UNIVERSITY OF OSLO
Department of Informatics

Investigation of parallel programming on heterogeneous multiprocessors

Masteroppgave

Håvard Espeland

Contents

List of Figures

Preface

Multi-core processors have become ordinary in modern commodity computers. Computationally intensive applications, like video processing, that previously only ran on specialized hardware, are now common on home computers. However, the demand for more computing power is ever-increasing, and with the introduction of high definition video, more performance is desired. As an alternative to having multiple identical processor cores, heterogeneous multiprocessors have cores with different capabilities. This allows tasks to be processed on simple cores with specialized functionality. The simplicity furthers low power consumption, small die usage, and low price.

Dealing with heterogeneous cores increases the complexity of writing programs for the architecture. The reasons for this includes different capabilities of the cores, and some heterogeneous architectures do not have shared memory. Without shared memory, accessing main memory requires explicit transfers to local memory. In this thesis, we consider two architectures, the STI Cell/B.E. and Intel IXP2400, and evaluate parallelization strategies and performance for real-world problems. Our tests show promising throughput for some applications, and we propose a scheme for offloading computationally intensive parts of an existing application.

Acknowledgements

I would like to express my gratitude to Håkon Stensland, and Paul Beskow, and my advisors, Carsten Griwodz and Pål Halvorsen. They have all contributed valuable feedback, and without them, this work would not have been possible.

Thanks to the guys at the Simula lab, where I worked during my master studies, for motivation, input, and great conversations.

I will also use this chance to say thank you to my friends and family for always supporting me.

Oslo, July 31. 2008
Håvard Espeland

Abbreviations

AES	Advanced Encryption Standard
ASMP	Asymmetric Multiprocessing
CBC	Cipher Block Chaining Mode of Operation
CBEA	Cell Broadband Engine Architecture
CTR	Counter Mode of Operation
DCT	Discrete Cosine Transform
DES	Data Encryption Standard
DMA	Direct Memory Access
ECB	Electronic Codebook Mode of Operation
EIB	Element Interconnect Bus
FIFO	First in, First out
FPU	Floating Point Unit
GPGPU	General-Purpose computing on Graphics Processing Units
GPU	Graphics Processing Unit
HPC	High-performance Computing
HPEL	Half pixel
HTTP	Hypertext Transfer Protocol
MFC	Memory Flow Controller
NAL	Network Abstraction Layer
NOP	No operation
NPU	Network Processing Unit
PPE	Power Processor Element
PPU	Power Processor Unit
PS3	Playstation 3
QPEL	Quarter pixel
RTP	Real-time Transport Protocol

RTSP	Real-time Streaming Protocol
RX	Receive
SIMD	Single Instruction Multiple Data
SMP	Symmetric Multiprocessing
SMT	Simultaneous Multithreading
SPE	Synergistic Processing Element
SPU	Synergistic Processing Unit
TCP	Transmission Control Protocol
TFRC	TCP Friendly Rate Control
TX	Transmit
UDP	User Datagram Protocol

Chapter 1

Introduction

1.1 Background and motivation

Strive for more processing power has been an ever-increasing demand since the introduction of the computer. The number of transistors that are used in a microprocessor has approximately doubled every 24 months, close to what Gordon Moore predicted in 1965, which is now famously known as Moore's Law. For decades, manufacturers have been using multiple processors to meet the demand for increasing computational capabilities. This can be done using a number of computers working together in a cluster configuration, or by using two or more processors in a single system.

Increasing the clock frequency of a CPU is limited by physical properties, and servers with multiple CPUs have been customary. This technique is known as symmetric multiprocessing (SMP), here the operating system treats each core alike. SMP is not a new concept, and was was used at least as early as 1961, when the Burroughs B5000 series mainframe became available with two processors [9]. SMP can be further scaled by clustering processors together near memory, and having multiple clusters. The clusters are able to access all memory directly, but at different latency and bandwidth, depending on how far the cluster operates from the memory. This technique is known as non-uniform memory architecture (NUMA), and allows scaling to many processors in a single system.

In 2001, IBM released the first commercial multi-core processor [10] with two independent cores on a single die. This marked the introduction of multiple CPUs to commodity hardware, and soon after, Intel and AMD also released multi-core CPUs. Multi-core is viewed, from the operating system point of view, as a regular SMP multi processor system, and applications written using threads can take advantage of them without adaptations.

There are also drawbacks of having multiple CPUs in a SMP configuration, in some architectures the cores share the same bus and memory bandwidth. The shared resources can become exhausted, when the number of cores increase. As a result of this, asymmetric architectures with local storage only available for a limited set of cores have become available. With asymmetric multiprocessing (ASMP), the operating system or application do not treat all cores identically, as done with SMP. This allows some cores to have special functions, limitations, or clock speed. Architectures that have cores with different capabilities, are often called heterogeneous architectures. Kumar et al. [11] performed an evaluation of two processors of the heterogeneous Alpha architecture. They achieved up to 63% better throughput compared to equivalent homogeneous processors occupying the same die area, when performing a set of benchmarks. The smaller die area used by heterogeneous architectures can both lead to reduced power consumption and cost of the processor. It also allows more processing elements to be placed in the die area.

Using co-processors for offloading tasks is an idea used since the beginning of computers. The familiar x86 architecture can use several processors for accelerating tasks, including the MMU for virtual memory, graphics processor for rendering, and network processor for offloading part of the network stack. What has happened in the recent years is the introduction of programmable co-processors which are able to perform general-purpose tasks in commodity hardware. The ability to run programmable shaders revolutionized the graphics processors, which used to have a fixed pipeline in hardware. This also allowed non-graphics related tasks to be executed on the processors. The use of heterogeneous multiprocessors in game consoles also furthers the availability of such architectures.

Heterogeneous multiprocessors are typically configured in such a way that there are a limited number (usually one) of fat cores and many thin cores. The fat cores can be based on existing architectures like PowerPC or ARM, and may run a normal operating system. The fat core's main task is to control a number of thin cores, which are simple, high-performing specialized cores, only capable of doing some tasks fast. Their simplicity makes it possible to have many thin cores on a single die, with a high-speed interconnect, and in some cases access to a shared memory and I/O.

1.2 Problem statement

Because the cores must be treated with regard to the individual core's features and connectivity, programming heterogeneous architectures can be difficult.

The complexity of developing for such architectures varies between platforms, and there is also differences in what types of applications that are suitable, and how to achieve high throughput. We want to investigate how to accelerate programs on two of these architectures, and to evaluate different strategies for parallel execution. Among the issues we investigate in the next chapters are job distribution schemes to accelerate existing projects by offloading CPU-intensive parts from the main processor to processing elements. We also look at strategies for parallelization, and limitations of the heterogeneous architectures.

1.3 Research Contributions

During the master studies, we have published two papers on protocol translation on an Intel IXP network processor [12, 13], released an parallel AES implementation for the STI Cell under the GPL license, and gained valuable insights on how to accelerate computationally intensive applications on heterogeneous architectures. This is, to the best of our knowledge, the first open source AES implementation for CBEA. We have also made progress on the goal of having an open-source H.264 encoder available for Cell, and looked at optimizing the encoder for the architecture without breaking the portability.

1.4 Outline

The rest of this thesis is organized as follows; Chapter 2 introduces the two heterogeneous architectures we use in the evaluations. In chapter 3, we look at our protocol translation proxy, and have a look at the pipelining strategy for processing of network packets on different cores. Chapter 4 looks at a branch-and-bound problem on the Cell architecture, and details how the architecture is programmed. In chapter 5, we investigate parallelization of block cipher encryption on the Cell platform, and in chapter 6, we study video encoding on the Cell platform. Chapter 7 contains the discussion of the results and lessons learned, and the conclusion and further work can be found in chapter 8.

Chapter 2

Heterogeneous multi-core architectures

In this chapter, we look at heterogeneous multiprocessors, and cover details of the architectures. In particular, we look at the two architectures used in the investigations of the next chapters, the Cell Broadband Engine, and the Intel IXP. Also, We highlight features and limitations of the architecture, and some of these will be looked further into during the next chapter's evaluations.

2.1 Cell Broadband Engine

The Cell Broadband Engine (CBE) [14] is a heterogeneous multiprocessor developed by Sony, Toshiba, and IBM. It is designed for fields such as high performance computing (HPC), game consoles, and video processing [15].

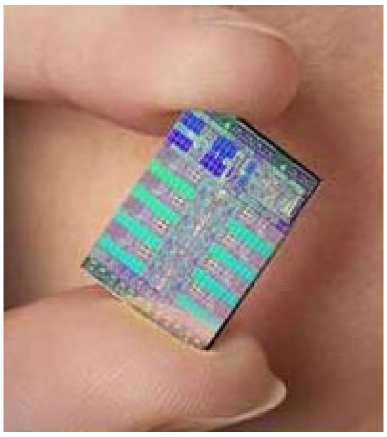


Figure 2.1: The Cell Broadband Engine multiprocessor (public domain image).

The project was initiated by Sony when they contacted IBM in 2000 and requested a processor with 1000 times the performance of the Playstation 2 Emotion Engine [16] for use in their next generation video game console [17]. As this performance goal turned out to be overly optimistic, the target performance was set to 100 times the Playstation 2 performance, and Toshiba joined as a technology partner. The three companies formed an alliance named STI, and in november 2006, Sony launched the Playstation 3 as the first commercial device embracing the CBE architecture (CBEA). For HPC, IBM and Mercury Computing offer blade servers featuring multiple Cell processors, and Toshiba offers laptops using CBE as an accelerator to offload media processing [18].

2.1.1 Hardware overview

The central components of the CBEA are the Power Processing Element (PPE) and a number of Synergistic Processing Elements (SPE), these are all connected by the Element Interconnect Bus (EIB). The system memory and the Flex IO interface are also connected to the EIB. The Flex IO interface further connects devices such as network interfaces and disk controllers. An overview of the CBEA main components are shown in figure 2.2. The current version of the CBE is clocked up to 3.2 GHz and have 8 SPEs.

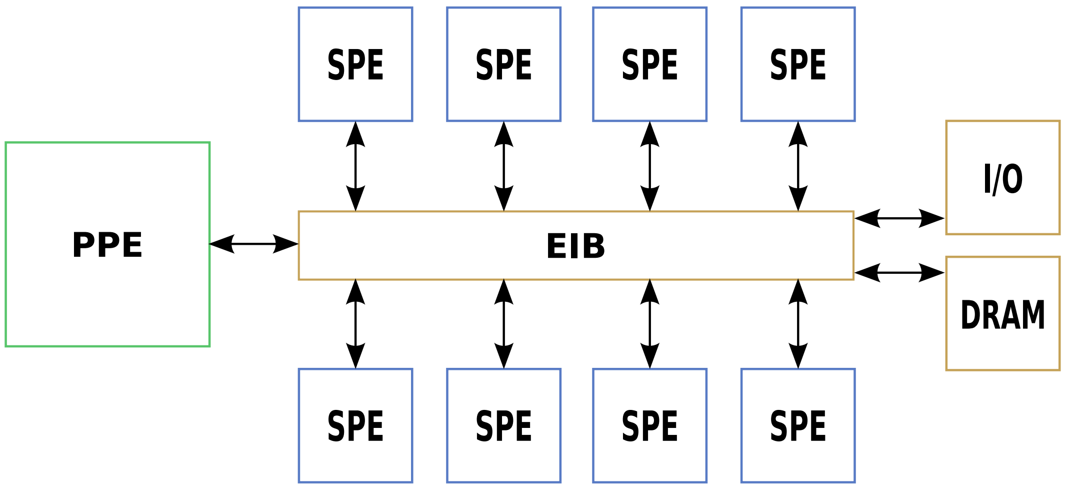


Figure 2.2: Main components of the Cell Broadband Engine

The PPE contains a general purpose 64-bit core based on PowerPC 970 (PPU), capable of executing two simultaneous hardware threads, similar to Intel's SMT technology described later. The primary job of the PPE is to control the SPEs, run the operating system, and manage system resources [19]. The PPE supports virtual memory, and features two levels of cache (32 kB level 1 instruction cache, 32 kB level 1 data cache, and 512 kB level 2 unified cache), a branch unit, and a vector (VMX) unit.

The vector unit in the PPE is a standard AltiVec compatible single instruction, multiple data (SIMD) unit. It is capable of processing four independent 32-bit words, eight 16-bit shorts, or sixteen 8-bit bytes, in a total of 128 bits. The unit can work on both integer and floating point values, but not all instructions are available for both types. Using SIMD instructions can improve the throughput in multimedia applications because of the parallel processing on units (see figure 2.4) executed as a single instruction. The SPEs are programmed mainly by SIMD instructions, but they are not AltiVec compatible. The available vector instructions on the PPE and SPEs have different names but are very similar.

An SPE contains a Synergistic Processing Unit (SPU) [1], and a Memory Flow Controller (MFC) as illustrated in figure 2.3. The MFC is unable to directly access the system memory, and can only use a small amount of fast local memory, known as the local storage. The local storage is 256 KB large and used for code, data, and stack without segmentation. The Memory Flow Controller passes data back and forth between the system memory and local storage. This is done by Direct Memory Access (DMA) transfers, which can be requested both from SPEs and the PPE. The SPEs have a large 128-entry 128-bit register file and support two simultaneous instructions using two execution pipelines. The instruction set used on SPEs is not PowerPC or PPE compatible, and operates primarily with vector operands.

The SPEs support both single and double precision floating point numbers, although the single precision is not fully IEEE 754 [20] compatible. The range has been extended by removing NaN and Infinity. Rounded numbers are always truncated down towards zero. The reason for this incompatibility is to trade accuracy for speed to better adapt the CBEA for real-time graphics and multimedia processing applications by simplifying the hardware [19].

2.1.2 Communication and synchronization

Programming the CBE often involves synchronization of several parallel tasks. To increase performance, the CBEA includes hardware support for several primitives, including atomic updates, message passing, and signals. The Memory Flow Controller supports atomic operations by DMA, which can be used to implement standard synchronization primitives such as mutexes and semaphores. Message queues for inter-

[1] The abbreviations SPE/SPU and PPE/PPU are often interchanged and used ambiguously in literature, and except for in this overview, SPE and PPE will be used, and a strict distinction between element and unit will not be enforced.

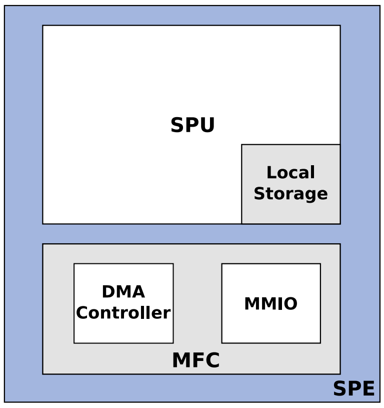


Figure 2.3: A Synergistic Processing Element

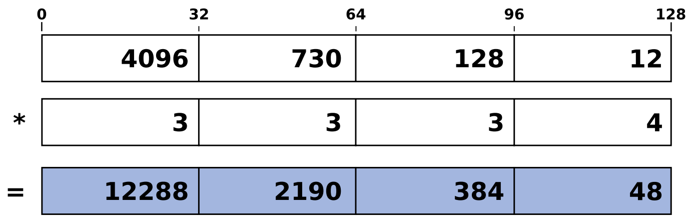


Figure 2.4: An example SIMD multiplication of two 128-bit vectors containing four 32-bit integer values. The multiplications are computed in parallel by a single instruction.

communication between the PPE and SPEs, and internally among the SPEs, are also available. Similarly, SPEs and the PPE can send signals, which together with messages are received either by polling, or by using an interrupt handler.

2.1.3 The Playstation 3 and Linux

The Playstation 3 uses a Cell multiprocessor clocked at 3.2 GHz, which can run Linux as an alternative to Sony's default operating system used for gaming. The featured processor is somewhat limited, when compared to a fully fledged CBE. One SPE is removed to get better yield in the production process, and one is reserved for a hypervisor, leaving six SPEs available for usage. The system has 256 MB of memory, with about 40 MB reserved by the hypervisor. The hypervisor also blocks access to certain parts of the hardware, including the graphics processing unit (GPU).

Both IBM and Sony are actively participating in improving the CBEA Linux support. Developers can fully utilize both the PPE and SPEs. Both have context switching abilities, including preemptive scheduling (SCHED_OTHER) for the SPEs. IBM also provides a cycle accurate, full system simulator for the CBEA, which can be used if suitable hardware is not available, and to aid debugging.

The CBE can be programmed using the standard gcc compiler to target both SPEs and the PPE. As an alternative, IBM provides the proprietary compiler XL. Accessing the SPEs from userspace in Linux is abstracted by the spufs [21] virtual file system, which provides the low level interface. Spufs is then used by gdb [22] and the libspe2 [23] runtime library for controlling the SPEs from the processes. The spufs virtual file system exposes the local storge, mailboxes, signals, DMA, execution state, and more to user space. It can be used to read resources and manipulate them directly. Before running a program on a SPE, the bytecode is uploaded to local storage using the libspe2 library, and is started by sending a signal.

2.1.4 Performance tuning

Successfully mapping a larger problem to the CBE can be a significant task. To fully utilize the computational capability of the CBE, the programmer must explicitly control all the resources on the processor. In particular, there is a need for strict data segmentation to fit relevant data in the local storage, and most importantly, avoid stalling the SPEs. Making sure the SPEs always have work pending includes hiding DMA latency, reducing synchronization overhead, eliminating branch misses, and optimizing for cache.

Hiding DMA latency is often accomplished by using multiple buffers, usually double buffering. This allows the SPE to work on a dataset while copying another. Reducing branch misses are especially important on SPEs, because of the lack of dynamic branch prediction. Writing branch free code is possible in many cases, and further speedup can be achieved by unrolling loops and hinting the processor on the likely outcome of branches. A good cache usage can make a big difference as well, i.e., DMAing to or from a non-aligned or non-multiple size of a cache line (128 B), yields a performance penalty.

A few components normally found in other processors are left out of a SPE to save power and cost. This includes above mentioned branch prediction and support for out-of-order execution. Instead, the compiler or the programmer is expected to take care of these issues. A special branch hint instruction is supported to guide the processor and reduce misses. Branch misses on the SPEs are extremely expensive. Correctly predicted branches run in a single cycle, while a mis-predicted branch cost up to 18 or 19 cycles depending on the address of the branch target. Furthermore, since the CBE uses an in-order execution model, shuffling instructions can provide a performance boost by avoiding pipeline stalling. Also, up to two instructions can run in parallel if they are compatible, and the data they work on is independent.

In an attempt to simplify programming, IBM developed an prototype auto-parallelizing compiler optimized for the CBEA, known as the OctoPiler. This compiler aims at reducing the complexity of using the architecture for novice programmers by automatically optimizing alignment, scalar operations by SIMD, and threads by SPEs. Eichenberger et al. claims in [24] that using IBM's Octopiler compiler increases productivity, and presents promising measurements compared to code not optimized for the CBE. He does not compare performance to hand written optimizations. On the contrary, Greene [25] recommends turning off all compiler optimizations (both on IBM's XL and the gcc compiler) and doing the optimize work manually to generate acceptable SPE code.

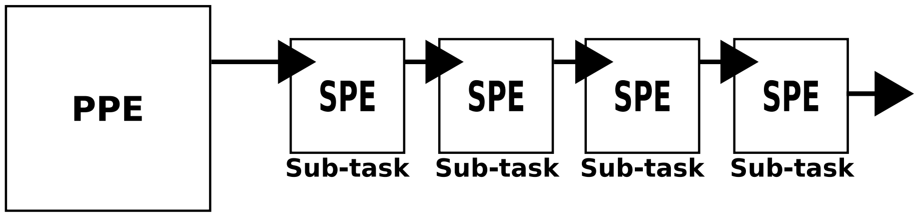
Another project for automatic parallelization is the Parallel For project [26], which targets multiple platforms including CBEA. The Parallel For project does automatic parallelization by analyzing dependencies in sequential code, and distribute the work to SPEs. Parallelizable loops are marked in a similar manner as OpenMP, resulting in generated PPE and SPE code. Zumbusch, the author of Parallel For, benchmarked smoothing of a multigrid in [27] using Parallel For compared to various SMP architectures and got decreasing execution time when using multiple SPEs. The SPE results does not scale as well as the SMP benchmarks. He argues that the performance is limited by the local memory size.

2.1.5 Parallelization schemes

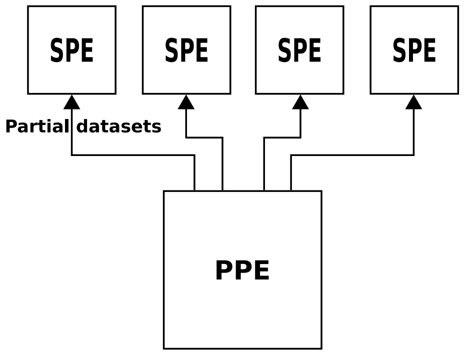
Traditional parallelization concepts, such as pipelining and data partitioning, are applicable on the CBEA (illustrated in figure 2.5), although they require consideration due to the non-shared memory architecture. The complicated part is to efficiently partition the data between parallel tasks, while keeping the amount of synchronization overhead to a minimum.

Several problems require steps to be completed sequentially, and some programs may benefit from a hybrid approach, where the main strategy is pipelining, while some parts of the code may gain an advantage by doing data partitioning on multiple cores. In the case where there are more SPE tasks than available cores, a scheduler is needed.

An SPE task is called an SPE context in Linux, and describes the complete SPE state for a running or suspended SPE task. Several scheduling policies are available in Linux including preemptive scheduling (SCHED_OTHER) and FIFO scheduling (SCHED_FIFO).



(a) A pipelining strategy where the data passes through several cores to solve sub-tasks, ending up with a complete solution.



(b) A parallelization strategy where data is partitioned and each partition is processed on a single core.

Figure 2.5: Common parallelization strategies

The current preemptive SPE scheduler runs every 100ms, and hands out time slices rewarding execution time to contexts with high priority. The FIFO scheduler uses a voluntarily model, where the context can be given full SPE control when it has work pending, or otherwise yields, and let another context run. The preemptive policy may not be suitable when overbooking the SPEs, since the scheduler is unaware of when a context has work pending, thereby giving execution time to idle contexts. The FIFO scheduling policy keeps threads running until they yield voluntarily, giving the programmer more explicit control of the scheduling.

Spawning more threads than available cores can make sense for at least two reasons; Several processes may be sharing the same CBE, and the type and amount of work the CBE has to perform on the data may vary. However, a trade-off with running more threads than available processing elements, is a possible performance penalty, because of the context switch and scheduling overhead. Writing programs in such a way can also turn out to be an advantage, since newer models of the CBE family, or alternative platforms, may have more cores available, making the code portable with a small effort.

2.1.6 Comparison to x86 symmetric multi processors

The x86 architecture supports symmetric multiprocessing (SMP), using more than one CPU socket, and recently also multiple CPU cores per socket. Intel also supports simultaneous multithreading (SMT) on some of its processors, this is known as HyperThreading. HyperThreading presents two virtual processors for every physical core to the operating system. Processes can be scheduled to both, and the CPU itself will perform context switching between them, allowing otherwise wasted cycles during stalls to be utilized.

The CBEA is different in key factors compared to the Intel x86 architecture. The two major differences are the heterogeneity of the cores, and the non-shared memory architecture with an exclusive local storage on each SPE. Data locality is an important issue on SMP as well, because of cache coherency, but scattered memory usage requires more attention on non-shared memory architectures. The heterogeneity of the cores leads to more complex development, where multiple compilers, and duplication of code is necessary to communicate between different types of cores..

Not all problems scale well on the CBEA compared to the x86 architecture. These are mainly related to the limitations of the hardware, covering scalar problems, branching problems, and problems with low data locality. Since the SPEs are vector processors they do not show their full potential processing scalar data. Good use of background DMA transfers and prefetching can mitigate the penalty of not having shared memory. Although some of these cases may also be troublesome on the x86, the added complexity of non-shared memory and large branch misprediction penalty is avoided.

2.1.7 Potential and applications

Williams et al. [28] have investigated the potential of the CBEA for scientific computing and conclude that the potential is tremendous for both single and double precision calculations. The slower double precision, compared to single precision speed is a limitation, but not a show-stopper. In [29], Buttari et al. report on using Playstation 3 nodes for scientific computing and describe several limitations, including limited main memory access speed and size, and the relatively slow gigabit ethernet adapter, compared to the processing speed. They conclude that the attractiveness for scientific computing on the Playstation 3 will depend on the application's characteristics.

In 2005, Sakai et al. [30] gave a keynote speech about performance evaluation of the CBE at Toshiba. They used a MPEG-2 to H.264 transcoder as an example application, running 2 simultaneous encoders each using 3 SPEs. In addition to the standard CBE optimization techniques, they discovered that bandwidth reservation between the encoders was necessary to avoid over-utilizing the system bus. Unfortunately, very few details are available in slides from the keynote, and apparently no paper was published.

2.2 IXP 2400 Network processor

The IXP2400 chipset [31] is a second generation, highly programmable network processor (NPU) and is designed to handle a wide range of access, edge and core network applications. The basic elements include a 600 MHz XScale (ARM compatible) core running Linux or VxWorks, eight 600 MHz special packet processors called micro-engines (μEngines), several types of memory and different controllers and buses.

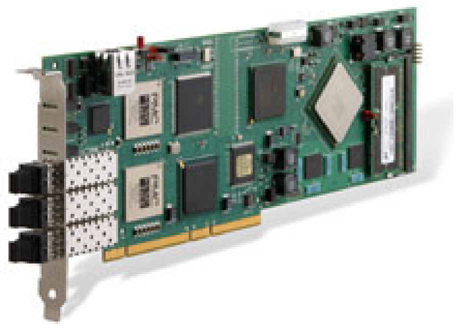


Figure 2.6: ENP-2611 from RadiSys with an IXP2400 Network Processor. The ENP-2611 is a PCI-X card working independently of the host computer. It allows DMA transfers to and from the host computer using the PCI bus.

With respect to the different CPUs, the XScale is typically used for the control plane (slow path), while the μEngines perform general packet processing in the data plane (fast path). The three memory types are used for different purposes, according to access time and bus bandwidth, the 256 MB SDRAM is used for packet store, the 8 MB SRAM for meta-data, tables and stack manipulation, and the 16 kB scratchpad (on-chip) for synchronization and inter-process communication. The physical interfaces are customizable, and can be chosen by the manufacturer of the device on which the IXP chipset is integrated. The number of network ports and the network port type are also customizable.

The major functional blocks of the IXP2400 architecture are shown in figure 2.7. The

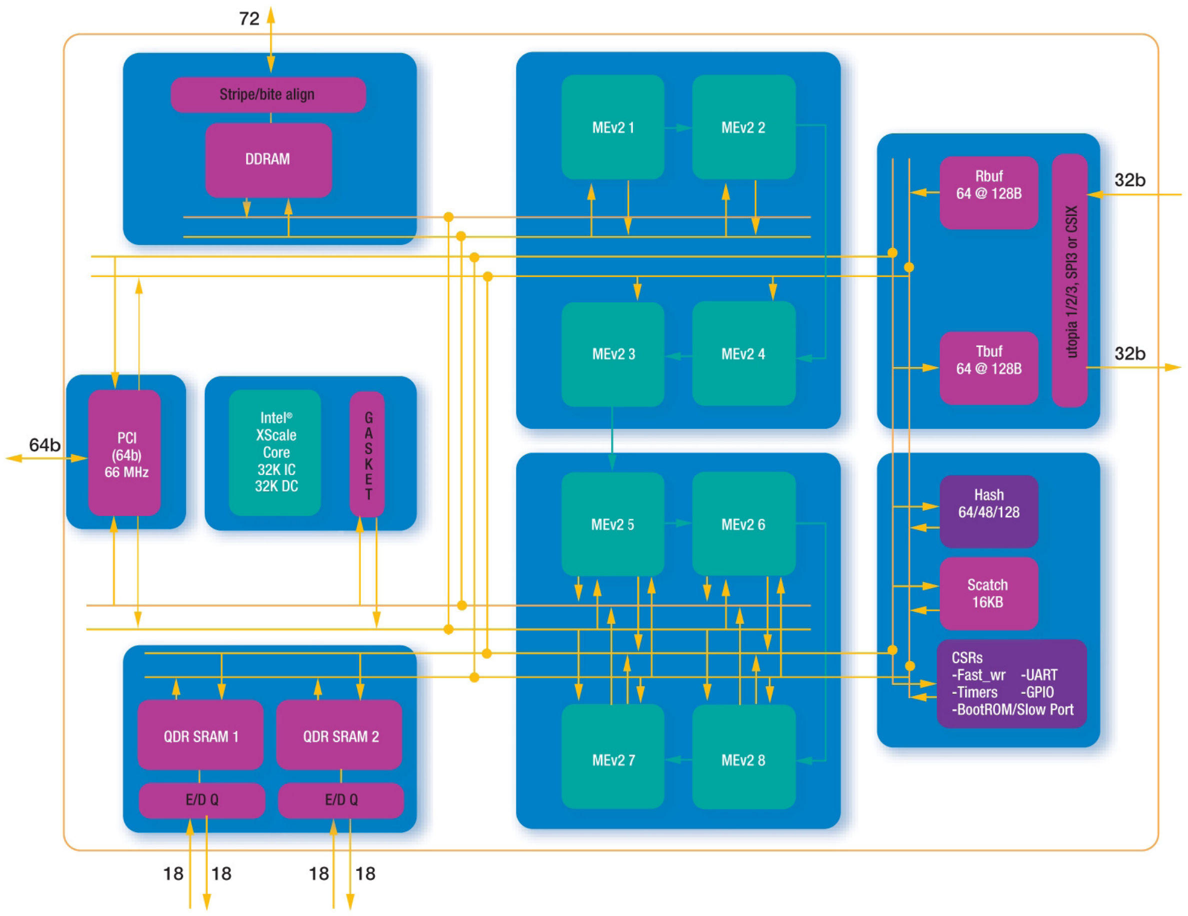


Figure 2.7: IXP2400 architecture overview. The figure is from an Intel product brief on IXP2400 [1]

μEngines are grouped together in clusters of four, which can communicate by next-neighbour registers internally. In normal configurations, two μEngines are reserved for low-level network receive and transmit functions, leaving six μEngines available for application usage.

The SDK includes a specialized C compiler for the MicroC language, in which the μEngines can be programmed. MicroC is a C-like language allowing rapid development and reuse of code. Synchronization and message passing between the cores can be performed as atomic operations on dedicated hardware channels known as scratch rings. Using a NPU for a network application allows processing on network wire speed, in the case of the IXP2400, up to 2.5 GBps, with very low latency [1]. Such performance can be achieved due to the dedicated architecture for network processing, and software implementation. Packet processing can be done with a limited protocol stack, allowing minimal processing overhead both for extracting information and updating network packets. This makes network processors ideal for high-performance, low-complexity network operations like statistics collection, deep packet inspection, or packet rewriting.

2.3 Other architectures

2.3.1 Graphics Processing Units

Graphics processing units (GPUs) are available in most commodity PCs, and the last generation are highly programmable. These includes recent graphics cards from AMD and nVIDIA. The GPUs are designed for graphics processing, but can provide a high performance for many computational intensive applications, known as general-purpose computing on graphics processing units (GPGPU). The traditional approach of using GPUs for general purpose applications has been to write vertex and fragment shaders using a specialized graphics API (OpenGL / DirectX). This imposed limitations, and in late 2006, ATI announced the Close-To-Metal [32] framework for GPGPU. Three months later, nVIDIA announced its GPGPU framework, CUDA [33].

The first generation of GPUs supporting CUDA was the G80 series from nVIDIA. The high-end G80 card Geforce 8800 GTX has 128 stream processors (also known as thread processors), organized as illustrated in figure 2.8. The stream processors have single-precision FPUs, and clusters of eight cores share a local memory of 16 kB. Each of the stream processors can handle up to 97 concurrent threads, and developers using the platform are expected to write applications using thousands of threads to get good performance.

Controlling such a large number of threads explicitly would render programming for the platform cumbersome. Instead, the CUDA framework uses a thread manager that dispatches and manages threads based on compiler directives. The programming language used is based on C, but with additional CUDA extensions.

GPUs can provide high throughput for computationally intensive applications that are suitable to the programming model required by the architecture. Schatz et al. [34] demonstrated that using an nVIDIA Geforce 8800 GTX, they could achieve a speedup of 3.79x compared to a similarly priced 3.0 GHz Xeon processor, when doing sequence alignment of DNA. The protein folding project Folding@Home [35] provides GPGPU clients for both nVIDIA and AMD cards. Another computationally intensive task that was successfully accelerated by a GPU is Reed-Solomon coding for use in RAID 6 disk

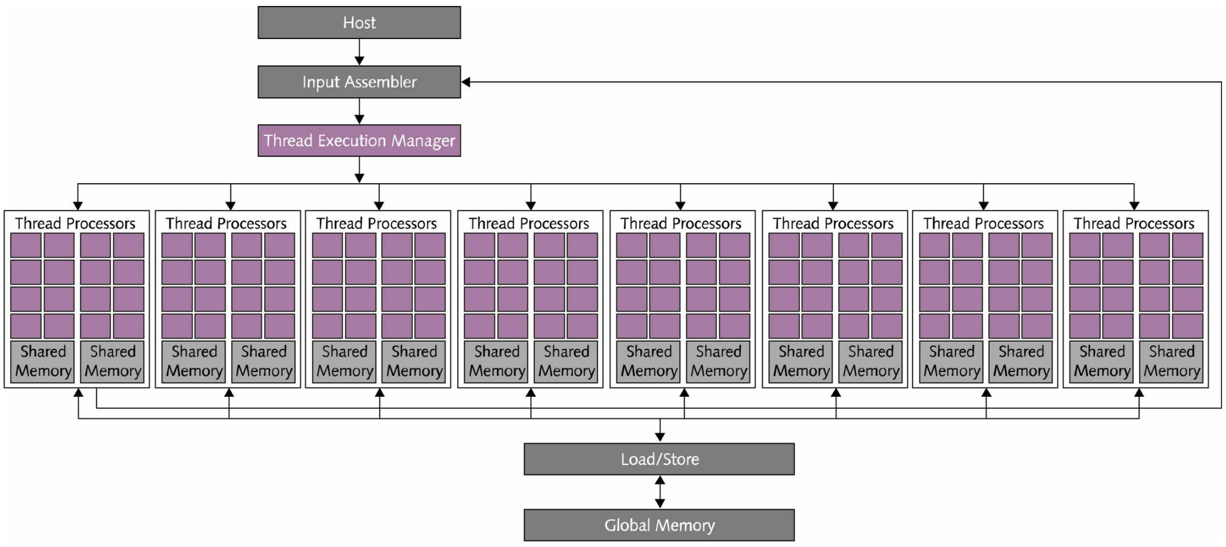


Figure 2.8: nVIDIA G80 architecture overview from [2]

systems by Curry et al. [36]. Their evaluations show that the GPU can outperform a modern CPU by an order of magnitude for this problem.

2.4 Summary

In this chapter we have presented an overview of the heterogeneous multi-core architectures STI Cell Broadband Engine and Intel IXP2400. Such architectures can deliver significant computational capabilities compared to traditional SMP in several fields. Unfortunately, utilizing the performance of these architectures leads to much extra work for the programmer, who has to be aware of how to structure the programs, considering the memory architecture and hardware limitations. Automatic tools for partitioning and memory management do exist, however they do not seem to be sufficient in areas where performance is first priority, like high performance computing or games.

In the next chapters, we consider some of the challenges and potential introduced here, and look at how this relates to programming for these architectures. This will be evaluated by implementations and benchmarks, both by adapting existing programs, and by writing from scratch. In chapter 3, we use a pipline scheme for processing packets on a network processor. In chapter 4, we introduce programming on the CBE, and compare the throughput of a branch-heavy application on SPE and PPE. In chapter 5, we introduce job dispatching of independent datasets, and look at what mechanism works best for distributing jobs, when the PPE is under heavy load. In chapter 6, we use what we have learned from the previous chapters, when offloading computationally intensive parts of a video encoder to SPEs.

Chapter 3

Protocol translation on Intel IXP network processor

In this chapter, we introduce a network protocol translating proxy for use in video streaming. The proxy is transparent, implying that both the client and server are unaware of its presence. It can be placed either near the client or the server, depending on the network link quality and the desired effects.

The proxy is implemented on the IXP asymmetric multi-core processor, and details of this proxy and network effects are available in our papers [12] and [13]. Some of the network related details presented in the papers are beyond the scope of this thesis, and we limit ourselves to present only the relevant details here.

This chapter is organized as follows; In section 3.1, we provide the background and motivation for deploying such a proxy. Section 3.2 contains an overview of the implementation of the proxy and some of the results from our network performance evaluations. In section 3.3 we look at what lessons that can be learned from this implementation.

3.1 Background information

The discussion about the most suitable protocols for streaming has been going on for years. Initially, streaming services on the Internet used UDP for data transfer, because multimedia applications often have large demands for bandwidth, reliability and jitter than could not be offered by TCP. Today, this approach is prevented by filters in Internet service providers (ISPs), firewalls in access networks and on end-user systems.

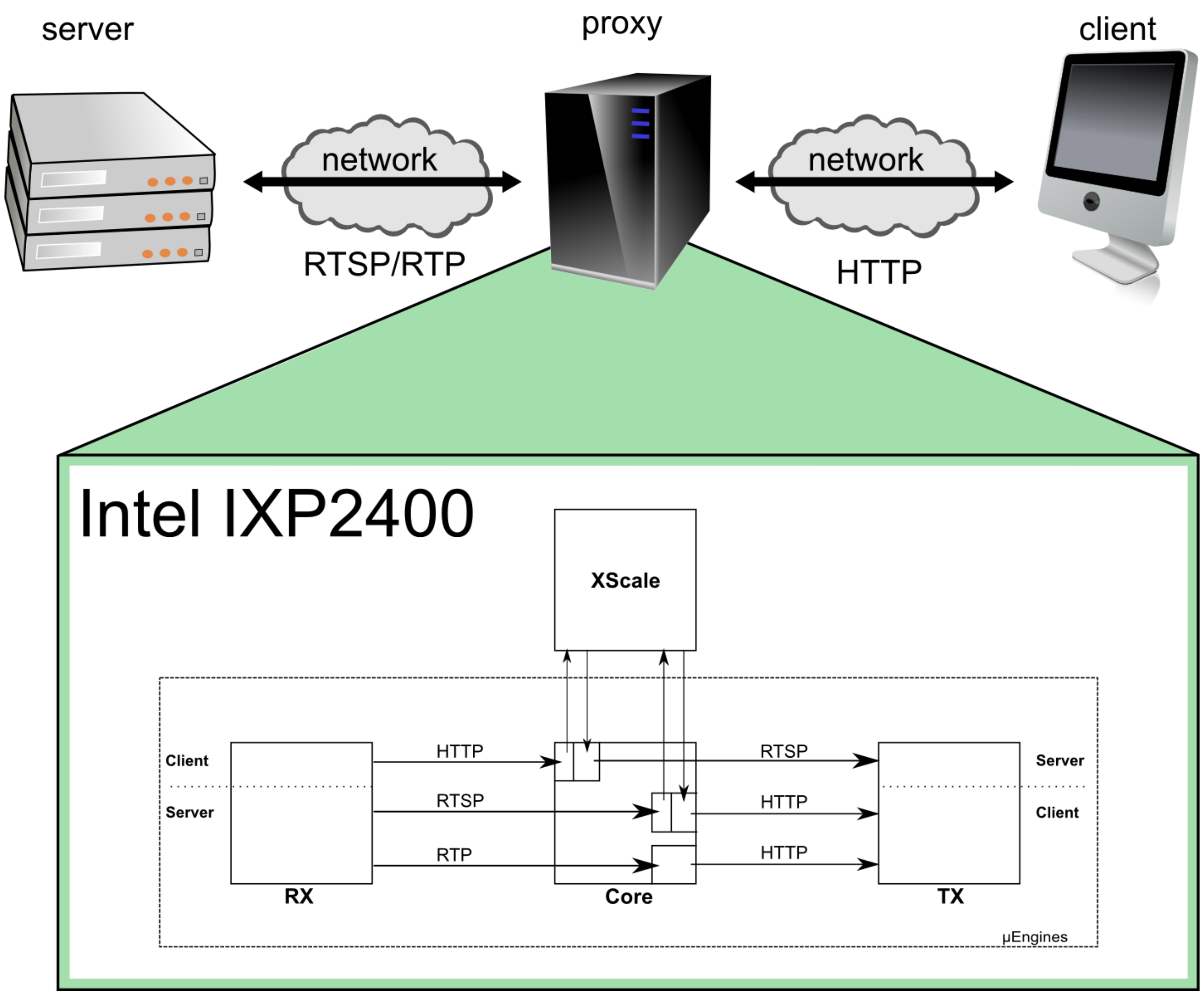


Figure 3.1: Network packet flow of the protocol translation proxy.

Some ISPs reject UDP because it is not fair against TCP traffic, and many firewalls reject UDP because it is connectionless and requires too much processing power and memory to ensure security. It has therefore become fairly common to use HTTP-streaming, which delivers streaming media over TCP. The disadvantage is that the end-user can experience playback hiccups and quality reductions because of the probing behavior of TCP, leading to oscillating throughput and slow recovery of the packet rate. A sender that uses UDP would, in contrast to this, be able to maintain a desired constant sending rate. Servers are also expected to scale more easily when sending smooth UDP streams and avoid dealing with TCP-related processing.

To exploit the benefits of both TCP and UDP, we experiment with a proxy that performs a transparent protocol translation. This is similar to the use of proxy caching that ISPs employ to reduce their bandwidth. We aim at live protocol translation in a TCP-friendly manner that achieves a high perceived quality to end-users. Our prototype proxy is implemented on an Intel IXP2400 network processor and enables the server to use UDP at the server side and TCP at the client side as illustrated in figure 3.1. The proxy also supports load-balancing of clients to a cluster of servers.

3.2 Implementation

The protocol translation proxy uses the XScale core and one μEngine application block. In addition, we use two μEngines for the receiving (RX) and the sending (TX) blocks. The transport protocol translation operation is shown in figure 3.1. According to the intended use of the different processors, incoming packets are classified by the μEngine based on the header. RTSP and HTTP packets are enqueued for processing on the XScale core (control path) while the handling of RTP packets is performed on the μEngine (fast path). TCP acknowledgements with zero payload size are processed on the μEngine for performance reasons.

The main task of the XScale is to set up and maintain streaming sessions, but after initialization, all video data is processed (translated and forwarded) by the μEngine. The proxy supports only a partial TCP/IP implementation, covering basic features. We do this to save both time and resources on the proxy. In our proxy, the protocol translation is performed by replacing headers (see figure 3.2). The translation reuses the space holding the RTP and UDP headers for the TCP header. It updates only the checksum and protocol number in the IP header before forwarding the packet. To deal with congestion control and fairness, we use TFRC [37], which is a specification for best effort flows that compete for bandwidth. The rate control is performed on a μEngine using fixed point arithmetics, and packets in excess of this rate are dropped.

The parallel approach used is a type of pipelining strategy, where a packet travels from core to core until transmitted. By only doing part of the work on each core, the level of load on the cores is reduced. We did not benchmark the scalability or utilization of the cores in our experiments, because the streaming server used could not handle more than a few connections. The proxy was designed in such a way that extra μEngines could be added in the pipeline for further scalability, if deemed necessary. This can be done by adding forward engines, which use the same channels for communicating with RX, XScale, and TX. The forward engines would work almost independently, by only sharing a common per-stream state in SRAM.

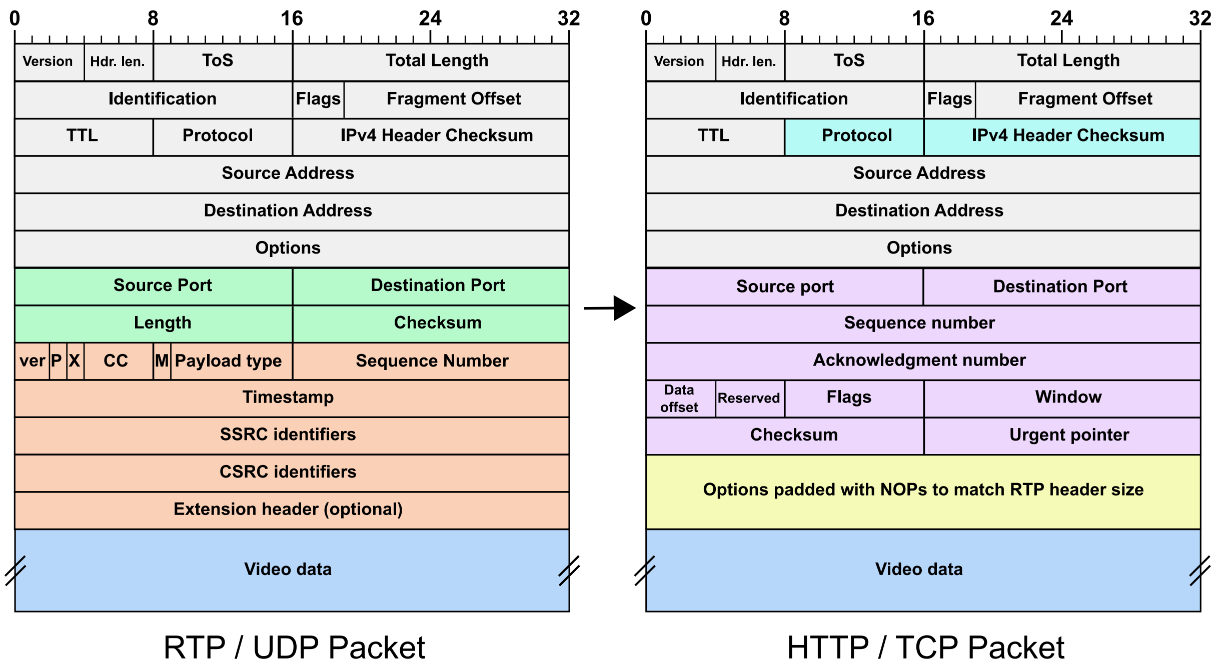
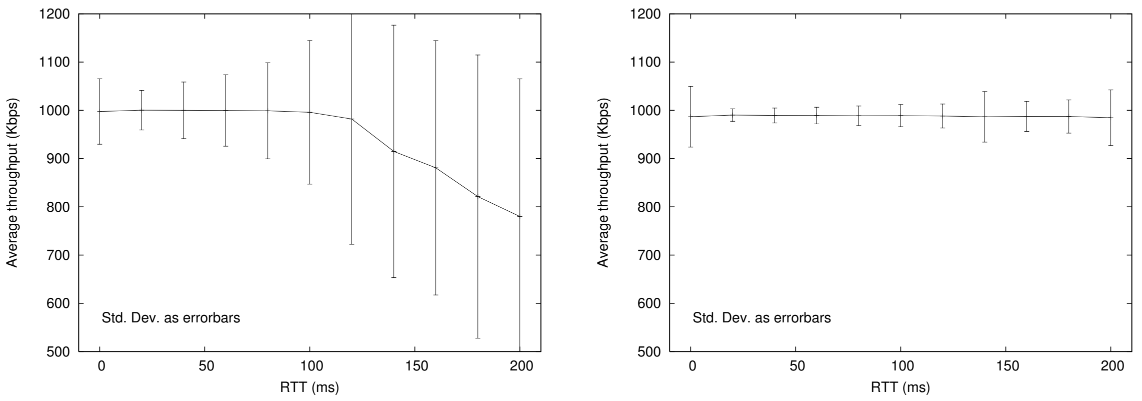


Figure 3.2: RTP/UDP to HTTP/TCP Translation

3.2.1 Network performance

We tested the proxy under different conditions of varying network latency (0-200ms) and drop rate (0-1%) by using a network emulator. These, and even harsher conditions are observed on the Internet [38]. Figure 3.3 shows examples of packet dropping between the server and the proxy. Dropping packets between the proxy and client has similar results. Figure 3.3(a) and 3.3(b) show the respective average throughput for HTTP and protocol translation scenarios under a constant loss rate of 1%. Keeping the link delay constant gives similar results. From the figures, we see that protocol translation achieves an average throughput that is hardly affected by link delays. In contrast, if using TCP end-to-end, the performance drops with an increasing loss rate and link delay, and the bandwidth oscillates heavily.

Using a network processor for implementing this proxy enables forwarding packets in the range of microseconds whereas typical existing systems require several milliseconds. The low overhead is caused by the minimal processing of network packets that does not have to traverse a network stack nor being copied to userspace when forwarding. The μEngines accesses the incoming packets directly, by only updating the changed packet headers. The XScale is used for session management, and only a few packets in every stream are processed on it. Since more μEngines are available, it is possible to use multiple forwarding cores, working in parallel. Benchmarks of scalability and core utilization were not within the scope of the papers, and are left as



(a) HTTP streaming (b) Protocol translation

Figure 3.3: Achieved bandwidth varying link latency with 1% loss

further work.

3.3 Lessons learned

The hardware-near approach on the IXP using no protocol stack delivers high throughput and low latency. Similar performance and transparency using a regular network card in a host PC, without overriding the network stack by writing applications as kernel modules seems unlikely. There is also an issue of overhead, since the host PC is not designed for low latency network applications, however, writing such programs on a network processor provides some interesting challenges.

The XScale processor runs both an operating system and a full protocol stack. This stack should not be used for the dataplane in high performance networking applications, as this would eliminate the advantage over a standard network card. Instead, it can be used for debugging and monitoring subsystems.

Network proxies like ours that manipulates packets, need a protocol implementation running on the μEngines. There are open source TCP/IP stacks available (e.g. uIP [39]) that are written for general embedded systems, but to reduce the processing overhead to a minimum, we implemented our own for this proxy. This gave us great flexibility in manipulating TCP behaviour, including the retransmission and congestion control mechanisms.

The pipeline strategy chosen allowed almost all packets to stay on the fast path. The only packets sent to the XScale by the control path were the session setup packets, initializing a session state in shared memory. The pipeline strategy uses, allows all packet processing to be done without synchronization primitives, even though multiple cores work on the same data in shared memory. The receive and transmit core never access the session states, since they only understand raw network frames. The XScale control processor, is the only core creating and deleting sessions, and after the initial session setup, it does not access session state until deletion. After this, all packets of this session stay on the fast path, and the session is handeled by only one forwarding engine. The use of a lock-free pipeline furthers throughput, since no time is spent waiting for acquiring locks.

The protocol state in shared memory of the IXP allows easy addition of more forwarding engines working in parallel. Distributing the incoming packets to the forwarding engines can be done in a simple round-robin schemes (which would require synchronization), or by more sophisticated schemes like adaptive load sharing [40]. The latter scheme assigns connections to forwarding engines based on the load of each engine, in such a way that packets from a connection in most cases are processed on the same μEngine. This is an advantage, otherwise packets from the same session processed in parallel would have to wait for the μEngine to acquire a lock on the session state.

3.4 Summary

In this chapter we have looked at the implementation of a protocol translating proxy running on a network processor. The proxy allows smooth video streaming on unreliable network scenarios, and uses three μEngines and the XScale processor on an Intel IXP network processor card. The use of multiple cores in a pipelining scheme and the hardware-near architecture of the network processor allow low latency and high throughput.

Chapter 4

n-Queens Problem

In this chapter, we use the Cell Broadband Engine to generate solutions to the *n*-queens problem using a branch-and-bound algorithm. The chapter briefly looks at the necessary steps of using libspe2 to start SPE threads. We also investigate how fast the algorithm runs on the PPE compared to an SPE. The lack of dynamic branch prediction on SPEs should impact the throughput of this algorithm. The results are also compared with an implementation running on x86.

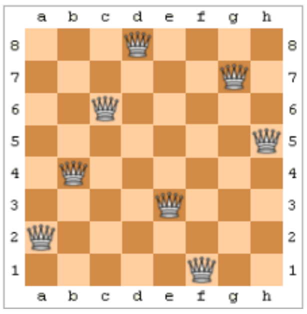


Figure 4.1: An example solution to the eight queens puzzle. Image from wikipedia [3]

This chapter is organized as follows; In section 4.1, we introduce the n-queens problem. Section 4.2 outlines the algorithm used, and we look at the details of running an SPE thread on the CBEA. In section 4.2.2, we present benchmarks of the implementation, and evaluates the results. In section 4.3, we look at the lessons learned.

4.1 Background Information

The n-queens problem is a generalization of the famous eight queens chess puzzle, where the player must place n hostile queens on an n x n chess board without any of the queens being able to take out another queen in one move. An example of a valid solution for an 8 x 8 board is shown in figure 4.1. The 8 queens problem has 92 distinct solutions, with most of them being symmetries, leaving a total of 12 unique solutions.

4.2 Adaptation for Cell

4.2.1 Implementation

Our implementation finds all unique solutions to the problem by using what is known as Wirth's algorithm [41]. The algorithm is of the class branch-and-bound, and permutates the board to all legal solutions. This is done by having three bit vectors representing horizontal, vertical and diagonal placement of queens. Combined, these three vectors represent the current state of the board, and placing a queen is a matter of setting the appropriate bits in the vectors representing the now illegal positions for a new queen. To check if a position is valid, a bitwise AND of the vectors gives a new vector of legal positions. To speed up execution, several constraints are introduced to avoid using time searching for known symmetries. The most obvious constraint is to only place the first queen in the upper half of the board, more constraints are honoured, but are not relevant for this overview.

The parallel approach is very simple; Each processing element gets a set of start positions in the first row, and is asked to find all unique solutions given placement of the first queen. This is not optimal, since it only scales to a number of cores limited by $\lceil n/2 \rceil$. It also means that the amount of work given to each core may be uneven, depending on the number of cores and the size of n. A better approach would be to split the problem space evenly by providing start positions for more queens.

Listings 4.1 to 4.5 show how an SPE job is dispatched on the PPE, and listing 4.6 displays the control set of operations running on the SPEs. The listings are from the *n*-queens implementation, but are modified to ease readability.

In listing 4.1 the data structures are set up. spu_params is shared to the SPEs, hence the 16 byte alignment. The binary of the compiled SPE code is loaded on line 7. The desired number of SPE contexts are created, and loaded binary is associated with the contexts. The running PPE thread is given a FIFO scheduling policy, meaning that they are not preemptively scheduled out until voluntarily yielded.

```
    pthread_t sputhread[num_threads];
    struct thread_arg arg[num_spus];
    struct queen_params spu_params[num_threads] __attribute__((aligned
       (16)));

    unsigned int          createflags = 0;

    spe_program_handle_t* program = spe_image_open("spu_queen");

    for (i=0; i<num_spus; ++i) {
        arg[i].queen_args = &spu_params[i];
        arg[i].spe = spe_context_create(createflags, 0);
        spe_program_load(arg[i].spe, program);
    }

    sched_setscheduler(getpid(), SCHED_FIFO, 0);
```

Listing 4.1: Parameter setup

The next step is job distribution in listing 4.2. The distribution is very coarse, as explained earlier, with the start positions spread among the threads.

```
    int remain = N_HALVED;
    int startpos = 0;

    for (i=0; i<num_threads; ++i) {
        spu_params[i].startpos = startpos;

        spu_params[i].num = remain / (num_threads - i);

        spu_params[i].N = N;
        remain -= spu_params[i].num;
        startpos += spu_params[i].num;
    }

    /* Get the rest */
    spu_params[num_threads - 1].num += remain;
```

Listing 4.2: Job distribution

In listing 4.3, the threads are given FIFO scheduling policy to avoid them being scheduled out when using more SPE contexts than the available SPEs. Instead, a blocked SPE context will wait until another SPE finishes before starting. Each SPE context needs a thread to run in, and we use regular pthreads for this. The thread runs as long as the SPE context exists, and the code to start it is listed in 4.4. After started, the threads will wait until they receive a signal to synchronize the work.

```
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    pthread_attr_setschedpolicy(&attr, SCHED_FIFO);

    for (i=0; i<num_threads; ++i) {

        ret = pthread_create(&sputhread[i], &attr, run_queen_spe, &
           arg[i]);

        if (ret) {
            perror("pthread_create");
            exit(1);
        }
    }

    /* Send start signal */
    for (i=0; i < num_spus; ++i)
        spe_signal_write(arg[i].spe, SPE_SIG_NOTIFY_REG_1, 2);
```

Listing 4.3: Thread creation and start signalling

```
void *run_queen_spe(void *thread_arg)
{
    int ret;
    struct thread_arg *arg = (struct thread_arg*) thread_arg;
    unsigned int entry = SPE_DEFAULT_ENTRY;
    spe_stop_info_t stop_info;

    ret = spe_context_run(arg->spe, &entry, 0, arg->queen_args, 0, &
        stop_info);

    if (ret < 0)
        perror("spe_context_run");

    return ret;
}
```

Listing 4.4: Starting an SPE

When a SPE finishes the work, a message is sent by a mailbox to the PPE. The PPE waits until it receives a message from all SPE contexts as listed in 4.5. This method of receiving mailbox messages is poll-based. Another approach is to create an event thread, and enable interrupts to be raised when a message is received.

```
    int work_left = num_spus;
    unsigned int val;

    while(work_left) {
        usleep(50000);
        for (i=0; i<num_spus; ++i) {
            if (spe_out_mbox_status(arg[i].spe)) {
                spe_out_mbox_read(arg[i].spe, &val, 1);
                fprintf(stderr, "Spe %d: Received mbox %d\n", i, val)
                    ;
                --work_left;
            }
        }
    }
```

Listing 4.5: Shutdown sequence

In listing 4.6, the SPEs start by waiting for a signal. When received, they DMA the parameters pointed to by the argp variable sent to main. All DMA operations are asynchronous, and are identified by a tag. spu_mfcstat on line 17 waits for the operations tagged to complete. When the parameters are received, the SPEs calculate the number of unique solutions given the start positions. When done, the result is DMAed back to main memory. After this operation is complete, the SPE use the mailbox to notify the PPE of its completion.

The SPE code is compiled using a version of gcc targeting the SPU instruction set, known as spu-gcc. The resulting binary can be embedded in the executable in a dedicated section, or as in this example, be loaded as a separate file at runtime.

```
1  int main(unsigned long long spe, unsigned long long argp, unsigned
       long long envp)
2  {
3      int tag = 1;
4
5      struct queen_params params;
6
7      do {
8          /* wait until start */
9      } while(!spu_readchcnt(SPU_RdSigNotify1));
10     spu_readch(SPU_RdSigNotify1);
11
12     /* Get meta */
13     spu_mfcdma64(&params, mfc_ea2h(argp), mfc_ea2l(argp),
14             sizeof(struct queen_params), tag, MFC_GET_CMD);
15
16     spu_writech(MFC_WrTagMask, 1 << tag);
17     spu_mfcstat(MFC_TAG_UPDATE_ALL);
18
19     unsigned int startpos = params.startpos;
20
21     N = params.N;
22     N_MIN_1 = N - 1;
23     N_HALVED = (N >> 1) + (N & 1);
24
25     unsigned int num_solutions = 0;
26
27     int i;
28     for (i=0; i<params.num; ++i) {
```

```
            num_solutions += partial_findqueen(startpos + i);
        }

        spu_mfcdma64(&num_solutions, mfc_ea2h(argp), mfc_ea2l(argp),
                4, tag, MFC_PUT_CMD);

        spu_writech(MFC_WrTagMask, 1 << tag);
        spu_mfcstat(MFC_TAG_UPDATE_ALL);

        unsigned int mb_value = 1;
        spu_writech(SPU_WrOutMbox, mb_value);

        return 0;
}
```

Listing 4.6: The main function on the SPEs

4.2.2 Evaluation

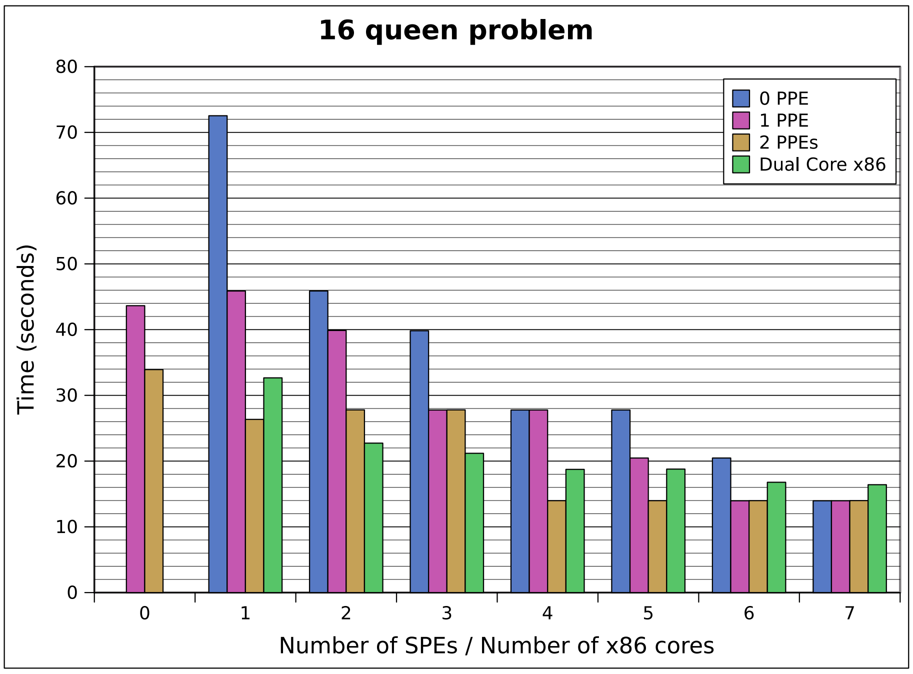


Figure 4.2: Benchmark of the n-queens problem using n=16. Lower is better.

The n-queens implementation was benchmarked on a PS3, and a dual core AMD Athlon X2 4200+ using various configurations of threads and SPEs. The results from

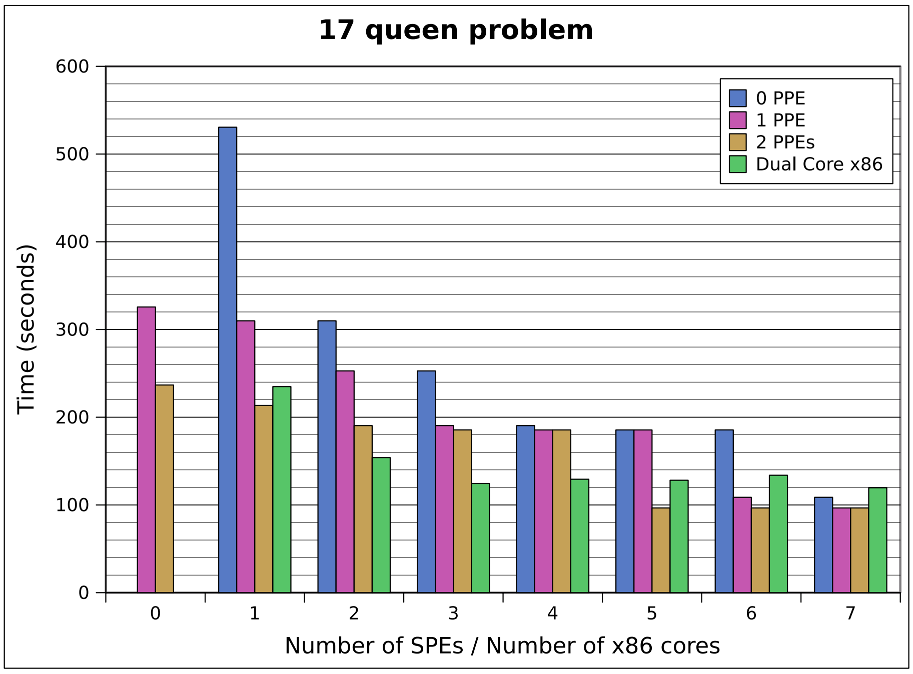


Figure 4.3: Benchmark of the n-queens problem using n=17. Lower is better.

n=16 and n=17 are shown in figure 4.2 and 4.3. The first, and most important observation from the graphs is the time spent solving the problem using a single SPE and a single PPE. For the 16-queens problem, calculating the solution using only a PPE thread take 43.6 seconds, while doing the same with a single SPE requires 72.5 seconds. Since our implementation does not use any dynamic job distribution, dispatch overhead is low. The large difference in performance is likely related to the lack of dynamic branch prediction, as this solution is based upon a branch-and-bound algorithm. On the x86, we benchmarked up to three threads, and compared that with the PS3 results. A single thread on x86 ran faster than on the PPE, and about twice as fast as an SPE thread.

The scalability on CBEA suffers a bit from the coarse split of work where each thread only gets a start position, and the number of rows to inspect. This leads to the jagged reduction of time with regard to the number of processing elements. In the case where the number of starting positions is non-dividable by the number of cores, one or more cores will get extra work compared to the others, and since we only split on the first start position, the other threads will idle until finished. This can be seen in the graphs by looking at the 2 PPE series, where in 4.2, we only take full advantage of 4 SPEs, and 5 SPEs in 4.3. For n=18 (not graphed), the number of starting positions is identical to n=17, and for n=19 all six SPEs should by utilized. As an effort to mitigate the effect of non-even distribution, we also tried over-booking the SPEs by using an 7th SPE. It is scheduled using Linux' SCHED_FIFO, meaning that it will only get execution time when one other SPE thread finishes. In 4.2 this provided the same execution time using only SPEs compared to dual processing on PPE and SPE. When increasing the number of starting positions (n=17, n=18), using only SPEs was not as fast as the combined approach. The scalability of the implementation is almost linear with regard to the number of SPEs used as long as there are enough starting positions available to evenly distribute among the processing elements.

4.3 Lessons learned

The algorithm used in this implementation required very little memory, with only three vectors to represent a state, and it fits into local storage with ease. The algorithm requires no synchronization or shared state, and is therefore well suited for architectures with many cores. With regards to SIMD usage for this problem, we have not considered if this is possible, and this is left as further work.

By evaluating the time spent for finding the number of solutions to the n-queens problem on SPE, PPE, and x86, we have an indication of the performance for branch heavy programs. For this type of application, an SPE provides a lower throughput than the PPE. However, the power of multiple SPEs is much higher than the PPE. We have also shown that even though this type of problem is not optimal for the CBEA, the combined throughput of the SPEs and PPE on the Playstation 3 are still faster than a dual core Opteron when running our implementation.

4.4 Summary

We have briefly looked at the n-queens problem, and outlined an branch-and-bound algorithm to solve it. The algorithm to calculate the number of unique solutions to the problem was implemented on the CBEA and x86. We have also benchmarked how an SPE thread is started, and how to distribute initial parameters to them. The implementation was then benchmarked on CBEA and x86. The n-queens problem is very simple, and the parallel strategy presented is not optimal, however, the benchmarks gives an indication of how the SPEs perform compared to the PPE and other platforms when branching is of importance.

Chapter 5

Advanced Encryption Standard

In this chapter, we parallelize the block cipher encryption algorithm AES. We split the work into job units, and focus on the job dispatching task of parallel programming. We benchmarked the scalability of throughput, when using different models for job dispatching.

Since the AES algorithm itself is not the subject of our investigation, we base our work on an existing single-threaded implementation of AES for x86, and do not explain details of the algorithm. To adapt the implementation for parallel processing, we designed a job dispatching framework suitable for PPE, SPE, and x86 usage. The scalability of job distribution is later evaluated.

The chapter is organized as follows; In section 5.1, we give an overview of the AES algorithm and modes of operation. Section 5.2 introduce the implementation we base our work on, and our adaptation of this is detailed in section 5.3. The lessons learned from this work can be found in section 5.4.

5.1 Background information

5.1.1 Cipher overview

The Advanced Encryption Standard (AES) [42] is a block cipher developed by the Belgian cryptographers Joan Daemen and Vincent Rijmen. It was submitted to the AES standardization process under the name Rijndael.

The standardization process started on January 2nd, 1997 when The National Institute of Standards and Technology (NIST) announced that a successor to the Data Encryption Standard (DES) was needed. This was because DES only had a small 56-bit key, which was becoming increasingly vulnerable to brute force attacks. 3DES solved the issue with small key-size by using three DES encryptions in sequence (with 2 or three different 56-bit keys), however DES was originally designed for hardware implementations, and not software. This made it unsuitable for software implementations on resource limited platforms, like embedded systems and mobile devices.

Similar to DES, AES was going to be an publicly disclosed encryption algorithm. In the nine months after the announcement, fifteen different designs were submitted. After the submission, all the candidates were analysed by cryptographers on the matter of security and performance. NIST held two conferences where the algorithms were discussed, and in august 1999, the fifteen algorithms were narrowed down to five finalists. A final round of reviews followed, and Rijndael was announced as the winner by NIST on October 2, 2000 and was standardized as AES in may 2002.

AES uses a fixed block size of 128-bit, and key sizes of 128, 192 and 256-bit. Because of the fixed block size, AES operates with a 4x4 array of bytes. Based on the size of the key, 10, 12, or 14 rounds with the basic steps below are performed:

1. Substitute Bytes
2. Shift Rows
3. Mix Columns
4. Add Round Key

The first step is KeyExpansion. This is a process also referred to as Rijndael key schedule, where the encryption key is expanded into a unique key for every round called a round key. In the *initial round*, only the *AddRoundKey* operation is applied. In the next rounds, the algorithm does the *SubBytes*, *ShiftRows*, *MixColums* and *AddRoundKey* operations. The *final Round* does not apply the *MixColums* operation.

Substitute Bytes

The first step in a round is the substitute bytes transformation step (Figure 5.1). Here, each byte in the array is updated using an 8-bit substitution box, also referred to as a Rijndael S-Box. This step is done to provide non-linearity in the cipher, and is important to prevent cryptographic attacks based on simple algebraic properties.

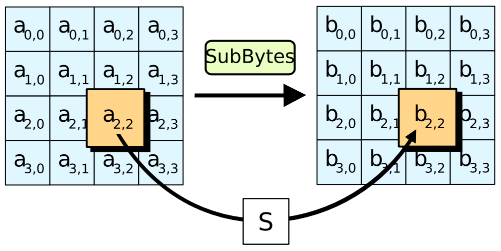


Figure 5.1: Substitute Bytes. Illustration from Wikipedia [4].

Shift Rows

The shift row transformation (Figure 5.2) called ShiftRow operates on the row of state (first row). Each byte on the second row is shifted one to the left. Similarly the third and forth rows are moved two and three places.

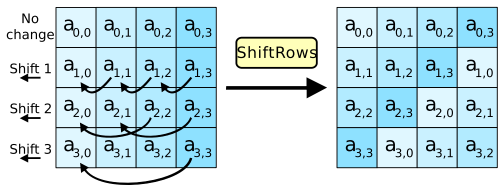


Figure 5.2: Shift Row. Illustration from Wikipedia [4].

Mix Column

The third step is mix column transformation (Figure 5.3). In this step, four bytes of each column of state are combined. The function takes a four byte input, and output four bytes. Each input byte affects all four output bytes.

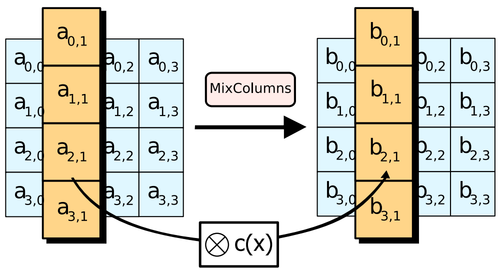


Figure 5.3: Mix Column. Illustration from Wikipedia [4].

Add Round Key

The Add Round Key step transforms the 128-bit state by XORing with the 128-bits round key. For each round a roundkey is derived from the main key using Rijndael's key schedule algorithm.

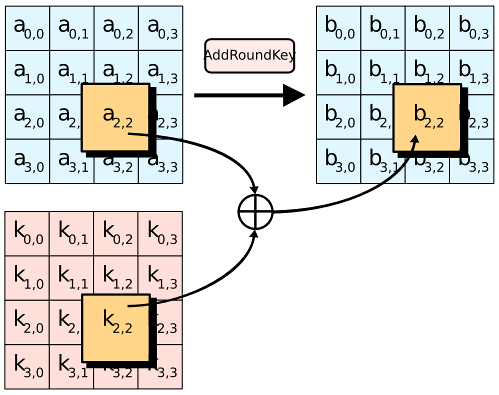


Figure 5.4: Add Round Key. Illustration from Wikipedia [4].

5.1.2 Modes of Operation

Mode of operation specifies how block ciphers supports enciphering data larger than one block. There are several standard modes available, and the most notable modes include electronic codebook (ECB), cipher block chaining (CBC) and counter (CTR). ECB is the simplest mode, using a single key to encrypt all blocks directly. As a result, it is easy to parallelize, since every block operation is done independently with the same key. The illustration in figure 5.5 shows why this mode is not suitable for real-world use; Identical blocks of input are encrypted to identical blocks of ciphertext. CBC solves this problem by XORing the block's plaintext with the ciphertext output from the last block. It also uses an initialization vector (IV) for the first block to be able to reuse the key without suffering from the same problem as ECB. The two main problems with this mode are single bit error in a block which renders all subsequent blocks undecryptable, and the process is strictly sequential i.e. not parallelizable.

The mode we use in our implementation is the counter mode. Counter combines the advantages of both ECB and CBC mode, by using a very different approach. Instead of enciphering the plain text directly, a counter is enciphered. This counter is initialized with a nounce, and is incremented for every block using a shared key. The encrypted counter is then XORed with the plain text producing the cipher text. This solves both the problems presented above, allowing the blocks to be encrypted independently of

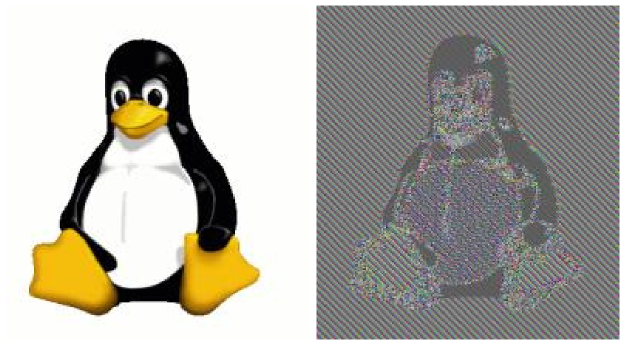


Figure 5.5: Block cipher encryption using ECB mode of operation. Tux to the left is the original, Tux to the right is encrypted. Illustration from Wikipedia [5].

each other, and errors in a block does not propagate to later blocks. Decrypting cipher text encrypted using counter mode is identical to the encryption process because of the involution property of the XOR operation, and the counter mode has provable bounded security no less than CBC mode [43]. Figure 5.6 illustrates the counter mode of operation.

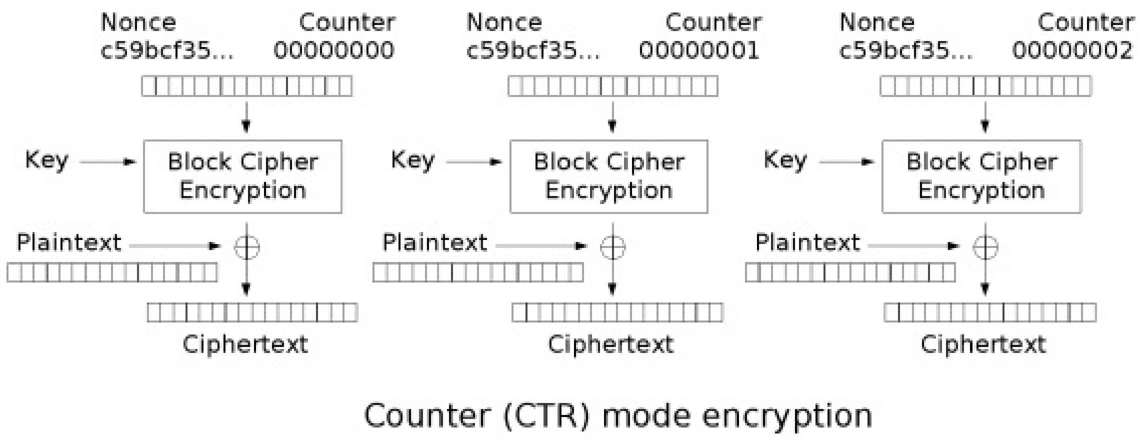


Figure 5.6: Counter mode of operation for block encryption. Illustration from Wikipedia [5].

5.2 Software basis

We found a fast, public domain implementation of AES made by Philip J. Erdelsky et al. [44], optimized for single core x86 usage. The implementation uses a lookup table scheme, combining the SubBytes, ShiftRows, and MixColumns steps. Each round requires 16 table lookups XORed 12 times, followed by an XOR operation with the round key. The lookup tables are hard coded, and every round is unrolled to minimize branching in the hot code path. The implementation we opted for does not exploit SIMD capabilities, nor is it easy to add, because of many load/store operations using large lookup-tables. Another approach to an AES implementation uses a bit-slicing techniques to benefit from SIMD architecture performance, and is discussed by Atri Rudra et al. [45]. Unfortunately, we did not find a suitable open source implementation using bit-slicing.

IBM has previously implemented AES encryption on an SPE using ECB, CBC, and EBC mode of operation, and the results are presented in [46]. Results for only a single SPE is presented, and for the ECB mode, they achieve 2.059 Gb/s. Their high performance on an SPE is likely obtained by using an implementation of AES that performs bit-slicing techniques to utilize the vector capabilities of the processing elements, but details of the implementation are scarce.

5.3 Adaptation for Cell

The implementation we based our work on does not support the counter mode of operation. This was implemented as part of our adaption of the code. We focused on CBEA and x86 architecture, and looked at what kind of speedup we could achieve by using multiple cores for encryption.

The AES encryption algorithm using counter mode fits well to the Cell architecture, considering the independent encrypting blocks, and the high locality of data. Because DMA operations on CBEA are expensive in terms of latency, and explicit in terms of programming, we had to shuffle the operations in a way that minimized DMA stalling. Since encrypting counter values is more time consuming than transferring the blocks, the DMA operations were hidden as background operations. As soon as a job is received, the SPE will start a DMA transfer of up to 16 kB of plain text to an unused buffer on the SPE. The SPE will then encrypt the nounce and the range of counter values specified in the job. The encryption of the counter values require more time than the background transfer, and after the encryption process is finished, the SPE makes sure the background DMA transfer has completed. The enciphered counter values are then XORed with the plain text data to produce the final encrypted content. The result is then transferred back to main memory using a backgrounded DMA transfer, and will continue while fetching the next job from the queue until all jobs are processed. The dispatch thread is notified by an interrupt, which avoids wasting resources polling for completed jobs.

5.3.1 Job Distribution

To expand the code for parallel processing, a job dispatching system was designed. A job includes a memory pointer to the plain text being encrypted, a destination pointer for the result, a start counter, and the number of blocks that should be encrypted in this job. To process the jobs, we implemented worker threads for x86, PPE, and SPE, and have looked at two models for distribution to worker threads. The first model uses a dispatch thread, which sends a job descriptor to every worker thread using the hardware channels on CBEA and waits for its completion before sending the next.

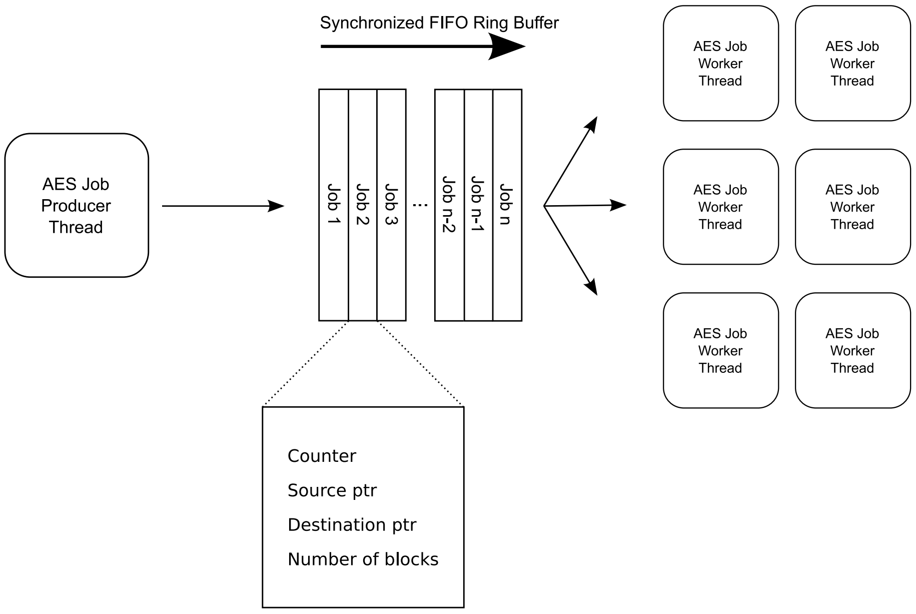


Figure 5.7: AES FIFO job distribution model

The second approach uses a shared FIFO job queue available for the worker threads to pull from. The FIFO is implemented as a ring buffer synchronized by a mutex, shared by the PPE and SPEs. The mutex is implemented as described in [19] by locking a cache line and DMAing the content after the cache line lock is acquired. Ideally, the FIFO should have a condition variable as well, but because of time constraints we did not implement the condition primitive. The FIFO can push and pop from both the PPE and SPEs, making it suitable for a range of applications.

Our Cell implementation of AES can scale to any number of threads. On our target machine, the Playstation 3, we use up to 6 SPEs, and 2 PPE threads. A job dispatching thread, and a single worker thread are running on the PPE core, while all available

SPEs are running worker threads. We also made a multi-core capable x86 version, reusing the FIFO for job dispatching using normal pthreads for synchronization. The worker threads running on the x86 architecture are similar to those performed on an equivalent PPE thread without the DMA operations.

5.3.2 Evaluation

We benchmarked the implementation on several architectures, and compared the encryption throughput. The test consisted of encrypting 4 GB of data and measure the elapsed time. This was done 5 times for each system, and the average values are presented in this section.

The benchmark results are collected from the following architectures:

- Intel Core 2 Duo E6750 2.66 GHz (64-bit)
- AMD Opteron 8220 Dual-Core 2.8 GHz (64-bit)
- STI Cell Broadband Engine 3.2 GHz (Sony Playstation 3 with 6 SPEs)
- Intel Pentium 4 3.0 GHz w/HyperThreading

Figure 5.8 graphs the performance of our AES implementation with regard to number of threads on CBEA using two different job distribution schemes. The 7th thread in the graph is a worker thread on the PPE. The graph displays a near linear throughput with regard to cores available, except for the case when including the PPE as a worker thread. In this case, the FIFO scheme scales to the last core, but the mailbox job distribution scheme is not able to utilize a worker thread running on the PPE. This happens because the PPE runs a worker thread while the SPEs are waiting for work. The job dispatch thread may be scheduled out when a job completes and, thereby is unavailable to dispatch a new job for idle SPEs. Because of time constraints, we did not measure the difference in dispatch overhead between the two job distribution schemes, and is left as further work.

In figure 5.9, we investigated the scalability of our AES implementation on different architectures. We tested it using a single thread, and an equal number of threads as there were cores available on the architecture. The Pentium 4 gains a small increase in speed when utilizing its SMT capabilities, by using two threads. The Opteron processor, and the Core 2 Duo approximately doubled their performance by running two threads. An interesting observation is that the Core 2 Duo performed better than the Opteron, even

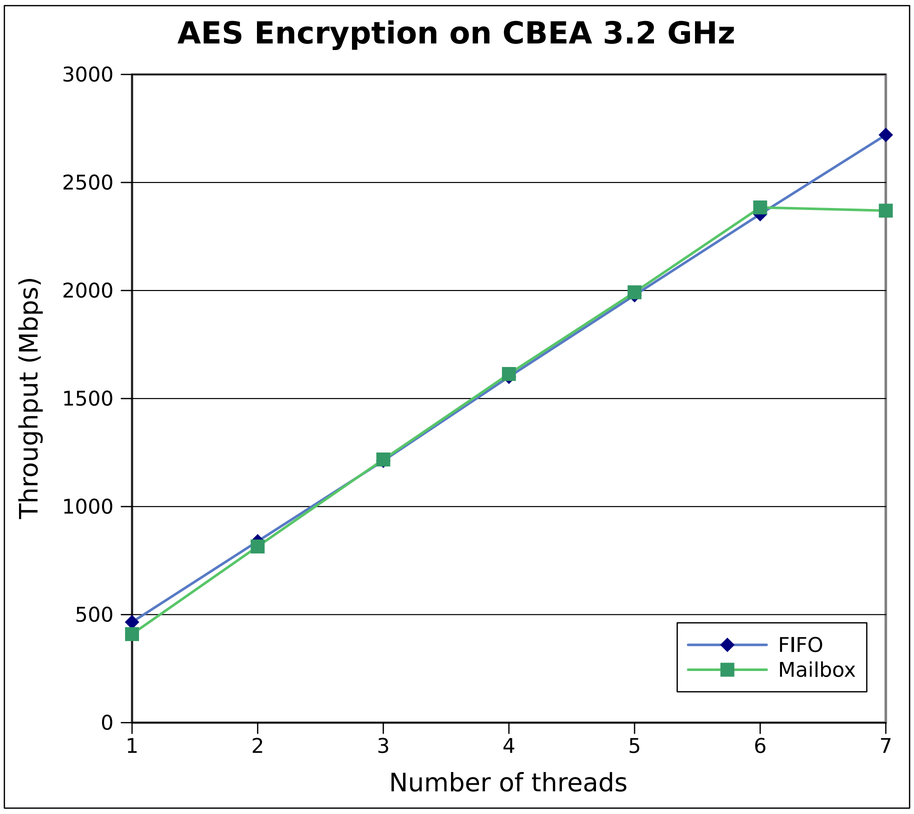


Figure 5.8: A benchmark of AES job distribution model on CBEA.

though it is clocked at a slightly lower speed. The best performer in the benchmark is the Cell processor, encrypting AES at 24.8% higher throughput than the Core 2 Duo.

5.4 Lessons learned

The AES algorithm works well on an architecture like the CBEA. The algorithm is easily parallelized with each thread working on an independent set of data using the counter mode of operation. Each SPE had to store approximately 10 KB of lookup tables, and every job request can be up to 1024 blocks large. Our background transfers and encryption required three buffers of 16 KB, which means that we only use a small amount of the available local storage. Since the SPEs work on blocks of continuous memory, the transfer overhead is very low.

Our proposed job scheduling model of using a shared FIFO with a dispatch thread and multiple worker threads, provided a linear scaling of throughput. Using mailboxes for job distribution gave a similar throughput, when not running a worker thread on the PPE. In this case, the PPE was not able to dispatch new jobs when the PPE worker

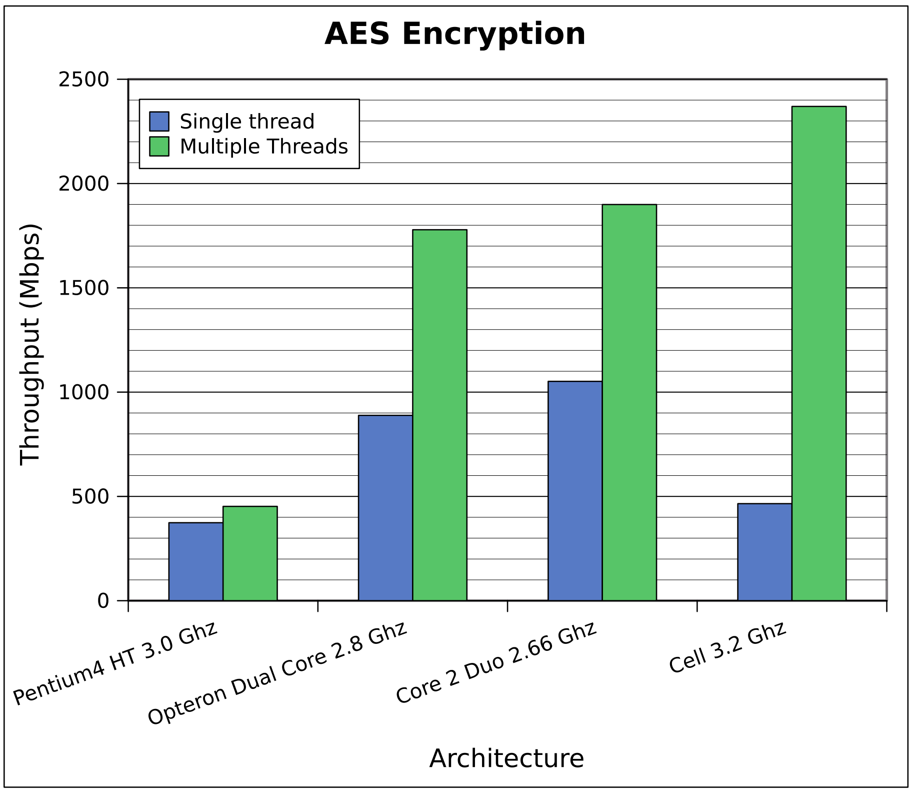


Figure 5.9: A benchmark of AES on different architectures.

thread was running, stalling SPEs, and ultimately resulting in reduced throughput. The results from this experiment indicate that when dealing with a busy PPE responsible for providing work for SPEs, work should be queued up to avoid starving the SPEs. Using mailboxes for distributing jobs seemed to work well when the PPE itself was not fully utilized.

5.5 Summary

In this chapter, we have given a brief overview of the Advanced Encryption Standard (AES), standardized by NIST in 2002. We have looked at common block cipher modes of operation, and their suitability for parallel processing. A fast single-threaded implementation was chosen and adopted for parallel processing by implementing the counter mode of operation and a job dispatching system. The performance of this dispatching system compared to using mailboxes was evaluated, and the FIFO system showed good scalability. The implementation was also benchmarked against other architectures.

Chapter 6

H.264/MPEG-4 Advanced Video Coding

In this chapter, we look at video encoding on the CBEA. Both IBM and Toshiba have done research in this area, with the results published as a whitepaper [47] and a keynote [30]. Both of these promise good performance on the architecture, but they are very scarce on details on how this was accomplished. In addition, no source code is available from the projects. If the above implementations are based on an existing encoder, or written from scratch for the CBEA is not conveyed.

In our investigation, we intend to base our work on an existing open-source encoder, and move time-consuming parts off to SPEs, without completely redesigning the encoder and thereby breaking the portability of it. We then evaluate the results, and find out whether our model for offloading computationally intensive parts of the encoder is sustainable.

The rest of this chapter is organized as follows; In section 6.1, we will provide a basic overview of the H.264 video codec. Section 6.2 introduces the x264, on which we have based our work. This includes a profiling of the code to determine critical areas. Section 6.3 contains details of our adaptation of the software for the target architecture, and the resulting benchmarks. Section 6.4 reflects on lessons learned.

6.1 Background information

The H.264/MPEG-4 Advanced Video Coding was standardized by ITU-T as H.264 [8] and by ISO as MPEG-4 Part 10 in 2003 [48]. It is a successor to earlier video standards like MPEG-2 and MPEG-4 Part 2, yielding improved image quality at lower bitrates; though at the expense of more demanding CPU computations. The H.264 codec is well

	Baseline	Extended	Main	High	High 10	High 4:2:2	High 4:4:4 P.
I- and P-Slices	Yes	Yes	Yes	Yes	Yes	Yes	Yes
B-Slices	No	Yes	Yes	Yes	Yes	Yes	Yes
Multiple Reference Frames	Yes	Yes	Yes	Yes	Yes	Yes	Yes
In-Loop Deblocking Filter	Yes	Yes	Yes	Yes	Yes	Yes	Yes
CAVLC Entropy Coding	Yes	Yes	Yes	Yes	Yes	Yes	Yes
CABAC Entropy Coding	No	No	Yes	Yes	Yes	Yes	Yes
Interlaced Coding	No	Yes	Yes	Yes	Yes	Yes	Yes
4:2:0 Chroma Format	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Monochrome Video Format	No	No	No	Yes	Yes	Yes	Yes
4:2:2 Chroma Format	No	No	No	No	No	Yes	Yes
4:4:4 Chroma Format	No	No	No	No	No	No	Yes
8 Bit Sample Depth	Yes	Yes	Yes	Yes	Yes	Yes	Yes
9 and 10 Bit Sample Depth	No	No	No	No	Yes	Yes	Yes
11 to 14 Bit Sample Depth	No	No	No	No	No	No	Yes

Table 6.1: An overview of the main differences of the profiles in H.264. The table was collected from Wikipedia's article on H.264 [49]

suited for High Definition (HD) content and is mandatory for Blu-Ray players and on the new terrestrial digital video broadcasting network (DVB-T) in Norway.

The H.264 standard describes the decoding process of compressed video, but does not detail how video encoding should be performed. This means that the only requirement for an encoder is that it produces a valid video bitstream, which is decodeable by a specification conforming decoder.

The standard supports a multitude of features. To limit the number of mandatory features in a decoder, several profiles are defined. Each of them has different mandatory features, ranging from the most simple baseline profile (most commonly used in mobile applications) to high 4:4:4 predictive profile, where high profile is very common, and is used in Blu-Ray and broadcasting. See table 6.1 for an overview of the main differences between the profiles.

Compared to MPEG-4 Part 2 Advanced Simple Profile (often referred to as MPEG-4), H.264 introduces new features to enhance quality. Most notably is the range of macroblock sizes, arbitrary frame order, well-defined integer DCT transformation, in-loop deblocking filter, and arithmetic entropy encoding. When correctly applied, the new features of H.264 can reduce the bitrate with approximately 50% with similar perceptual quality compared to prior standards [50].

This bitrate reduction is primarily achieved by reducing redundancy in the spatial and temporal domains. The spatial domain exists within a single frame, and is compressed by predicting samples from nearby areas, known as Intra prediction. The temporal domain is between frames, and temporal redundancy can be reduced by copying an identical or similar area from the same or different locations in an earlier encoded frame. The latter is called inter-prediction, and the position that the area is copied from is

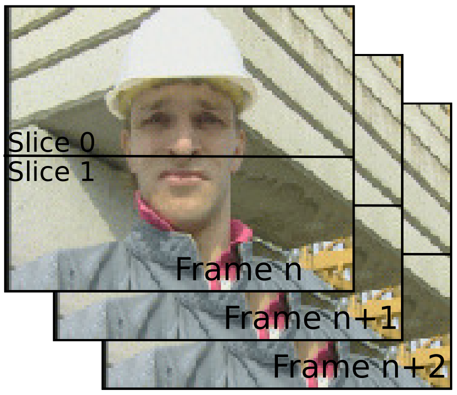


Figure 6.1: A video using two slices per frame. Each slice contains macroblocks, but are not shown.

known as a motion vector. The difference between the original frame and the predicted frame is stored as residual data, and good prediction reduces the entropy of the residuals, and thus the size of the encoded video.

6.1.1 Frames, Slices, and macroblocks

An input video frame to be encoded is split into smaller units according to the intended usage. A frame can be split into one or more slices, and a slice contains several macroblocks describing the frame. See figure 6.1 for an illustration of using two slices, and figure 6.2 for a figure showing a frame containing a single slice and its macroblocks. When a frame contains only a single slice, the terms frame and slice are used interchangeably.

The area covered by a macroblock range from 4x4 up to 16x16 pixels in size. The size of the macroblocks is chosen as a trade-off between storing multiple macroblocks, and the accuracy of a prediction of a large area, ultimately resulting in as few bits used in the output as possible. There are two main types of macroblocks:

Intra-macroblock An intra-macroblock may extrapolate or interpolate pixels from the macroblock(s) next to itself within the same slice.

Inter-macroblock Inter-macroblocks uses a motion vector to predict an area based on a previous frame, within the same slice.

Slice support is a feature included in H.264 to further increase error resistance and recovery at the cost of compression efficiency. It can also be used to aid parallel approaches for both the encoder and the decoder since predictions are limited within slices. This allows slices in the same frame to be processed independently and simul-

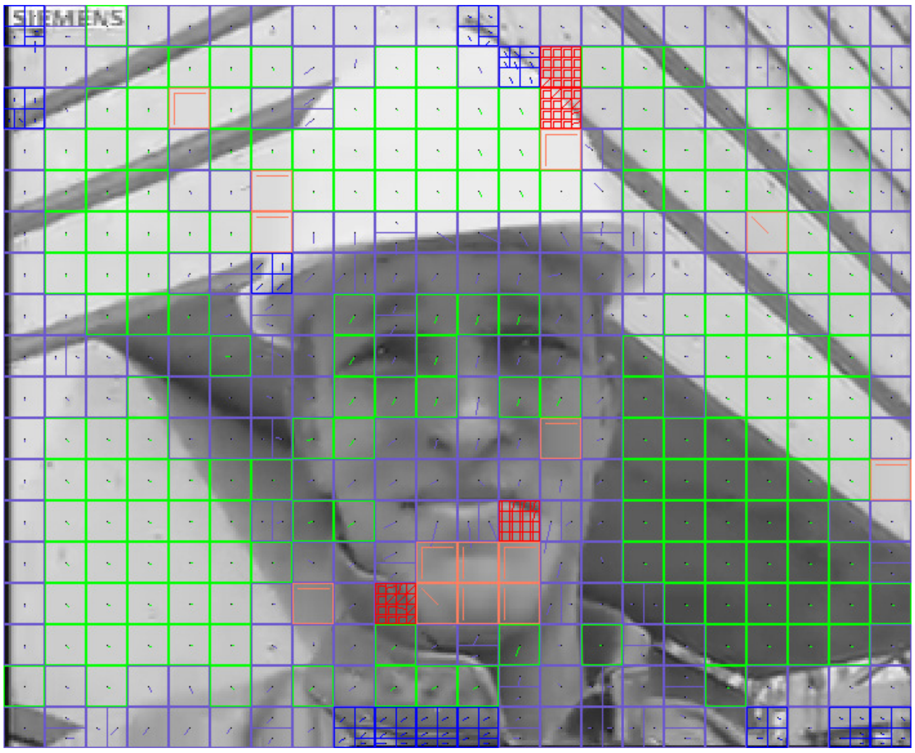


Figure 6.2: An image displaying macroblock types for the foreman test sequence using x264 and one slice per frame. Green macroblocks are skipped (unchanged from last frame), red and orange macroblocks are intra-predicted, while the blue macroblocks are inter-predicted depicted with motion vectors.

taneously. H.264 supports 3 main types of slices; I, B, and P. Slice types SI and SP are also available and can be used in scalable video, but are outside the scope of this overview. Based on the slice type, the encoder restricts itself to a set of prediction modes described below.

Intra (I) Slices can only contain intra-macroblocks and do not reference other slices. Intra-macroblocks predict by referencing near macroblocks in the same slice and interpolating the values in a direction. I-slices never depend on earlier encoded slices, and can always be used as a decoding starting point.

Predicted (P) Slices can contain intra-macroblocks and inter-macroblocks that predict the macroblock area by referencing an earlier encoded slice with a motion vector.

Bi-Predicted (B) Slices can contain intra-macroblocks and inter-macroblocks. The inter-macroblocks can be predicted from more than one slice (called bi-predicted) with an optional weighted average. In earlier video standards, B-slices could not be referenced by other slices, but this is supported in H.264.

A sequence of slices and their dependencies are shown in figure 6.3. The I-slices do not depend on earlier slices, and can be used as a decoding starting point. P- and B- slices depend on prior decoding of one or more slices that are used in the prediction of the slice.

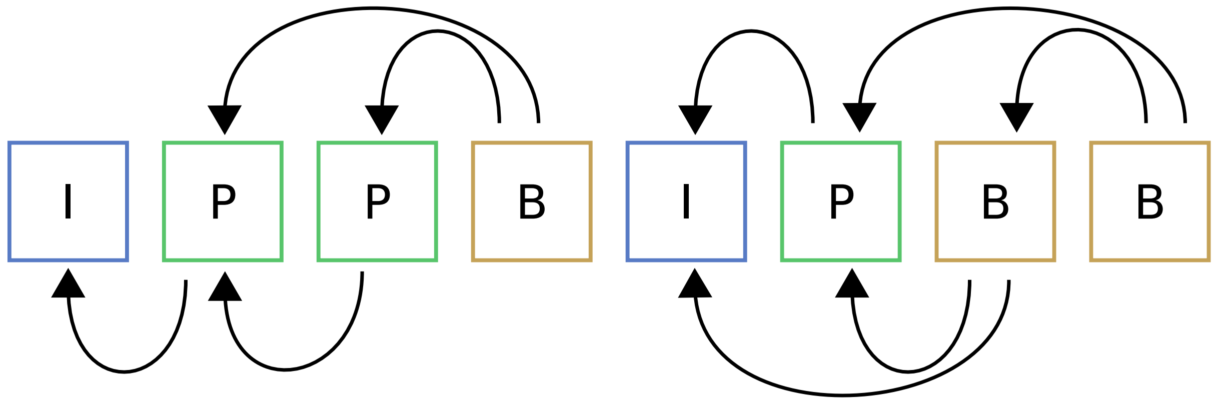


Figure 6.3: An example sequence of slices in display order and their dependencies to earlier ones.

6.1.2 Color space

The pixel colors in a macroblock are represented in the YCbCr colorspace, which is a transformation of the RGB values that will eventually be displayed on a screen. The Y channel is known as the luma channel, and is a grayscale representation of the target image. Cb and Cr are chroma channels, where the former channel represents blue, and the latter channel represents red. Together they amount to the color information of the image, and can be converted back and forth to RGB using a mapping known as a color profile. The main advantage of storing the luma separate from the chromas is that resolution of the chroma channels can be reduced with limited perceptual loss of quality compared to the luma channel. This is called chroma sub-sampling, and is illustrated in figure 6.4. Chroma sub-sampling factor 4:2:0 is very common, and is supported in all profiles of H.264. 4:2:0 means that for every 4 luma component Y, a single blue Cb and red Cr component are stored. Since each chroma component is shared between 4 luma components the bandwidth is effectively halved compared to 4:4:4 or no sub-sampling.

6.1.3 Encoding process overview

Encoding a video using the H.264 codec is a process performed in 3 main stages. The first stage is the analysis stage. Here, the encoder decides which modes and parameters

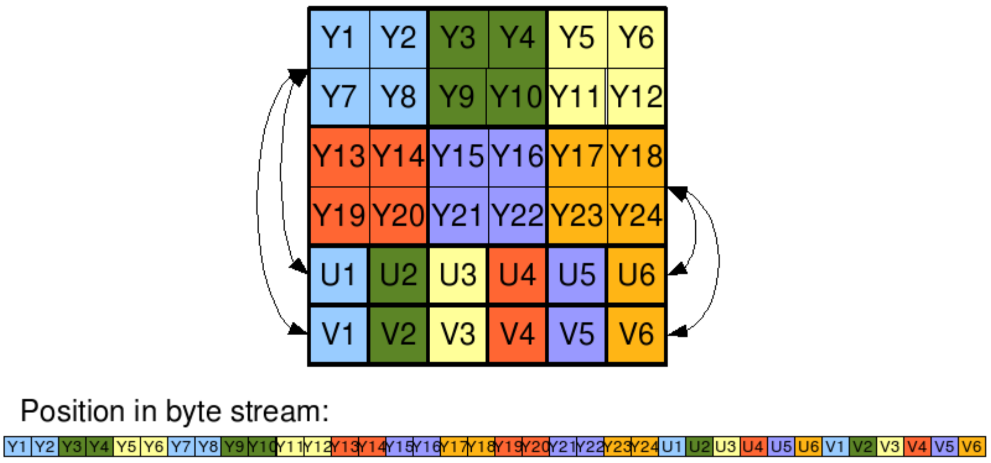


Figure 6.4: 4:2:0 YUV packed chroma subsampling. This is similar to H.264's 4:2:0 YCbCr with Cb and Cr represented as U and V. The figure is from Wikipedia's article on YUV [6] with modifications.

should be used in the encoding process. How these decisions are formed, and what is taken into account, is undefined and up to the encoder designers. A common goal is to produce the best possible result given the target bitrate, quality and encoding time constraint. The next stage is the encoding stage, where the parameters chosen during the analysis stage are used as input. The final stage is the entropy coder, where the redundancy in the bitstream output is reduced. The encoding process is presented in figure 6.5, and each stage is explained in detail below.

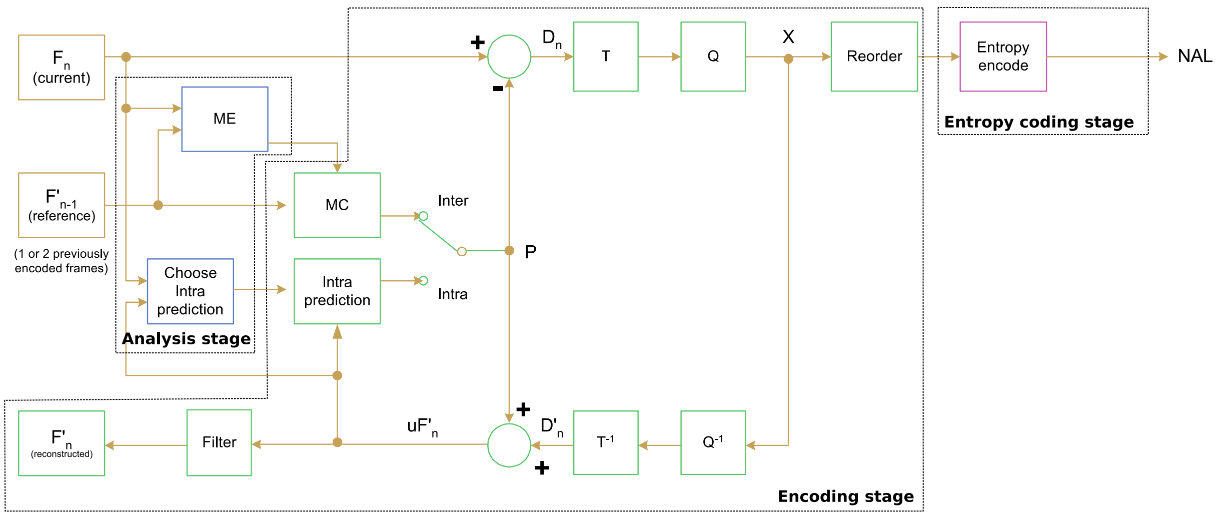


Figure 6.5: H.264 encoder overview, borrowed from [7] and slightly modified.

Analysis

F_n is the current (to-be) encoded frame. Predicting a macroblock in F_n is done in two main phases; First, the encoder analyses the frame using the motion estimator (ME block) and the "choose intra-prediction block", as seen in figure 6.5. The second phase is the actual encoding phase, which generates a prediction P from either the motion compensator (MC) or intra-prediction using parameters obtained in the analysis phase. P is later used to reconstruct the frame, explained in the section 6.1.3.

The prediction modes available for a macroblock are limited by the slice type, as explained in section 6.1.1. How the encoder chooses the type of slice to use is implementation specific and can depend on the intended player of the video. A common reason for choosing I-slice is to create starting points for decoding by limiting the number of frames between I-slices. In this case, the encoder will restrict itself to intra-prediction for F_n. Otherwise, when it decides to use a P- or B-slice, the encoder is free to choose inter- or intra-prediction per macroblock as it sees fit. When opting for intra- prediction, the encoder examines different inter-prediction modes to estimate the original frame as closely as possible. Inter-prediction works by searching for the most similar reference area in one or more earlier encoded frames (F'_{n-1}).

To put a metric on how similar two blocks are, and thus what prediction mode works best, is encoder specific. The evaluated samples of the blocks being compared is often restricted to the luma channel, but may also include chroma channels, depending on the encoder. A common way of comparing blocks is to compute a cost using a sum of absolute differences (SAD) or a sum of absolute transformed differences (SATD). SAD works exactly as named, it sums the absolute values of the differences between the two blocks being compared to create a cost. SATD work similarly, but sums the differences after a frequency transformation (e.g., Hadamard transform) and produces a better cost estimate. After producing a cost for all interesting prediction modes, the encoder can decide what works best.

Encoding

When prediction mode and related parameters are decided, the encoder completes the intra-prediction or motion compensation to obtain prediction P. The difference between P and the original frame F_n is residual block D_n.

D_n is then transformed to the frequency domain (T) using discrete cosine transform (DCT). By looking at frequencies instead of raw pixel data, it is possible to remove high frequency data from the residuals, reducing the entropy of the residuals. This is the lossy step in most image compression formats, and is called quantisation (Q). The amount of quantisation is decided by the quantisation parameter (QP), and is adjusted to match the desired output bitrate and quality. It is possible to skip the quantisation step, and produce lossless video.

The output from the quantisation, X, is then used to reconstruct the encoded frame by applying inverse quantisation (Q^{-1}) and inverse transformation (T^{-1}). This data is then filtered with a deblocking filter to smooth edges between macroblocks which reduces the artificial blocking created by the macroblocks. The output of this filter is a reconstructed frame (F'_n) identical to that produced by a compliant H.264 decoder using the final bitstream as input. It is worth noting that the encoder must use the reconstructed frame as input to prediction and not the original frame. Otherwise, since the decoder will see the reconstructed frame, the errors introduced by quantisation will propagate and distort predictions.

Entropy coding

The prediction parameters and the quantised residuals are compressed using one of two entropy coding methods supported in H.264. Context Adaptive Variable Length Coding (CAVLC), which uses Exp-Golomb codes and is supported in all profiles. Alternatively the encoder may use Context Adaptive Binary Arithmetic Coding (CABAC) which can be used in all profiles except baseline and extended. CABAC results in a better compression ratio, but at the cost of higher computational complexity of the arithmetic coding. Both compression methods are context adaptive, meaning that the probability tables used during coding are influenced by the context and earlier data.

The resulting bitstream of the entropy encoder is stored in network abstraction layer (NAL) units. NAL units contain a header and an integer number of bytes of payload. Several NAL units are defined to aid using H.264 as a stream format (e.g RTP), or including redundancy or scalability.

6.1.4 Decoding overview

The H.264 video decoding process is similar to the encoding process, and an overview is shown in figure 6.6. Reconstructing the frame F'_n is performed on individual macroblocks by predicting a macroblock's data and adding this to stored residuals. The residual data is read from the Network Abstraction Layer (NAL) and decompressed using either CABAC or CAVLC entropy decoding. The video frames are not necessarily stored in order of display, and may have to be re-ordered before reconstruction.

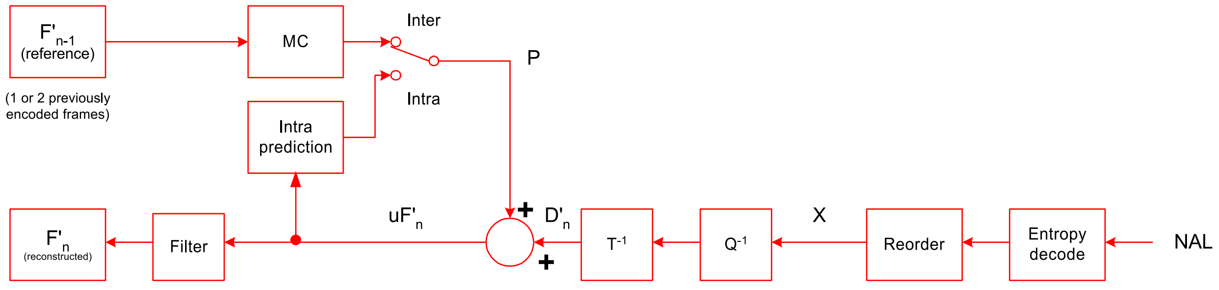


Figure 6.6: H.264 decoder overview, borrowed from [7]

.

The residuals are stored in the frequency domain using a transformation algorithm based on Discrete Cosine Transform (DCT), but with some adaptations to support integer operations and zero mismatch between the encoder and the decoder. To reconstruct the residuals, the decoder applies inverse quantisation (Q^{-1}) by multiplying each element in the residual block coefficients by a scaling factor. After this, D'_n can be computed by applying an inverse transformation function (T^{-1}).

Depending on the type of macroblock, the prediction mode can be either intra-prediction or inter-prediction (MC). Intra-prediction uses previously decoded macroblocks from the same frame to extrapolate pixels in one of 9 different directions for a 4x4 sized macroblock. Inter-prediction uses one or more previously decoded frames to compensate for moving objects. This is done in quarter pixel resolution by copying an area pointed to with a motion vector from the referenced frames.

When both the prediction step and residual step are completed, they are added together resulting in an almost completely reconstructed frame. The final step is the in-loop deblocking filter which is used to reduce blocking distortion produced by subdividing the frame into macroblocks. When all macroblocks in a frame are decoded, the frame is fully reconstructed and ready for display. Depending on further references, it may still be used by the decoder to predict later frames.

6.2 Software basis

6.2.1 The x264 video encoder

x264 [51] is an open source high profile H.264 encoder licensed under the GPL. It is widely used, and ranked as the second best H.264 encoder available, in terms of quality and speed, in the fourth annual H.264 video codec comparison by MSU [52]. x264 supports a number of platforms and architectures including PowerPC. It can run on the PPU of the Cell/B.E, but does not exploit the Cell's computational capabilities, since the x264 code is only optimized for a general PowerPC, it does not take advantage of the available SPEs.

The x264 code is parallelized using pthreads, and scales well on SMP systems. According to a benchmark included with the source code, the current x264 implementation achieves up to 3.745 scaling with a four CPU x86 machine using eight threads. x264's parallelization scheme previously used multiple slices per frame. It was abandoned because of bitrate penalty of using multiple slices, since predicted macroblocks can not reference other macroblocks across slice boundaries. In the new version, a better scheme was devised, spawning a new encoding thread for every frame up to a maximum number of threads. Disadvantages of this model are a requirement for early decision of frame type, and limited range for motion vectors to only reference macroblocks from the encoded (upper) part of reference frames. Another issue with this model is rate control, which has to make decisions for the current frame before earlier frames finish.

6.2.2 Profiling x264

To determine the most CPU intensive parts of x264, we profiled an encoding session using the valgrind tool callgrind [53]. The tool dumps cycle estimations in each function at regular intervals, and when analysed using Kcachegrind [54], callgraphs can be produced. The profiling session was executed on x86 because of limited support for PowerPC in callgrind, and the amount of work done in the different parts of the encoding process is assumed similar running on PowerPC. Encoding parameters for this profiling are subpixel refinement level 5, all macroblock sizes are analysed, b-frames allowed, and hexagonal motion estimation. Figure 6.7, shows absolute cycle estimations for the major blocks of the encoder during a periodic dump. The graph shows that most of the cycles are spent in *x264_slice_write* and its children.

The rank below *x264_slice_write* outlines the 3 major components of the encoding process, namely analysing (*x264_macroblock_analyse*), encoding (*x264_macroblock_encode*), and entropy coding (*x264_macroblock_write_cabac*). The filter functions (*x264_fdec_filter_row*) include deblocking filter (2.4% of cycles), hpel filter (2.2% of cycles), and a border for fast prediction calculation (0.2% of cycles).

The analysis stage is clearly the most time consuming part of the encoder with about 63% of the total cycles spent in the profiling. Figure 6.8 shows a callgraph of major functions used during the analysis stage. The *x264_macroblock_analyse* function is called per macroblock and decides a prediction mode by iterating through the different modes and assigns a cost to each mode. Intra-prediction is fairly fast with only 6.3% of the cycles spent analysing the macroblock. About 54% of the cycles during analysis are spent in inter-prediction functions. The inter-prediction functions spend about 70 % of their time in motion estimation searching (*x264_me_search_ref*) and its subroutine subpixel refinement (*refine_subpel*).

6.3 Adaptation for Cell

Based on the profiling results, two parts are evaluated as candidates for offloading to SPEs; The subpixel filter (*x264_frame_filter*) and motion estimation searching (*x264_me_search_ref* with its subroutine *refine_subpel*). The former does not consume many cpu cycles, but has a clear advantage over other potential functions - it works on rows of macroblocks instead of single ones. Motion estimation is selected because it uses a large number of cycles.

Offloading the mentioned functions to a SPE does not speed up encoding by itself - the PPE must be able to do other work while waiting for the job to complete. The idea is to spawn multiple encoding threads using x264's existing parallelization, with each thread dispatching jobs to a FIFO for subpixel filter and another FIFO for motion estimation searching. By doing this, we free up resources for the PPE to work on other functions while the jobs are performed.

6.3.1 Job distribution

The FIFO used is the same synchronized ring buffer developed for the AES job dispatcher. It was benchmarked to be less PPE-intensive than using mailboxes for job distribution, and adds the capability of having multiple SPEs pulling jobs from the

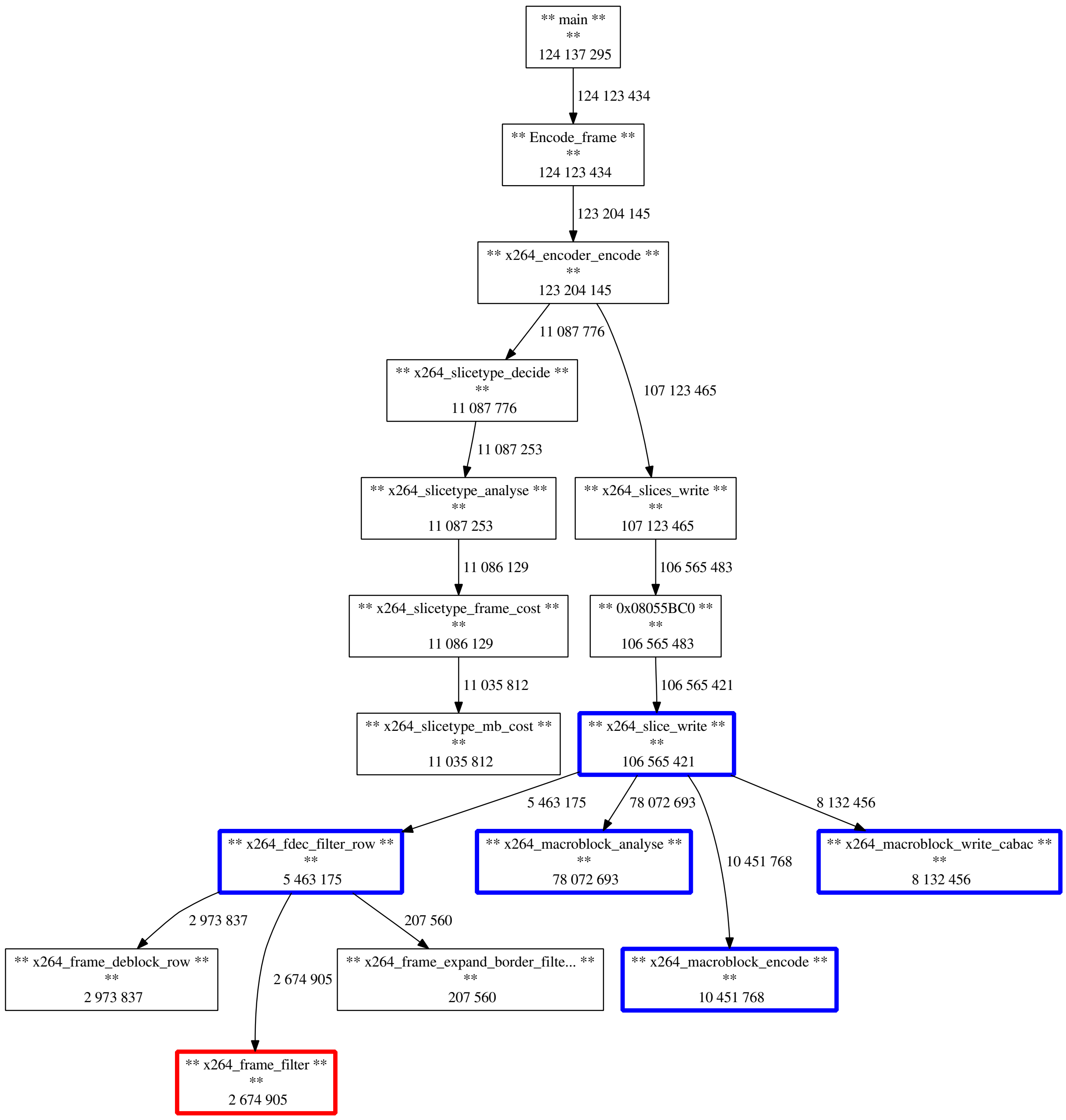

** main ** ** 124 137 295
124 123 434
** Encode_frame ** ** 124 123 434
123 204 145
** x264_encoder_encode ** ** 123 204 145
11 087 776
107 123 465
** x264_slicetype_decide ** ** 11 087 776
11 087 253
** x264_slicetype_analyse ** ** 11 087 253
** x264_slices_write ** ** 107 123 465
11 086 129
106 565 483
** x264_slicetype_frame_cost ** ** 11 086 129
** 0x08055BC0 ** ** 106 565 483
11 035 812
106 565 421
** x264_slicetype_mb_cost ** ** 11 035 812
** x264_slice_write ** ** 106 565 421
5 463 175
78 072 693
8 132 456
10 451 768
** x264_fdec_filter_row ** ** 5 463 175
** x264_macroblock_analyse ** ** 78 072 693
** x264_macroblock_write_cabac ** ** 8 132 456
2 973 837
207 560
2 674 905
** x264_frame_deblock_row ** ** 2 973 837
** x264_frame_expand_border_filte... ** ** 207 560
** x264_macroblock_encode ** ** 10 451 768
** x264_frame_filter ** ** 2 674 905

Figure 6.7: Profiling graph of x264 main functions during encoding. Numbers are absolute cycle estimations.

same FIFO. Job dispatching is done by creating a job descriptor that includes parameters and memory addresses required to process a job. It also contains a pointer to a condition for when the job completes, which will be signalled to resume to calling thread. The dispatcher locks a job mutex, pushes the job descriptor to the FIFO, and waits for the complete condition to be signaled. The job mutex is necessary because of an unlikely race condition when the job completes before the thread starts waiting for the condition variable.

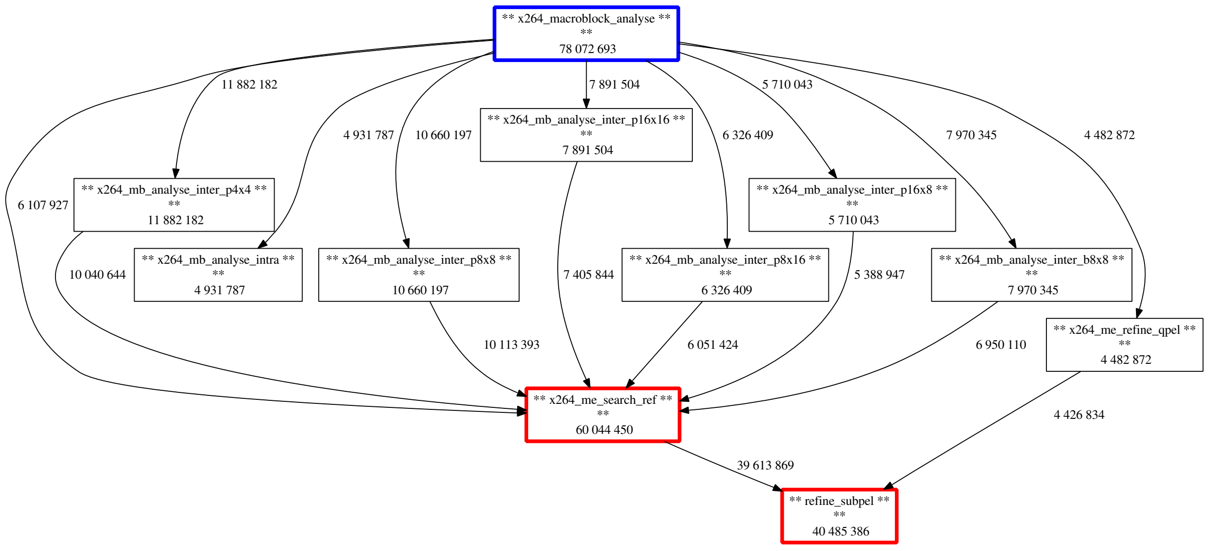


Figure 6.8: Profiling graph of x264 analyse stage during encoding. Numbers are absolute cycle estimations.

When a job finishes, the SPE will send an message to the PPE, which is catched by an event thread running on the PPE. This message contains the memory address of the job complete condition, which is signaled after locking the shared job mutex. The calling thread that is waiting for the condition, and the job to complete, can now resume execution.

6.3.2 Half-pixel filtering

Half-pixel filtering (HPEL filter) is used as an intermediate step to speed up calculating quarter-pixel values for luma motion estimation and compensation. The filter runs once for every row of macroblocks and produces half-pels (hpels) which are halfway between integer samples and is a weighted sum of six values called a 6-tap filter. Three such arrays of interpolated hpels are produced; a horizontal, a vertical, and an interpolation of six hpel samples. The latter can be interpolated from either horizontal or vertical hpel samples previously calculated. By using hpels at the later stages, quarter-pixels can be found by simply averaging two samples. The subpels are illustrated in figure 6.9, where the gray boxes are integer samples from the luma plane, and the rest are either half pixels or quarter-pixels. Since hpels are halfway between integer samples, we want to calculate b, h, m, s and j. b is interpolated using the horizontal filter from E, F, G, H, I, and J, and similarly h is interpolated using the vertical filter from A, C, G, M, R, and T. j is interpolated from either the previously calculated horizontal or vertical hpels, e.g., cc, dd, h, m, ee, and ff.

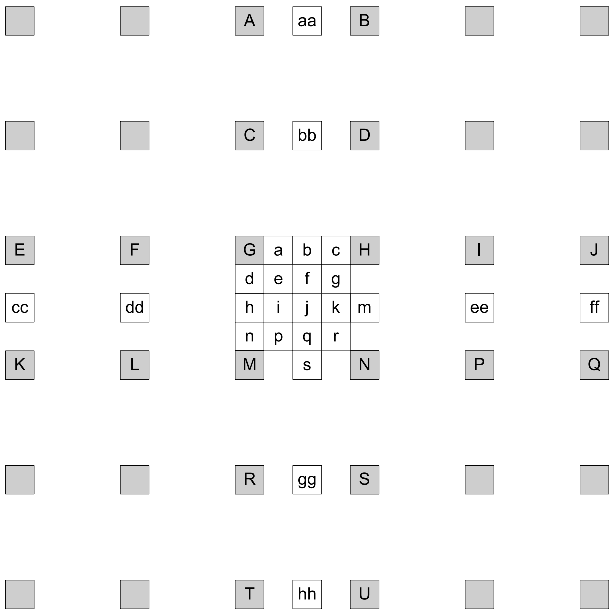


Figure 6.9: Half-pixels (b,h,m,s,j) are calculated by interpolation using a 6-tap filter. The illustration also shows integer-pixels (upper case), and quarter-pixels. Figure from [8].

A job descriptor structure for a job related to motion compensation is found in listing 6.1. For a hpel filter job, the strides, width, height, and src[0] represent the luma plane. The dst array points to the 3 hpel destination planes. The rest of the job descriptor is unused for hpel filter jobs.

```
typedef struct mc_job
{
    enum {
        HPEL_FILTER = 1,
        MC_LUMA,
    } type;

    uint8_t pad[3];
    ea_t done; /* Pointer to job done condition */

    int32_t i_src_stride, i_dst_stride;
    int32_t i_width;
    int32_t i_height;

    int32_t mvx, mvy;

    ea_t src[4];
    ea_t dst[4];

} __attribute((aligned(16))) mc_job_t;
```

Listing 6.1: hpel filter job descriptor

```
planeaddr.e = job.src[0].e − 8 − 2*job.i_src_stride;
planesize = (5 + job.i_height) * job.i_src_stride + 16;
```

Listing 6.2: Luma plane DMA alignment

The hpel filter is adapted for SPE usage from x264's general purpose C implementation. When a job descriptor is received by an SPE, an area of the luma plane is copied to local storage for filtering. The plane is misaligned by 8 bytes, and the filter requires 2 rows above, and 3 rows below the macroblock for the 6-tap filter shown in listing 6.2. The plane as well as the computed hpels must fit in local storage, and the width of the plane is limited to 1920 (padded to 2048 for boundary and alignment) resulting in up to 144 KB of used local storage for data. When the filtering is finished, the SPE issue an DMA request to copy the resulting hpels to the given memory locations in dst[].

6.3.3 Motion Estimation

Motion estimation is the most time-consuming stage of the encoding process as demonstrated by the profiling session. It is performed multiple times for every macroblock, exploring different sizes, reference frames, and ranges. As a result of the parameters, it returns a motion vector with an associated cost, which depicts the lowest cost found in the search area given the current constraints. It is called from the *x264_mb_analyse_inter_*x** set of functions, for every size of macroblock tried, and for every possible reference frame. Within a macroblock, for different combinations of smaller blocks, the motion estimation must be run sequentially because of prediction from the other small blocks. This does not exclude performing cost calculation on different block sizes or reference frames in parallel, as long as they do not depend (be predicted) on each other. Performing parallel searches within the same macroblock was not examined in this thesis due to time constraints, instead we focused on offloading the search and refining functions, and dispatched search jobs from the analysing functions.

```
typedef struct me_search_job
{
    ea_t done; /* Pointer to job done condition */

    int32_t bw, bh;
    int32_t i_pixel;
    int32_t i_me_range;

    int32_t mv_x_min, mv_x_max;
    int32_t mv_y_min, mv_y_max;

    int mvp[2];
    int32_t mvc[4][2];
    int32_t i_mvc;

    int32_t i_halfpel_thresh;
    int32_t i_subpel_refine;

    ea_t p_cost_mv;

    ea_t p_fref[6];
    int i_stride[2];

    ea_t p_fenc[3];
    ea_t p_fenc_buf;

    int32_t mv_min_spel[2];
    int32_t mv_max_spel[2];

    int32_t b_chroma_me;

    ea_t result;

    /* Pointer to x264_t and x264_me_t used by PPE version on ME */
    ea_t h, me;

} __attribute((aligned(16))) me_search_job_t;
```

Listing 6.3: Motion Estimation job description

The motion estimation job descriptor is listed in 6.3 and includes all parameters necessary for performing the search. *bw* and *bh* are the width and height of the macroblock, and *i_pixel* is a numeric representation of the same. *i_me_range* is the desired search range, and $mv\_\{x, y\}\_\{min, max\}$ is the valid range of the resulting motion vector with integer precision limited by the boundaries of the plane. *mvp* is the predicted motion vector based on earlier vectors stored in a cache, and *mvc* is up to 4 earlier motion vectors from nearby macroblocks, which may aid the search in the right direction. *i_halfpel_thresh* is a threshold for early termination, and *i_subpel_refine* is a value from 1 to 7 which specifies the level of refinement the motion vector search algorithm should perform. *p_cost_mv* is a lookup table of costs associated with a motion vector given a quantisation parameter. *p_fref* is the planes of the referenced plane we are searching in, being the luma plane, 3 hpel planes for luma, and 2 chroma planes. The *i_stride* is the stride for referenced luma and chroma planes. *p_fenc* has pointers to the luma and chroma planes of our current macroblock, and *p_fenc_buf* is an aligned pointer to the whole current luma and 16x16 block which may be split into several smaller macroblocks. The ranges $mv\_\{min, max\}\_spel$ are motion vector range limitations when using subpixel estimation. *b_chroma_me* decides if we are doing refinements in the chroma plane, and finally result points to where the results are stored. The result structure is listed in 6.4 and is DMAed back containing the best motion vector found and its associated cost.

```c
typedef struct me_search_result
{
    int cost_mv;        /* lambda * nbits for the chosen mv */
    int cost;           /* satd + lambda * nbits */
    int mv[2];
    int i_halfpel_thresh;
} __attribute((aligned(16))) me_search_result_t;
```

Listing 6.4: Motion Estimation search result

The SPE starts a job by copying all planes of the current macroblock (*p_fenc*) to local storage. It maintains a cache of the *p_cost_mv* lookup table and will copy the 65538 bytes large table if the QP has changed, or if the cache is empty.

When searching, the SPE uses a cache which is a rectangle subset of the referenced frame, and is polled for every comparison to check if it is within range. The cache supports up to 64 pixels of ME search range in each direction. The macroblock may be up to 16x16 pixels in size, and the source address may be misaligned, thus $(2 * 64 + 32) * (2 * 64 + 16) = 23040$ bytes are necessary for the luma and hpel planes. Each chroma plane requires only $(2 * 32 + 24) * (2 * 32 + 8) = 6336$ bytes because of chroma subsampling. A total of up to 104832 bytes may be used as cache for the frame. Updating the referenced frame cache is done by a DMA list to obtain scatter/gather functionality. The SPE creates a list of aligned addresses of rows in the requested planes limited by the boundaries set in the job. When transferred, the rectangle is stored in a single buffer for each plane.

Searching for the best motion vector is mostly copied from the x264 source code implemented in C, and adapted for use with an SPE. Depending on the refine subpel value, integer or subpel estimation is performed using SAD in a hexagon shaped search pattern. When an area of interest is determined by a preliminary vector, and the subpel refine value is not 1, the motion vector is refined using the more time-consuming comparison function SATD and by also evaluating chroma planes. When completed, the results are stored in the me_search_result structure and DMAed back to the PPE.

6.3.4 Evaluation

We benchmarked the offloading of the hpel filter by varying the amount of subpel refinement and the number of threads used during encoding. The encoder is instructed to analyse all macroblock sizes, use hexagonal motion estimation, and has a target bitrate of 100 kB/s. The video used in this benchmark is the 300 frame CIPR foreman sequence [55]. All tests are run 5 times with the average value used in the graphs. The built-in average fps counter in x264 provides the data points. The HPEL filter runs on a single SPE as it is assumed that not enough work is produced from the threads to utilize more based on the profiling data.

Figure 6.10, 6.11, and 6.12 are the results from the hpel benchmark. Using only one thread, the PPE is faster in all tests. The reason for this is two-fold - There is an overhead penalty for dispatching a job to an SPE, DMAing the luma rows, and the hpels back to main memory. The other reason is that the PPE is generally faster than a single SPE if the code is run sequential (e.g. branch prediction).

By increasing the number of threads, both the PPE and the SPE gets a performance boost. The PPE's speed-up is due to its SMT capability, which permits two simultaneous threads to run on a single core. The time spent filtering hpels is now available to other threads on the PPE. Increasing the number of threads means more background work for the PPE to perform, while the hpels are filtered on the SPE. When increasing the level of subpixel refinement the number of cycles spent on filtering become less significant.

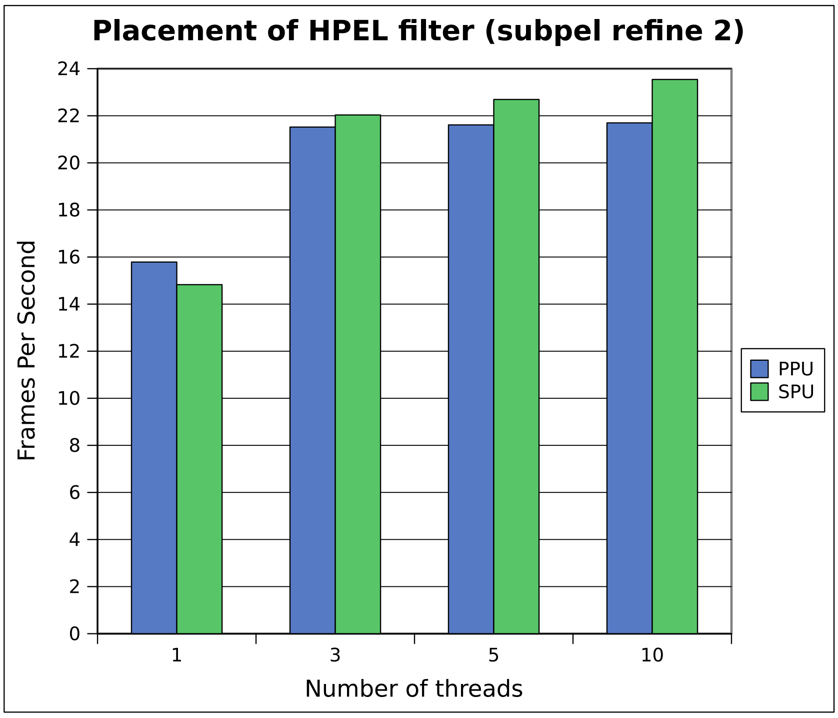


Figure 6.10: HPEL filter offloading to SPE using subpixel refine parameter 2.

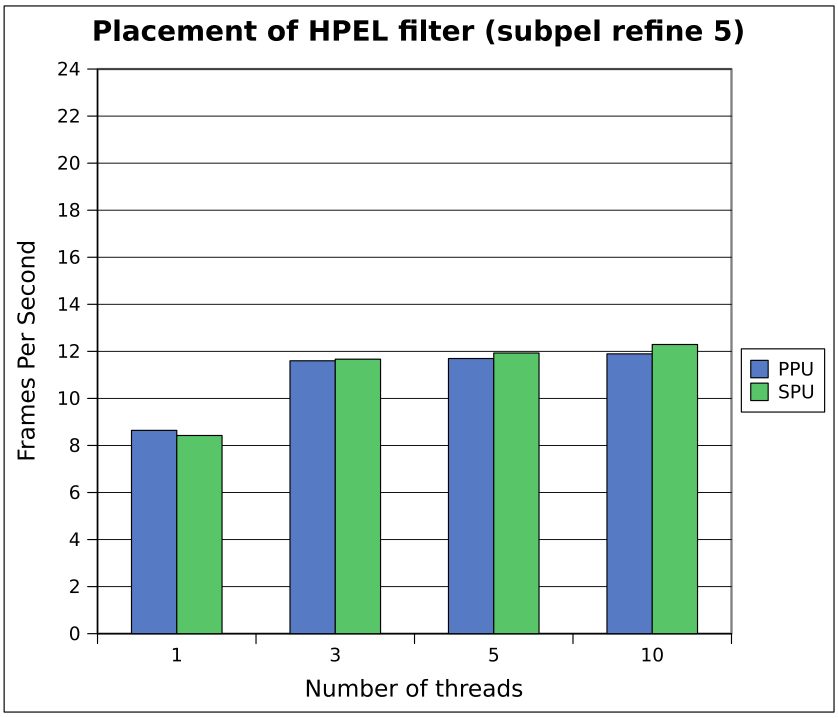


Figure 6.11: HPEL filter offloading to SPE using subpixel refine parameter 5.

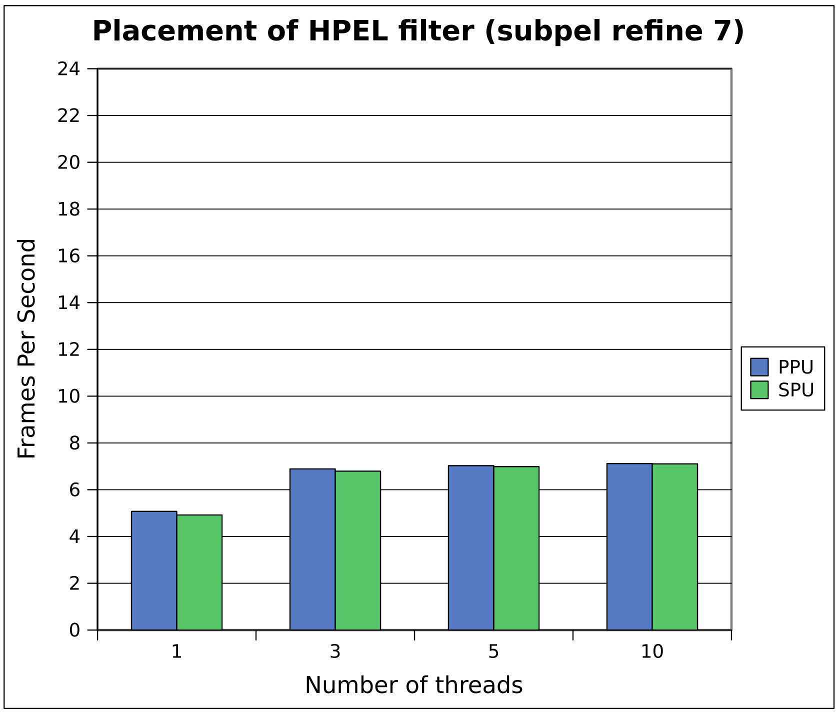


Figure 6.12: HPEL filter offloading to SPE using subpixel refine parameter 7.

Offloading the motion estimation function was benchmarked in a similar manner by varying the amount of subpel refinement, and number of threads during encoding. We also tried using two SPEs dequeuing from the job FIFO, since the motion estimation accounts for so many cycles it may benefit from having cpu time available. All data points are 5 run averages, and the graphs are shown in figure 6.13, 6.14, and 6.15.

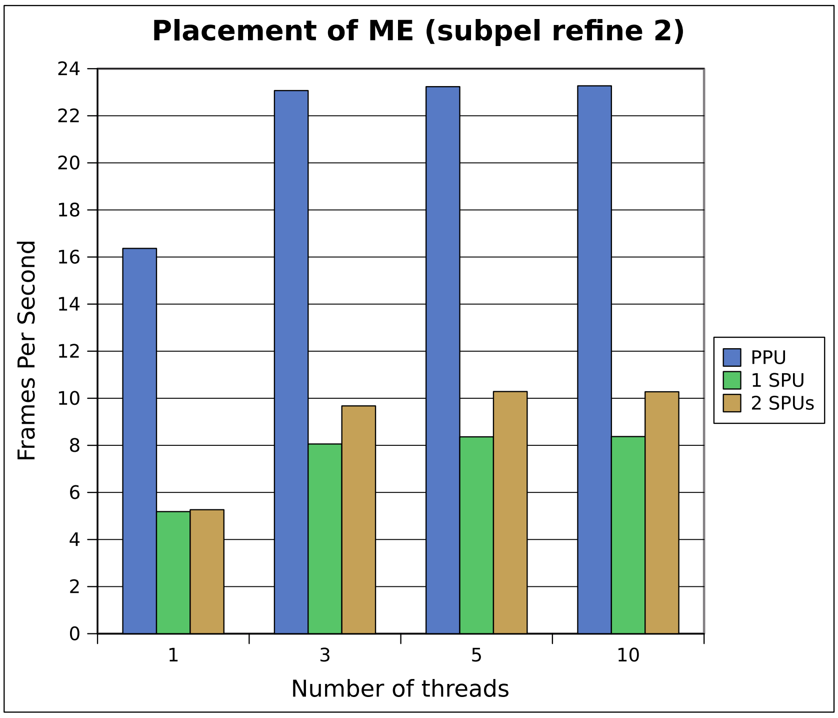


Figure 6.13: Motion estimation function offloaded using subpixel refine parameter 2.

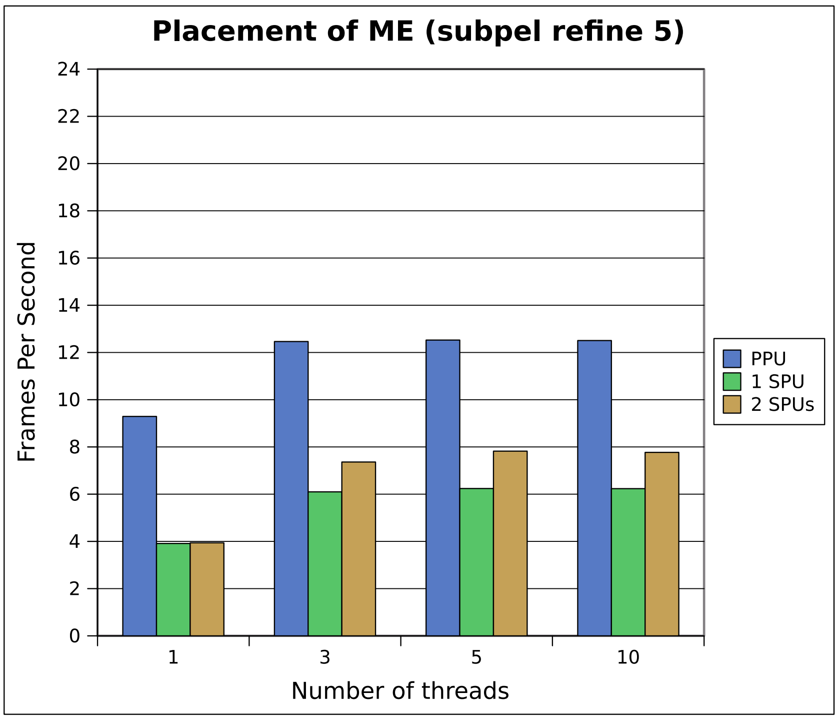


Figure 6.14: Motion estimation function offloaded using subpixel refine parameter 5.

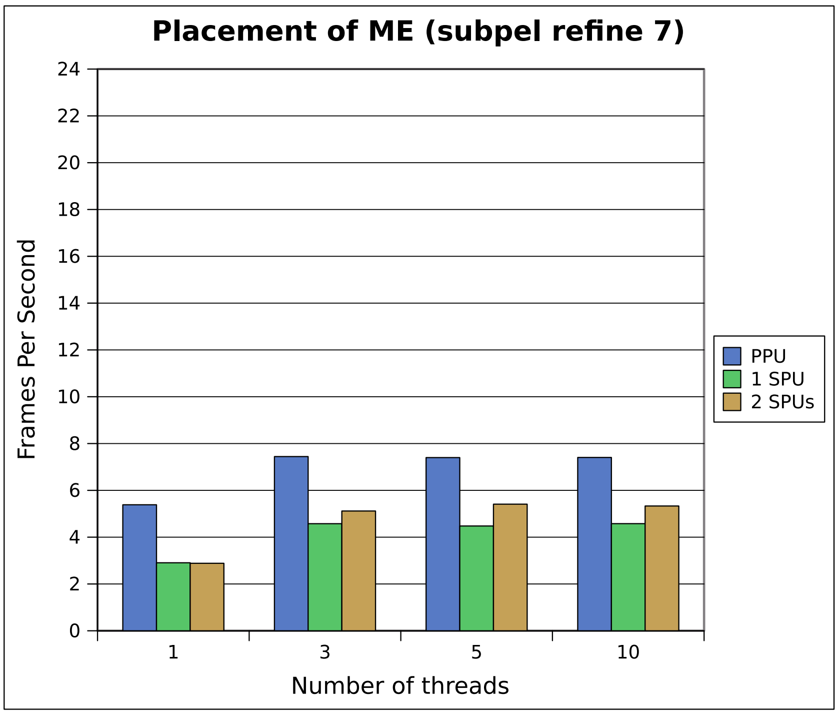


Figure 6.15: Motion estimation function offloaded using subpixel refine parameter 7.

The graphs show that the original PPE based *me_search_ref* runs faster in all tested scenarios. Increasing the number of SPEs used for ME increases the overall FPS somewhat, but it is still short of the PPE implementation. The performance gap between the PPE and SPE version diminishes when increasing the subpixel refinement parameter and hence the amount of work performed in the offloaded function. This suggests that there is a high constant overhead by dispatching a motion estimation job to an SPE.

Figure 6.16 and 6.17 show the relative encoding speed (fps) compared to subpixel refinement 2. For subpixel refinement level 5 the PPE version runs at 56.8 % of the refinement level 2 version, while using a SPE runs at 75.4 % of the refinement level 2 version for a single thread. The relative comparison graphs show that increasing the refinement parameter has less impact on the encoding throughput when offloading to SPE(s) than using the regular PPE code. This is likely caused by a large constant overhead of dispatching a job to an SPE compared to the actual job elapsed time, and when increasing the amount of work, the overhead becomes less significant.

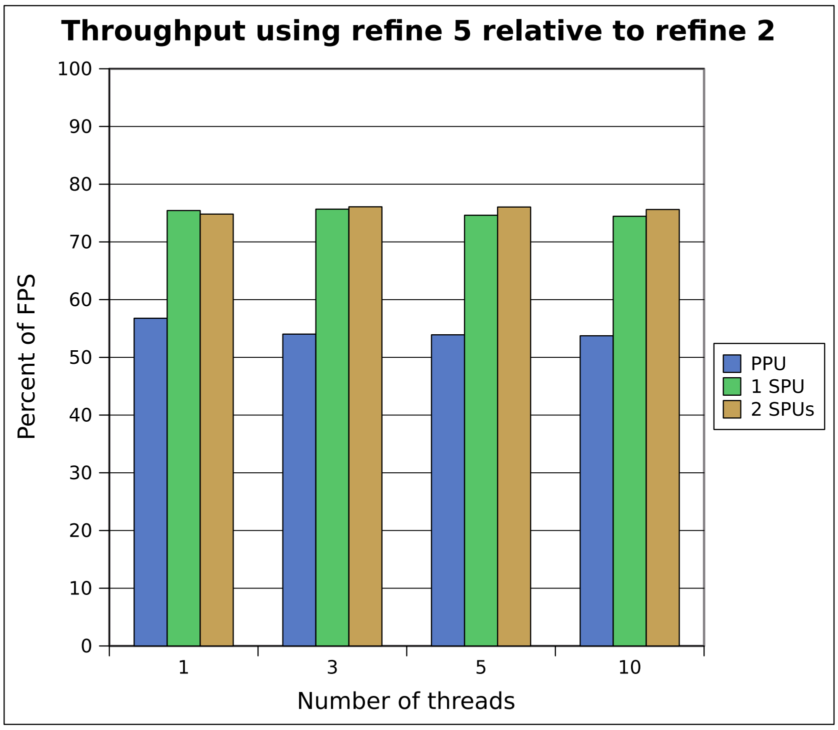


Figure 6.16: Relative encoding speed of motion estimation function offloaded using subpixel refine parameter 5.

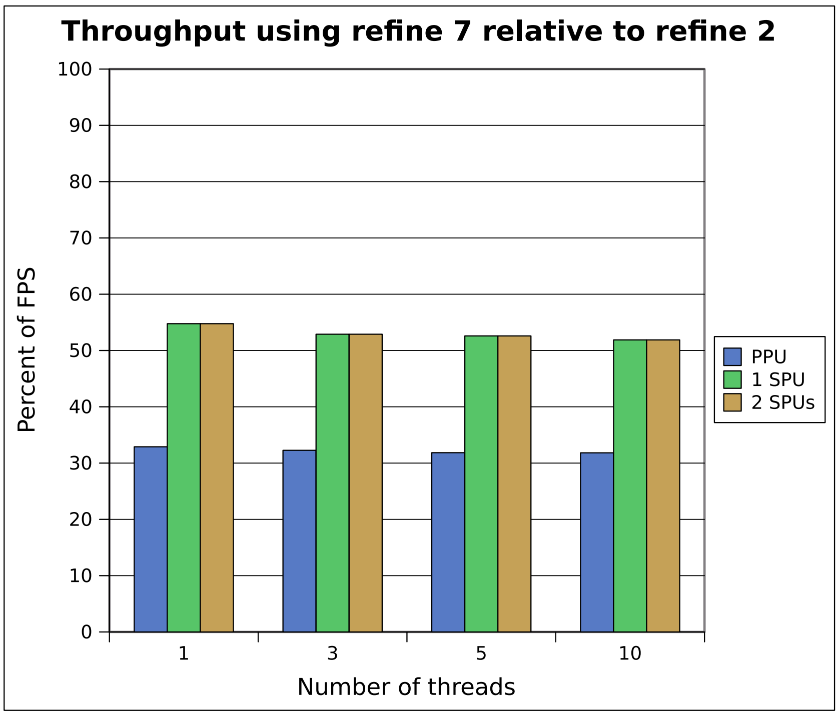


Figure 6.17: Relative encoding speed of motion estimation function offloaded using subpixel refine parameter 7.

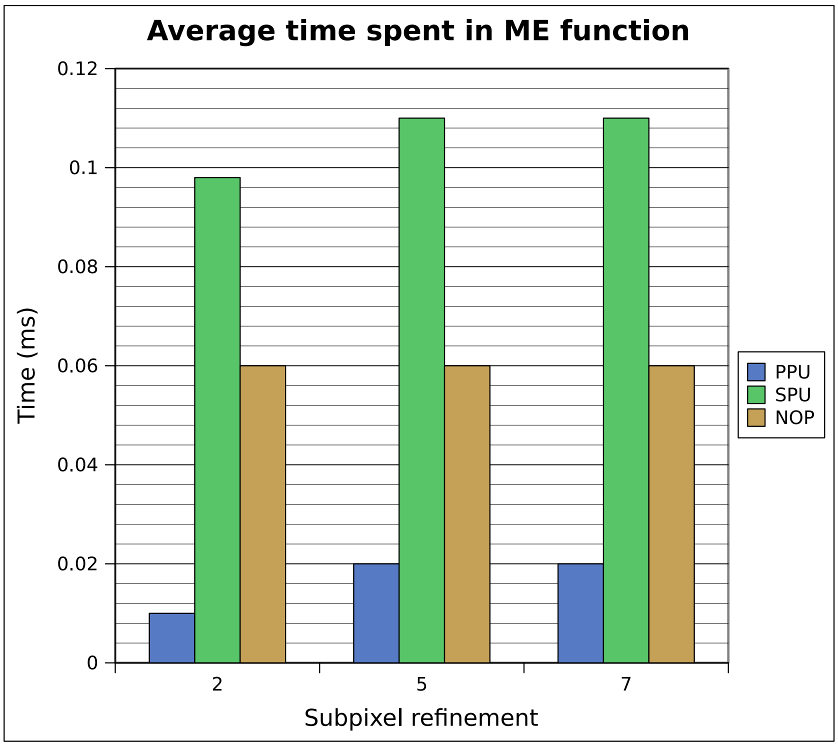


Figure 6.18: Average time spent in motion estimation search and refine function including job dispatching.

We also looked at the average time spent in the motion estimation search functions that were offloaded. Figure 6.18 shows benchmarks using the PPE version, the SPE version (including job dispatching), and a NOP function which we use to measure the overhead of dispatching a job to an SPE using our framework. The measurements were made using the gettimeofday() system call, and hence the precision of the data with sub-millisecond granularity is a bit uncertain. However, the data is consistent with the relative benchmarks, displaying an increase in ME running time of about 0.01 ms for both the PPE and SPE version when increasing the level of refinement. The NOP benchmark shows that dispatching a job accounts for more than half the time spent in the SPE function. The SPE overhead compared to the PPE version relates to DMA transaction overhead of copying planes and reference frames. The slowdown can also be caused by the SPE being slower than the PPE for some tasks, as shown in the n-queens experiment, although the relative performance when incrementing the refinement level suggest otherwise.

6.4 Lessons learned

Our scheme for offloading parts of the code to SPEs by using job dispatching worked well when each job was large enough to overcome the dispatch overhead. The HPEL filter processed rows of macroblocks on every request, and by offloading this function to an SPE, the overall performance was increased relative to the cycles it accounted for in the profiling session. The offloading of the motion estimation function, which accounted for most cycles in the profiling, did not give an performance increase because each call to the function worked on too little data to overcome the overhead.

To improve the feasibility of offloading motion estimation in x264, the amount of work performed during a job must be significantly larger. The calls to *x264_me_get_ref* can not simply be batched up, since many of them must run in sequence because of predictions. As mentioned earlier, some of the motion estimation searches for a macroblock are independent, e.g., when trying different reference frames, and can run in parallel on multiple SPEs simultaneously.

For our evaluated functions the local storage was large enough to accommodate useful data to process a job. For the hpel filter, we had room for one row of macroblocks having a frame width up to 1920 pixels. Supporting larger frames is possible, but may require running the filter in multiple passes. For motion estimation, we had room in local storage for searching up to 64 pixels in all directions. If the motion estimation requested an area outside of this region, a new rectangle was copied, centered around this request. This happened rarely, but was not benchmarked. Since motion estimation is likely the most memory- consuming operation of video encoding, the size of the local storage seems sufficient for video encoding/decoding.

The benchmarks did not take into account the processing speed of an SPE compared to the PPE, but looking at the decrease of performance when increasing the refinement level during ME shows that when the amount of work goes up, the increase in time spent performing the work on the PPE vs. SPE is about the same. This suggests that when ignoring the overhead, the PPE and SPE run motion estimation at about the same speed.

As further work, one may try to move parts of the analyse functions to SPEs to avoid job dispatching for every ME search request. This increases the amount of work performed for every ME request by grouping together jobs to cover multiple searches. If then, the amount of work exceed the dispatching overhead, a large increase in throughput should be observed. It is worth noting that moving analyse functions to SPEs may lead to a shortage of local storage, since the analyse functions consist of fairly large amounts of code, even though the profiling shows they account for a fraction of the cycles.

The overall impression from this experiment is that it is possible to gain performance by offloading time-consuming parts to SPEs without completely rewriting the code to map the target architecture. This allows portability, and reuse of existing code.

6.5 Summary

In this chapter, we have given an overview of the H.264 codec features, and described the basics of encoding and decoding a video. An open source implementation of the H.264 standard - x264 - was chosen as the basis of our experiment. x264 was profiled, and based on this result, two parts of the encoder, namely the hpel filter and motion estimation searching, were offloaded to SPEs as a step in mapping the encoder to the Cell platform. The benchmarks show the first part successfully gains overall encoder speedup. The second part did not improve overall efficiency, because of the high constant overhead of job dispatching compared to the time spent on the actual work of the motion estimation search and refine functions.

Chapter 7

Discussion

7.1 Performance and potential

We have investigated using heterogeneous architectures and their ability to support different applications. Of the problems we looked at, the AES encryption was the most successful, scaling almost linearly with regards to the number of processing units. This was because of the completely independent processing for blocks, and data was used in such a way that it could be transferred as a background operation.

The n-queens threads work independently, and provided decent scalability, but the benchmarks showed that the SPEs were much slower than the PPE. This was related to the CBEA's lack of branch prediction unit on the SPEs. Compared to a dual core Opteron, the PPE was a bit slower than a core on the x86, however, the PPE and all SPEs combined were faster than the two cores of the Opteron. For the AES experiment, the PPE and SPE throughput were roughly equal because of the branch-free implementation. This demonstrates that the efficiency of solving a given problem is highly architecture dependent.

The video encoding experiment was split in two parts. The moving of HPEL calculations to an SPE provided good offloading for the PPE, by working on a large amount of data (row of macroblocks). This was in-line with the profiling session with a speed-up relative to the cycles the filter accounted for. The offloading of motion estimation, however, did not work as well as could be hoped for. This part also uses the majority of cycles in the encoding process. By looking at the benchmarks, it became clear that a single motion estimation search job is too small in terms of time to perform a search request compared to the dispatch overhead, which ultimately led to original version being faster than the offloaded version. This does not mean that offloading the motion estimation to an SPE is a bad idea, but each job must be more time-consuming. This can be accomplished by moving a larger part of the analysis functions to SPEs.

Based on the experiences with x264, it seems likely that to fully take advantage of architectures like CBEA, programs must be designed specifically for the target architecture, otherwise major rewriting of the code may be necessary. Getting decent performance requires working units large enough to neglect the dispatching overhead when using a job-based model. It is also clear that computationally intensive programs with working units able to fit in local processing unit memory are preferred when using an architecture without shared memory. Otherwise, the processing elements can not get working units in advance, and must fetch the data from main memory when needed. The small size of the local storage is one of the challenges when programming for the CBEA.

The throughput of SPEs demonstrated in the AES evaluation shows the architecture potential in high-performance areas when care is taken to adapt the program to the architecture. The AES algorithm used is branch-free during encryption, and work units are small enough to fit in local storage. Such type of work is ideal for platforms like CBEA, and shows potential for a number of computationally expensive applications.

7.2 Parallelization strategies

In our experiments, we have looked at different strategies for distributing workloads to processing units. In the protocol translation application we used hardware channels to communicate between the cores. The address of the packet in shared memory was sent from the RX core, and based on packet inspection it was either handled locally, or sent to another core for further processing. In the latter case, it was received back on a different channel, and sent for transmitting on another channel. This model allows for simultaneous processing of packets, with very low overhead, because of the shared memory, and by communicating with only a pointer to the packet. Another advantage of this model is that using mutexes on packets are unnecessary, because a core stops accessing the packet as soon it enqueues it for another core. The model used is a pipeline model, where only a subset of the packets are forwarded to the XScale core. Weng and Wolf [56] have evaluated parallelization strategies for network processors by simulating scenarios with different pipeline width and length. Their results show that increase in throughput for pipeline width is limited by the memory bandwidth. The throughput is proportional to pipeline depth. This model is in line with our proxy scheme using a dedicated receive core, a forward core, and transmit core in the fast path. The IXP2400 has two memory channels, and our proposed strategy of using multiple forwarding engines should give an increase in throughput until the memory bandwidth is saturated.

The simplest work distributing strategy we tested was used in the n-queens problem. Because of the nature of solving the problem, no data or shared state had to be distributed after the initial start parameters, meaning that only communication taking place after the start is the finished message containing the result.

When benchmarking the AES experiment, we tried two different work distribution techniques. Both were job-based, meaning that a job was dispatched with parameters and data, and when finished, the results were returned. The first technique dispatched a job to an SPE by using the hardware channels, and when the job finished, a new job was dispatched. This scaled well as long as the PPE did nothing else, but when the PPE was under load, performance dropped. To avoid this, we added a software-implemented queue that all the SPEs (and PPE) could pull jobs from. This allowed the PPE to be utilized as long as the queue was large enough to avoid it being emptied while the dispatch thread was scheduled out.

The technique investigated on x264 was an offloading approach, where we combined the built-in parallelism written for SMP scalability, with SPE offloading of time-consuming functions. The offloading allowed the main processor to schedule in an idle thread, while the offloaded function executed. The job dispatching was done in a similar manner as the AES experiment. The offloading model worked well when jobs were reasonably sized, but slowed down execution in an experiment with small job sizes. The advantage of this approach is that it allows existing programs to be gradually ported by offloading individual parts to processing units, without completely rewriting the flow of the program.

We did not get a chance to explore pipelining techniques on CBEA since the problems we investigated did not seem applicable. Offloading more of the analysis phase of x264 to SPEs may very well benefit from such techniques, but this was, because of time constraints, not been looked into further. Based on the experience obtained from the above applications, it seems unlikely that a single scheme that works in all cases is sufficient to distribute work to processing elements on heterogeneous architectures.

7.3 Manual labour

A common observation for all the problems we have looked at, is that significant extra work has to take place to utilize the different processing units compared to traditional SMP. The processing units are often stripped of features, and only provide high performance working on specialized tasks. This simplicity is also the reason why these architectures have many cores. They are efficient when working with problems where the majority of the execution time does not require extra hardware like branch prediction, out-of-order execution, and shared memory. Other problems completely rely on these features, and are much better off using traditional fat-core architectures like x86.

The first architecture we investigated, the Intel IXP has a shared memory accessible from all its cores. This makes programming easy, by being able to use shared data structures and passing pointers between cores. Developers must still be aware of alignment, and what types of memory that are used for different data. The challenging part of using the IXP architecture is mainly developer tool related. The SDK 3.5 provided C compiler is not very robust, and the debugging support is limited.

A common method for writing parallel programs today involve using threads (e.g. pthreads) optimized for shared memory architectures. There are also higher-level threading libraries available, noteworthy open multi-processing (OpenMP) [57] and Intel threading building blocks (TBB) [58]. Both work by adding statements advising the compiler of parallel independency of functional blocks. They produce code that works with shared memory architectures, and are considerable easier to use than doing low-level threading and synchronization manually. Another approach is the message passing interface (MPI). It is often used for HPC applications because of its distributed nature. MPI does not require shared memory, because it was designed to communicate by messages. These messages can be sent by network interconnect or to another process on the same machine. Ohara et al. [59] developed a framework for MPI microtasking targeting the CBEA. This framework allows very small code snippets to be automatically distributed to SPEs, and the microtasks communicate by MPI messages. The ease of programming in this model comes at the cost of scheduling overhead and communications for the microtasks. The MPI microtasking framework is a research project of IBM, and is not yet available for evaluation. Other frameworks for parallelization on CBEA, both free and commercial, includes CorePy [60], RapidMind [61], and Cell SuperScalar [62]. Others are also available, but the frameworks have not been evaluated in this work.

It is likely that programming models like MPI are necessary to get high performance out of heterogeneous architectures without hand-coding memory transfers. Having multiple threads does not improve performance if there is no data available for the cores to process, leaving the processing units idle. For non-shared memory architectures, the main challenge is often memory management. This includes partitioning data such that multiple threads can use the data simultaneously. Another issue is the overhead of synchronization between threads.

Instead of parallelizing all of the program, it is possible to narrow down the effort to certain cpu intensive parts. Using heterogeneous cores as accelerators is nothing new, in fact even the x86 architecture has supported offloading floating point instructions to a dedicated math co-processor (FPU) since the first x86 processor - the 8086. Now, the FPU is on-die with the main processor(s), but parallel execution of instructions remains; leaving the CPU able to execute instructions while running other instruction on in multiple pipelines. Modern processors of the x86 architecture have multiple execution units yielding large instruction-level parallelism scalability. This parallelism is mostly hidden by the out-of-order execution model of x86 and optimizing compilers, leaving many programmers completely unaware of the underlying parallelism. Optimizing compilers are not perfect, and while reordering and rewritings of functions, especially with regard to cache usage may provide better performance, hand-optimizing is often not prioritized.

7.4 Architectural considerations

The use of Intel IXP in the protocol translator was chosen because of its properties as a network processor. There are other similar network processor on the marked, but the Intel IXP was chosen because we had access to one. The Cell Broadband Engine, on the other hand, is a special architecture with no direct competitors. The success of the Playstation 3 also makes the architecture available in millions of homes all over the world. This combined computing power is already utilized by the Folding@Home project [35] which is a distributed protein folding project. Even though the CBEA is a new architecture, the wide deployment and use has made it mature fast. The development tools are good, and Linux support for the platform is continuously improving. This made the CBEA a good choice our investigation.

Even more common heterogeneous architectures are GPUs. They are virtually in every commodity PC sold, and there is even a GPU in the PS3. The usability of GPUs for non-graphics related work has surged with the introduction of high-level programming frameworks like Nvidia's CUDA. However, getting decent performance from GPUs requires, like CBEA, code written to take advantage of the target hardware. Using GPUs can result in many times the performance compared to a regular x86 cpu, but as with the other heterogeneous architectures, GPUs require platform consideration of the application to deliver throughput.

The requirement of specializing the code to match the target platform, suggests that most programs will not undergo the effort of optimizing for architectures like CBEA or GPUs, even though they may be available as accelerators on the target platform. Still, general purpose computation-intensive tasks like encryption, image filtering, motion compensation, or DCT can be exposed to applications by a framework that offloads the task to an accelerator. The XvMC [63] is an example on an API that provides such a framework for video decoding on Linux. By using such frameworks, applications can take advantage of available accelerators in the system without having to care for what architecture that handles the task.

7.5 The future

The complexity of writing code for use on heterogeneous architectures is daunting compared to traditional single-threaded programming, and I believe that unless the compiler, or a high-level library provides decent performance without doing explicit memory management, non-shared memory architectures will mostly be used by high-performance applications or as accelerators when appropriate.

Chapter 8

Conclusion

8.1 Summary

In this thesis, we have investigated the potential of parallelization schemes on heterogeneous multi-core architectures. This was done using a hands-on approach, by either adapting existing projects to the target platform, or by writing the programs from scratch.

During our investigations, we have examined the potential of two recent architectures; the Intel IXP, and the STI Cell Broadband Engine. The IXP network processor was evaluated with a pipeline scheme for packet processing in a network proxy using three specialized cores and one general purpose processor.

For the CBEA, we have evaluated efficiency of branch-heavy applications performed on the two different types of cores on the CBEA. We also compared this to the x86 architecture, and found that the SPE on the CBEA was the slowest, but since there are several processing elements available, their combined throughput outperformed the competitors. A comparison for AES encryption, shows that for our implementation both the SPE and PPE provide similar throughput. We have also looked at job distribution of independent jobs, and evaluated using the CBEA hardware channels compared with a software based FIFO implementation for job distribution. In contrast to the limited length built-in channels, the SPEs received jobs at a steady rate when using the FIFO, even when the PPE was fully utilized. Based on this result, we examined using the job dispatching scheme in a video encoding project by offloading computationally intensive parts. Based on a profiling session, we evaluated offloading two different parts of the encoder, the HPEL filter and the motion estimator, and achieved an increase in throughput for the former. The latter part suffered from high dispatch overhead compared to the job execution time.

8.2 Further work

During our investigation, we have looked at a number of applications and related parallelization challenges. Unfortunately, we did not find time to follow up on all loose ends, and some are left as further work. This includes work both on implementations and evaluations. With regards to implementation, the most promising work that should be further looked into is the motion estimation offloading of the video encoder. According to the profiling session done on the video encoder, accelerating motion estimation has huge potential for increased throughput.

Further increase in throughput for both the video encoding and AES encryption may be achieved by utilizing SIMD capabilities of the architectures. Since SPE, PPE, and x86 all support 128 bit SIMD instructions, however for our comparisons, we did not use SIMD for any architectures. Although, it would be interesting to investigate at how this relates to our evaluations. Adapting the AES implementation to SIMD requires a major rewrite, since the table lookup scheme used is too large to fit in registers. Another implementation approach is to use a bit-slicing technique for encryption. Exploring SIMD capabilities will likely not effect our job distribution strategy, instead it may increase the total throughput, and is by itself an interesting evaluation of the applicability of architectures.

The n-queens implementation did not have a parallel strategy that scales to many cores for a small size of n. This was not a problem for our evaluation, but later architectures may feature a high number of cores, and further improvements should take this into account when distributing jobs. Another implementation specific feature we did not look further into was using more cores of the network processor. The IXP2400 platform used in the tests have eight μEngines in total, and we only used three of them. Strategies for using multiple forwarding engines, or integrating the extra cores in the pipeline may affect both latency and throughput, and should be look into.

As for further work in the evaluation part of this thesis, latencies in dispatch overhead during job distribution of the AES and video encoder should be investigated. The transparent proxy was also not evaluated with regards to network throughput, latency, and stability. It would also be interesting to compare the proxy running on a specialized platform, with a standard host PC using a full protocol stack.

8.3 Conclusion

We have shown that when using heterogeneous architectures, we were able to achieve good throughput for some applications, and good potential for other problems. The heterogeneous architectures need adaptations of the code to show their full potential, and the parallelization strategy chosen is essential. Not all problems benefit from heterogeneous architectures like the CBEA, where the processing elements are simple, with limited hardware, and no shared memory available. Thus, it is important to consider the target architecture's capabilities and limitations when programming for heterogeneous multiprocessors.

Appendix

Attached is a CD-ROM containing all source code used and benchmark results for the CBEA evaluation. The protocol proxy source code is unfortunately not distributable because of NDA protection on the IXP SDK.

The content of the CD-ROM can also be found at the following address:

`http://www.ping.uio.no/~gus/master/`

Bibliography

[1] Intel Corporation. Intel(r) ixp2400 network processor product brief. http://www.intel.com/design/network/prodbrf/279053.htm.

[2] Tom R. Halfhill. Parallel processing with cuda. www.mpronline.com, January 2008.

[3] Wikipedia. Eight queens puzzle — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Eight_queens_puzzle&oldid=222480125, 2008. [Online; accessed 16-July-2008].

[4] Wikipedia. Advanced encryption standard — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Advanced_Encryption_Standard&oldid=222103768, 2008. [Online; accessed 01-July-2008].

[5] Wikipedia. Block cipher modes of operation — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Block_cipher_modes_of_operation&oldid=221317021, 2008. [Online; accessed 02-July-2008].

[6] Wikipedia. Yuv — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Y'UV&oldid=222386744, 2008. [Online; accessed 01-July-2008].

[7] Iain E. G. Richardson. H.264 mpeg-4 part 10 white paper, 2002. www.vcodex.com.

[8] ITU-T. *Advanced video coding for generic audiovisual services*, 2003. ITU-T Recommendation H.264.

[9] Ballistic Research Laboratories. A fourth survey of domestic electronic digital computing systems report no. 1227. http://ed-thelen.org/comp-hist/BRL64.html#TOC, 1964.

[10] Ibm multicore research and design. http://domino.research.ibm.com/comm/research_projects.nsf/pages/multicore.index.html, 2007.

[11] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 81, Washington, DC, USA, 2003. IEEE Computer Society.

[12] Håvard Espeland, Carl Henrik Lunde, Håkon Kvale Stensland, Carsten Griwodz, and Pål Halvorsen. Transparent protocol translation for streaming. In *MULTIMEDIA '07: Proceedings of the 15th international conference on Multimedia*, pages 771–774, New York, NY, USA, 2007. ACM. Short paper.

[13] Håvard Espeland, Carl Henrik Lunde, Håkon Kvale Stensland, Carsten Griwodz, and Pål Halvorsen. Transparent protocol translation and load balancing on a network processor in a media streaming scenario. In *NOSSDAV '08: Proceedings of the 2008 international workshop on Network and operating systems support for digital audio and video*, New York, NY, USA, 2008. ACM. Demo paper.

[14] IBM. *Cell Broadband Engine Architecture*, 2006.

[15] Samuel K. Moore. Winner: Multimedia monster, 2006. http://www.spectrum.ieee.org/jan06/2609.

[16] Ars Technica. A technical overview of the emotion engine. http://arstechnica.com/reviews/hardware/ee.ars.

[17] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, , and D. Shippy. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5), 2005.

[18] Toshiba. G50 laptop. http://explore.toshiba.com/laptops/qosmio/G50.

[19] IBM. *Cell Broadband Engine Programming Handbook*, 2007.

[20] IEEE. Standard for binary floating-point arithmetic. *ANSI/IEEE Std 754-1985*, 1985.

[21] Arnd Bergmann. Linux on cell broadband engine status update. In *Proceedings of the Linux Symposium*, pages 21–28, June 2007.

[22] The gnu project debugger. http://sourceware.org/gdb/.

[23] IBM. *SPE Runtime Management Library*, 2007.

[24] A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo. Using advanced compiler technology to exploit the performance of the cell broadband engine architecture. *IBM Systems Journal*, 45(1), 2006.

[25] John Greene. Developing optimized spu assembly code for the cell. In *Summit on Software and Algorithms for the Cell Processor*, October 2006.

[26] Gerhard Zumbusch. Parallel for. `http://www.xfree86.org/~mvojkovi/XvMC_API.txt`.

[27] Gerhard Zumbusch. A container-iterator parallel programming mode. *Lecture Notes in Computer Science*, 4967:1130–1139, 2008.

[28] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. The potential of the cell processor for scientific computing. In *Proceedings of the 3rd conference on Computing frontiers*, pages 9–20, Ischia, Italy, 2006. ACM Press.

[29] Alfredo Buttari, Piotr Luszczek, Jakub Kurzak, Jack Dongarra, and George Bosilca. A rough guide to scientific computing on the playstation 3, May 2007.

[30] Ryuji Sakai, Seiji Maeda, Christopher Crookes, Mitsuru Shimbayashi, Katsuhisa Yano, Tadashi Nakatani, Hirokuni Yano, Shigehiro Asano, Masaya Kato, Hiroshi Nozue, Tatsunori Kanai, Tomofumi Shimada, and Koichi Awazu. Keynote speech: Programming and performance evaluation of the cell processor. Number 17. HOT CHIPS, 2005.

[31] Intel Corporation. Intel IXP2400 network processor datasheet, February 2004.

[32] AMD. Amd close to metal technology unleashes the power of stream computing. `http://www.amd.com/us-en/Corporate/VirtualPressRoom/0,,51_104_543~114147,00.html`.

[33] Nvidia. Compute unified device architecture. `http://www.nvidia.com/object/cuda_home.html`.

[34] Michael Schatz, Cole Trapnell, Arthur Delcher, and Amitabh Varshney. High-throughput sequence alignment using graphics processing units. *BMC Bioinformatics*, 8(1):474, 2007.

[35] Folding at home. http://folding.stanford.edu/.

[36] M.L. Curry, A. Skjellum, H.L. Ward, and R. Brightwell. Accelerating reed-solomon coding in raid systems with gpus. *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–6, April 2008.

[37] M. Handley, S. Floyd, J. Padhye, and J. Widmer. TCP Friendly Rate Control (TFRC): Protocol Specification. RFC 3448 (Proposed Standard), January 2003.

[38] Batu Sat and Benjamin W. Wah. Playout scheduling and loss-concealments in voip for optimizing conversational voice communication quality. In *MULTIMEDIA '07: Proceedings of the 15th international conference on Multimedia*, pages 137–146, New York, NY, USA, 2007. ACM.

[39] Adam Dunkels. uip. http://www.sics.se/~adam/uip/index.php/Main_Page.

[40] L. Kencl and J. Le Boudec. Adaptive load sharing for network processors, 2002.

[41] Niklaus Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4):221–227, 1971.

[42] Nation Institute of Standards and Technology. Fips 197 advanced encryption standard (aes), 2001.

[43] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A concrete security treatment of symmetric encryption. *Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on*, pages 394–403, Oct 1997.

[44] Philip J. Erdelsky et al. Rijndael encryption algorithm. http://www.efgh.com/software/rijndael.htm.

[45] Atri Rudra, Vijay Kumar, Pradeep K. Dubey, Raghav Bhaskar, and Animesh Sharma. Data-sliced implementation of reed-solomon and rijndael on simd processors. http://citeseer.ist.psu.edu/553736.html.

[46] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation¿a performance view. *IBM Journal of Research and Development*, 2006.

[47] Ashish Jagmohan, Brent Paulovicks, Vadim Sheinin, and Hangu Yeo. H.264 video encoding algorithm on cell broadband engine. IBM Whitepaper.

[48] ISO/IEC. *ISO/IEC 14496-10:2003*, 2003. Information technology - Coding of audio-visual objects - Part 10: Advanced Video Coding.

[49] Wikipedia. H.264/mpeg-4 avc — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=H.264/MPEG-4_AVC&oldid=223178169, 2008. [Online; accessed 04-July-2008].

[50] Thomas Wiegand, Gary J. Sullivan, Gisle Bjøntegaard, and Ajay Luthra. Overview of the h.264/avc video coding standard. *IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS FOR VIDEO TECHNOLOGY*, 13(7), 2003.

[51] x264 - a free h264/avc encoder. http://www.videolan.org/developers/x264.html.

[52] MSU Graphics & Media Lab (Video Group). Fourth annual msu mpeg-3 avc/h.264 video codec comparison, December 2007.

[53] Valgrind tool suite. http://valgrind.org/.

[54] Kcachegrind - profiling visualization. http://kcachegrind.sourceforge.net/.

[55] Cipr video test sequences. http://www.cipr.rpi.edu/resource/sequences/.

[56] Ning Weng and Tilman Wolf. Pipelining vs. multiprocessors - choosing the right network processor system topology. In *Proceedings of Advanced Networking and Communications Hardware Workshop (ANCHOR)*, June 2004.

[57] Open multi-processing. http://openmp.org.

[58] Intel. Threading building blocks. http://www.threadingbuildingblocks.org/.

[59] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani. Mpi microtask for programming the cell broadband engine processor. *IBM Systems Journal*, 45(1), 2006.

[60] Christopher Mueller, Ben Martin, and Andrew Lumsdaine. Corepy: High-productivity cell/b.e. programming. GA Tech/STI Cell/B.E. Workshop, 2007.

[61] Rapidmind. Rapidmind. http://www.rapidmind.net/.

[62] Barcelona Supercomputing Center. Cell superscalar. http://www.bsc.es/cellsuperscalar.

[63] X-video motion compensation. http://www.xfree86.org/~mvojkovi/XvMC_API.txt.