A);
25
26   #pragma css start
27   //start measurement
28   for(i=0; i<64; i++){
29     for(j=0; j<64; j++){
30       my_params.y = i;
31       my_params.x = j;
32       task(&my_params, A[i][j-1], A[i-1][j+1], A[i][j]);
33     }
34   }
35   #pragma css barrier
36   //stop measurement
37   #pragma css finish
38 }
```



(a) SD (b) CD

Figure 6.1: The task dependency patterns of the SD and CD benchmarks. The ND benchmark has no task dependencies.

tion the number of available parallel tasks gradually increases, until halfway, after which the available parallelism similarly decreases back to one. Therefore, the amount of available parallelism at the start and end of the benchmark is lower than the number of cores. Due to this ramping effect in the CD benchmark, the maximum obtainable parallelism is 14.5 using 16 cores. The other benchmarks do not suffer from this ramping effect and have a maximum obtainable parallelism of 16 when using 16 cores.

6.3 Scalability of the StarSS Runtime System

The main purpose of this section is to quantify the task management overhead of the StarSS runtime system. One of the main metrics is scalability, which is defined as the speedup of a parallel execution on N SPEs compared to execution using one SPE. Besides measuring performance and scalability, we have also generated Paraver traces [12] to study the runtime behavior in more detail.

First, we study the CD benchmark in detail, especially for small task sizes, as it resembles H.264 decoding, which has an average task size of approximately $20\mu s$.

6.3.1 CD Benchmark

Using the default $(16, 128, 8)$ configuration of StarSS, which represents (scheduler.min_tasks, scheduler.initial_tasks, scheduler.max_strand_size), a maximum scalability of 7.3 is obtained for the CD benchmark using 16 SPEs.

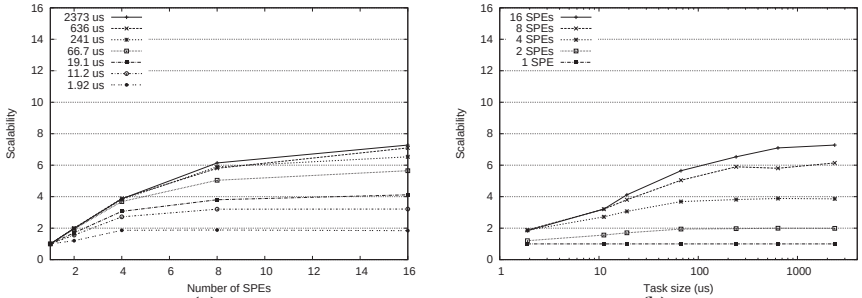


Figure 6.2: Scalability of StarSS with the CD benchmark and using the default configuration.

Figure 6.2 shows the scalability of the CD benchmark with the default configuration for several task sizes.

Even when very large tasks of $2373\mu s$ are used, the SPEs are mostly idle and waiting for tasks, as illustrated in Figure 6.4, which shows a part of the trace. The colored parts indicate the execution phases, of which the legend is depicted in Figure 6.3. The first and second row of the trace corresponds to the main and helper thread, respectively. The other rows correspond to the 16 SPEs. The yellow markers indicate communication events, where the direction of the arrow indicates whether it is a receive or a send. In Figure 6.4 only the receives are depicted.

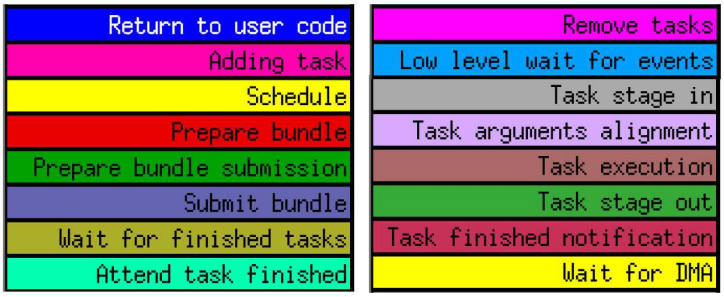


Figure 6.3: Legend of the StarSS Paraver traces.

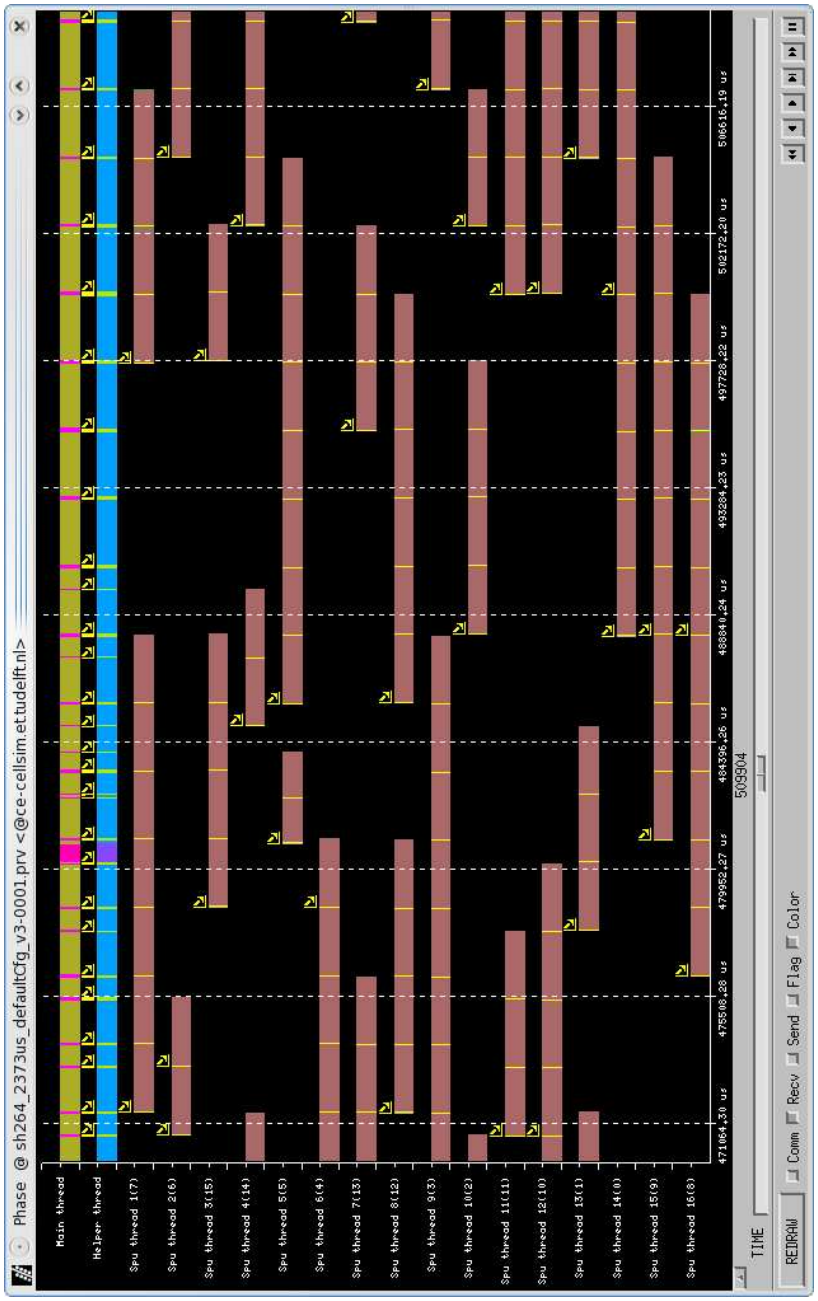


Figure 6.4: Partial Paraver trace of the CD benchmark with 16 SPEs, a task size of $2373\mu s$, and the default configuration.

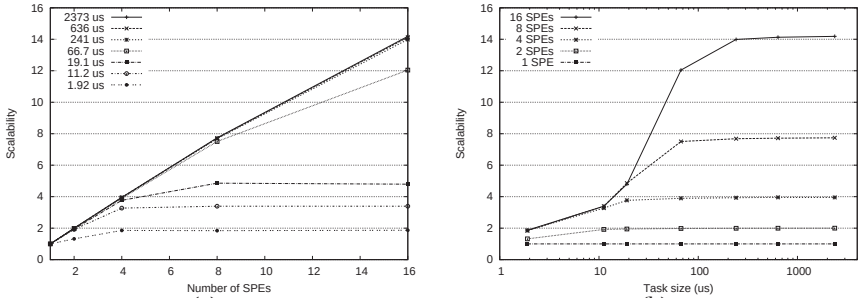


Figure 6.5: Scalability of StarSS with the CD benchmark and the $(1, 1, 1)$ configuration.

The reason for the low scalability is found in the configuration. The runtime system waits for 16 tasks to be ready before it schedules them to SPEs. As there is a limited amount of parallelism available, cores are left idle, waiting for the runtime system to schedule tasks. Therefore, immediate scheduling of ready tasks is required, which is attained with a $(1, y, z)$ configuration.

Furthermore, the runtime system tries to group up to 8 tasks to a single SPE. Again, because of the limited amount of available parallelism, cores are left idle because the load is not properly balanced. Disallowing large groups, by using an $(x, y, 1)$ configuration, forces the runtime system to balance the load among the SPEs. Furthermore, the absolute performance is improved by immediate start of scheduling at the beginning of the application, enforced by using a $(x, 1, z)$ configuration. Scalability is not affected by this modification.

Summarizing, the optimal configuration for this benchmark and with large tasks is $(1, 1, 1)$. The default configuration is only beneficial for situations with ample parallelism. With the optimal settings, finally, a scalability of 14.2 is obtained for large tasks, as shown in the graphs of Figure 6.5. This scalability is very close to the maximum obtainable parallelism of 14.5. Figure 6.6 depicts a part of the Paraver trace. It shows that indeed all SPEs are almost continuously busy.

This configuration, however, stresses the runtime system, especially for small tasks, as every single task is treated separately. Figure 6.7 depicts a part of the Paraver trace of the CD benchmark with a task size of $19\mu s$, for which a scalability of 4.2 is obtained. The yellow lines connecting the rows indicate communication. The helper thread signals to the SPEs what task to execute. The SPEs, in turn, signal back when the task execution is finished.

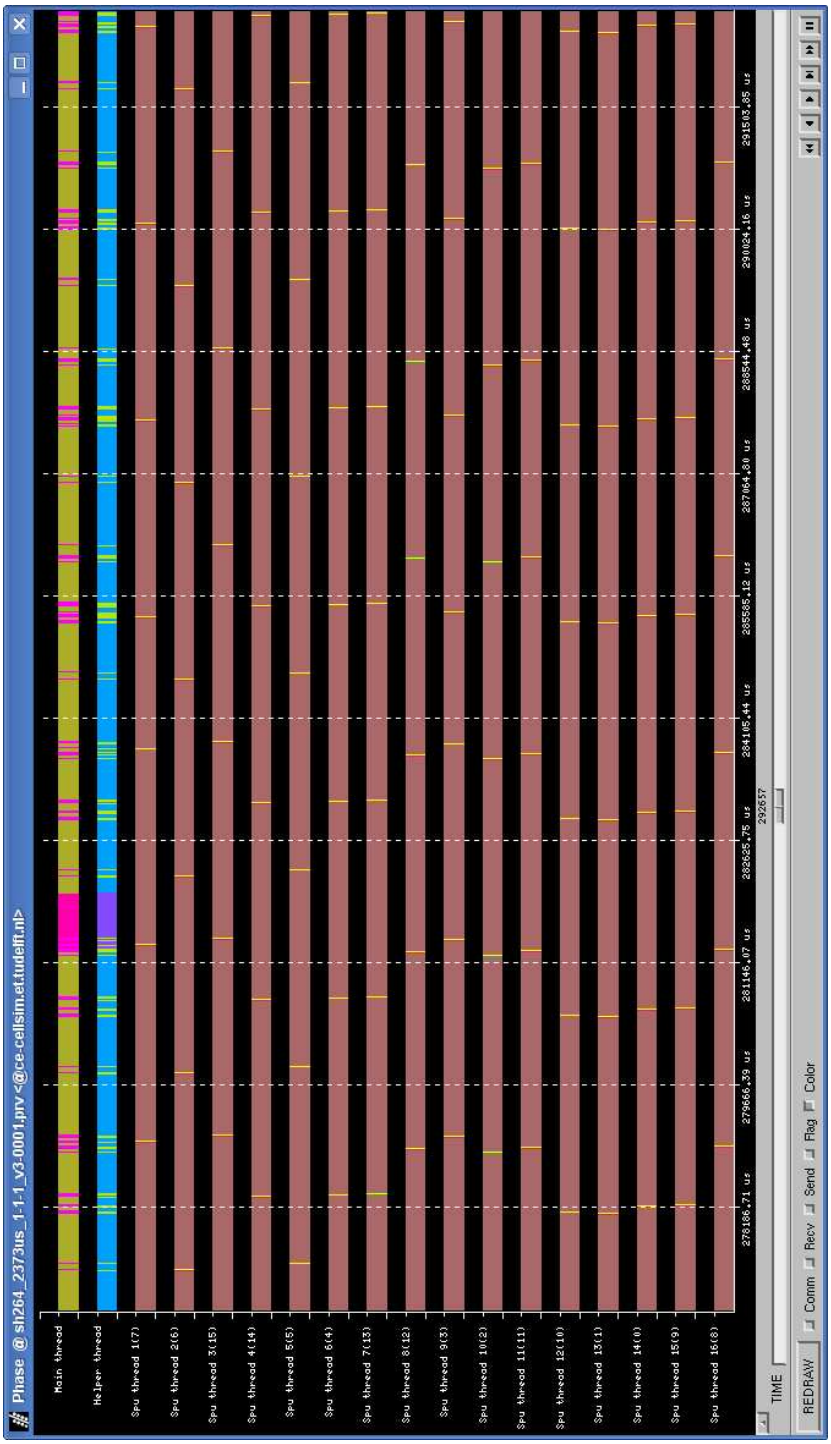


Figure 6.6: Partial Paraver trace of the CD benchmark with 16 SPEs, a task size of $2373\mu s$, and the $(1, 1, 1)$ configuration.

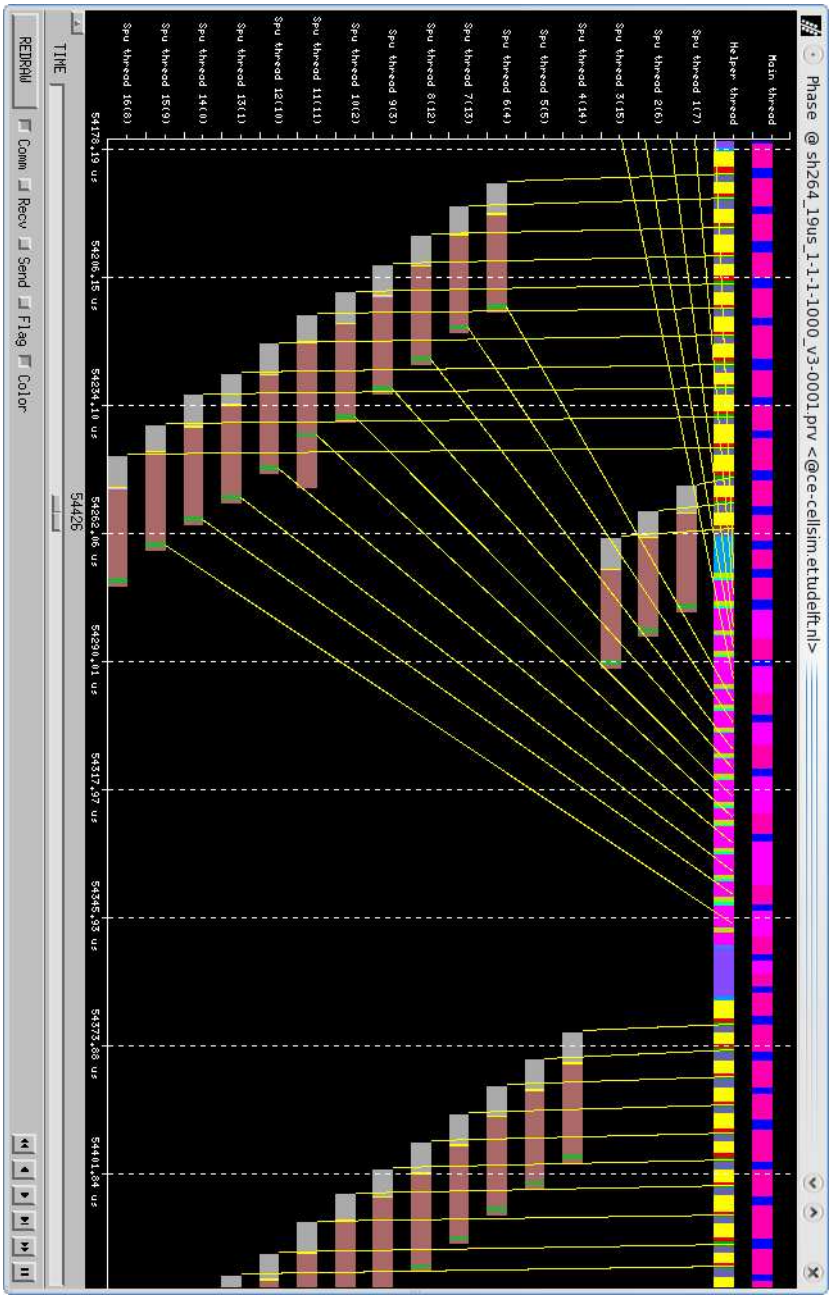


Figure 6.7: Partial Paraver trace of the CD benchmark with 16 SPEs, a task size of $19\mu s$, and the (1,1,1) configuration.

The second row of the trace shows that scheduling (yellow), preparing (red), and submitting a bundle (green) takes approximately $6\mu s$, while removing a task (magenta) takes approximately $5.8\mu s$. As the total task size, including task stage in, wait for DMA, task execution, task stage out, and task finished notification, is $22.5\mu s$, the helper thread can simply not keep up with the SPEs. Before the helper thread has launched the sixth task, the first one has already finished. However, its ready signal (yellow line) is processed much later, and before this SPE receives a new task, even more time is spent.

For such small tasks, the performance can be improved by using the $(1, 1, 4)$ configuration. Grouping 4 tasks in a bundle reduces the scheduling time with 45%, the preparation and submission of tasks with 73%, and the removal of tasks with 50%. In total, the per-task overhead is reduced from $11.6\mu s$ to $5.5\mu s$ resulting in an overall performance improvement of 20%.

Despite this improvement in the runtime system's performance, the scalability is only 5.2. Both the helper thread and main thread remain bottlenecks. Although using the $(1, 1, 4)$ configuration the per-task overhead in the helper thread is reduced to $5.5\mu s$, it is far from sufficient to leverage 16 SPEs efficiently. For that, at most a per-task overhead of $22.5/16 = 1.4\mu s$ is required.

The main reason for the bottleneck in the main thread is very subtle. Although the benchmark exhibits sufficient parallelism, due to the order in which the main thread spawns tasks (from left to right and top to bottom), at least 16 "rows of tasks" (corresponding to 1/4th of the matrix) have to be added to the dependency graph before 16 independent tasks are available that can keep all cores busy. Thus, to exploit the parallelism available in the benchmark, the addition of tasks in the main thread must therefore run ahead of the parallel execution. The trace reveals that the main thread can only keep up with the parallel execution, and thus it limits the scalability of the system.

Figure 6.8 depicts the scalability of StarSS with the CD benchmark using the $(1, 1, 4)$ configuration. Although this configuration outperforms the $(1, 1, 1)$ configuration for small task sizes, for large task sizes it limits the scalability to less than 10. To obtain a high scalability, the $(1, 1, 1)$ configuration must be used in combination with a medium task size. More specifically, Figure 6.5 shows that in order to obtain a scalability efficiency of 80% when using 16 SPEs, which is a scalability of 12.8, at least a task size of approximately $100\mu s$ is required. Similarly, for 8 SPEs a task size of $50\mu s$ is required.

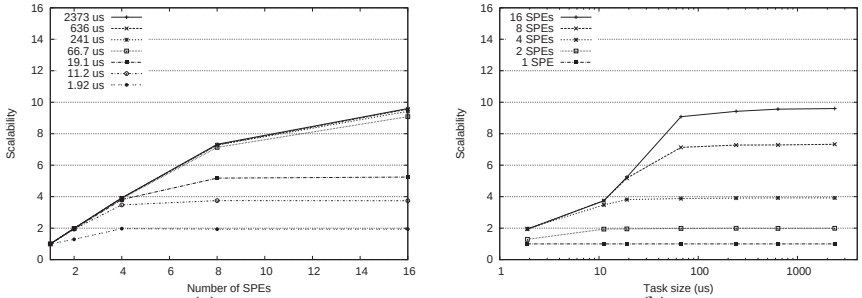


Figure 6.8: Scalability of StarSS with the CD benchmark and the $(1, 1, 4)$ configuration.

6.3.2 SD Benchmark

In the SD benchmark, tasks corresponding to different rows are completely independent of each other, because tasks only depend on their left neighbor. The difference in dependency pattern has several effects on the runtime system. First, this greatly simplifies the task dependency graph, which in turn might lead to decreased runtime overhead. Second, the dependency pattern enables the runtime system to create efficient bundles, further reducing the overhead. Our measurements, however, show only a small improvement in scalability compared to the CD benchmark.

Analysis of the scalability of the SD benchmark with the $(1, 1, 1)$ configuration shows hardly any difference with the CD benchmark. The creation of bundles improves scalability a little. The $(1, 1, 8)$ configuration is optimal in terms of performance and scalability and provides a maximum scalability of 14.5. Allowing larger bundles does not improve performance. Figure 6.9 depicts the scalability of the SD benchmark for different task sizes.

For a task size of $19\mu s$ a scalability of 5.6 is obtained. Analysis of the Paraver trace, depicted in Figure 6.10, shows that the main thread is the bottleneck. It is continuously busy with user code (26.6%), adding tasks (50.9%), and removing tasks (20.6%). On average per task these phases take $2.0\mu s$, $3.8\mu s$, and $1.5\mu s$, respectively. This indicates that the time taken by building and maintaining the task dependency graph is not much affected by the dependency pattern, unlike expected.

The stress on the helper thread, on the other hand, is significantly reduced by the use of bundles. On average, the time spent on scheduling, preparation of

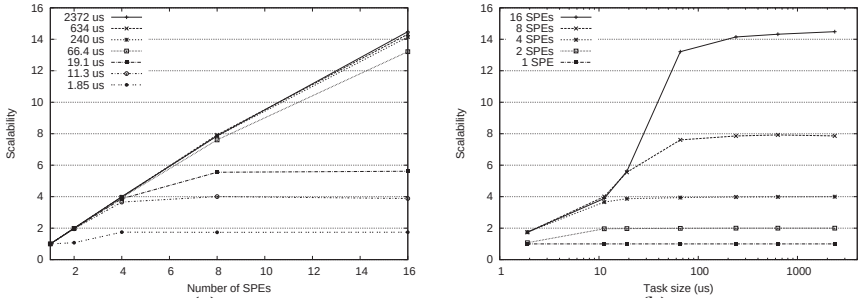


Figure 6.9: Scalability of StarSS with the SD benchmark and the $(1, 1, 8)$ configuration.

the bundles, and submission of the bundles is $1.86\mu s$, $0.10\mu s$, and $0.20\mu s$, respectively.

From this, one might conclude that the helper thread to SPE communication is sufficiently efficient to be scalable. This is not the case, however, due to the round-robin polling mechanism used by the helper thread. The Paraver trace depicted in Figure 6.7 shows that the helper thread checks the status of the SPEs in a round-robin fashion. The SPE's finish signal is not directly handled by the helper thread, but only when it is its turn to be checked. Even if the helper thread spends only $2.16\mu s$ per SPE, in the worst case situation it takes $34.56\mu s$ before the finish signal is handled.

6.3.3 ND Benchmark

The scalability of the ND benchmark is depicted in Figure 6.11. Similar to the SD benchmark, the $(1, 1, 8)$ configuration is optimal as it allows the runtime system to create bundles. The maximum obtained scalability is 15.8, which is significantly higher than with the two other benchmarks. This is because all tasks are independent. Thus, first, there is no ramping effect and second, as soon as 16 tasks have been added, the runtime system detects 16 independent tasks to execute.

The improvement for smaller task sizes is also due to the increased available parallelism. The time spent on building and maintaining the task graph, however, is approximately equal to the time spent on these for the SD benchmark, limiting scalability. We conclude that not the dependencies by itself are time-consuming, but the process of checking for dependencies.

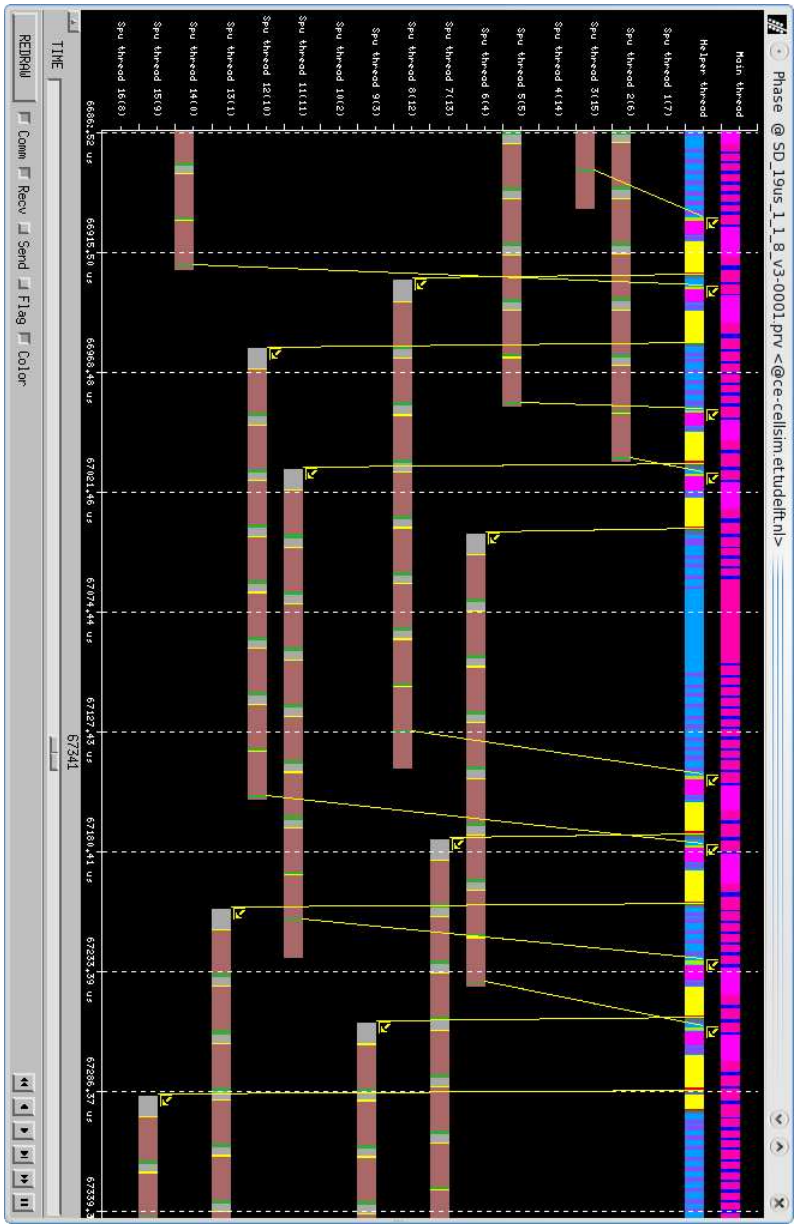


Figure 6.10: Partial Paraver trace of the SD benchmark with a task size of $19\mu s$ and the $(1, 1, 8)$ configuration. The main thread bottlenecks the system.

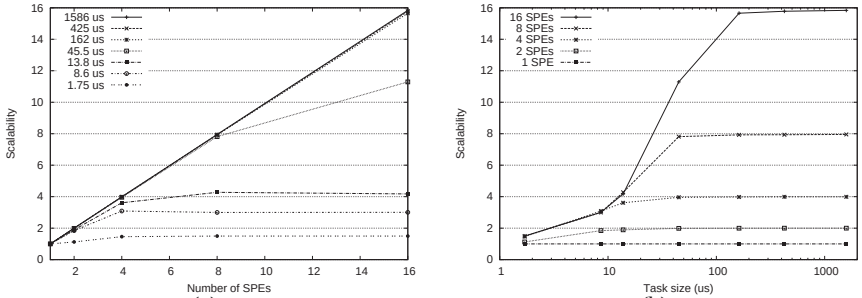


Figure 6.11: Scalability of StarSS with the ND benchmark and the $(1, 1, 8)$ configuration.

6.4 A Case for Hardware Support

In the previous section we analyzed the StarSS system and showed that the current runtime system has a low scalability for fine-grained tasks. To show the potential gains of accelerating the runtime system, in this section we compare StarSS to manually parallelized implementations of the benchmarks using a dynamic task pool. Based on the analysis presented in this and the previous section, we identify the bottlenecks in the StarSS runtime system. Those lead to a set of requirements that hardware support for task management should comply with.

6.4.1 Comparison with Manually Parallelized Benchmarks

The benchmarks are also manually parallelized. The task functions were offloaded to SPE code. On the PPE, tasks are added to a task pool, which dynamically keeps track of task dependencies. An interface between the SPEs and the PPE was implemented, such that the task pool, running on the PPE, assigns tasks to SPEs. As for each task the dependencies are predetermined, the process of dependency resolution has become very simple and fast. A simple Last-In-First-Out (LIFO) buffer is used to minimize the scheduling latency. The implementation of these manually parallelized benchmarks took a lot of time compared to the implementations of the StarSS benchmarks. Especially, building and optimizing the task pool took significant effort [43].

The scalability of the benchmarks with the manually coded task pool is shown in Figures 6.12, 6.13, and 6.14. The first figure shows that for a task size of

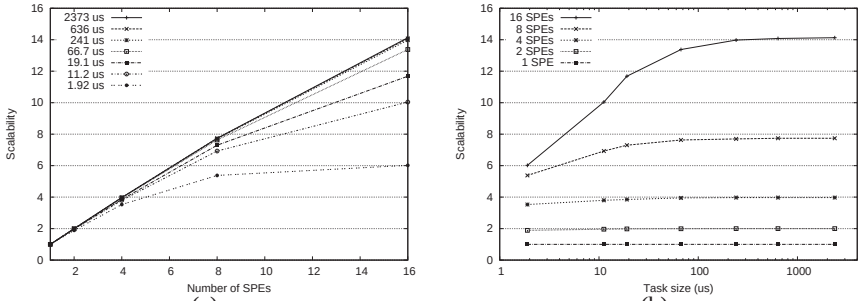


Figure 6.12: Scalability of the manually parallelized CD benchmark.

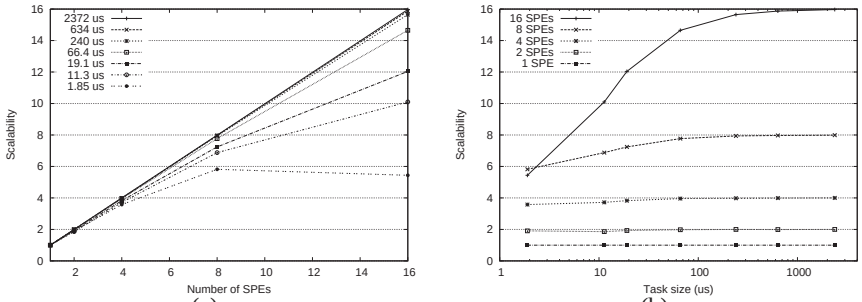


Figure 6.13: Scalability of the manually parallelized SD benchmark.

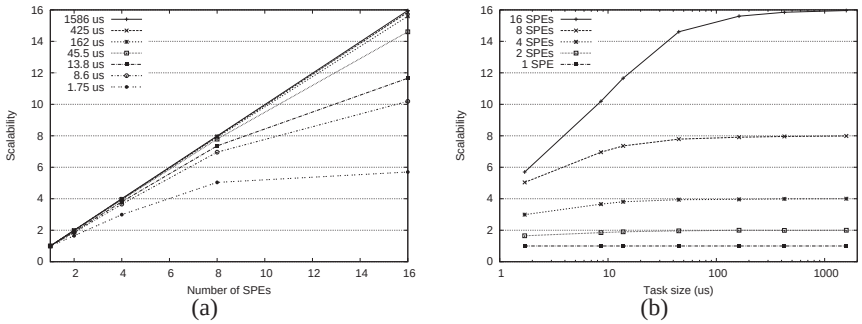


Figure 6.14: Scalability of the manually parallelized ND benchmark.

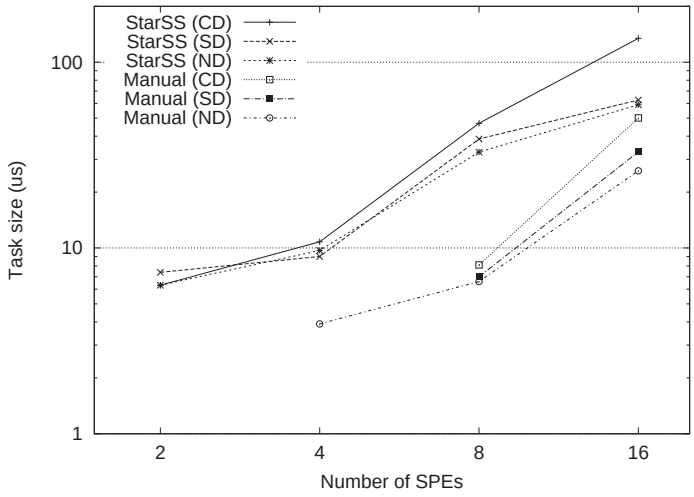


Figure 6.15: The iso-efficiency lines of the StarSS and Manual systems (80% efficiency).

$19\mu s$, the scalability of the CD benchmark using 16 SPEs is almost 12. This is much higher than the scalability of 4.8 obtained with StarSS. Similarly, the scalability of the other two benchmarks for small task sizes is significantly higher compared to StarSS.

Although the scalability of the manually parallelized benchmarks is much better than the scalability obtained with StarSS, for small task sizes it is still limited. Similar to StarSS, these benchmarks suffer from the round-robin polling mechanism used by the PPE to handle mailbox messages from the SPEs. The PPE spends $3.5\mu s$ per task on updating the dependency table, submitting tasks to the ready queue, and submitting a new task to the SPE. As SPEs have to wait their turn, typically they are waiting $16\mu s$ before receiving the new task. A significant performance improvement can be made when the hardware would provide faster synchronization primitives.

Figure 6.15 compares the scalability of the StarSS system to that of the manually parallelized (Manual) system. It displays the iso-efficiency lines, which represent the minimal task size required for a certain system in order to obtain a specific scalability efficiency. Scalability efficiency is defined as S/N, where S is the scalability and N is the number of cores. As scalability is defined relative to one SPE, in this case we define N as the number of SPEs, ignoring the PPE. In Figure 6.15 the iso-efficiency graph for an efficiency of 80% is depicted. Table 6.1 lists the values of the iso-efficiency function. For the Manual

Table 6.1: The iso-efficiency values of the StarSS and Manual systems (80% efficiency).

no. SPEs	Task size (μs)					
	StarSS			Manual		
	ND	SD	CD	ND	SD	CD
2	6.3	7.4	6.3			
4	9.7	9	10.8	3.9		
8	32.8	38.6	46.9	6.6	7	8.1
16	59	62.6	134.4	26	32.9	50.1

system and a small number of SPEs, an efficiency of 80% is always obtained, irrespective the task size. Therefore, the corresponding iso-efficiency points do not exist and are not displayed in the graph and table.

From the iso-efficiency graph it is clear that the Manual system achieves an efficiency of 80% for much smaller task sizes than StarSS does. For eight SPEs the difference in task size is between a factor of 5 and 6, while for 16 SPEs the difference is between 2 and 3 times. The main cause for this difference is the dependency resolution process of the StarSS runtime system. The iso-efficiency lines of both system increase with the number of SPEs. This is expected as the overhead increases with the number of SPEs, and thus longer tasks are required to hide the task management latency on the PPE. Especially the polling mechanism used to synchronize incurs increasing overhead for larger core counts. For the Manual system this is the main reason for the line increase. Furthermore, increasing the core count beyond eight requires off-chip communication, increasing the synchronization latency and reducing scalability.

Besides a lower scalability, also the absolute performance of StarSS is lower, up to four times, than that of the Manual System for fine-grained tasks. Table 6.2 depicts the execution times of the three benchmarks. For larger task sizes the difference diminishes as the task management overhead becomes negligible compared to the task execution time. The task sizes used in the ND benchmark are slightly different from those used in the other benchmarks. This is due to the way the `spend_some_time()` function is implemented.

6.4.2 Requirements for Hardware Support

From the prior analysis, we conclude that the StarSS runtime system currently does not scale well for fine-grained task parallelism. The manually parallelized

Table 6.2: Absolute execution times, in ms, for different task sizes, in μs, of the three benchmarks for the StarSS and Manual systems, and using 8 SPEs.

Benchmark	System	Execution time per task size						
		1.92	**11.2**	**19.1**	**66.7**	**241**	**636**	**2373**
CD	StarSS	23	23	23	41	132	340	1250
	Manual	6.0	10	13	39	130	337	1250
		1.85	**11.3**	**19.1**	**66.4**	**240**	**634**	**2372**
SD	StarSS	17	17	18	39	128	330	1230
	Manual	4.1	9.4	13	37	126	327	1210
		1.75	**8.6**	**13.8**	**45.5**	**162**	**425**	**1586**
ND	StarSS	18	19	18	26	86	221	818
	Manual	4.5	7.0	10	26	86	221	813

benchmarks, though, showed that it is possible to achieve a much higher scalability, while using dynamic task management.

The main bottleneck of the StarSS runtime system is the dependency resolution process. Irrespective of the actual dependency pattern, determining whether there are task dependencies is a laborious process that cannot keep up with the parallel task execution. In order to efficiently utilize 8 or 16 SPEs for H.264 decoding, roughly corresponding to the CD benchmark with a task size of $20\mu s$, the process of building and maintaining the task dependency graph should be reduced from $9.1\mu s$ to $2.5\mu s$ and $1.3\mu s$, respectively. Therefore, we describe the first requirement for task management hardware support as follows:

Requirement 1 *The hardware support system should accelerate the task dependency resolution process by at least a factor of 3.6 and 7.3, when using 8 and 16 SPEs, respectively.*

Preferably, this process is accelerated even more, such that it can run ahead of execution in order to reveal more parallelism. As such a large speedup is required we expect that some hardware acceleration, such as special instructions, will not be sufficient. Instead, we propose to perform this process completely in hardware.

The second bottleneck is the scheduling, preparation, and submission of tasks, performed by the helper thread in StarSS. In total this per-task overhead is $5.5\mu s$, where at most a latency of $2.8\mu s$ and $1.4\mu s$ is required, for 8 and 16 SPEs, respectively.

Requirement 2 *The hardware support system should accelerate scheduling, preparation, and submission of tasks by at least a factor of 2.0 and 3.9, when using 8 and 16 SPEs, respectively.*

The scheduler tries to minimize the overhead by creating bundles of tasks, and tries to exploit data locality by grouping a dependency chain within one bundle. For small tasks, however, the scheduling costs more than the benefits it provides. A rudimentary, yet fast, scheduling approach might therefore be more efficient than a sophisticated one.

The latency of synchronization on the Cell processor is too long due to the polling mechanism. Interrupts are more scalable than the polling mechanism, but still need to execute a software routine. Instead, SPEs should be able to retrieve a new task themselves instead of waiting for one to be assigned. Hardware task queues, such as Carbon [94], is an example of such an approach. Each core can read a task from the queue autonomously, avoiding off-chip communication and execution of a software routine. This approach does not only provide benefit for the Cell processor, but also using a more conventional multicore [94]. Assuming a 5% overhead for synchronization is acceptable, the synchronization latency should be at most $1\mu s$, irrespective of the number of cores.

Requirement 3 *The hardware support system should provide synchronization primitives for retrieving tasks, with a latency of at most* $1\mu s$*, irrespective of the number of cores.*

When compliant with these three requirements, a hardware support system will effectively improve the scalability of StarSS, while maintaining its ease of programming. In the next section we propose the Nexus system, which is developed based on this set of requirements.

6.5 Nexus: a Hardware Task Management Support System

In this section we present the design of the Nexus system, which provides the required hardware support for task management in order to efficiently exploit fine-grained task parallelism with StarSS. Nexus can be incorporated in any multicore architecture. In this section, as an example, we present a Nexus design for the Cell processor.

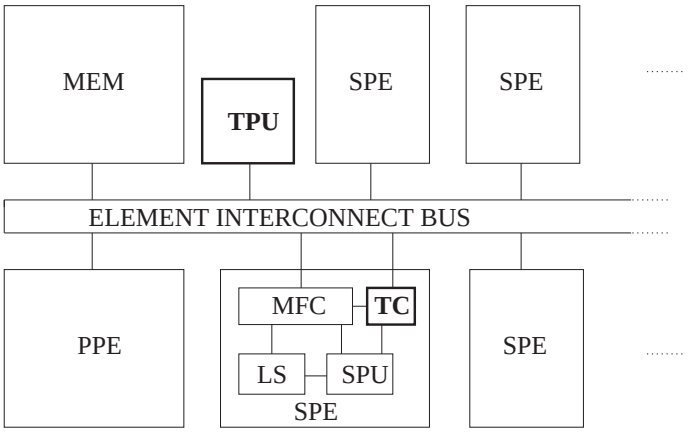


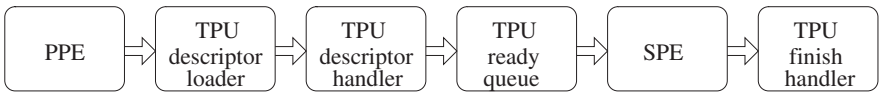
Figure 6.16: Overview of the Nexus system (in bold) implemented in the Cell processor. The depicted architecture could be one tile of a larger system.

First, we provide an overview of the Nexus system and explain its main operation based on the life cycle of a task. Second, we present the design of the Task Pool Unit (TPU), which is the main hardware unit of the Nexus system. Finally, we present the Nexus API.

6.5.1 Nexus System Overview

The Nexus system contains two types of hardware units (see Figure 6.16). The first and main unit is the Task Pool Unit (TPU). It receives tasks descriptors from the PPE, which contain the meta data of the tasks, such as the function to perform and the location of the operands. It resolves dependencies, enqueues ready tasks to a memory mapped hardware queue, and updates the internal task pool for every task that finishes. The TPU is designed for high throughput and therefore it is deeply pipelined. It is directly connected to the bus to allow fast access from any core.

The second unit in the system is the Task Controller (TC), which can be placed at individual cores. The TC fetches tasks from the TPU, issues the DMAs of input and output operands, and implements double buffering. While the first task is executed, the inputs of the next are loaded. TCs are optional as the TPU is memory mapped and thus can be accessed by any core directly. No design of the TC has been developed as of yet. In the future work section of this chapter we discuss some implementation details, though.



Phase 1	Phase 2	Phase 3	Phase 4	Phase 5	Phase 6
Create task descriptor; write to in buffer	Load task descriptor to task storage	Process task descriptor; fill the tables	Add ready tasks to ready queue	Read ready queue; load task descriptor; execute task; write task id to finish buffer	Handle finish buffer; process kick−off lists

Figure 6.17: The phases of the task life cycle and the corresponding hardware units.

Storing all tasks in a central unit provides low latency dependency resolution and scheduling. The hardware queues ensure fast synchronization. Such a centralized approach, however, eventually creates a scalability bottleneck for increasing core count. In such a case, a clustered approach can be used. Tasks are distributed among TPUs such that inter-TPU communication is low while task stealing allows system-wide load balancing.

6.5.2 The Task Life Cycle

The main operation of the Nexus system is described by the task life cycle illustrated in Figure 6.17. The life cycle of tasks starts at the PPE which prepares the task descriptors dynamically. It sends a task descriptor pointer, as well as the size of the task descriptor, to the TPU. In the second phase, the TPU loads the descriptor into a task storage, where it remains until the task is completely finished. The task storage can be thought of as the local store of the TPU.

Once the task descriptor is loaded, in the third phase, the information in the descriptor is processed. The new task is added to the task graph, by checking for dependencies on other tasks. The TPU performs this dependency resolution by using three tables, as explained in the next section. This phase, therefore, consists of filling the tables with the required information, which consumes only little time as will be shown.

The task resides in the tables until its dependencies are resolved. This process is handled by the TPU in the fourth phase. It maintains the dependency graph throughout the execution, detects when the task can be executed, and then adds the task to a ready queue.

In the fifth phase, the task is executed. First, the SPE reads the task id from the ready queue. This process can be performed by either its TC or by software running on the SPU. The task descriptor is loaded from the task storage after which the task operands are loaded into the local store using DMA commands. When execution is finished and the task output is written back to main memory, the task id is signaled to the TPU.

Finally, in the sixth phase, the TPU handles the finish signal sent by the SPE. Based on the id of the task, it updates the tables, which thereafter indicate which tasks have no dependencies left. It then adds these tasks to the ready queue. Thus, the sixth phase of the current task equals the fourth phase of the tasks that are added to the ready queue.

6.5.3 Design of the Task Pool Unit

The block diagram of the Nexus TPU is depicted in Figure 6.18. Its main features are the following. The task life cycle is pipelined to increase throughput. Dependency resolution consists of table lookups only, and therefore has low latency. All task descriptors are stored in the task storage to avoid off-chip communication. These three features are described in detail below.

Pipelining

The task life cycle is pipelined using the phases described in the previous section. For example, while the TPU is loading the descriptor of the first task, the PPE prepares the descriptor of the second task. Similarly, the other phases are pipelined. As a consequence, the TPU contains separate units for phases 2, 3, and 4/6 that operate concurrently. As explained in the previous section, phases 4 and 6 belong to the same operation and therefore are mapped to a single unit.

Phase 2 is performed by the *descriptor loader*. It reads the *in buffer* and issues a DMA to load the task descriptor into the *task storage*. All ports are memory mapped and can be accessed from anywhere in the system. The *descriptor handler* is responsible for phase 3. It reads the task descriptor from the *task storage* and performs the dependency resolution process, as described below. Finally, phases 4 and 6 are performed by the *finish handler*. It reads task ids from the *finish buffer* and updates the tables.

The three tables, the two buffers, the *ready queue*, and the *task storage* all have a fixed size. Thus, they can be full in which case the pipeline stalls. For example, if the *task storage* is full, the *descriptor loader* stalls. If no entry

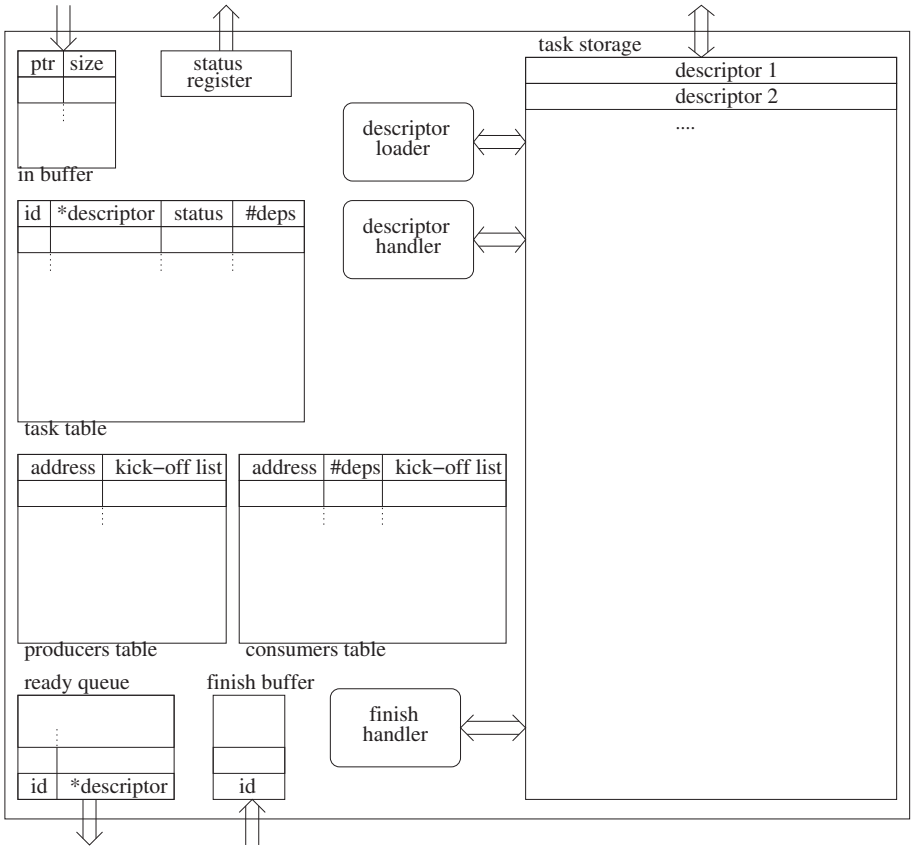


Figure 6.18: Block diagram of the Nexus TPU. The ports are memory mapped and can be accessed anywhere from the system.

of the *task storage* is removed, the *in buffer* will quickly be full too, which stalls the process of adding tasks by the PPE. Deadlock cannot occur, however, because in the StarSS programming model tasks are added in serial execution order. That means that if tasks are processed in this order sequentially, the execution will be correct.

Dependency Resolution

Generally, task dependencies are expressed by using a task dependency graph. Building such a task dependency graph, however, is very time-consuming. In the TPU, dependencies are resolved by using three tables. Thus, the dependency resolution process can be performed fast, as it consists of simple lookups

only. There is no need to search through the tables to find the correct item.

The *task table* contains all tasks in the system and records their status and the number of tasks it depends on. Tasks with a dependency count of zero are ready for execution and added to the *ready queue*. The *task table* is indexed by the task id.

The *consumers table* is used to prevent write-after-read hazards. It contains the addresses of data that is going to be produced by a pending task. Any task that requires that data can subscribe itself to that entry by adding its id to the kick-off list and increasing its own dependency count in the *task table*. When a task finishes, the *finish handler* processes the kick-off lists of the entries in the *consumers table* corresponding to the addresses of the output operands of the task. That is, for each task id in the kick-off lists, it decreases the dependency count of that task in the *task table*.

As example, assume task $T1$ writes to address A, and task $T2$ reads from address A. Thus, task $T2$ depends on task $T1$. When task $T1$ is added by the *descriptor handler*, an entry for address A is generated in the *consumers table*, with an empty kick-off list. Task $T1$ is also added to the *task table*. When task $T2$ is added, the *consumers table* is checked for address A. As it is present, id $T2$ is added to the kick-off list of that entry. Further, task $T2$ is added to the *task table* with a dependency count of one. Once task $T1$ finishes, the *finish handler* reads the entry of address A in the *consumers table*. In the kick-off list, it finds the id of task $T2$ and decreases the dependency count of task $T2$ in the *task table*. As the dependency count is zero, task $T2$ is added to the *ready queue*.

Similarly, the consumers table prevents read-after-write hazards. This table contains the addresses of data that are going to be read by pending tasks. Any new task that will write to these locations can subscribe itself to the kick-off list.

As example, assume task $T1$ reads from address A while task $T2$ writes to address A. In this case task $T2$ is indirectly depending on task $T1$. Task $T2$ does not need the output of task $T1$, but it should not write to address A before task $T1$ has read from it. This read-after-write hazard is handles as follows. When task $T1$ is added by the *descriptor handler*, an entry for address A is generated in the consumers table. The dependency count of that entry is set to one, while the kick-off list remains empty. In case an entry for address A already exists only the dependency count of that entry is increased by one. Further, task $T1$ is added to the *task table*. When task $T2$ is added by the *descriptor handler*, the consumers table is checked for address A. As the entry

is found, the id of task $T2$ is added to the kick-off list of that entry. Further, task $T2$ is added to the *task table* with a dependency count of one. When task $T1$ finishes, the *finish handler* decreases the dependency count of entry A in the consumers table. If it reaches zero, all consumers of address A have read the data, and thus the kick-off list is processed. The id of task $T2$ is found in the kick-off list and thus its dependency count in the *task table* is decreased by one. It reaches zero, and thus it is added to the *ready queue.*

Finally, write-after-write hazards are handled by the *producers table.* A task writing to a particular address that is already in the *producers table,* subscribes itself to the corresponding entry. This subscription, however, is performed in a different manner, as explained in Section 6.5.3.

The lookups in the producers and consumers tables are address-based. As the lists are much smaller than the address space, a hashing function is used to generate the index.

Task Storage

The *task storage* contains only task descriptors. The amount of data maintained within a task descriptor depends on the number of task operands. Thus variable sized task descriptors might be used to optimally utilize the available memory. That requires complicated memory management, however.

Instead, fixed-size task descriptors are used. Typically, tasks have a few operands only. Assuming a maximum of five operands, the task descriptor is 64 bytes large. The *task storage* is divided in slots of 64 bytes. For each slot one bit is required to indicate if it is occupied. Task ids correspond to the slot of its task descriptor in the *task storage* and are assigned by the descriptor loader.

Note that the source-to-source compiler can detect tasks with more than five operands. It can warn the programmer or schedule the tasks to the PPU without intervention of the TPU.

Kick-off List Barriers

The *consumers table* contains one kick-off list per address. This is sufficient as long as there is only one producer per address. If multiple tasks write to the same address, multiple kick-off lists for that address are required.

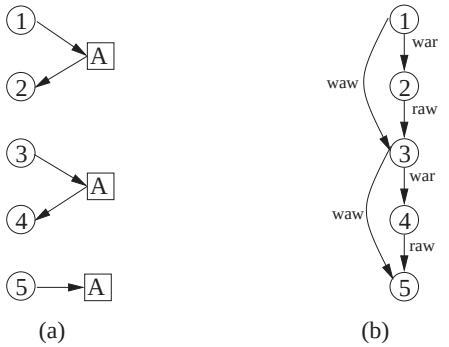


Figure 6.19: (a) An example of tasks writing to and reading from the same address A and (b) the corresponding task dependency graph.

Consider the example depicted in Figure 6.19. The left side of the figure depicts the data that is produced or consumed by tasks $T1$ through $T5$. The right side of the figure illustrates the dependencies. First, task $T1$ writes to location A, after which task $T2$ reads from location A. Adding task $T1$ by the *descriptor handler*, adds the address of A to the *consumers table*. Adding task $T2$ adds id 2 to kick-off list of address A in the *consumers table*. Next, task $T3$ also writes to location A, after which task $T4$ reads the new value from location A. Addition of task $T3$ does not change the producer table. Addition of task $T4$, however, adds id 4 to the kick-off list of address A in the producer table, assuming one kick-off list per address is maintained. If now task $T1$ finishes, the kick-off list of address A in the producer table is processed and both task $T2$ and $T4$ are put in the *ready queue*. Task $T4$, however, is depending on task $T3$ which has not been executed. Similarly, task $T5$ is kicked-off from the consumers table before task $T4$ is executed.

We solved this issue by using barriers in the kick-off lists. Barriers divide kick-off lists in multiple parts, as if they are separate lists. A task that writes to an address that is already in the *producers table*, adds a barrier to the kick-off list of the corresponding entry. Similarly, a task that reads from an address already in the *consumers table*, and which entry has a non-empty kick-off list, adds a barrier to that kick-off list.

This process is illustrated in Figure 6.20 for the example of Figure 6.19. Each row shows the changes in the relevant entries of the three tables for a particular step. In the top five rows, the tasks 1 through 5 are added to the system, which is done by the *descriptor handler*. The bottom five rows show how the *finish handler* updates the tables when it processes ids 1 through 5.

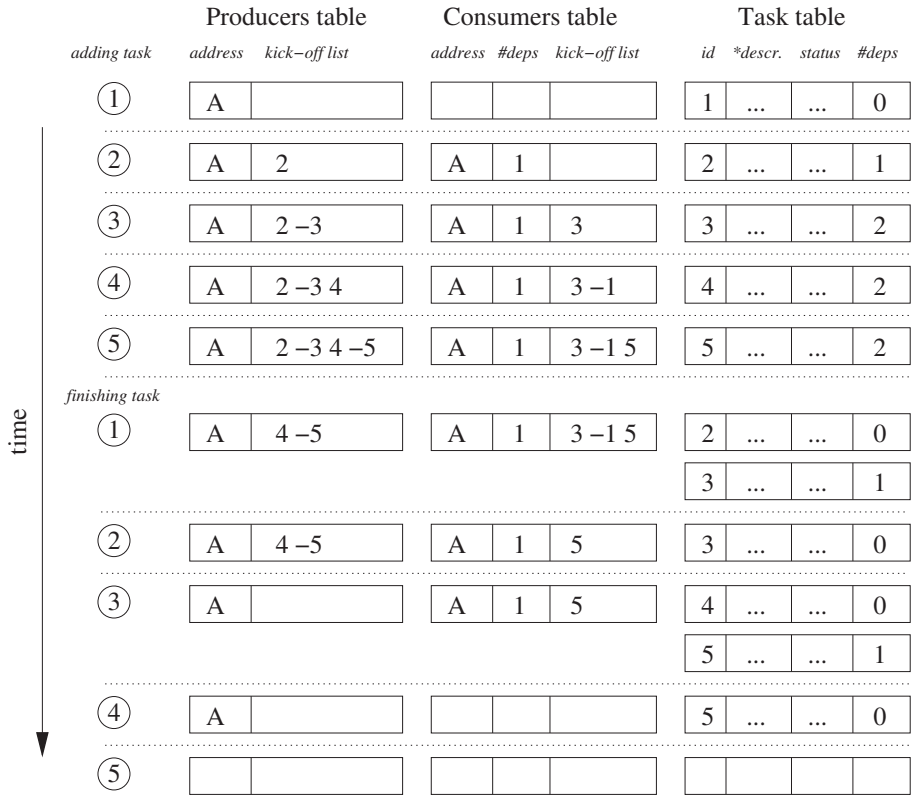


Figure 6.20: The dependency resolution process for the example of Figure 6.19.

In the first step, address A is added to the *consumers table* and task $T1$ is added to the *task table* with a dependency count of 0. In the second step, an entry for address A is created in the consumers table, task $T2$ subscribes to address A in the *consumers table*, and task $T2$ is added to the *task table* with a dependency count of 1. In the third step, task $T3$ has to subscribe to address A in the *consumers table* to prevent a write-after-write hazard. It also has to add a barrier. These two are combined by writing the negative task id in the kick-off list. Task $T3$ also subscribes to address A in the consumers list, because of the dependency on task $T2$. Task $T3$ is added to the *task table* with a dependency count of 2. In the fourth step task $T4$ is added. It subscribes to address A in the *consumers table* as normal due to its dependency on task $T3$. Further, it places a barrier in the consumers table by adding a value of -1 to the kick-off list. In the consumers table the barriers are indicated with negative values too. The

absolute value of the barrier indicates the number of tasks that need to read the data in the corresponding address. Thus, it is similar to the dependency count value in the entries of the consumer table. If another task that reads from address A would be added now, the barrier would be decremented instead of adding another barrier. The addition of task $T5$ in the fifth step is similar to the third step.

When task $T1$ finishes, the *finish handler* processes the kick-off list in the producer table for address A up to and including the barrier. Thus, the dependency counts of both task $T2$ and $T3$ are decremented. The dependency count of the first reaches zero, and thus task $T2$ is added to the *ready queue*. Further, task $T1$ is removed from the *task table*. The consumers table is unaltered. When task $T2$ finishes, the entry for address A in the consumers table is processed. The dependency count in that entry is decremented and becomes zero. Thus the kick-off list is processed up to the barrier, which adds task $T3$ to the *ready queue*. The barrier itself is negated and becomes the new dependency count of that entry. Similarly, in the final three steps the tables are updated until all tasks have completed and the tables are empty.

6.5.4 The Nexus API

In this section the software interface to the Nexus system is described. It shows the relation between the programming model, the runtime system, and the TPU.

Initialization

The Cell processor works with virtual addresses and physical addresses, referred to as effective and real addresses, respectively. The real address space is the set of all addressable bytes in physical memory and on memory mapped devices, such as the TPU. Programs, though, use effective addresses only, and thus a translation is always necessary.

The real address of the TPU is fixed and known a priori. Its mapping to the effective address space is a runtime issue, and requires an *mmap* system call. Therefore, before any program can use the TPU, an initialization must take place, in which the effective address of the TPU is retrieved. Listing 6.3 shows how the TPU can be initialized. The function `init_tpu()` is part of libtpu, which is the runtime system of Nexus.

Listing 6.3: Initialization retrieves the effective address of the TPU from the OS.

```
1 #include <libtpu.h>
2
3 tpu_handle_t     tpu_handle;
4
5 int main(){
6     tpu_handle = init_tpu();
7     ...
8 }
```

Adding Tasks

The user specifies tasks with pragmas as shown in Listing 6.2, while the TPU requires a task descriptor. The source-to-source compiler analyzes the user code and transforms it into code that creates and initializes the task descriptor. For example, the code of Listing 6.2 would be translated into that of Listing 6.4.

The task descriptor has a header and body, which contains the parameter information. The header specifies the task function and the number of parameters it takes. In this case the function `task` was assigned number 1 by the source-to-source compiler. This number is used by the SPEs to call the correct function. These functions are extracted from the source code by the source-to-source compiler and put in the SPE code with an automatic generated main function. For each task parameter several fields are specified in the task descriptor. The io type can be IN, OUT, or INOUT, which are defined in libtpu. The other four entries specify the location and size of the parameter, which can be stored in a strided fashion.

Once the task descriptor is filled, the `tpu_add_task` runtime system function is called. It requires as arguments the tpu handle, a pointer to the task descriptor, and its size, although the last parameter is not necessary for the current TPU implementation. The runtime system copies the pointer and the size to the *in buffer* of the TPU, and not the descriptor itself. If the *in buffer* is full, it overwrites the last item. Thus, the function first checks the status register of the TPU. The implementation of the `tpu_add_task` function is shown in Listing 6.5. The status register is a 32-bit value, of which the first byte contains the number of empty elements in the *in buffer*, and the second byte contains the state of the TPU. The other two bytes are currently not used.

Listing 6.4: Preparation of the task descriptor and submission of it to the TPU.

```
1  task_descriptor_t* task_d_list = malloc(....);
2
3  task_descriptor_t* td_ptr = task_d_list;
4  for (i=0; i<64; i++){
5      for (j=0; j<64; j++){
6          task_descriptor_t td;
7          td_ptr->task_func = 1;
8          td_ptr->no_params = 4;
9          td_ptr->p1_io_type = IN;
10         td_ptr->p1_start_ptr = &my_params;
11         td_ptr->p1_x_length = sizeof(params_t);
12         td_ptr->p1_y_length = 1;
13         td_ptr->p1_y_stride = 0;
14         td_ptr->p2_io_type = IN;
15         td_ptr->p2_start_ptr = A[i][j-1];
16         td_ptr->p2_x_length = 16*sizeof(int);
17         td_ptr->p2_y_length = 16;
18         td_ptr->p2_y_stride = 16*sizeof(int);
19         td_ptr->p3_io_type = IN;
20         td_ptr->p3_start_ptr = A[i-1][j+1];
21         td_ptr->p3_x_length = 16*sizeof(int);
22         td_ptr->p3_y_length = 16;
23         td_ptr->p3_y_stride = 16*sizeof(int);
24         td_ptr->p4_io_type = INOUT;
25         td_ptr->p4_start_ptr = A[i][j];
26         td_ptr->p4_x_length = 16*sizeof(int);
27         td_ptr->p4_y_length = 16;
28         td_ptr->p4_y_stride = 16*sizeof(int);
29
30         tpu_add_task(tpu_handle, td_ptr,
31             sizeof(task_descriptor_t));
32         td_ptr++;
33 }   }
```

The Nexus library also provides the `tpu_add_task_do_while`. It tries to add a task to the TPU. If the *in buffer* is full, instead of waiting it retrieves tasks from the *ready queue* to execute in the mean time. Once there is a empty place in the *in buffer*, it adds the task and returns. In this way, the PPE is utilized optimally and performance can be improved.

Listing 6.5: The `tpu_add_task` runtime system function implementation.

```
1  void tpu_add_task(tpu_handle_t tpu, void* descriptor,
2                                                  size_t size){
3    tpu_status_t status;
4
5    assert(tpu);
6
7    do {
8      memcpy(&status, (tpu + STATUS_REGISTER), 4);
9    } while (status.in <= 0);
10
11   memcpy((tpu + IN_BUFFER), &descriptor, 4);
12   memcpy((tpu + IN_BUFFER + 4), &size, 4);
13 }
```

Listing 6.6: The runtime system provides a wait function which returns once all tasks have finished.

```
1  int main(){
2
3      ...
4      tpu_wait(tpu_handle);
5      return;
6  }
```

Finalization

After the PPE has added all tasks to the TPU it has to wait for them to finish before continuing with the shutdown or next phase of the program. The runtime system provides a blocking wait, which continuously checks the status of the TPU and returns once it indicates that all tasks are finished. Listing 6.6 shows the utilization of this function.

The Nexus runtime system also provides the functionality for the PPE to execute tasks while it is waiting for the parallel execution to finish, as shown in Listing 6.7. The function `tpu_finished` returns true if all tasks are finished. The function `tpu_try_get_task` returns the task id if there was a task in the *ready queue*, else it returns zero. The function `execute_task` is provided by the source-to-source compiler. It calls the task function, indicated by the function number in the descriptor, with the operands, which are also

Listing 6.7: The PPE can also obtain tasks from the TPU to execute them.

```
1  int main(){
2
3      ...
4      while(!tpu_finished(tpu_handle)){
5          task_descriptor_t td;
6          int task_id;
7          if(tpu_try_get_task(tpu_handle, &td, &task_id)){
8              execute_task(&td);
9              tpu_finish_task(tpu_handle, task_id);
10         }
11     }
12     return;
13 }
```

indicated in the descriptor. Finally, with `tpu_finish_task` the task id is sent to the *finish buffer* of the TPU.

So far the API concerned the PPE side of the code. The SPE side is depending on whether a TC is included in the system or not. For now, we assume there is no TC. Instead, runtime system functions are used to obtain and finish a task.

Listing 6.8 shows an example of the SPEs main function. With the `tpu_get_task` function, the SPE receives a task descriptor. The function first reads from the *ready queue*, until it finds a task there. Next, it DMAs in the task descriptor and returns. The `tpu_finish_task` is equal to that on the PPE, except for its internal implementation. The SPE code never returns. Instead, the SPEs are stopped by the PPE once all tasks have finished.

The function `load_and_execute_task` is provided by the source-to-source compiler. It reads the task descriptor, DMAs in all the input parameters, calls the proper task function, and finally DMAs back the output operands. Except for the task functions, all SPE code is generated automatically.

6.6 Evaluation of the Nexus System

In the previous section we have presented the design of the Nexus system. In this section we evaluate the implementation of Nexus for the Cell processor. First, we describe the experimental setup. Second, we present the performance and scalability analysis of the Nexus system.

Listing 6.8: The SPE main code retrieving tasks from the TPU.

```
1  #include <tpu_spu.h>
2
3  void main(unsigned long long id, unsigned long long argp) {
4      mm_ptr tpu = argp;
5
6      task_descriptor_t td __attribute__ ((aligned (16)));
7      int task_id;
8      while(1){
9          if(tpu_get_task(tpu, &td, &task_id)){
10             load_and_execute_task(&td);
11             tpu_finish_task(tpu, task_id);
12         }
13     }
14 }
```

6.6.1 Experimental Setup

Similar to Chapters 4 and 5, we use CellSim as evaluation platform (see also Section 4.3.3). In these previous chapters, however, we were only interested in the SPE, while in this chapter, the entire Cell processor is of importance. Therefore, the configuration of the PPE, cache, memory, and bus modules are validated, to ensure these simulator models are accurate.

In contrast to previous chapters, we validated CellSim against a real Cell processor, instead of validating against SystemSim, for the following reason. The additional information provided by SystemSim was not needed for this validation, while execution on the simulator is much slower than using the real processor. Although the Cell blade provides 16 SPEs, we used only 8 for this validation. CellSim can simulate 16 SPEs, but connects all of them to the same on-chip bus. As on the Cell blade these 16 SPEs are divided between two chips, the two architectures would be different. There is one architectural difference between the simulator and the real Cell processor that was not resolved. The real processor contains both an L1 and L2 cache on the chip. In the simulator only the L1 cache is available. We account for this difference in the configuration of CellSim.

We adjusted the configuration parameters to match the real processor. As CellSim is not cycle-accurate, an empirical methodology was used to determine the correct parameter values. More precisely, we measured several properties

Table 6.3: CellSim configuration parameters.

L1-latency	10
mem-latency	45
dma-latency	900
bus-latency	8
bus-width	16
ppu-fetch-rate	1.4
spu-fetch-rate	1.4

on both the real Cell processor and on the simulator. The corresponding parameters were adjusted until the properties measured on the simulator matched these of the real processor. For example, the latency of DMA transfers was adjusted with the parameters 'dma-latency', 'bus-latency', 'bus-bandwidth', and 'mem-latency'. The properties that we measured are the following:

- **Mailbox round-trip time:** The round-trip time of a mailbox signal from the PPE to the SPE and back.
- **DMA latency:** The time spent on issuing and waiting for DMAs.
- **SPU execution time:** The execution time of one task on the SPE. This does not include any DMAs, mailbox signals, or other communications.
- **PPE execution time:** The execution time of the serial CD benchmark running on the PPE.
- **Cell execution time:** The execution time of the parallel CD benchmark, when it is executed on the PPE and eight SPEs.
- **Cell scalability:** The scalability of the parallel CD benchmark, when it is executed on the PPE and eight SPEs. The scalability is relative to the parallel execution time on one SPE.

Most of the configuration parameters influence several properties resulting in a multi-dimensional matching problem. The most accurate configuration, presented in Table 6.3, was determined by performing many measurements. The memory access latency in the simulator is 45 cycles only, while using the real processor, off-chip memory accesses take in the order of a thousand cycles. As mentioned before, the simulator does not model the L2 cache. Thus a miss in the L1 cache goes directly to main memory while in the real processor it

Table 6.4: Validation of CellSim by comparing several properties of the simulator against those measured on the real Cell processor. All measures are in cycles, except scalability.

	Cell	CellSim	Error
Mailbox round-trip	1,248	1,216	-2.6%
DMA latency	7,048	6,524	-7.4%
SPU ex. time	61,446	61,898	0.7%
PPE ex. time	559,184	546,097	-2.3%
Cell ex. time	41,996,703	41,471,514	-1.3%
Cell scalability	7.4	7.0	-5.5%

goes to the L2 cache first. A memory access latency of 45 cycles corresponds to the average L1 miss penalty of the real processor. The memory access latency parameter, however, also affects the memory access time of DMAs. This was compensated by increasing the dma-latency parameter (an MFC module parameter) to 900 cycles, whereas the default value is 50 cycles.

Table 6.4 compares the properties of CellSim to that of the real Cell processor. Most properties are modeled within 3% accuracy. The DMA latency, however, has an error of 7.4%. This error can be lowered by increasing the dma-latency parameter. This would, however, negatively affect the scalability, which is 5.5% off in the current configuration. The current trade-off seems reasonable. Overall, the simulator accuracy is fairly well and sufficient for its purpose.

The TPU is implemented as one module in CellSim. Although the size of the tables and the storage is still subject to research, we use the following values in the experiments. The *task table* and the *task storage* have 1024 entries. This is the minimum size needed for the CD benchmark to detect 16 independent tasks. It is much less, though, than the total number of tasks (4096) in the benchmarks. Thus, the process of adding tasks is stalled once the *task table* and *task storage* are full.

The *producers* and *consumers tables* each have 4096 entries. This is sufficient to contain all inputs and outputs of the 1024 tasks in the *task table*. A significant number of hash collisions, however, happen. A coalesced hash algorithm [164] is used to resolve hash collisions. It allows the tables to be filled efficiently to a high density, while maintaining low latency table lookups.

The latencies of the TPU are determined as follows. The TPU functionality consists mainly of accessing the tables and the storage. In the real processor,

Table 6.5: Execution time of the CD benchmark, with a task size $19\mu s$, for the three systems. Three configurations of the Nexus runtime system are analyzed.

Nexus configuration	Execution Time (ms)			Speedup StarSS + Nexus	
	StarSS + Nexus	StarSS	Manual	vs StarSS	vs Manual
8 SPEs	11.1	22.3	13.0	2.02	1.17
8 SPEs - PPE	10.8	22.3	13.0	2.06	1.20
8 SPEs - PPE+	10.4	22.3	13.0	2.14	1.24

accessing the local store, which is 256 kB large, takes six cycles. Thus, it is reasonable to assume that the *task storage*, which is 64 kB large, can be accessed in two cycles. Similarly, the three tables, which are smaller than the *task storage*, were assigned a latency of one cycle. The *producers* and *consumers table* are accessed several times per lookup, due to hash collisions. Measurements showed that on average one lookup takes six cycles. Per task several lookups are required, depending on the number of parameters.

The latencies of the handlers are mainly based on the latencies of the tables and storages. Also cycles for control logic and arithmetic operations are added to the handler latencies. The total latency of the handlers, for each operation they perform, is depending on the actual input and the state of the tables. In the next section, which presents the performance evaluation, these latencies are analyzed for the benchmark.

6.6.2 Performance Evaluation

In this section we analyze the performance and the scalability of the Nexus system. We present results for the CD benchmark. Analysis for the other two benchmarks has yet to be performed. When comparing to the StarSS system that uses the software task management, we refer to that system as 'StarSS', while the system using the Nexus hardware task management is referred to as 'StarSS + Nexus'.

The execution time of the CD benchmark, with a task size of $19\mu s$ is shown in Table 6.5. Three configurations of the Nexus runtime system are analyzed. The first (8 SPEs) uses only the eight SPEs to execute tasks. In the second configuration (8 SPEs - PPE), the PPE also executes tasks, but only after it has added all tasks to the TPU. Finally, in the third configuration (8 SPEs - PPE+), the PPE also executes tasks while the addition of tasks to the TPU is stalled because the *in-buffer* is full.

The table compares the StarSS + Nexus system to the StarSS and the Manual systems. The execution of the benchmark using the StarSS system is measured on the Cell blade, as the code generated by the StarSS compiler cannot be executed on the simulator. Thus, for this particular analysis we used the hardware configuration for which the simulator was validated, i.e., we used eight SPEs.

The table shows that using the Nexus hardware instead of the software task management, a speedup of more than 2x is obtained, when using eight SPEs. As shown in Figure 6.8, for this task size and number of cores, the scalability of the StarSS system is 5.2. Section 6.3 showed that task management overhead limits the scalability and performance. As the performance has been improved by a factor of over 2x, the Nexus system has successfully removed the performance bottleneck for this case.

More details on the parallel execution of the benchmark are provided in Figure 6.22, which shows a part of the Paraver trace. The legend for this trace is depicted in Figure 6.21. The x-scale of the trace is in cycles, although the figure states ns. This is due to a mismatch in the metrics used in the simulator and the visualization tool. The trace shows that the SPEs are almost continuously executing tasks. The time it takes for an SPE to obtain a new task is extremely short when compared to the trace of the StarSS system (Figure 6.7). It takes 985 cycles, which is $0.3\mu s$, for the SPEs to read from the *ready-queue* of the TPU.

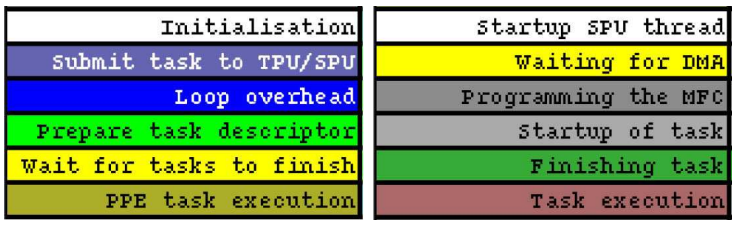


Figure 6.21: Legend of the StarSS + Nexus Paraver traces.

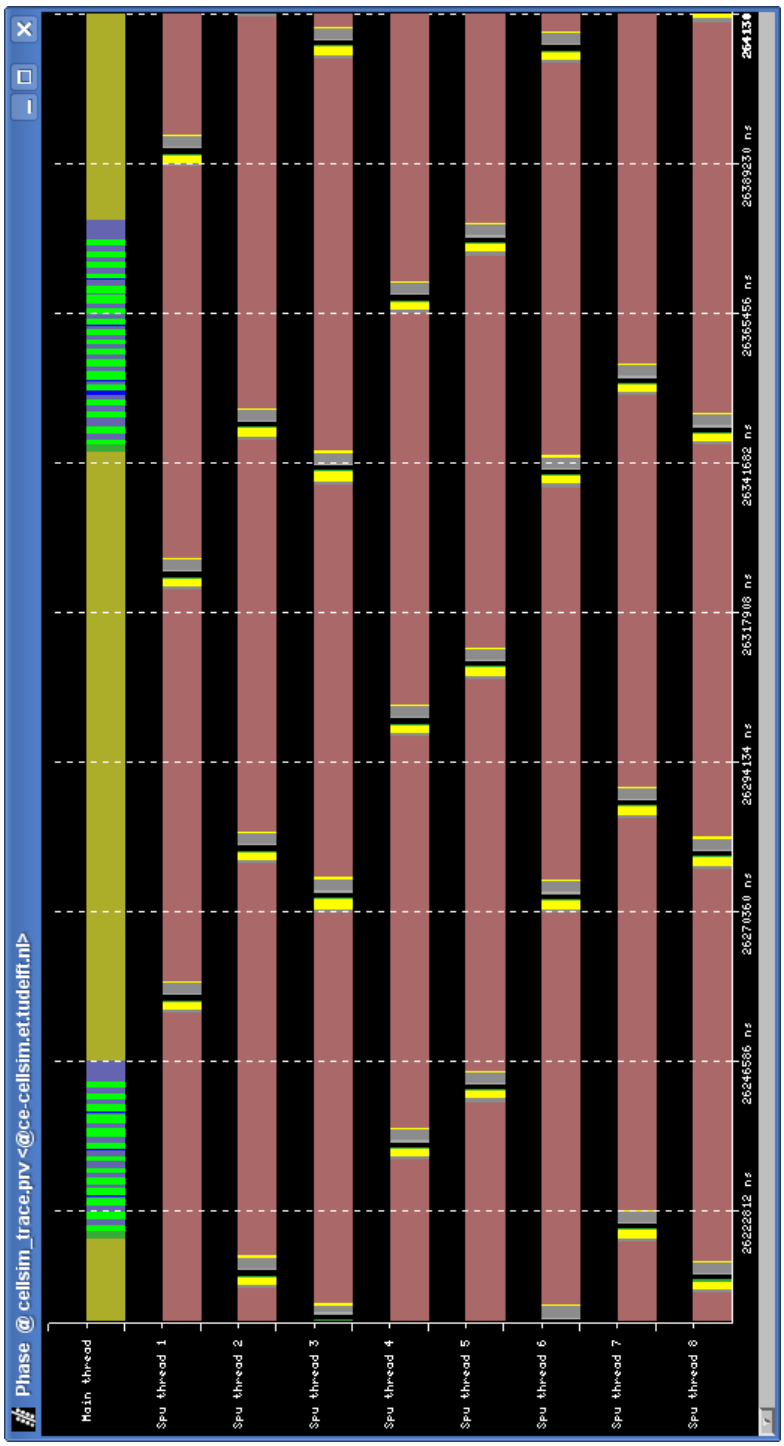


Figure 6.22: Partial Paraver trace of the CD benchmark with 8 SPEs, a task size of $19\mu s$, and using the Nexus system.

Table 6.6: The task throughput of the PPE and the TPU.

PPE (preparation and submission of tasks)	1116	cycles/task	2.9	tasks/μs
Descriptor loader	6	cycles/task	533	tasks/μs
Descriptor handler	81.8	cycles/task	39.1	tasks/μs
Finish handler	45.1	cycles/task	71.0	tasks/μs

Furthermore, the trace shows that the PPE needs very little time to prepare and add tasks to the TPU, when compared to the StarSS traces. On average, the PPE spends 1116 cycles on preparing and submitting one task. The trace also shows that the PPE executes a task when the *in-buffer* is full. After finishing the task, it continuously adds new tasks to the TPU. The *in-buffer* runs full because the *task table* is full, and not because the PPE generates tasks faster than the TPU can process them, as we show next.

Table 6.6 shows the throughput of the PPE and those of the TPU. The first was measured from the trace, while the others were extracted from the simulator. The PPE has the lowest throughput with 2.9 tasks/μs. The *descriptor loader* only has to issue a DMA, and thus has a very high throughput. The *descriptor handler*, on average, needs 81.8 cycles to process a task descriptor and has the lowest throughput of the TPU units as it performs the most complex operation of all the units. The *finish handler*, on average, needs 45.1 cycles per task id it processes.

These numbers show that the addition of tasks is slower than the finishing of tasks. Furthermore, the throughput of the addition process is limited by the PPE. The memory bandwidth utilization has not been considered so far, but it does not limit the throughput. Assume the PPE writes one task descriptor, of 64B, to main memory every 1116 cycles, and assume the TPU reads them at the same pace. Then the total bandwidth utilization is $2 * 64B * 3200MHz/1116cycles = 367MB/s$, while the available bandwidth to main memory is $25.6GB/s$. Thus, only 1.4% of the bandwidth is taken by the process of adding tasks. We conclude that the throughput of the Nexus system is limited by the PPE preparing and submitting tasks, and is maximally 2.9 tasks/μs.

The throughput of the Nexus system determines the maximum scalability that can be obtained. In other words, it determines how many cores can efficiently be used within the system. This scalability, however, also depends on the task size. In Figure 6.23, the limit to the scalability that can be obtained is shown

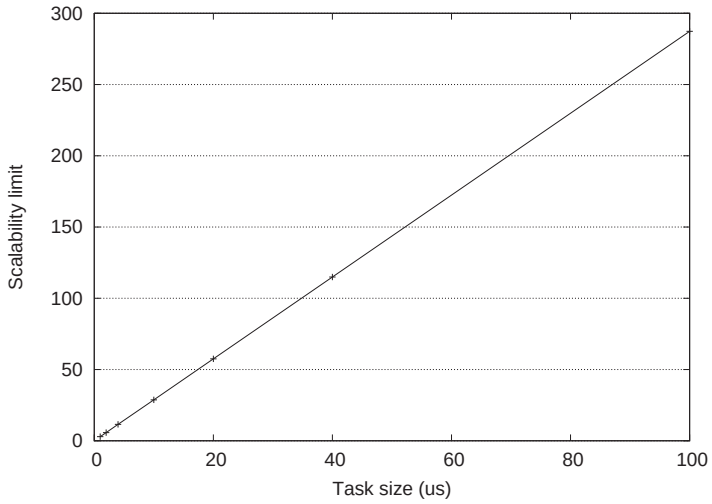


Figure 6.23: The limit to the obtainable scalability using the Nexus system, based on the maximum throughput.

for several task sizes. For example, for tasks of $20\mu s$, the scalability limit is $20 \times 2.9 = 58$. The figure shows that the scalability limit increases with the task size.

The TPU latencies are modeled as accurately as possible. As no real hardware has been developed, however, they might be incorrect to some extent. Therefore, we analyze the impact of increased TPU latencies on the performance. We use the CD benchmark with a task size of $19\mu s$, and only SPEs are executing tasks. The result is shown in Figure 6.24. The performance is normalized to that using the default latencies. The x-axis is the normalized TPU latency. A value of 10 indicates that all latencies of the TPU are multiplied by a factor of 10.

The figure shows that the performance is hardly influenced for up to a normalized latency of 100. This is explained as follows. Eight SPEs are used and the task size of $20\mu s$. Therefore, the execution rate is 0.4 tasks/μs. Within the TPU, the *descriptor handler* has the lowest throughput, which is 39.1 tasks/μs and which is directly related to the TPU latencies. When the latencies are increased with a factor of 100, the throughput increases also with a factor of 100 and becomes equal to the execution rate. When the latencies increase further, the throughput of the *descriptor handler* limits the throughput of the entire system and the performance decreases as shown in the figure.

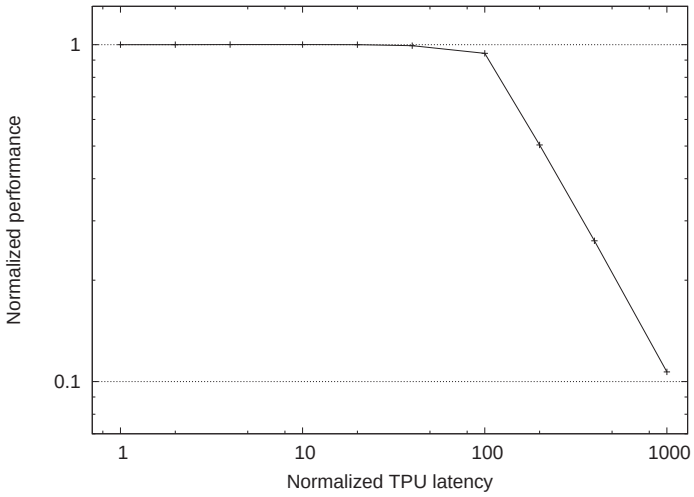


Figure 6.24: Impact of the TPU latency on the performance for a task size of $20\mu s$ and using eight SPEs.

The point in Figure 6.24, from where the performance drops, therefore depends on the task size and the number of SPEs used. The minimum value, however, is 13.5 because the performance is limited by the throughput of the PPE, which is 13.5 times larger than that of the *descriptor handler*. From this experiment we conclude that the claimed performance improvements, which can be obtained with the Nexus system, are reliable.

So far we have considered the single case of the CD benchmark with a task size of $19\mu s$ and using 8 SPEs. We have shown that the Nexus hardware provides large performance improvements and that the throughput of the Nexus system potentially allows scaling to large number of cores. Figure 6.25 shows the measured scalability of the StarSS + Nexus system, for several task sizes and up to 16 SPEs. The maximum scalability obtained is 14.3, which is very close to the theoretical scalability limit of 14.5 for the benchmark. The figure shows that for task sizes of $11.2\mu s$ and larger, the scalability is almost perfect. Therefore, using the Nexus system, large speedups can be obtained compared to StarSS without hardware task management support. For example, for a task size of $11.2\mu s$ and using 16 SPEs, the StarSS + Nexus system is 5.3 times faster than the StarSS system (measured on real hardware). In case 32 SPEs would be used and the application exhibits sufficient parallelism, the speedup would double approximately, as for that task size, the StarSS system does not scale beyond 4 cores, while the StarSS + Nexus system theoretically scales up

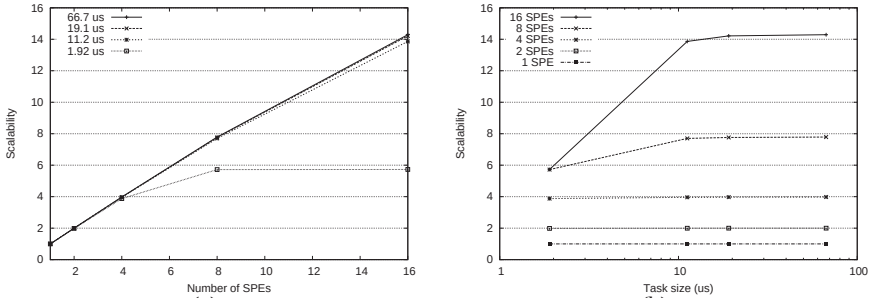


Figure 6.25: Scalability of the StarSS + Nexus system with the CD benchmark.

to approximately 30 cores. For a task size of $1.92\mu s$ and using 16 SPEs, the StarSS + Nexus system is 7.4 times faster than the StarSS system (measured on real hardware), although both system have a very limited scalability of 5.7 and 1.9, respectively.

The scalability of the StarSS + Nexus system is compared to that of the StarSS and the Manual systems, in Figure 6.26 and Table 6.7. They show the iso-efficiency function of the three systems, for a scalability efficiency of 80%. The StarSS + Nexus system obtains an 80% efficiency for much smaller task sizes than the StarSS system. The scalability of StarSS was limited by the task management overhead, which we successfully accelerated with the Nexus hardware.

The task size, required to obtain an 80% efficiency for the StarSS + Nexus system, is even lower than that required for the Manual system. The scalability of the Manual system was limited by the polling mechanism used to synchronize, which scaled very poorly. The difference between the line labeled StarSS + Nexus in Figure 6.26 and the line labeled Manual, shows the benefits of fast hardware synchronization primitives. The Manual line is much steeper, indicating the scalability difference.

To conclude this evaluation, we compare the measurements to the requirements that have been formulated in Section 6.4.2. The first requirement, states that the task dependency resolution process must be accelerated by a factor of 3.6 and 7.3, when using 8 and 16 SPEs respectively. In the StarSS system, the dependency resolution process takes $9.1\mu s$. The most time-consuming part of the TPU is the *descriptor handler*, which takes on average 81.81 cycles per task, which is $0.026\mu s$ per task. Therefore, the dependency resolution process is accelerated by a factor of 356.

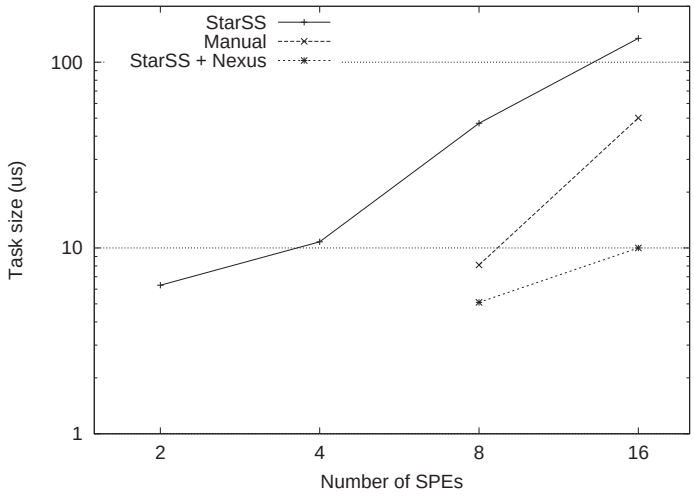


Figure 6.26: The iso-efficiency lines of the three systems for the CD benchmark (80% efficiency).

Table 6.7: The iso-efficiency values of the three systems for the CD benchmark (80% efficiency).

no. SPEs	Task size (μs)		
	StarSS	Manual	StarSS + Nexus
2	6.3		
4	10.8		
8	46.9	8.1	5.1
16	134.4	50.1	10.0

The second requirement is that scheduling, preparation, and submission of tasks to the SPEs, should be accelerated by a factor of 2.0 and 3.9, when using 8 and 16 SPEs, respectively. These processes, which can be viewed as the back-end of the task pool, were executed by the helper thread of the StarSS software task management system. Using the Nexus hardware, this back-end process of retrieving tasks from the task pool and bringing them to the SPEs for execution, is performed differently. Scheduling is currently performed by simply writing to a FIFO queue. Preparation of tasks is not needed anymore, as all the necessary information is in the *task storage* already. Submission of tasks to SPEs is replaced by SPEs reading tasks from the *ready queue* themselves. The best to compare the back-end of the StarSS software task management system

to is the process of the *finish handler*. It takes $0.014\mu s$ per task, which is 390 times faster than the back-end process of the StarSS system.

Finally, the third requirement is that the hardware should provide synchronization primitives for retrieving tasks, with a latency of at most $1\mu s$, irrespective of the number of cores. As mentioned before, it takes 985 cycles ($0.3\mu s$) for the SPE to read an item from the ready queue. This latency is independent of the number of cores used, neglecting contention on the bus and the queue. In our experiments, however, we have not detected any contention effects. Another effect not considered here, is the increasing bus (or NoC) latency for an increasing size of the on-chip system.

All three requirements are met by the Nexus system. Therefore, by using the Nexus system, the StarSS programming model can efficiently be used for much smaller tasks and more cores. Provided the application exhibits sufficient parallelism, task sizes as small as $10\mu s$ can efficiently be used on a multicore containing approximately 30 cores.

6.7 Future Work

The research described in this chapter is work in progress at the time of writing. In this section we describe the future work, consisting of features that yet have to be implemented, issues that have to be resolved, analyses that have to be performed, and bottlenecks that have to be removed.

6.7.1 Dependency Resolution

The TPU resolves dependencies by means of three tables. This system, however, is based on the start address of a piece of data. This puts certain restrictions on the application, which are not always met.

As long as data is divided into fixed elementary blocks, the system works well. For example, if a matrix is divided in blocks of 16×16 , the address of the top-left element is all that is needed to refer to the entire block. This assumes the system is aware of the block width, height, and stride. In the benchmark programs this is the case.

In some applications or kernels, such as motion compensation, the data blocks have an unknown alignment. Thus, blocks can partially overlap as depicted on the left side of Figure 6.27. Computing the overlap is very laborious and easily annihilates the gains of task level parallelism. Imagine, the input operand

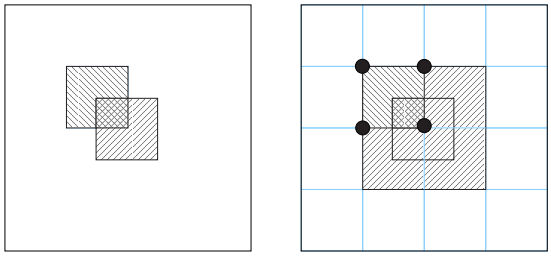


Figure 6.27: Calculating task dependencies through overlapping memory ranges (left) is laborious. Only the addresses of left-top corners have to be checked, if all blocks are translated into several blocks that are aligned with the grid.

of a task has to be compared against a thousand output operands of blocks inside the TPU. There is no clear solution to this issue as of yet, but we are investigating a few directions.

The first solution direction is to map at runtime blocks that are not aligned with a grid onto several aligned blocks and maintain the current hardware dependency resolution scheme. The source-to-source compiler can extract certain information from the annotated code and pass it on to the runtime system. The runtime system checks the operands for their alignment by computing the start addresses of the blocks it is overlapping and adds those to the task descriptor. If data is maintained within matrices, this computation can be performed quickly by checking the matrix indices.

Take for example the right-hand side of Figure 6.27. All blocks are 16×16 elements large and assume the matrix is named `frame`. In this case the block `frame[24:39][24:39]` is accessed which overlaps four aligned blocks. The system knows the block size, and thus the alignment can be checked by computing the x and y index, modulus the block width and height, respectively. If the modulus is not zero, it computes the start addresses of the blocks it overlaps, i.e., frame[16:31][16:31], frame[32:47][16:31], frame[16:31][32:47], frame[32:47][32:47].

These computations are rather simple and have to be performed only once per misaligned block. The TPU can check the dependencies among other tasks by using the lookup mechanisms as described before. This method, though, stresses the runtime system, and it has to be investigated what the impact of it on the performance and scalability will be.

A second solution direction is to accelerate the overlap detection in hardware. Such a solution is not as much tied to a particular runtime system and programming model, and thus can be more general applicable. We are collaborating with Yoav Etsion and Alex Ramirez from the Barcelona Supercomputer Center (BSC) to investigate such hardware.

6.7.2 Task Controller

In the current implementation, the SPE has to read a task from the TPU, load the descriptor, issue DMAs to fetch the input operands, execute the task, issue DMAs to write back the output operands, and finally signal to the TPU. Performance can be improved by exploiting concurrency on a single SPE. Possibly a software double buffering strategy can be deployed. A hardware unit, however, could increase concurrency even more. Therefore, we will investigate the benefits of adding a Task Controller (TC) to each SPE.

Exploiting double buffering allows hiding the latency of access to main memory. This is done through the MFC, which works autonomously and concurrently with the SPU. Similarly, the TC could perform local task management in an autonomous and concurrent way. For example, while the SPU is executing a task, the TC could read a new task id from the *ready queue* of the TPU, load the task descriptor, and program the MFC to load the input operands into the local store. Similarly, the TC could issue DMAs to store the output of a task back to main memory and write the task id to the *finish buffer*. All information needed to perform these operations is contained within the task descriptor.

To apply such a double buffering scheme, twice as many tasks should be available. For applications with limited available parallelism, the double buffering scheme might provide little performance improvement. For such situations, the Nexus system can be extended with task scheduling that exploits data locality. For example, a tail-submit strategy (see Section 3.7.2) can be deployed. We will investigate how the TC and TPU have to be modified to support such task scheduling.

6.7.3 Other Features

Currently, the Nexus system is using a fixed size task descriptor. Although this eases the implementation of the TPU, quite some memory *task storage* is waisted. We intend to investigate in future work the trade-offs of using variable sized task descriptors instead.

To scale the Nexus system to the many-core area, a clustered approach can be used. We will investigate how to best implement the clusterization. A TPU-to-TPU interface has to be designed, allowing efficient task stealing. Application scheduling is typically managed by the OS. We will investigate an OS interface that allows such management.

StarSS is one of the task based programming models, currently being developed. We will investigate compatibility of the Nexus system with other programming models.

6.7.4 Evaluation of Nexus

An important evaluation that has yet to be performed is the area and power consumption of the Nexus system. For both the TPU and the TC, the costs have to be compared against the benefits. The TPU consists of mainly memory, and might consume a significant amount of area and power. On the other hand, if additional cores cannot be deployed efficiently because of limited application scalability, the area and power is better spent on the Nexus system.

Related is the question what the proper size is for the storage and the tables of the TPU. These are depending on the amount of available parallelism in the applications, the dependency patterns, and the number of cores in the system. We will investigate the relation between these sizes, the performance, and the area and power consumption.

Finally, we will evaluate the benefits of the Nexus system for several other benchmarks. We will take benchmarks from several domains and with varying task sizes.

6.8 Conclusions

The task-based StarSS programming model is a promising approach to simplify parallel programming. In this chapter we evaluated the StarSS runtime system. It was shown that for fine-grained task parallelism, the task management overhead significantly limits the scalability. Specifically, for a task size of approximately $20\mu s$, as used in task based parallel H.264 decoding, the scalability is limited to five cores. In order to efficiently utilize 8 or 16 cores for such a task size, the runtime system should be accelerated by at least a factor of 3.6 and 7.3, respectively. Mainly the task dependency resolution process is too laborious to perform in software. Furthermore, the scheduling, prepara-

tion, and submission of tasks to cores take too much time to supply more than four cores continuously with tasks. Finally, we observed that the lack of fast synchronization primitives causes many unnecessary stalls.

We proposed the Nexus system for hardware task management support. It performs dependency resolution in hardware using simple table lookups. Hardware queues are used for fast scheduling and synchronization. The Nexus system was evaluated by means of simulation. It was shown that the dependency resolution process is accelerated by a factor of 356. Furthermore, the bottleneck of scheduling, preparation, and submission of tasks to the SPEs was removed. Scheduling is performed by writing to a hardware FIFO queue, preparation of tasks is not needed anymore, and the submission of tasks was replaced with a distributed mechanism, in which SPEs read tasks from the queue themselves. This mechanism using the hardware queue also provides the fast synchronization primitive that was necessary.

Simulations showed that when using 16 SPEs, for the CD benchmark a near optimal scalability is obtained for task sizes of approximately $10\mu s$ and larger. For such a task size, the scalability limit, based on the task throughput, is 29 and increases with the task size.

The Nexus system is currently still work in progress. The optimal parameters of the Nexus hardware, such as the size of the tables, the queues, and the storage, have yet to be determined. Also additional features, such as dependency detection of potentially overlapping memory regions, will be developed. Further, the area and power consumption of the Nexus hardware will be analyzed. Finally, several other benchmarks will be used to evaluate the performance and scalability of the Nexus system.

Chapter 7

Conclusions

In this dissertation, we have addressed several challenges in the multicore area. The overall aim of this work was to improve the scalability of multicore systems, with a focus on H.264 video decoding. More specifically, the following four objectives were targeted. First, to analyze the H.264 decoding application, to find new ways to parallelize the code in a scalable fashion, and to analyze the parallelization strategies and determine bottlenecks in current parallel systems. Second, to specialize the cores of the Cell processor for H.264 in order to obtain performance and power benefits. Third, to design hardware mechanism to reduce the overhead of managing the parallel execution. Finally, to enhance efforts in improving programmability by providing hardware support.

In this final chapter, we first summarize the research presented in this thesis and show how the objectives have been met. Further we specify the main contributions of this work. Second, we discuss possible directions for future research.

7.1 Summary and Contributions

This dissertation started in Chapter 2 with an analysis of the power wall problem for multicores. Specifically, we investigated the limits to performance growth of multicores due to technology improvements with respect to power constraints. As a case study we modeled a multicore consisting of Alpha 21264 cores, scaled to future technology nodes according to the ITRS roadmap. Analysis showed that the power budget significantly limits the performance growth. Despite this, the power constrained performance can double every three years

due to technology improvements.

When taking Amdahl's Law into account, a serial fraction of 1% decreases the power constrained performance by up to a factor of 10, assuming a symmetric multicore. For an asymmetric multicore, where the serial code is executed on a core that runs at a higher clock frequency, 1% of serial code reduces the achievable speedup by a factor of up to 2. This confirms that serial performance remains a challenge. On the other hand, there is Gustafson's Law which assumes that the input size grows equally with increasing number of processors. Using Gustafson's Law, for a serial fraction of 10% the performance loss is maximally 11%. Whether Amdahl's or Gustafson's Law is applicable to a certain application domain remains to be determined.

The contributions made in Chapter 2 can be summarized as follows:

- It was shown, based on the ITRS roadmap predictions, that technology improvements allow the performance of multicores to double every three years, despite being limited by the power budget.
- Our model confirms that in case Amdahl's Law is applicable, the power constrained multicore performance can significantly be improved by using an asymmetric architecture.

The first objective of this dissertation is met in Chapter 3, which investigates the parallelization of H.264 video decoding. Most promising is macroblock-level parallelism, of which it has been proven before that it can be exploited inside a frame in a diagonal wavefront manner. We have observed that macroblocks (MBs) in different frames are only dependent through motion vectors. Therefore, we have proposed a novel parallelization strategy, called the Dynamic 3D-Wave, which combines intra-frame and inter-frame MB-level parallelism.

Using this new parallelization strategy we have performed a limit study of the amount of MB-level parallelism available in H.264. We modified the FFmpeg H.264 decoder and analyzed real movies. The results have revealed that motion vectors are typically small on average. As a result a large amount of parallelism is available. For FHD resolution the total number of parallel MBs ranges from 4000 to 7000.

Finally, we have discussed a few implementations of the 2D-Wave and 3D-Wave strategy. Using the 3D-Wave parallelization strategy, a scalability of 55 was obtained on a 64 core system, whereas the scalability of the 2D-Wave saturates around 15. Experiences with these implementations have shown that low

latency synchronization is essential for fine-grained parallelism as exploited in these macroblock-level parallelizations of H.264.

The contributions made in Chapter 3 can be summarized as follows:

- A novel parallelization strategy for H.264 has been proposed, which is more scalable than any previous parallelization.
- We have performed a limit study of the amount of macroblock-level parallelism available in H.264, which has shown that for a FHD resolution the total number of parallel macroblocks ranges from 4000 to 7000.
- Analysis of parallel implementations of H.264 has shown that low latency synchronization is essential for efficient fine-grained parallelization.
- Although H.264 was assumed to be hard to parallelize, we were able to create a scalable implementation. This suggests that many other applications can be parallelized in a scalable fashion.

A specialization of the Cell SPE architecture was presented in Chapter 4, with which the second objective was met. We enhanced the architecture with twelve instruction classes, most of which have not been proposed before. These instruction classes can be divided in five groups. First, we added instructions to enable efficient scalar-vector operations, because the SPE has a SIMD only architecture. Second, we added several instructions that perform saturation and packing as needed by the H.264 kernels. Third, we added the swapoe instructions to perform fast matrix transposition, which is needed by many media kernels. Fourth, we added specialized arithmetic instructions that collapse frequently used simple arithmetic operations. Finally, we added support for non-quadword aligned access to the local store, which is especially needed for motion compensation. The speedup achieved on the H.264 kernels is between 1.84x and 2.37x, while the dynamic instruction count is reduced by a factor between 1.75x and 2.67x.

The contributions made in Chapter 4 can be summarized as follows:

- We have identified the bottlenecks in the execution of the H.264 kernels on the Cell SPE architecture.
- We have proposed novel application specific instructions for H.264 video decoding, of which several are applicable to other media applications as well.

- We have shown that a speedup of approximately 2x can be obtained on the H.264 kernels by adding about a dozen of application specific instructions.

One of the proposed instruction classes performs intra-vector operations. By using these intra-vector instructions, the matrix transpositions, generally required in two-dimensional kernels, were avoided. In Chapter 5 it has been investigated what the benefits of such intra-vector SIMD instructions for other kernels are. For six kernels intra-vector instructions have been developed and simulations have shown that a speedup of up to 2.06 was achieved. We have discussed the results and have shown how the characteristics of the kernels affect the applicability of intra-vector instructions.

The contributions made in Chapter 5 can be summarized as follows:

- We have proposed intra-vector SIMD instructions to avoid matrix transposition in two-dimensional kernels.
- We have shown that intra-vector SIMD instructions can speed up kernels by a factor of up to 2.06.

The last two objectives are met in Chapter 6. In this chapter we have first evaluated the runtime system of the task-based StarSS programming model. It was shown that for fine-grained task parallelism, the task management overhead significantly limits the scalability of StarSS applications. Specifically, for a task size of approximately $20\mu s$, as used in task based parallel H.264 decoding, the scalability is limited to five cores.

We have contributed by proposing the Nexus system for hardware task management support. It performs dependency resolution in hardware using simple table lookups. Hardware queues are used for fast scheduling and synchronization. The Nexus system was evaluated by means of simulation. It was shown that the dependency resolution process is accelerated by a factor of 356. As a result, the StarSS programming model can be efficiently used for much smaller task sizes than was possible before. Simulations have shown that when using 16 SPEs, a near optimal scalability is obtained for task sizes of approximately $10\mu s$ and larger. For such a task size, the scalability limit, based on the task throughput, is 29 and increases with the task size.

The contributions made in Chapter 6 can be summarized as follows:

- The StarSS runtime system was analyzed, and it was shown that the task management overhead is too large to efficiently exploit fine-grained parallelism for more than five cores.

- We have proposed the Nexus system for hardware task management support.
- We have proposed a novel dependency resolution scheme, based on low latency table lookups.
- We have shown that the Nexus system can greatly reduce the task management overhead, as a result of which the StarSS programming model can efficiently be used for much smaller task sizes (down to $10\mu s$).

7.2 Possible Directions for Future Work

In Chapter 3, the limit study has shown that thousands of parallel macroblocks are available, when using the 3D-Wave parallelization strategy. The implementation based on the TriMedia multicore, showed a speedup of 55 using 64 cores. The scalability curve, however, indicates that this implementation scales to higher core counts. Future work could investigate the scalability limit of the 3D-Wave implementation. Furthermore, it would be interesting to study the performance of the 3D-Wave on different architectures. Especially, the impact of the memory hierarchy would be interesting research topic. Such research, however, requires a large scale system. Potentially, such system could be based on the Tilera's Tile-GX processor, or on a second generation Cell processor. Finally, the integration of slice-level parallel entropy decoding into a 3D-Wave implementation, could be analyzed.

The work presented in Chapter 4 could be extended with a thorough analysis of the power consumption. It was shown that the specialization of the Cell SPEs improves performance and decreases the dynamic instruction count for the H.264 kernels. This indicates that the power efficiency has been improved, although it has not been quantified. To perform such a power analysis, tools and/or power models are needed. Such tools and models exist, but require a lower level implementation than we provide. With modern large scale systems on chip, containing billions of transistors, it should be able to estimate the power consumption in an early design stage, where a low-level implementation is not yet available. Therefore, new power models and tools are needed. Such power tools could also be used to evaluate the power consumption of the hardware proposed in Chapters 5 and 6.

It was suggested in Chapter 5, that the proposed intra-vector instructions can be implemented by mapping them to a special functional unit. Such a functional unit, could consist of a series of ALUs and interconnect, as has been used

in [158, 172]. Future work can include the design of such a functional unit. It can also be investigated if the proposed intra-vector instructions can be mapped to extensible processors, such as the ARC700 [2], Tensilica's XTensa [154], or Altera's NioSII [18].

The Nexus hardware support for task management, proposed in Chapter 6, is ongoing work at the time of writing and many issues remain to be investigated. The most interesting are dependency detection of overlapping memory regions, the implementation of the task controller (including task scheduling), analysis of the area and power consumption, evaluation with several other benchmarks, and investigation of the applicability of the Nexus hardware to other emerging programming models.

Throughout Chapters 4 to 6, we used the CellSim simulator. The simulator is not very fast, but for the benchmarks that were used, it was not a very large problem. Simulations took at most 10 hour. One of the limitations that we encountered was that the simulator cannot execute normal Cell binaries. As the simulator does not run an OS, the code has to be recompiled with special libraries. For applications such as FFmpeg this is not feasible within reasonable time, and individual kernels or synthetic benchmarks had to be used instead. For future multicore research, the current development of multicore simulators is of great importance.

Bibliography

[1] AMD Phenom Processors. http://www.amd.com/uk/products/desktop/processors/phenom/Pages/AMD-phenom-processor-X4-X3-at-home.aspx.

[2] ARC 700 Core Family. http://www.arc.com/configurablecores/arc700/.

[3] ARM Cortex-A9 MPCore. http://www.arm.com/products/CPUs/ARMCortex-A9_MPCore.html.

[4] Cell Superscalar. http://www.bsc.es/plantillaG.php?cat_id=179.

[5] CellBench. https://gso.ac.upc.edu/projects/cellbench/.

[6] CellBuzz: Georgia Tech Cell BE Software. http://sourceforge.net/projects/cellbuzz.

[7] CellSim: Modular Simulator for Heterogeneous Multiprocessor Architectures. http://pcsostres.ac.upc.edu/cellsim/doku.php/.

[8] CEVA-XC Communication Processor. www.ceva-dsp.com/products/cores/ceva-xc.php.

[9] DaVinci Digital Media Processors - Texas Instruments. http://www.ti.com/.

[10] Intel CE 2110 Media Processor. http://www.intel.com/design/celect/2110/.

[11] Intel Core i7 Processor. http://www.intel.com/cd/products/services/emea/eng/processors/corei7/overview/405796.htm.

[12] Paraver - trace visualization tool. http://www.bsc.es/plantillaA.php?cat_id=485.

[13] Six-Core AMD Opteron Processor. http://www.amd.com/uk/products/server/processors/six-core-opteron/Pages/six-core-opteron.aspx.

[14] Tensilica Technology Helps Power Worlds Fastest Router. http://www.tensilica.com/news/110/330/Tensilica-Technology-Helps-Power-World-s-Fastest-Router.htm.

[15] Tensilica's Diamond 388VDO Video Engine. http://www.tensilica.com/products/video/video_arch.htm.

[16] Intel Itanium Architecture Software Developer's Manual - Revision 2.2, 2006. http://download.intel.com/design/Itanium/manuals/24531905.pdf.

[17] Ghiath Al-Kadi and Andrei Terechko. A Hardware Task Scheduler for Embedded Video Processing. In *Proc. High Performance Embedded Architectures and Compilers (HiPEAC) Conference*, 2009.

[18] Altera. Nios II Processor: The World's Most Versatile Embedded Processor. http://www.altera.com/products/ip/processors/nios2/ni2-index.html.

[19] 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 *Proc. Int. Conf. on Parallel and Distributed Systems (ICPADS)*, 2009.

[20] M. Alvarez, A. Ramirez, M. Valero, A. Azevedo, C.H. Meenderinck, and B.H.H. Juurlink. Performance Evaluation of Macroblock-level Parallelization of H.264 Decoding on a cc-NUMA Multiprocessor Architecture. *Avances en Sistemas e Informtica*, 2009.

[21] M. Alvarez, E. Salami, A. Ramirez, and M. Valero. A Performance Characterization of High Definition Digital Video Decoding using H.264/AVC. In *Proc. IEEE Int. Workload Characterization Symp.*, 2005.

[22] M. Alvarez, E. Salami, A. Ramirez, and M. Valero. HD-VideoBench: A Benchmark for Evaluating High Definition Digital Video Applications. In *Proc. IEEE Int. Workload Characterization Symp.*, 2007. http://personals.ac.upc.edu/alvarez/hdvideobench/index.html.

[23] M. Alvarez, E. Salami, A. Ramirez, and M. Valero. Performance Impact of Unaligned Memory Operations in SIMD Extensions for Video Codec Applications. In *Proc. Int. Symp. on Performance Analysis of Systems & Software*, 2007.

[24] G.M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proc. AFIPS Conf.*, 1967.

[25] AnandTech. Lab Update: SSDs, Nehalem Mac Pro, Nehalem EX, Bench and More. http://anandtech.com/weblog/showpost.aspx?i=603.

[26] Marnix Arnold and Henk Corporaal. Automatic Detection of Recurring Operation Patterns. In *Proc. Int. workshop on Hardware/software codesign (CODES)*, 1999.

[27] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, et al. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department University of California, Berkeley, 2006.

[28] A. Azevedo, B.H.H. Juurlink, C.H. Meenderinck, A. Terechko, J. Hoogerbrugge, M. Alvarez, A. Ramirez, and M. Valero. A Highly Scalable Parallel Implementation of H.264. *Trans. on High-Performance Embedded Architectures and Compilers (HiPEAC)*, 2009.

[29] A. Azevedo, C.H. Meenderinck, B.H.H. Juurlink, M. Alvarez, and A. Ramirez. Analysis of Video Filtering on the Cell Processor. In *Proc. Int. Symp. on Circuits and Systems*, 2008.

[30] P. Bellens, JM Perez, RM Badia, and J. Labarta. CellSs: a Programming Model for the Cell BE Architecture. In *Proc. Supercomputing Conf. (SC)*, 2006.

[31] M. Berekovic, H.J. Stolberg, M.B. Kulaczewski, P. Pirsch, H. Möller, H. Runge, J. Kneip, and B. Stabernack. Instruction Set Extensions for MPEG-4 Video. *Journal of VLSI Signal Processing*, 23(1), 1999.

[32] S. Borkar. Thousand core chips: a technology perspective. In *Proc. Design Automation Conference*, 2007.

[33] M. Briejer, C.H. Meenderinck, and B.H.H. Juurlink. Extending the Cell SPE with Energy Efficient Branch Prediction. In *Proc. Int. Euro-Par Conf.*, 2010.

[34] Jeffery A. Brown, Rakesh Kumar, and Dean Tullsen. Proximity-Aware Directory-Based Coherence for Multi-Core Processor Architectures. In *Proc. Symp. on Parallel Algorithms and Architectures*, 2007.

[35] J. Castrillon, D. Zhang, T. Kempf, B. Vanthournout, R. Leupers, and G. Ascheid. Task Management in MPSoCs: an ASIP Approach. In *Proc. Int. Conf. on Computer-Aided Design*, 2009.

[36] Cavium Networks. OCTEON Multi-Core Processor Family. http://www.caviumnetworks.com/OCTEON_MIPS64.html.

[37] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. In *Proc. IEEE Int. Symp. on High-Performance Computer Architecture (HPCA)*, 2005.

[38] H. Chang and W. Sung. Efficient Vectorization of SIMD Programs with Non-aligned and Irregular Data Access Hardware. In *Proc. Int. Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2008.

[39] J.W. Chen, C.R. Chang, and Y.L. Lin. A Hardware Accelerator for Context-Based Adaptive Binary Arithmetic Decoding in H. 264/AVC. In *Proc. Int. Symp. Circuits and Systems (ISCAS)*, 2005.

[40] 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.

[41] 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 *Proc. Int. Parallel and Distributed Processing Symp.*, 2004.

[42] D. Cheresiz, B.H.H. Juurlink, S. Vassiliadis, and H. A. G. Wijshoff. The CSI Multimedia Architecture. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 2005.

[43] C.C. Chi. Parallel H.264 Decoding Strategies for Cell Broadband Engine, 2009. Msc thesis - ce.et.tudelft.nl.

[44] C.C. Chi, B.H.H. Juurlink, and C.H. Meenderinck. Evaluation of Parallel H.264 Decoding Strategies for the Cell Broadband Engine. In *Proc. Int. Conf. on Supercomputing (ICS)*, 2010.

[45] J. Chong, N.R. Satish, B. Catanzaro, K. Ravindran, and K. Keutzer. Efficient Parallelization of H.264 Decoding with Macro Block Level Scheduling. In *Proc. IEEE Int. Conf. on Multimedia and Expo*, 2007.

[46] ClearSpeed. The CSX600 Processor. http://www.clearspeed.com.

[47] ClearSpeed. The CSX700 Processor. http://www.clearspeed.com/products/csx700.php.

[48] J. Corbal, R. Espasa, and M. Valero. MOM: a Matrix SIMD Instruction Set Architecture for Multimedia Applications. In *Proc. ACM/IEEE Conf. on Supercomputing (ICS)*, 1999.

[49] T.L. da Silva, C.M. Diniz, J.A. Vortmann, L.V. Agostini, A.A. Susin, S. Bampi, and B. Pelotas-RS. A Pipelined 8x8 2-D Forward DCT Hardware Architecture for H. 264/AVC High Profile Encoder. In *Proc. Second Pacific Rim Symp. on Advances in Image and Video Technology (PSIVT)*, 2007.

[50] M.A. Daigneault, J.M. Pierre Langlois, and J.-P. David. Application Specific Instruction Set Processor Specialized for Block Motion Estimation. In *Proc. IEEE Int. Conf. on Computer Design*, 2008.

[51] I. Daubechies and W. Sweldens. Factoring Wavelet Transforms into Lifting Steps. *Journal of Fourier Analysis and Applications*, 4(3), 1998.

[52] Advanced Micro Devices. *3DNow! Technology Manual*. Advanced Micro Devices, Inc, 2000.

[53] K. Diefendorff, PK Dubey, R. Hochsprung, and H. Scale. AltiVec Extension to PowerPC Accelerates Media Processing. *IEEE Micro*, 20(2), 2000.

[54] EE Times. IBM's Power7 Heats up Server Competition at Hot Chips. http://www.eetimes.com/news/semi/showArticle.jhtml?articleID=219400955.

[55] EE Times. Sun, IBM Push Multicore Boundaries. http://www.eetimes.com/news/semi/showArticle.jhtml?articleID=219500130.

[56] J.O. Eklundh. A Fast Computer Method for Matrix Transposing. *IEEE Trans. on Computers*, 100(21), 1972.

[57] Y. Etsion, A. Ramirez, R.M. Badia, and J. Labarta. Cores as Functional Units: A Task-Based, Out-of-Order, Dataflow Pipeline. In *Proc. Int. Summer School on Advanced Computer Architecture and Compilation for Embedded Systems*.

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

[59] B. Flachs, S. Asano, S.H. Dhong, H.P. Hofstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty, et al. The Microarchitecture of the Synergistic Processor for a CELL Processor. In *Proc. IEEE Int. Solid-State Circuits Conference*, 2005.

[60] M. Flierl and B. Girod. Generalized B Pictures and the Draft H.264/AVC Video-Compression Standard. *IEEE Trans. on Circuits and Systems for Video Technology*, 13(7), July 2003.

[61] Freescale. MPC8641D: High-performance Dual Core Processor. http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MPC8641D.

[62] Freescale. QorIQ Communications Platforms. http://www.freescale./webapp/sps/site/overview.jsp?nodeId=02VS0l25E4.

[63] T. Fujiyoshi, S. Shiratake, S. Nomura, T. Nishikawa, Y. Kitasho, H. Arakida, Y. Okuda, Y. Tsuboi, M. Hamada, H. Hara, et al. A 63-mW H. 264/MPEG-4 Audio/Visual Codec LSI with Module-Wise Dynamic Voltage/Frequency Scaling. *IEEE Journal of Solid-State Circuits*, 41(1), 2006.

[64] R. Giorgi, Z. Popovic, and N. Puzovic. DTA-C: A Decoupled Multi-Threaded Architecture for CMP Systems. In *proc. Int. Symp. on Computer Architecture and High Performance Computing*, 2007.

[65] R. Giorgi, Z. Popovic, and N. Puzovic. Introducing Hardware TLP Support in the Cell Processor. In *Int. Workshop on Multi-Core Computing Systems (MuCoCoS)*, 2009.

[66] D. Gong, Y. He, and Z. Cao. New Cost-Effective VLSI Implementation of a 2-D Discrete Cosine Transform and its Inverse. *IEEE Trans. on Circuits and Systems for Video Technology*, 14(4), 2004.

[67] S. Gordon, D. Marpe, and T. Wiegand. Simplified use of 8x8 Transforms–Updated Proposal & Results. Joint Video Team (JVT) of

ISO/IEC MPEG and ITU-T VCEG, doc. JVT–K028, Munich, Germany, March 2004.

[68] A. Gulati and G. Campbell. Efficient Mapping of the H.264 Encoding Algorithm onto Multiprocessor DSPs. In *Proc. Embedded Processors for Multimedia and Communications II*, 2005.

[69] J.I. Guo, R.C. Ju, and J.W. Chen. An Efficient 2-D DCT/IDCT Core Design Using Cyclic Convolution and Adder-Based Realization. *IEEE Trans. on Circuits and Systems for Video Technology*, 14(4), 2004.

[70] J.L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31(5), 1988.

[71] L. Gwennap. Digital, MIPS add Multimedia Extensions. *Microprocessor Report*, 10(15), 1996.

[72] B. Hanounik and X.S. Hu. Linear-time Matrix Transpose Algorithms Using Vector Register File With Diagonal Registers. In *Proc. Int. Parallel & Distributed Processing Symp.*, 2001.

[73] J.L. Hennessy and D.A. Patterson. *Computer Architecture - A Quantative Approach*. Morgan Kaufman Publishers, 4th edition, 2007. page 3.

[74] M.D. Hill and M.R. Marty. Amdahl's Law in the Multicore Era. *IEEE Computer*, 41(7), 2008.

[75] M.D. Hill and M.R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7), 2008.

[76] HirooHayashi. SpursEngine - Architecture Overview & Implementation, 2008. http://www.ksrp.or.jp/fais/sec/cell/fais/news/pdf/kenkyutheme/hayashi_workshop2008_v2.pdf.

[77] H.P. Hofstee. Power-Constrained Microprocessor Design. In *Proc. IEEE Int. Conf. on Computer Design: VLSI in Computers and Processors*, 2002.

[78] H.P. Hofstee. Power Efficient Processor Architecture and the Cell Processor. In *Proc. Int. Symp. on High-Performance Computer Architecture (HPCA)*, 2005.

[79] J. Hoogerbrugge and A. Terechko. A Multithreaded Multicore System for Embedded Media Processing. *Trans. on High-Performance Embedded Architectures and Compilers*, 3(2), 2008.

[80] IBM. *Cell Broadband Engine Programming Handbook*, 2006. http://www.bsc.es/plantillaH.php?cat_id=326.

[81] IBM. *Synergistic Processor Unit Instruction Set Architecture*, 2007. http://www.bsc.es/plantillaH.php?cat_id=326.

[82] Intel. Single-chip Cloud Computer . http://techresearch.intel.com/articles/Tera-Scale/1826.htm.

[83] Intel Integrated Performance Primitives. http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/302910.htm.

[84] ITRS. Int. Technology Roadmap for Semiconductors, 2007 Edition. http://www.itrs.net.

[85] ITRS. Int. Technology Roadmap for Semiconductors, 2009 Edition. http://www.itrs.net.

[86] ITU-T and ISO/IEC JTC 1. Generic Coding of Moving Pictures and Associated Audio Information - Part 2: Video (ITU-T Recommendation H.262 - ISO/IEC 13818-2 (MPEG-2)), 1994.

[87] ISO/IEC JTC1. ISO/IEC 14496-2 - Information Technology - Coding of Audio-Visual Objects - Part 2: Visual, 1999.

[88] S. Kang and D.A. Bader. Optimizing JPEG2000 Still Image Encoding on the Cell Broadband Engine. In *Proc. Int. Conf. on Parallel Processing*, 2008.

[89] R.E. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2), 1999.

[90] 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. *IEEE Micro*, 21(2), 2001.

[91] J. Kim and D.S. Wills. Quantized Color Instruction Set for Media-on-Demand Applications. In *Proc. Int. Conf. on Multimedia and Expo (ICME)*, 2003.

[92] K.D. Kissell. MIPS MT: A Multithreaded RISC Architecture for Embedded Real-Time Processing. In *"Proc. High Performance Embedded Architectures and Compilers (HiPEAC) Conf."*, 2008.

[93] C.E. Kozyrakis and D.A. Patterson. Scalable, Vector Processors for Embedded Systems. *IEEE Micro*, 23(6), 2003.

[94] S. Kumar, C.J. Hughes, and A. Nguyen. Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors. In *Proc. Int. Symp. on Computer Architecture (ISCA)*, 2007.

[95] B. Lee and D. Brooks. Effects of Pipeline Complexity on SMT/CMP Power-Performance Efficiency. In *Proc. Workshop on Complexity Effective Design*, 2005.

[96] J. Lee, S. Moon, and W. Sung. H.264 Decoder Optimization Exploiting SIMD Instructions. In *Proc. Asia-Pacific Conf. on Circuits and Systems*, 2004.

[97] R. Lee and J. Huck. 64-bit and Multimedia Extensions in the PA-RISC 2.0 Architecture. In *Proc. Compcon Conf.*, 1996.

[98] S.M. Lee, J.H. Chung, and M.M.O. Lee. High-Speed and Low-Power Real-Time Programmable Videomulti-Processor for MPEG-2 Multimedia Chip on 0.6 μm TLM CMOS Technology. In *Proc. Asia and South Pacific Design Automation Conf. (ASP-DAC)*, 1999.

[99] L. Li, Y. Song, S. Li, T. Ikenaga, and S. Goto. A Hardware Architecture of CABAC Encoding and Decoding with Dynamic Pipeline for H. 264/AVC. *Journal of Signal Processing Systems*, 50(1), 2008.

[100] H. Liao and A. Wolfe. Available Parallelism in Video Applications. In *Proc. Int. Symp. on Microarchitecture (Micro)*, 1997.

[101] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner. SODA: A High-Performance DSP Architecture for Software-Defined Radio. *IEEE Micro*, 27(1), 2007.

[102] P. List, A. Joch, J. Lainema, G. Bjntegaard, and M. Karczewicz. Adaptive Deblocking Filter. *IEEE Trans. on Circuits and Systems for Video Technology*, 13(7), 2003.

[103] L. Liu, S. Kesavarapu, J. Connell, A. Jagmohan, L. Leem, B. Paulovicks, V. Sheinin, L. Tang, and H. Yeo. Video Analysis and Compression on the STI Cell Broadband Engine Processor. Technical report, IBM T.J. Watson Research Center, 2006.

[104] Y. Liu and S. Oraintara. Complexity Comparison of fast Block-Matching Motion Estimation Algorithms. In *Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, 2004.

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

[106] G. Long, D. Fan, and J. Zhang. Architectural Support for Cilk Computations on Many-Core Architectures. *ACM SIGPLAN Notices*, 44(4), 2009.

[107] S.G. Mallat. A Theory for Multiresolution Signal Decomposition: the Wavelet Representation. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 11(7), 1989.

[108] H. Malvar, A. Hallapuro, M. Karczewicz, and L. Kerofsky. Low-complexity Transform and Quantization with 16-bit Arithmetic for H. 26L. In *Proc. IEEE Int. Conf. on Image Processing (ICIP)*, 2002.

[109] H.S. Malvar, A. Hallapuro, M. Karczewicz, and L. Kerofsky. Low-Complexity Transform and Quantization in H. 264/AVC. *IEEE Trans. on Circuits and Systems for Video Technology*, 13(7), 2003.

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

[111] C.H. Meenderinck. Implementing SPU instructions in CellSim. http://pcsostres.ac.upc.edu/cellsim/doku.php/.

[112] C.H. Meenderinck, A. Azevedo, B.H.H. Juurlink, M. Alvarez, and A. Ramirez. Parallel Scalability of Video Decoders. Technical report, Delft University of Technology, April 2008. http://ce.et.tudelft.nl/publications.php.

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

[114] A. Mendelson. How Many Cores are too Many Cores? Presentation at 3rd HiPEAC Industrial Workshop.

[115] K. Moerman. NXP's Embedded Vector Processor can Tune into Software-Defined Radio. 2007. http://www.eetimes.eu/germany/202403704.

[116] G.E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 1965.

[117] M. Moudgill, J. Glossner, S. Agrawal, and G. Nacer. The Sandblaster 2.0 Architecture and SB3500 Implementation. In *Proc. Software Defined Radio Technical Forum*, 2008.

[118] T. Nishikawa, M. Takahashi, M. Hamada, T. Takayanagi, H. Arakida, N. Machida, H. Yamamoto, T. Fujiyoshi, Y. Maisumoto, O. Yamagishi, et al. A 60 MHz 240 mW MPEG-4 Video-Phone LSI with 16 Mb Embedded DRAM. In *Proc. IEEE Int. Solid-State Circuits Conference (ISSCC)*, 2000.

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

[120] Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG. Int. Standard of Joint Video Specification (ITU-T Rec. H. 264— ISO/IEC 14496-10 AVC), 2005.

[121] R.R. Osorio and J.D. Bruguera. An FPGA Architecture for CABAC Decoding in Many-core Systems. In *Proc. IEEE Int. Conf. on Application-Specific Systems, Architectures and Processors (ASAP)*, 2008.

[122] S.H. Ou, T.J. Lin, X.S. Deng, Z.H. Zhuo, and C.W. Liu. Multithreaded Coprocessor Interface for Multi-Core Multimedia SoC. In *Proc. Asia and South Pacific Design Automation Conference (ASPDAC)*, 2008.

[123] Y. Pang, L. Sun, S. Guo, and S. Yang. Spatial and temporal data parallelization of multi-view video encoding algorithm. In *Proc. IEEE Workshop on Multimedia Signal Processing (MMSP)*, 2007.

[124] S. Park, S. Kim, K. Byeon, J. Cha, and H. Cho. An Area Efficient Video/Audio Codec for Portable Multimedia Applications. In *Proc. IEEE Int. Symp. on Circuits and Systems (ISCAS)*, 2000.

[125] A. Peleg, S. Wilkie, and U. Weiser. Intel MMX for Multimedia PCs. *Communications of the ACM*, 40(1), 1997.

[126] H. Peters, R. Sethuraman, A. Beric, P. Meuwissen, S. Balakrishnan, C.A.A. Pinto, W. Kruijtzer, F. Ernst, G. Alkadi, J. van Meerbergen, et al. Application Specific Instruction-Set Processor Template for Motion Estimation in Video Applications. *IEEE Trans. on Circuits and Systems for Video Technology*, 15(4), 2005.

[127] D. Pham, S. Asano, M. Bolliger, M.N. Day, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, et al. The Design and Implementation of a First-Generation CELL Processor. In *Proc. IEEE Int. Solid-State Circuits Conference (ISSCC)*, 2005.

[128] PicoChip. PC205 High Performance Signal Processor. http://www.picochip.com/page/76/.

[129] J. Planas, R.M. Badia, E. Ayguadé, and J. Labarta. Hierarchical Task-Based Programming With StarSs. *Int. Journal of High Performance Computing Applications*, 23(3), 2009.

[130] Plurality. Hypercore Processor. http://www.plurality.com/hypercore.html.

[131] S.K. Raman, V. Pentkovski, and J. Keshava. Implementing Streaming SIMD Extensions on the Pentium III Processor. *IEEE Micro*, 2000.

[132] A. Rodriguez, A. Gonzalez, and M.P. Malumbres. Hierarchical Parallelization of an H.264/AVC Video Encoder. In *Proc. Int. Symp. on Parallel Computing in Electrical Engineering*, 2006.

[133] M. Roitzsch. Slice-Balancing H.264 Video Encoding for Improved Scalability of Multicore Decoding. In *Proc. IEEE Real-Time Systems Symp.*, 2006.

[134] R. Sasanka, M.L. Li, S.V. Adve, Y.K. Chen, and E. Debes. ALP: Efficient Support for all Levels of Parallelism for Complex Media Applications. *ACM Trans. on Architecture and Code Optimization (TACO)*, 4(1), 2007.

[135] K. Schöffmann, M. Fauster, O. Lampl, and L. Böszörmenyi. An Evaluation of Parallelization Concepts for Baseline-Profile Compliant H.264/AVC Decoders. In *Proc. Euro-Par Conf.*, 2007.

[136] A. Shahbahrami, B.H.H. Juurlink, and S. Vassiliadis. Performance Impact of Misaligned Accesses in SIMD Extensions. In *Proc. Workshop on Circuits, Systems, and Signal Processing (ProRISC)*, 2006.

[137] A. Shahbahrami, B.H.H. Juurlink, and S. Vassiliadis. Implementing the 2D Wavelet Transform on SIMD-Enhanced General-Purpose Processors. *IEEE Trans. on Multimedia*, 10(1), 2008.

[138] Z. Shen, H. He, Y. Zhang, and Y. Sun. VS-ISA: A Video Specific Instruction Set Architecture for ASIP Design. In *Proc. Int. Conf. on Intelligent Information Hiding and Multimedia Signal Processing*, 2006.

[139] Y. Shi. Reevaluating Amdahl's Law and Gustafson's Law, 1996. http://www.cis.temple.edu/~shi/docs/amdahl/amdahl.html.

[140] H. Shojania, S. Sudharsanan, and Chan Wai-Yip. Performance Improvement of the H.264/AVC Deblocking Filter Using SIMD Instructions. In *Proc. IEEE Int. Symp. on Circuits and Systems (ISCAS)*, 2006.

[141] M. Sima, S. D. Cotofana, J. T. J. van Eijndhoven, S. Vassiliadis, and K. A. Vissers. An 8x8 IDCT Implementation on an FPGA-augmented TriMedia. In *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, 2001.

[142] Magnus Själander, Andrei Terechko, and Marc Duranton. A Look-Ahead Task Management Unit for Embedded Multi-Core Architectures. In *Proc. EUROMICRO Conf. on Digital System Design Architectures, Methods and Tools (DSD)*, 2008.

[143] N. Slingerland and A.J. Smith. Measuring the Performance of Multimedia Instruction Sets. *IEEE Trans. on Computers*, 2002.

[144] Sony. BCU-100 Computing Unit with Cell/B.E. and RSX. http://pro.sony.com/bbsccms/ext/ZEGO/files/BCU-100_Whitepaper.pdf.

[145] Sony. Media Backbone. http://pro.sony.com/bbsccms/ext/BroadcastandBusiness/minisites/NAB2010/docs/prodbroch_medbacksol.pdf.

[146] K. Stavrou, C. Kyriacou, P. Evripidou, and P. Trancoso. Chip Multiprocessor Based on Data-Driven Multithreading Model. *Int. Journal of High Performance Systems Architecture*, 1(1), 2007.

[147] K. Stavrou, D. Pavlou, M. Nikolaides, P. Petrides, P. Evripidou, P. Trancoso, Z. Popovic, and R. Giorgi. Programming Abstractions and Toolchain for Dataflow Multithreading Architectures. In *Proc. IEEE Int. Symp. on Parallel and Distributed Computing (ISPDC)*, 2009.

[148] P. Stenström. Chip-Multiprocessing and Beyond. In *Proc. Int. Symp. on High-Performance Computer Architecture (HPCA)*, 2006.

[149] 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 *Proc. SPIE Conference on Applications of Digital Image Processing*, 2004.

[150] K. Suzuki, M. Daito, T. Inoue, K. Nadehara, M. Nomura, M. Mizuno, T. Iima, S. Sato, T. Fukuda, T. Arai, et al. A 2000-MOPS Embedded RISC Processor with a Rambus DRAM Controller. *IEEE Journal of Solid-State Circuits*, 34(7), 1999.

[151] H. Takata, T. Watanabe, T. Nakajima, T. Takagaki, H. Sato, A. Mohri, A. Yamada, T. Kanamoto, Y. Matsuda, S. Iwade, et al. The D30V/MPEG Multimedia Processor. *IEEE Micro*, 19(4), 1999.

[152] D. Talla and L.K. John. Cost-Effective Hardware Acceleration of Multimedia Applications. In *Proc. Int. Conf. on Computer Design (ICCD)*, 2001.

[153] A. Tamhankar and K.R. Rao. An Overview of H. 264/MPEG-4 Part 10. In *Proc. EURASIP Conference focused on Video/Image Processing and Multimedia Communications*, 2003.

[154] Tensilica Inc. Product Brief - Xtensa LX3 Customizable DPU. http://www.tensilica.com/uploads/pdf/LX3.pdf.

[155] Texas Instruments. OMAP Applications Processors. http://focus.ti.com/paramsearch/docs/parametricsearch.tsp?family=dsp&sectionId=2&tabId=2218&familyId=1525&paramCriteria=no.

[156] Tilera. TILE-Gx Processors Family. http://www.tilera.com/products/TILE-Gx.php.

[157] Tilera. TILE64 Processor Family. http://www.tilera.com.

[158] N. Topham. EnCore: A Low-power Extensible Embedded Processor. In *Presentation at HiPEAC Industrial Workshop*, 2009. http://groups.inf.ed.ac.uk/pasta/pub/hipeac-wroclaw.pdf.

[159] M. Tremblay, JM O'Connor, V. Narayanan, S.M. Inc, and CA Mountain View. VIS Speeds new Media Processing. *IEEE Micro*, 16(4), 1996.

[160] K. van Berkel, P. Meuwissen, N. Engin, and S. Balakrishnan. CVP: A Programmable Co Vector Processor for 3G Mobile Baseband Processing. In *Proc. World Wireless Congress*, 2003.

[161] J.W. van de Waerdt, K. Kalra, P. Rodriguez, H. van Antwerpen, S. Vassiliadis, S. Das, S. Mirolo, C. Yen, B. Zhong, C. Basto, et al. The TM3270 Media-Processor. In *Proc. IEEE/ACM Int. Symp. on Microarchitecture*, 2005.

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

[163] I. Verbauwhede and C. Nicol. Low Power DSP's for Wireless Communications. In *Proc. Int. Symp. on Low Power Electronics and Design*, 2000.

[164] J.S. Vitter. Implementations for Coalesced Hashing. *Communications of the ACM*, 25(12), 1982.

[165] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A.Luthra. Overview of the H.264/AVC Video Coding Standard. *IEEE Trans. on Circuits and Systems for Video Technology*, 13(7), 2003.

[166] M. Wien. Variable Block-Size Transforms for H. 264/AVC. *IEEE Trans. on Circuits and Systems for Video Technology*, 13(7), 2003.

[167] D.H. Woo and H.H.S. Lee. Extending Amdahls Law for Energy-Efficient Computing in the Many-Core Era. *IEEE Computer*, 41(12), 2008.

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

[169] X264-devel – Mailing list for x264 developers, July-August 2007. Subject: Out-of-Range Motion Vectors.

[170] Z. Zhao and P. Liang. Data Partition for Wavefront Parallelization of H.264 Video Encoder. In *IEEE Int. Symp. on Circuits and Systems (ISCAS)*, 2006.

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

[172] M. Zuluaga and N. Topham. Resource Sharing in Custom Instruction Set Extensions. In *Proc. IEEE Symp. on Application Specific Processors (SASP)*, 2008.

List of Publications

International Journals

1. **C.H. Meenderinck**, A. Azevedo, B.H.H. Juurlink, M. Alvarez, A. Ramirez, *Parallel Scalability of Video Decoders*, Journal of Signal Processing Systems, Volume 57, Issue 2, 2009.
2. 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.
3. 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 Informtica, Volume 6, Number 1, 2009, ISSN 1657-7663.

International Conferences

4. **C.H. Meenderinck**, B.H.H. Juurlink, *A Case for Hardware Task Management Support for the StarSS Programming Model*, Proceedings of the Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD), 2010.
5. M. Briejer, **C.H. Meenderinck**, B.H.H. Juurlink, *Extending the Cell SPE with Energy Efficient Branch Prediction*, Proceedings of the International Euro-Par Conference, 2010.
6. C.C. Chi, B.H.H. Juurlink, **C.H. Meenderinck**, *Evaluation of Parallel H.264 Decoding Strategies for the Cell Broadband Engine*, Proceedings of the International Conference on Supercomputing (ICS), 2010.

7. **C.H. Meenderinck**, B.H.H. Juurlink, *Intra-Vector SIMD Instructions for Core Specialization*, Proceedings of the IEEE International Conference on Computer Design (ICCD), 2009.

8. **C.H. Meenderinck**, B.H.H. Juurlink, *Specialization of the Cell SPE for Media Applications*, Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP), 2009.

9. 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.

10. 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.

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. **C.H. Meenderinck**, B.H.H. Juurlink, *(When) Will CMPs hit the Power Wall?*, Proceedings of Euro-Par Workshops - Highly Parallel Processing on a Chip (HPPC), 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. 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.

15. 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.

Local Conferences

16. M Briejer, **C.H. Meenderinc**k, B.H.H. Juurlink, *Energy Efficient Branch Prediction on the Cell SPE*, Proceedings Workshop on Circuits, Systems and Signal Processing (ProRISC), 2009.

17. **C.H. Meenderinck**, B.H.H. Juurlink, *A Chip MultiProcessor Accelerator for Video Decoding*, Proceedings Workshop on Circuits, Systems and Signal Processing (ProRISC), 2008.

18. **C.H. Meenderinck**, B.H.H. Juurlink, *(When) Will CMPs hit the Power Wall?*, Proceedings Workshop on Circuits, Systems and Signal Processing (ProRISC), 2007.

19. 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

20. **C.H. Meenderinck**, B.H.H. Juurlink, *The SARC Media Accelerator - Specialization of the Cell SPE for Media Acceleration*, CE technical report, 2009, CE-TR-2009-01.

21. **C.H. Meenderinck**, B.H.H. Juurlink, *(When) Will CMPs hit the Power Wall?*, CE technical report, 2008, CE-TR-2008-04.

22. **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.

Tutorials & Manuals

23. **C.H. Meenderinck**, *Implementing SPU instructions in CellSim*, http://pcsostres.ac.upc.edu/cellsim/doku.php/, 2007.

24. **C.H. Meenderinck**, *CellSim/SarcSim Profiling Tool Manual*, 2008.

International Journals (not directly related to this thesis)

25. **C.H. Meenderinck**, S. D. Cotofana, *Computing Division Using Single-Electron Tunneling Technology*, IEEE Transactions on Nanotechnology, Volume 6, Number 4, 2007.

26. **C.H. Meenderinck**, S. D. Cotofana, *An Analysis of Basic Structures for Effective Computation in Single Electron Tunneling Technology*, Romanian Journal of Information Science and Technology, Volume 10, number 1, 2007.

Conferences (not directly related to this thesis)

27. **C.H. Meenderinck**, S. D. Cotofana, *Basic Building Blocks for Effective Single Electron Tunneling Technology Based Computation*, Proceedings of International Semiconductor Conference (CAS), 2006.

28. **C.H. Meenderinck**, S. D. Cotofana, *Computing Division in the Electron Counting Paradigm using Single Electron Tunneling Technology*, Proceedings of IEEE Conference on Nanotechnology, 2006.

29. **C.H. Meenderinck**, S. D. Cotofana, H*igh-Radix Addition and Multiplication in the Electron Counting Paradigm Using Single Electron Tunneling Technology*, Proceedings of International Workshop on Computer Systems: Architectures, Modelling, and Simulation (SAMOS), 2006.

30. **C.H. Meenderinck**, S. D. Cotofana, *Electron Counting based High-Radix Multiplication in Single Electron Tunneling Technology*, Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS), 2006.

31. **C.H. Meenderinck**, C. R. Lageweg, S. D. Cotofana, *Design Methodology for Single Electron Based Building Blocks*, Proceedings of IEEE Conference on Nanotechnology, 2005.

32. **C.H. Meenderinck**, S. D. Cotofana, C. R. Lageweg, *High Radix Addition Via Conditional Charge Transport in Single Electron Tunneling Technology*, Proceedings of International Conference on Application-Specific Systems, Architectures and Processors, 2005.

33. **C.H. Meenderinck**, S. D. Cotofana, *Periodic Symmetric Functions and Addition Related Arithmetic Operations in Single Electron Tunneling Technology*, Proceedings of Workshop on Circuits, Systems and Signal Processing (ProRisc) 2005.

34. **C.H. Meenderinck**, S. D. Cotofana, *Computing Periodic Symmetric Functions in Single Electron Tunneling Technology*, Proceedings of International Semiconductor Conference (CAS), 2005, **Best paper award**.

Samenvatting

De zoektocht naar steeds krachtigere procesoren heeft ertoe geleidt dat steeds vaker multicore architecturen worden toegepast. Omdat de klok frequentie niet meer hard steeg, en omdat ILP technieken steeds minder opleverden, was het vermeerderen van het aantal cores per chip een natuurlijke keuze. Het transistorbudget groeit nog steeds, en het is daarom te verwachten dat binnen tien jaar een chip honderd krachtige cores kan bevatten. Maar het opschalen van het aantal cores vertaalt zich niet noodzakelijkerwijs in een gelijke opschaling van de prestaties. In dit proefschrift presenteren wij een aantal technieken om de schaalbaarheid van de prestaties van multicore systemen te verbeteren. Met deze technieken gaan wij in op een aantal belangrijke uitdagingen op het gebied van multicores.

Ten eerste onderzoeken wij het effect van de *power wall* op de architectuur van toekomstige multicores. Ons model bevat een voorspelling van technologische verbeteringen, een analyse van symmetrische en asymmetrische multicores, alsmede de invloed van de wet van Amdahl.

Ten tweede onderzoeken wij de parallellisatie van een H.264 video decoderingsapplicatie, waarmee we ingaan op de schaalbaarheid van applicaties. Bestaande parallellisatiemethodes worden besproken en een nieuwe strategie wordt gepresenteerd. Analyse laat zien dat met het gebruik van de nieuwe parallellisatiestrategie de hoeveelheid beschikbare parallellisme in de orde van duizend is. Verscheidene implementaties van de strategie worden besproken, welke de moeilijkheden en de mogelijkheden laten zien van het benutten van het beschikbare parallellisme.

Ten derde presenteren wij een Applicatie Specifieke Instructie Set (ASIP) processor voor H.264 decodering, welke gebaseerd is op de Cell SPE. ASIPs zijn energiezuinig en maken het mogelijk om de prestatie op te schalen in systemen waar energieverbruik een beperkende factor is.

Ten slotte presenteren wij hardware ondersteuning voor taak management, waarvan de voordelen tweevoudig zijn. In de eerste plaats ondersteunt het het SARC programmeermodel, welke een taak gebaseerde dataflow programmeermodel is. Door te voorzien in hardwarematige ondersteuning voor de meest tijdrovende onderdelen van het runtime systeem, wordt de schaalbaarheid verbeterd. In de tweede plaats vermindert het de parallellisatieoverhead, zoals synchronisatie, door te voorzien in snelle hardware primitieven.

Curriculum Vitae

Cor Meenderinck was born in Amsterdam, the Netherlands, on January 29th 1978. From 1990 he took the secondary education at the Herbert Vissers College in Nieuw-Vennep, where he graduated in 1996. In the same year he became a student at the Electrical Engineering, Mathematics, and Computer Science Faculty of Delft University of Technology. He obtained a BSc degree in electrical engineering and later an MSc degree, with honors, in Computer Engineering.

In 2005, Cor started working as a PhD student with the Computer Engineering Laboratory of Delft University of Technology. The first year he worked on single electron tunneling based arithmetic operations, under supervision of Sorin Cotofana. From 2006 on he worked on the SARC project under supervision of Ben Juurlink. He was actively involved in this European project (16 partners) as well as in research clusters of the HiPEAC network of excellence. He has closely worked together with researchers from several universities and paid a long term visit to the Barcelona Supercomputer Center (BSC) twice.

His research interests include computer architecture, multicores, media accelerators, parallelization, design for power efficiency, task management, programming models, and runtime systems.